

IMAGE AND VIDEO PROCESSING
ON XILINX ZYNQ ULTRASCALE+ MPSoC

A THESIS SUBMITTED TO
THE FACULTY OF ARCHITECTURE AND ENGINEERING
OF
EPOKA UNIVERSITY

BY

KEVIN SELMANHASKO

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRONICS AND DIGITAL COMMUNICATION ENGINEERING

February, 2024

Approval sheet of the Thesis

This is to certify that we have read this thesis entitled “**Image and video processing on Xilinx Zynq Ultrascale+ MPSoC**” and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Arban Uka
Head of Department
Date: February, 28, 2024

Examining Committee Members:

Dr. Florenc Skuka (Computer Engineering) _____

Dr. Shkëlqim Hajrulla (Computer Engineering) _____

Assoc. Prof. Dr. Arban Uka (Computer Engineering) _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name Surname: Kevin Selmanhasko

Signature: _____

ABSTRACT

IMAGE AND VIDEO PROCESSING ON XILINX ZYNQ ULTRASCALE+ MPSoC

Selmanhasko, Kevin

M.Sc., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Arban Uka

Field-programmable gate arrays (FPGAs) have become a popular choice for high-speed data transmission systems due to their high performance and flexibility. FPGA-based systems are increasingly being used in various applications, such as radar, medical imaging, autonomous driving, and quantum computing. Within this paper, a detailed examination is presented, offering insights into FPGA-centric image and video processing, accompanied by a thorough exploration of the associated design considerations. The FPGA used in this thesis is the ZCU104. The advantages of FPGA-based systems, including high-speed data processing, low latency, and reconfigurability are discussed. The paper also emphasizes certain challenges and limitations associated with the design of FPGA-based data transmission systems. To illustrate the practicality of FPGA data processing, PYNQ has been used as it contains Open Computer Vision Library (CV). The main focus has been implementation and comparison of different methods used for image and video processing. In order to measure the performance of these algorithms, several aspects have been taken into consideration such as accuracy or frame rate. In addition, the limitations of this hardware have been discussed. The aim of this thesis is analyzing the performance of these processing algorithms and what can be done for future improvements. Overall FPGA are an excellent choice compared to the traditional approaches.

Keywords: *FPGA, PYNQ, Image Processing, Video Processing.*

ABSTRAKT

PROCESIMI I VIDEOVE DHE IMAZHEVE NE XILINX ZYNQ ULTRASCALE+ MPSoC

Selmanhasko, Kevin

M.Sc., Departamenti i Inxhinierisë Kompjuterike

Udheheqësi: Assoc. Prof. Dr. Arban Uka

Hardware-t e rikonfigurueshëm (FPGA) janë bërë një zgjedhje popullore për sistemet e transmetimit të të dhënave me shpejtësi të lartë për shkak të performancës dhe fleksibilitetit të tyre të lartë. FPGA-te po përdoren gjithnjë e më shumë në aplikacione të ndryshme, si radarët, imazhet mjekësore, drejtimi autonom dhe llogaritja kuantike. Në këtë teme shkencore paraqitet një shqyrtim i hollësishëm, duke ofruar perspektivë në procesimin e imazheve dhe videove të përqendruar në FPGA. FPGA-ja e përdorur në këtë teme është ZCU104. Jane diskutuar avantazhet e sistemeve bazuar në FPGA, duke përfshirë procesimin e të dhënave me shpejtësi të lartë, vonesa të ulëta dhe rikonfigurueshmëria. Jane theksuar gjithashtu disa sfida dhe kufizime të lidhura me dizajnin e sistemeve të transmetimit të të dhënave bazuar në FPGA. Për të ilustruar procesimin e të dhënave në FPGA, është përdorur PYNQ pasi përmban Open Computer Vision Library (CV). Fokusi kryesor është te implementimi dhe krahasimi i metodeve të ndryshme të përdorura për procesimin e imazheve dhe videove. Për të vlerësuar performancën e këtyre algoritmave, janë marrë në konsideratë disa aspekte si saktësia ose frekuenca vizuale. Gjithashtu janë diskutuar kufizimet e këtij hardueri. Qëllimi kryesor I ketij punimi shkencor është analizimi i performancës së këtyre algoritmave të procesimit dhe çfarë mund të ndërmerret për tu permirsuar ne te ardhmen. Në përgjithësi, FPGA-te janë një zgjedhje shumë e mirë krahasuar me metodat tradicionale.

Fjalet kyçe: FPGA, PYNQ, Procesim Imazhesh, Procesim Videosh.

Dedicated to my family and friends!

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Assoc. Prof. Dr. Arban Uka for the help he has given me to write this thesis and for all his advices, encouragement and support during all my years studying in this University. I sincerely appreciate the time and effort he has spent to improve my experience during these years.

TABLE OF CONTENTS

ABSTRACT	iii
ABSTRAKT	iv
ACKNOWLEDGMENTS	vi
LIST OF TABLES	ix
LIST OF FIGURES	x
LIST OF ABBREVIATIONS	xii
CHAPTER 1	1
INTRODUCTION	1
1.1 Field-Programmable Gate Arrays	1
1.2 Video Processing Pipelines.....	1
1.3 Video Processing on FPGAs.....	3
1.4 High-level Synthesis	4
1.5 Python Productivity for Zynq (PYNQ).....	4
CHAPTER 2	9
METHODS AND MATERIALS	9
2.1 Materials	9
2.1.1 Hardware Components	9
2.1.1.1 ZCU104 board – XILINX ZYNQ ULTRASCALE+ MPSoC EV.....	9
2.1.2 Software components	13
2.1.2.1 PYNQ 3.0 with VIVADO 2022.1	13
2.1.2.2 PYNQ Overlays.....	14

2.1.2.3 AXI DMA.....	18
2.1.2.4 AXI VIDEO DMA	20
2.1.2.5 Open Computer Vision (CV).....	22
2.2 Methodology	23
2.2.1 Steps for connecting PYNQ to FPGA	23
2.2.2 Image Processing Filtering	24
2.2.3 2D Convolution	25
2.2.3.1 Sobel Filter	26
2.2.3.2 Laplacian Filter.....	27
2.2.3.4 Gaussian Filter.....	28
2.3.1 Steps for implementation of the custom overlay	30
RESULTS AND DISCUSSION.....	33
3.2 Results of the Real time video in the Jupyter Notebook	38
3.2.1 The Frame Rate Comparison:.....	38
3.2 Results from of Real-Time Video from USB camera and pre-saved Video) ..	39
3.2.1 USB Camera Input (1280x720).....	39
3.2.2 Pre-saved video Input (768x576)	39
3.2.3 Power Consumption Comparison:.....	40
3.2.4 Data read from disk:	40
3.3 Results from the HLS IP C simulation for Gaussian Filter 5x5:	41
3.4 Results from the HLS IP C simulation for Sobel Filter:.....	42
CONCLUSION	44
REFERENCES	45

LIST OF TABLES

Table 1. ZCU104 Resources [34]	12
Table 2. PYNQ Versions compatible with VIVADO.....	13
Table 3. Side to Side Comparison of Filters Implementation on Software	38
Table 4. Comparison of Hardware and Software with the input coming from the USB Camera Input	39
Table 5. Comparison of Hardware and Software with the input coming from the pre=saved video "vtest.mp4"	39
Table 6. Power Comparison 1240x720(Watts).....	40
Table 7. Data read from disk during hardware implementation	41

LIST OF FIGURES

Figure 1. The PYNQ Framework	6
Figure 2. Xilinx ZYNQ Ultrascale+ Evaluation Kit components and tools [32]	9
Figure 3. A detailed picture of Zynq UltraScale+ MPSoC ZCU104 and its interfaces	11
Figure 4. ZCU104 High Level device Diagram	12
Figure 5. Connection of the peripherals in the Base Overlay of ZCU104 board	14
Figure 6. Base Overlay Block design in PYNQ 3.0	16
Figure 7. Block design of Video on Base Overlay	16
<i>Figure 8.</i> Design of HDMI INPUT	17
Figure 9. Design of HDMI OUTPUT	17
Figure 10. The AXI interfaces on the AXI DMA Controller	18
Figure 11. An example of the AXI DMA connected in the PL of the Zynq MOSoC [38]	20
Figure 12. The AXI VDMA block diagram and AXI interfaces.....	21
Figure 13. The AXI VDMA video frame buffer example.....	22
Figure 14. Illustration for processing of pixel through neighborhood operations [42] .	24
Figure 15. 2D Convolution of Image and Kernel Window [50]	26
<i>Figure 16.</i> The illustration of both sobel operators Gy and Gx for the detection of edges in the horizontal and vertical direction respectively.....	27
<i>Figure 17.</i> The resulting image from MatLab of the complete Sobel filter as the magnitude of the x and y direction gradients	27
Figure 18. The result from MatLab of the convolution of the image with the Gaussian filter. [45].....	29
Figure 19. The result from the Canny Filter(left) and Sobel Filter(right) Compared side- by-side	29

Figure 20. Implementation of the IP in the block design for ZCU104.....	32
Figure 21. Initialization of the HDMI input and HDMI output.....	33
Figure 22. Code cell for the real-time video displayed on the HD monitor	34
Figure 23. Real time display of Laptop screen on PYNQ	34
Figure 24. Real Time Edge Detection Using Laptop Screen as an input	35
Figure 25. Cell code for the real-time video with Laplacian Edge Detection	36
Figure 26. Cell code for the real-time video with Canny Edge Detection	37
Figure 27. Cell code for the real-time video with Sobel Edge Detection.....	37
Figure 28. Implementation of real time canny filter on USB CAMERA.....	38
Figure 29. The simulation results of the HLS IP (1280x720) image	42
Figure 30. Graphical representation of the device and placed logic resources	43

LIST OF ABBREVIATIONS

ASIC	Application-Specific Integrated Circuits
AXI	Advanced eXtensible Interface
FPS	Frames Per Second
SRAM	Static Random-Access Memory
CPU	Central Processing Unit
GPU	Graphics Processing Unit
HDL	Hardware Description Language
IDE	Integrated Development Enviroment
MMIO	Memory Mapped INPUT/OUTPUT
RTL	Register Transfer Level
IPYHTON	Interactive PYTHON
OS	Operating System
AR	Augmented Reality
RF	Radio Frequency
TCM4	Tightly Coupled Memory
MMU	Memory Management Unit
API	Application Programming Interface
PMBus	Power Management Bus
CV	Computer Vision
DMA	Direct Memory Access
EV	Embedded Vision
FPGA	Field Programmable Gate Array
HD	High-Definiton
HDMI	High-Definition Multimedia Interface
HLS	High Level Synthesis
IP	Intellectual Property
MPSoc	Multiprocessor System on a Chip

OpenCV	Open-source Computer Vision
PL	Programmable Logic
DSP	Digital Signal Processing
PS	Processing System
PYNQ	Python Productivity on Zynq
RH	Reconfigurable Hardware
SoC	System on Chip
USB	Universal Serial Bus
VDMA	Video Direct Memory Access
LUT	Look Up Table
FF	Flip Flops
DSP	Digital Signal Processor
BRAM	Block Random-Access Memory
SRL	Shift Register Logic

CHAPTER 1

INTRODUCTION

1.1 Field-Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) are powerful integrated circuits that offer unparalleled flexibility and performance in digital hardware design. Unlike traditional application-specific integrated circuits (ASICs) that are fixed in functionality, FPGAs can be reprogrammed and customized to perform a wide range of tasks [1]. They consist of a matrix of configurable logic blocks and programmable interconnects, allowing designers to create complex digital circuits by programming the interconnections and functionality of these blocks [2]. This flexibility enables FPGAs to be used in diverse applications, including digital signal processing, embedded systems, high-speed communication, artificial intelligence, and more. High processing capability, parallelism, low-latency, and real-time responsiveness have been provided by FPGA-s, making them ideal for applications that require high-performance computing and hardware acceleration [3] [4] [5]. With the advancement of development tools and frameworks, FPGAs are becoming more accessible to designers, enabling them to leverage the benefits of hardware customization without the need for ASIC design expertise [6] [7] [8] [9] [10]. As a result, FPGAs continue to revolutionize the world of digital design and offer endless possibilities for innovation and optimization in various domains.

1.2 Video Processing Pipelines

Video processing is a form of digital signal processing (DSP) targeting video frame data. Video processing techniques can be used to improve the image, isolate target features, compress the video, or integrate intentional artificial features. Some examples

of video filters include grayscale, inversion, erosion, dilation, file compression, edge detection, and superimposed text or images. A standard video application includes several different filters implemented as sequential processing cores to produce the final processed image. These processing cores can be pipelined together in different stages to process a video frame from the raw input to the desired output. The pipeline improves throughput by having each stage perform a different video filter on sequential elements at the same time. Additional stages can be added to the pipeline while potentially maintaining the throughput rate.

Video pipelines can be generated to operate on a new pixel of the video each clock cycle and produce the processed video at a fixed latency. This pipelined processing allows the system to handle the bandwidth of a live video feed. Custom video processing pipelines can be tailored to meet the requirements of a specific application. For critical systems like surveillance video, custom pre-processing of live video can help provide better and quicker responses to any given situation. These pipelines can be used to improve the color quality output of the CMOS/CCD image sensor [11]. Another example shows smart video surveillance using a custom pipeline to improve the response time of the system and be able to make critical choices [12]. Central Processing Units (CPUs), Graphics Processing Units (GPUs), FPGAs, and custom application-specific integrated circuits (ASICs) have been used to produce these high-performance and low-latency video processing pipelines. One example of related work using video pipelines is stitching together images from separate cameras to create a live panoramic view of an event. Using a CPU and GPU, their pipeline provided live 7000×960pixel panorama images at 30 frames per second (fps). As video data bandwidth increases and more complex processing is required, custom ASIC solutions can be used to provide the necessary processing performance. This architecture operates on one pixel per clock cycle and generates a number of different processing primitives to apply to the incoming video data. It can be applied to a pipeline with CPU, GPU, and high-speed volatile memory to process live video from a sensor of 720p resolution at 30 fps. When video pipelines are implemented as software on CPUs and GPUs, they are limited by the given instruction set and the memory latency. Implementing the pipeline in the custom hardware of an ASIC can overcome these limitations and provide more efficient processing, but can lack

adaptability to change functionality. Though ASICs can be highly efficient, they require a great deal of time to develop and test and are impractical to deploy in small quantity applications due to their high costs. The research presented in this thesis illustrates that FPGAs can serve as an intermediary solution, striking a balance between the flexibility offered by software-driven general-purpose hardware (CPUs and GPUs) and the performance achieved by expensive custom ASICs.

1.3 Video Processing on FPGAs

FPGAs can implement custom video processing pipelines using a vast quantity of reprogrammable resources. SRAM-based FPGAs can be programmed repeatedly, an unlimited number of times, to implement new processing pipelines. These pipelines are defined in hardware description language (HDL) and take advantage of the Look-Up Table (LUT), DSP, and BRAM resources available on FPGAs [13]. FPGAs provide a powerful platform to implement complex video pipelines for custom applications. These pipelines have been employed to incorporate video filters such as Harris Corner, Sobel, Robert, Prewitt, and Laplacian filters on live video streams reaching 600×800 pixels at 60 Hz [14] [15]. Modern FPGAs could manage full 4k resolution at a 60fps rate while processing multiple inputs into one output. A considerable number of researches have applied FPGAs as pre-processors for CPUs or DSPs to implement the pipeline in a hybrid system of both hardware and software. Nevertheless, developing these pipelines on FPGAs can prove to be a difficult and time-consuming task. [16] FPGA programmers may require complex vendor tools in developing and testing RTL designs. As the digital design becomes more complex, the compilation and simulation times greatly increase. To overcome this low design productivity, researchers have explored various methods to rapidly deploy custom pipelines within an FPGA [17] [18] [19].

1.4 High-level Synthesis

FPGA vendors provide technologies to increase productivity with simple software to FPGA implementation transfer using already existing software libraries for video processing. Many of these technologies make use of high-level synthesis (HLS), which defines an algorithm and then compiles it down to a level of digital circuitry using software programming languages. An HLS design approach provides developers with a simpler entry point, quicker development time with less code, and hardware acceleration of software functions [20]. The Xilinx HLS tool defines how the hardware is implemented using vendor-specific pragmas in C++ while taking latency, pipelining, and throughput into account. As video processing pipelines are being developed, HLS enables quick design and testing [21]. The implementation of video encoders, Sobel edge detection, and other practical video processing operations up to 4k at 60 frames per second has been demonstrated to be successful [22] [23] [24]. With examples of how to use it across its devices, Xilinx has made the OpenCV library for software video processing available as an HLS implementation under the name OpenCV [25] [26].

1.5 Python Productivity for Zynq (PYNQ)

PYNQ is a project developed by AMD that simplifies the utilization of Adaptive Computing platforms. By leveraging the Python language and its libraries, designers can take advantage of programmable logic and microprocessors to construct electronic systems that are more advanced and captivating [27]. PYNQ is compatible with various platforms, including Zynq, Zynq Ultrascale+, ZynqRFSoc, Alveo accelerator boards, and AWS-F1. This compatibility enables the creation of high-performance applications that exhibit exceptional capabilities. PYNQ is designed to cater to a diverse set of designers and developers, including:

1. Software Developers: PYNQ allows software developers to leverage the capabilities of Adaptive Computing platforms without the need to use ASIC-style design tools for hardware development. They can utilize the software interface and framework

provided by PYNQ to harness the potential of platforms such as Zynq, Alveo, and AWS-F1.

2. **System Architects:** PYNQ offers system architects an effortless software interface and a framework that facilitates rapid prototyping and development of their designs on Zynq, Alveo, and AWS-F1. This allows them to quickly iterate and test their ideas, expediting the design process.
3. **Hardware Designers:** PYNQ caters to hardware designers who aim to maximize the reach and usability of their designs. By utilizing PYNQ, they can provide a user-friendly software interface and framework, enabling a wider audience to utilize their designs effectively.

PYNQ addresses the complexity of co-design by providing a pre-configured software stack, augmented by libraries of hardware and software components that can be selectively reused, depending on the target application. Figure 2 illustrates the general concept of the PYNQ framework and shows how the various PYNQ layers relate to those of a typical Zynq-based embedded system [28].

Upper Layer (Applications): At the top of the PYNQ stack, user interaction is facilitated by one or more Jupyter Notebooks. It is an open-source project that emerged from academia, with its roots in data science, and a key aim of the project was to further “reproducible science” [29]. Briefly, Project Jupyter facilitates scientific results to be presented in a manner that enables readers to reproduce and validate the claims of the authors, with reasonable effort [30]. It allows users to create interactive documents, known as ‘notebooks’, which are served via a standard web browser. These notebooks contain a variety of different content, including live executable code and visualizations, as well as textual, graphical, and mathematical documentation. Considering them being organized in cells makes the individual execution possible.

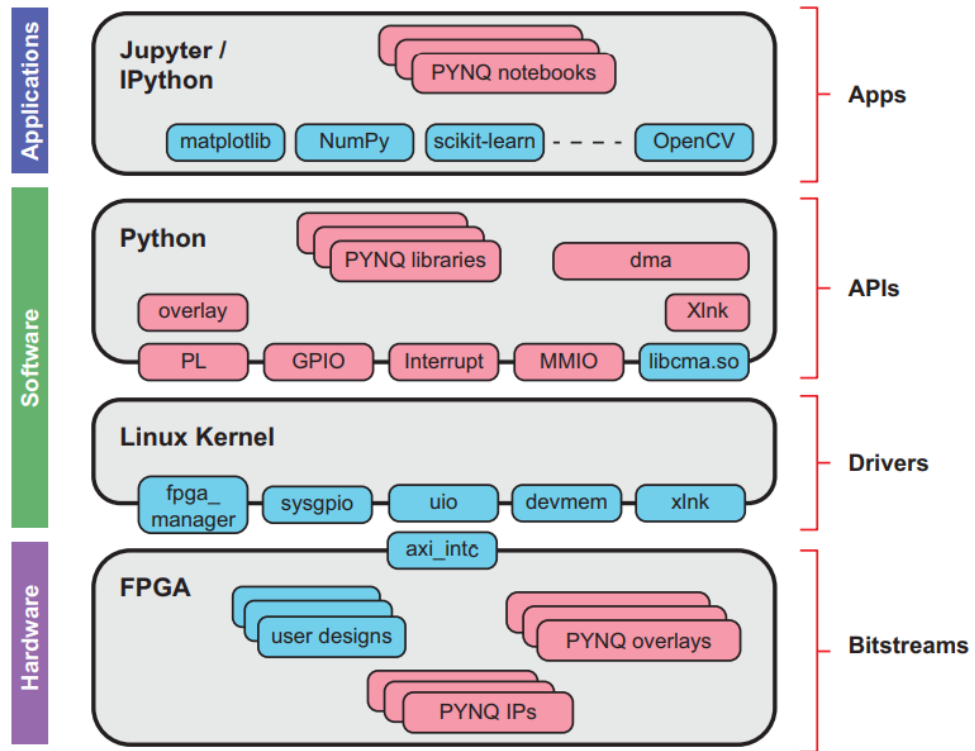


Figure 1. The PYNQ Framework

Originally known as IPython (Interactive Python) Notebooks and featuring only Python programming, Jupyter Notebooks now support a variety of programming languages [31]. More recently, Jupyter Labs has extended the concept to a complete web-based Integrated Development Environment (IDE). Jupyter Notebooks are an integral part of Jupyter Labs. A distinguishing feature of PYNQ is that its Jupyter Notebooks are hosted on Zynq’s Arm processor (i.e., an embedded device), whereas the Jupyter project was originally conceived for desktop and server computing. The notebooks reside on a webserver on the Zynq PS, and the user accesses them from a standard web browser over a network connection. An example of Jupyter Notebook is shown in Section 2.2.2.4 Within a PYNQ Jupyter Notebook, the developer creates their own custom functionality by writing their own Python code and selectively reusing third-party code from the many open-source Python libraries that are available. In addition they can add documentation and visualization content to help others understand and use the design. [30]

Middle Layers (Software): The mid-layers of the PYNQ stack consist of Python software, the Operating System (OS) and the low-level software drivers. In the upper middle layer, the PYNQ framework includes Python libraries and APIs for interacting with various elements of Zynq-based systems. For instance, there are Python APIs for downloading overlays (bitstream files) to the Programmable Logic (PL), communicating with GPIO resources, and handling interrupts generated in the PL. Additionally, there are Python APIs for memory-mapped transfers (MMIO) and DMA transfers. One of the most significant advantages of Zynq and Zynq MPSoC compared to other devices (or combinations of devices) is the ability to quickly move large amounts of data between CPU and PL, and vice versa. The use of PYNQ enables these transfers to be controlled in a very straightforward fashion using Python code. In the lower middle layer, the PYNQ framework includes a Linux-based OS, bootloaders to initiate system start-up, and a web server to host Jupyter Notebooks. Hence, the design effort of developing common software elements of an embedded system is removed, allowing new users to get started quickly with Zynq making this is a key benefit of the PYNQ framework. The lower middle layer includes a set of drivers for interacting with elements of the Zynq hardware system [30].

Lower Layer (Hardware): The bottom layer of the stack represents a hardware system design, which would normally be created in Vivado using Intellectual Property (IP) integrator and associated design tools, and then generated to a bitstream (*.bit) file. The bitstream file is transferred onto the memory card inserted into the target board. The process of programming the hardware system onto the PL can then be initiated directly from within a Jupyter Notebook (running on the PS), using a single line of code:

```
my_overlay = Overlay("/path/to/your/overlay/file/bitstream.bit")
```

In PYNQ, hardware system designs are often referred to as overlays. They have been used in a manner analogous to software libraries, wherein a hardware system has been developed for a particular application domain, but with an aspect of generality that facilitates enhanced sharing and reuse. Details of hardware designs can be abstracted and their functionality has been exposed in Python via an API, which enables a very software-centric style of using PYNQ. As outlined in the section above, one of the objectives of

PYNQ is to enable designers without hardware expertise to develop applications in software, based on pre-existing overlays. Furthermore, it helps hardware engineers create designs that can be evaluated and used by software engineers. Although overlays are often generalized designs, a more traditional hardware/software co-design approach could also be taken, wherein highly customized hardware is developed for a specific use case. Here, several advantages of the PYNQ framework can be leveraged, including the availability of a ready-made software stack, the ease of interfacing with elements of the developed hardware design and the potential to adapt and extend the software programming environment. The set of available PYNQ IPs can be freely reused — this includes interfacing blocks for DMA, audio, video, and I2C, and components from logic tools. Considering all these reusable components, the term hardware libraries could be defined. This is an umbrella term, and it refers to the set of IPs and overlays that are available as part of the PYNQ framework for flexible reuse. Hardware libraries may be considered analogous to software libraries. Certainly, additional hardware libraries can be created by developers for their own use, or for sharing with others [30].

In summary, PYNQ aims to provide an accessible and efficient platform for software developers, system architects, and hardware designers to harness the capabilities of Adaptive Computing platforms, fostering rapid development and expanding the usability of their designs.

CHAPTER 2

METHODS AND MATERIALS

2.1 Materials

2.1.1 Hardware Components

2.1.1.1 ZCU104 board – XILINX ZYNQ ULTRASCALE+ MPSoC EV



Figure 2. Xilinx ZYNQ Ultrascale+ Evaluation Kit components and tools [32]

The ZCU104 Evaluation Kit allows designers to quickly initiate projects for embedded vision applications such as surveillance, Advanced Driver Assisted Systems (ADAS), machine vision, Augmented Reality (AR), drones, and medical imaging. This kit incorporates a Zynq™ UltraScale+™ MPSoC EV device with a video codec and provides support for various common peripherals and interfaces tailored for embedded vision use cases. The ZU7EV device is provided with a quad-core ARM® Cortex™-

A53 applications processor, a dual-core Cortex-R5 real-time processor, a Mali™-400 MP2 graphics processing unit, a 4KP60 capable H.264/H.265 video codec, and 16nm FinFET+ programmable logic [33].

Xilinx produces a range of System-on-Chips (SoCs) that combine the software programmability of a processor with the hardware programmability of an FPGA. They offer a diverse selection of boards to cater to customers in need of SoC platforms for design, classified into three categories: cost-optimized, mid-range, and high-end. The cost-optimized category includes devices like the Zynq-7000 series and Artix, providing an economical solution for developers working on applications with less demand for extensive software processing. These boards can be acquired with either single-core or dual-core ARM Cortex-A9 processors. On the opposite end of the spectrum, the high-tier category encompasses various versions of the Zynq UltraScale+ RFSoc board, featuring options with Radio Frequency (RF) converters, SD-FEC cores, or a combination of both [34].

The Zynq UltraScale+ MPSoC family comprises three distinct models: CG, EV, and EG. In comparison to the CG variant, the EG variant enhances the dual application processor setup by introducing a quad application processor and GPU. On the contrary, the EV variant integrates the features of the EG variant while enhancing video codec capabilities by incorporating both H.264 and H.265 standards. These devices are well-suited for multimedia vision applications that require the processing of video streams or a substantial number of frames. For the purposes of this thesis, the EV model is selected as it excels in image and video processing. Figure 3 illustrates the physical layout of the FPGA.

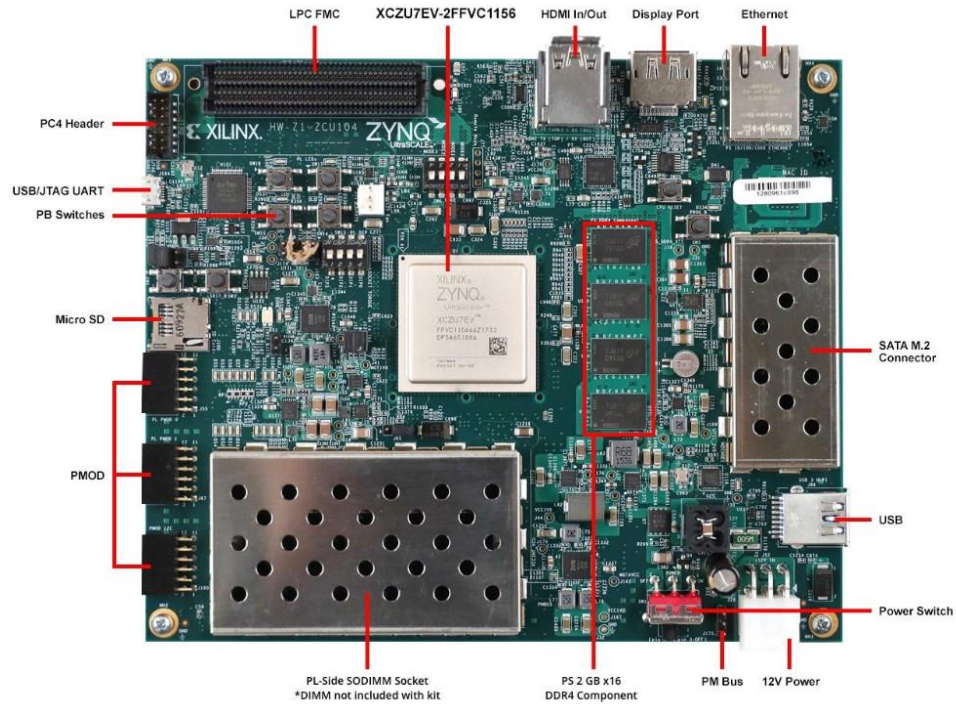


Figure 3. A detailed picture of Zynq UltraScale+ MPSoC ZCU104 and its interfaces

On the ZCU104's physical features, as shown in Figure 3, the FPGA has 464 General Purpose I/O (GPIO) pins for connecting other external devices, a Micro-USB/JTAG port for programming, a Micro SD port for expandable memory and boot options, dual HDMI 2.0 ports for input and output, a display port, a PHY tri-mode Ethernet port, a USB 3.0 port, and a display port. A quad-core ARM Cortex-A53 CPU with an Infineon Power Management Bus (PMBus), a floating-point unit, a Memory Management Unit (MMU), a 32 KB instruction cache, and a 32 KB data cache makes up the Application Processing Unit (APU) on the board [34].

Table 1. ZCU104 Resources [34]

ZCU104 Resources	
System Logic Cells(K)	504
Memory	38 MB
DSP Slices	1,728
Video Codec Unit	1
Maximum I/O Pins	464

“Each core of the dual-core ARM Cortex-A5 processor found in the Real-time Processing Unit (RPU) has a vector floating-point unit, a Memory Protection Unit (MPU), 128 KB of Tightly Coupled Memory (TCM), a 32 KB instruction cache, and a 32 KB data cache. Two-pixel processors, a geometry processor, an MMU, and a 64 KB L2 cache were all part of the GPU on the board. Figure 4 below shows the high-level device diagram for the ZCU104. The ZCU104 device, which is on the low end, has the programmable logic features listed in Table 1 above” [34].

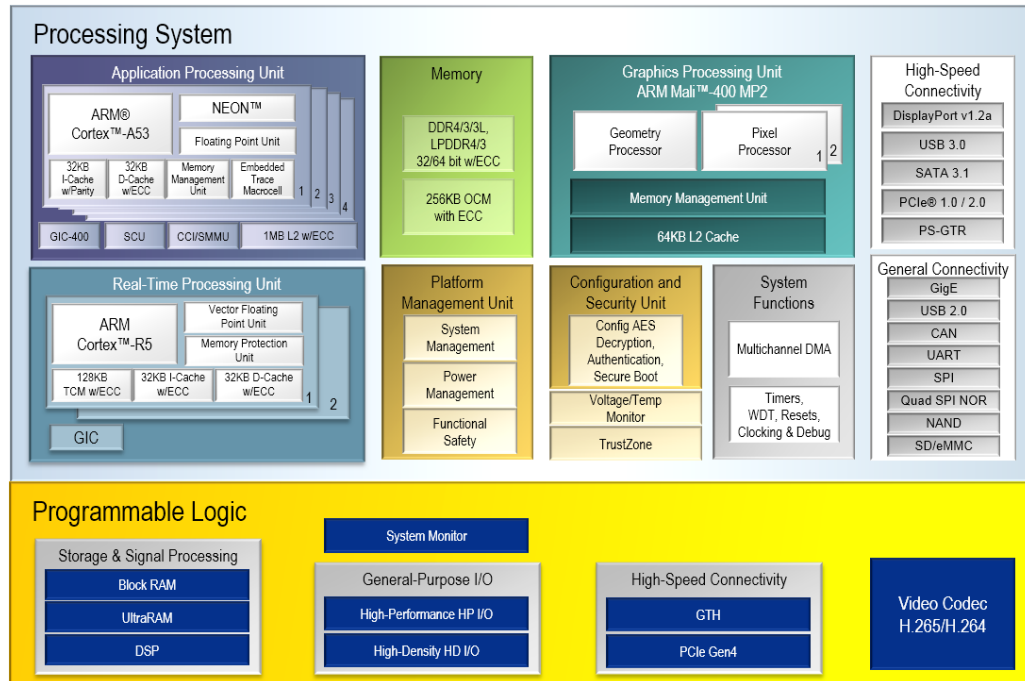


Figure 4. ZCU104 High Level device Diagram

Additional Components:

- Workstation
- Two HDMI Cables
- SD card at least 8GB

2.1.2 Software components

2.1.2.1 PYNQ 3.0 with VIVADO 2022.1

For the purpose of rebuilding the base overlay in VIVADO firstly the version of the PYNQ should be compatible with the version of VIVADO which will be used. [35] In this thesis PYNQ 3.0 is being used with overlays built in VIVADO 2022.1. On Table 2 there are all PYNQ versions with its corresponding VIVADO.

Table 2. PYNQ Versions compatible with VIVADO

VIVADO Version	PYNQ Version
VIVADO 2022.1	Version 3.0
VIVADO 2020.1	Version 2.6
VIVADO 2019.1	Version 2.5
VIVADO 2018.3	Version 2.4
VIVADO 2018.2	Version 2.3
VIVADO 2017.4	Version 2.2
VIVADO 2017.4	Version 2.1
VIVADO 2016.1	Version 2.0
VIVADO 2015.4	Version 1.4

2.1.2.2 PYNQ Overlays

The Xilinx Zynq All Programmable devices combine a dual-core Arm Cortex-A9 processor known as Processing System (PS) with FPGA fabric known as Programmable Logic (PL) in order to create a system-on-chip (SOC). Some dedicated peripherals such as USB, UART, SPI, memory controllers are included in PS, but this subsystem can also be enhanced by adding extra hardware intellectual property (IP) by using a PL Overlay.

PYNQ overlays play a very important role in the PYNQ framework especially for this case with ZCU104 board. An overlay in PYNQ refers to a hardware design that is implemented in the programmable logic (PL) section of the Zynq system-on-chip (SoC). These overlays give a massive level of abstraction and provide users the ability to accelerate their applications by leveraging the FPGA resources available on the ZCU104.

In order to create a PYNQ overlay the most common way is by using high-level synthesis (HLS). This way makes it easier for the developers to design it in languages like Python or C++ and to automatically generate the corresponding FPGA bitstream.

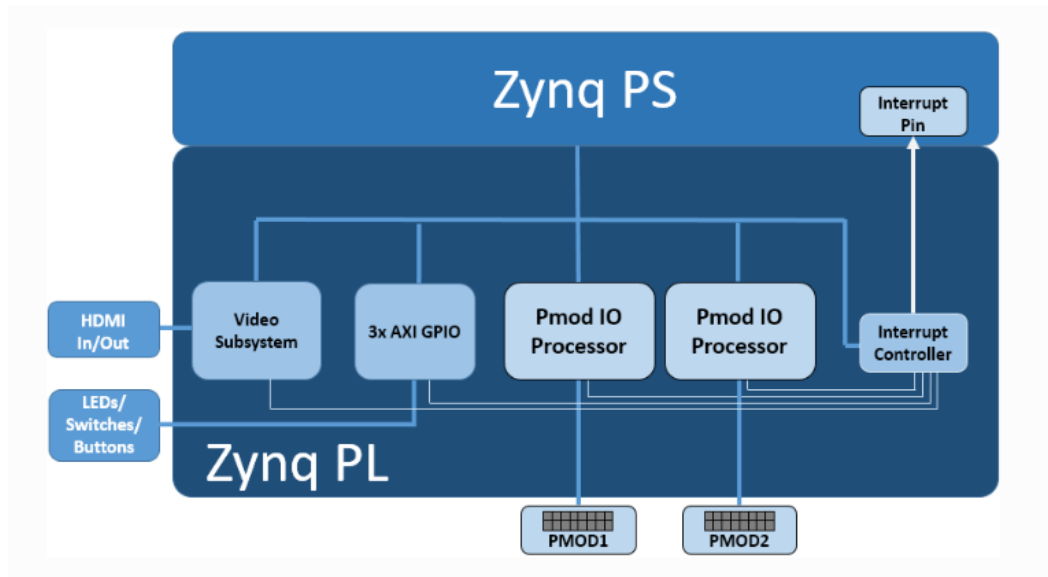


Figure 5. Connection of the peripherals in the Base Overlay of ZCU104 board

The overlays in PYNQ are highly customizable and can be tailored to specific application requirements. They include dedicated hardware accelerators, custom IP cores, or interfaces that enable efficient communication between the PL and the PS. By

offloading computationally intensive tasks to the overlay, users can achieve significant performance improvements compared to running their applications solely on the processing system.

PYNQ overlays also provide a software interface that allows developers to interact with the hardware accelerators and other custom IP cores from the Python environment. This interface, known as the PYNQ API, enables seamless integration of the overlay's functionality into the software application stack, providing a unified programming model.

Additionally, the PYNQ ecosystem offers a range of pre-built overlays that target specific domains or applications [36]. These overlays can be readily downloaded and used as a starting point for building custom overlays, saving development time and effort. Moreover, the PYNQ community actively contributes to the creation and sharing of overlays, fostering collaboration and knowledge exchange.

In summary, PYNQ overlays in the context of the PYNQ 3.0 framework and the ZCU104 platform provide a means to accelerate applications by leveraging FPGA resources. They offer a higher level of abstraction, customizable design options, software interfaces for seamless integration, and a thriving ecosystem of pre-built overlays. These features empower developers to harness the power of programmable logic and achieve enhanced performance for their applications.

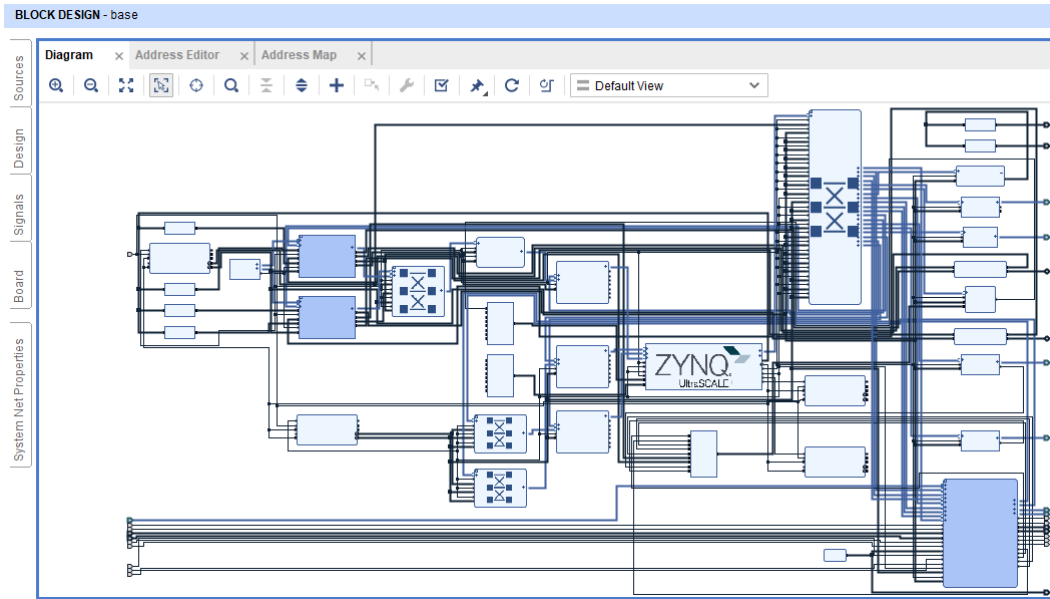


Figure 6. Base Overlay Block design in PYNQ 3.0

Video Peripheral structure:

- HDMI IN
- HDMI OUT
- PHY CONTROLLER
- AXI VDMA

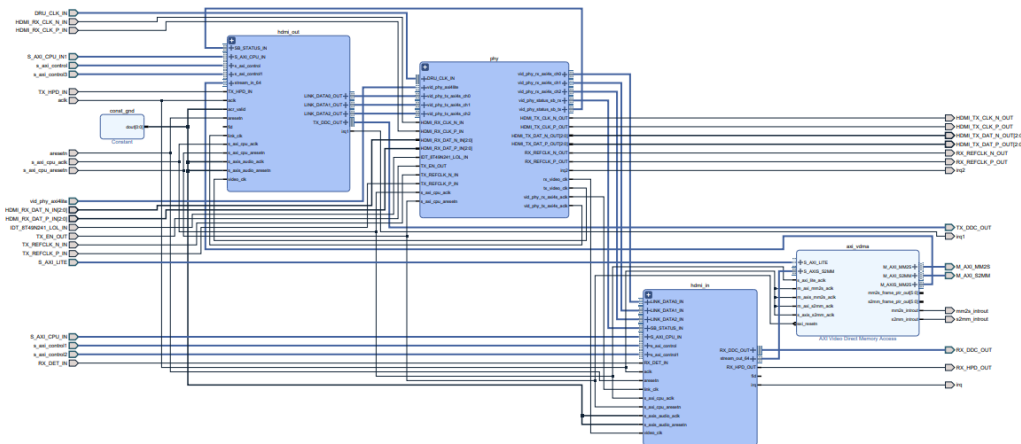


Figure 7. Block design of Video on Base Overlay

Both HDMI-s have a similar design as can be shown in Figure 5 and Figure 6. The components are listed below:

- Color convert (HLS IP)
- Pixel pack (HLS IP)
- AXI4-Stream Subset Converter
- AXI4-Stream Register Slice

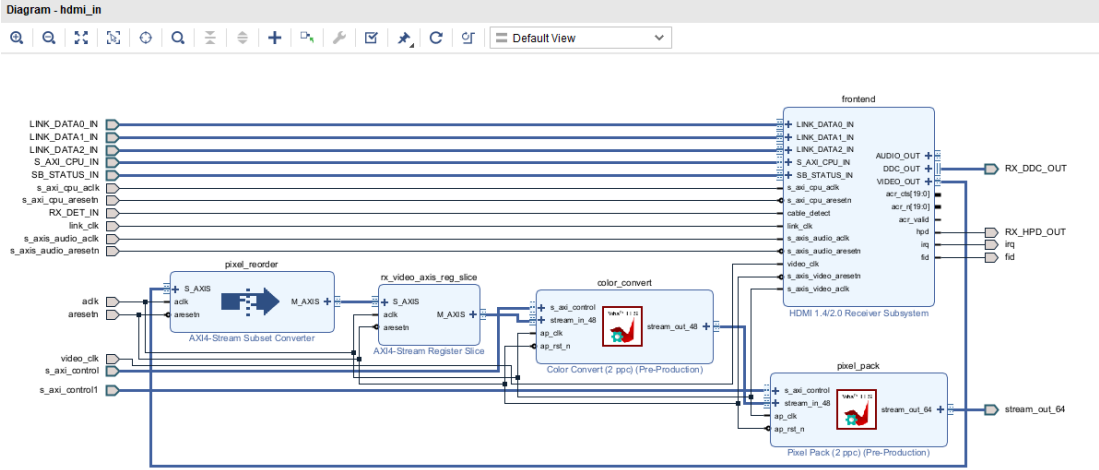


Figure 8. Design of HDMI INPUT

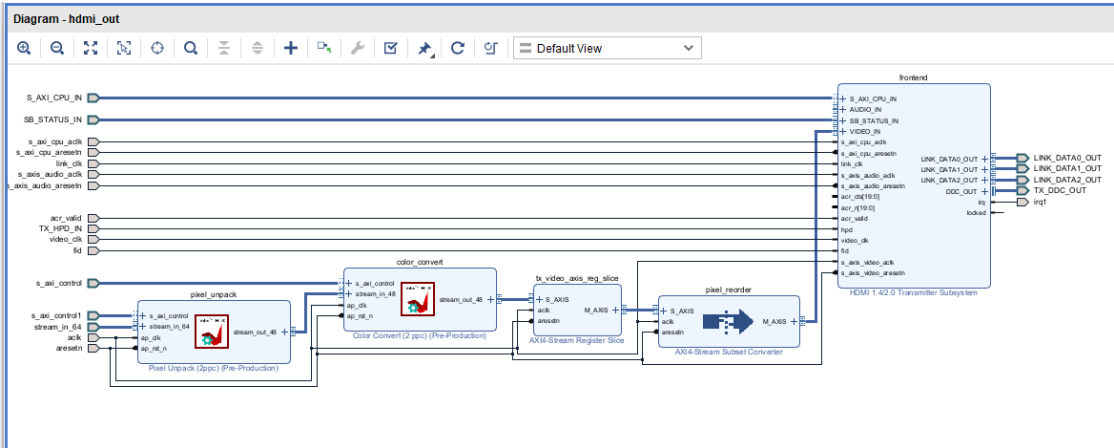


Figure 9. Design of HDMI OUTPUT

2.1.2.3 AXI DMA

Xilinx offers the AXI Direct Memory Access (DMA) IP Core to facilitate communication between hardware accelerators in the PL and the main system memory [37]. This AXI DMA enables high-bandwidth communication through the utilization of AXI Memory-Mapped and AXI stream interfaces. Figure 10 presents a diagram illustrating the primary input and output ports of the AXI DMA IP.

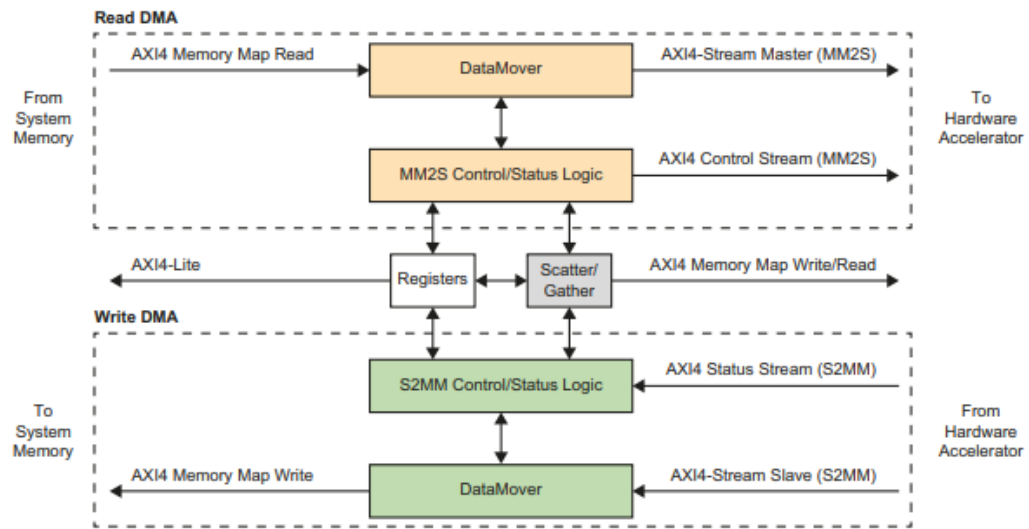


Figure 10. The AXI interfaces on the AXI DMA Controller

As depicted, the AXI DMA IP Core incorporates two data movers within its structure. One data mover facilitates reading from system memory, as indicated by the orange blocks, while the other handles writing data to system memory, represented by the green blocks. Each channel functions independently and can be activated or deactivated during the hardware system development phase. The process of reading from system memory utilizes the AXI4-Stream Master interface, identified as Memory-Mapped to Stream (MM2S). The AXI4 Control Stream (MM2S) interface provides the target IP Core with supplementary application and control data. Similarly, the write DMA employs the AXI4-Stream Slave interface for writing data to system memory, which may also be referred to as Stream to Memory-Mapped (S2MM). The write DMA includes an additional interface,

AXI4 Status Stream (S2MM), to receive status updates and application data from the target IP Core. The AXI4-Lite interface facilitates low-bandwidth communication with the PS. The optional scatter/gather interface enables the DMA to retrieve preloaded descriptors from system memory with minimal assistance from a processor core. Subsequently, the DMA can self-configure for its target address, transaction length, and other control parameters.

The Zynq MPSoC features an AXI DMA connection, illustrated in Figure 11 as an example of its integration in the PL. This representation is just one of the numerous system configurations possible with the Zynq MPSoC. The AXI DMA employs the AXI4 Memory-Mapped interface when interacting with the DDR Controller, utilizing the burst-transfer capability of the AXI4 Memory-Mapped protocol. This protocol supports efficient data transfers by incorporating an address and an access pattern specified by the master, determining subsequent addresses for the following data.

Burst transfers within a single transaction are facilitated by the access pattern, reducing the overhead and latency associated with data transfer. When transmitting data to the target IP Core, the AXI DMA utilizes the AXI4-Stream interface, enabling burst transfers of an unrestricted (infinite) size. In this protocol, no address channel is necessary, as it is designed for a seamless flow of data directly between the source and destination within the device.

In Figure 11, various connections are depicted, including the primary link between the DMA and S_AXI_HP1_FPD port through the AXI interconnect. This serves as the DMA's main pathway for reading and writing to the primary system memory. By linking it to the PL's high-performance ports, a high-throughput route to the DDR controller is established. Specifically, the first high-performance port (HP1_FPD) is utilized, having an exclusive link to port 4 of the DDR controller. This configuration is illustrated as an example of the AXI DMA connected in the Zynq MPSoC PL.

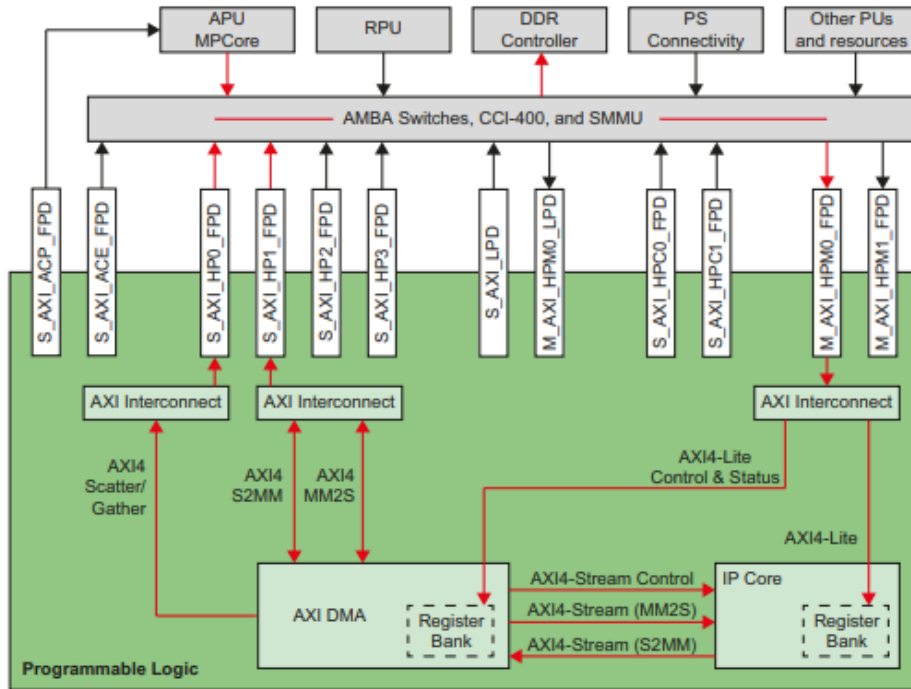


Figure 11. An example of the AXI DMA connected in the PL of the Zynq MOSoC [38]

The optional scatter/gather port is connected to S_AXI_HP0_FPD to fetch buffer descriptors from main memory. The AXI4-Lite control and status interface links to M_AXI_HPM0_FPD for communication with the Arm processors in the PS, enabling Arm processors to configure the AXI DMA and obtain status information. During reads from system memory, the DMA employs the AXI4 MM2S channel, also known as Memory-Mapped to Stream, transferring data to the AXI4-Stream (MM2S) channel for further transmission to the IP Core. Conversely, when writing to system memory, the DMA controller utilizes the AXI4 S2MM channel for data transfer.

2.1.2.4 AXI VIDEO DMA

The AXI Video DMA, represented by the VDMA IP Core [39], facilitates high-performance transfer of video frame data between DDR memory and the PL. Similar to the AXI DMA, the VDMA features control and status logic, a data mover block, and

AXI4-Lite registers, as depicted in Figure 12. Additionally, a new component, the Line Buffer, has been introduced. This asynchronous buffer serves as a temporary storage for pixel data before it is written to the AXI4 Memory-Mapped interface or AXI4-Stream interface.

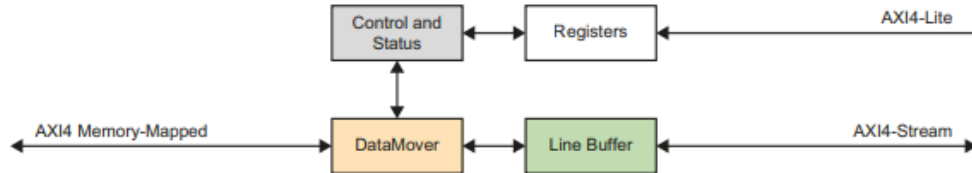


Figure 12. The AXI VDMA block diagram and AXI interfaces

Similar to the AXI DMA, the VDMA IP Core offered by Xilinx supports two data movers, designated for reading from and writing to system memory. Each data mover follows the structure illustrated in Figure 12, featuring its own AXI4 Memory-Mapped interface for communication with DDR memory and an AXI4-Stream interface for data transfer onto the PL. The choice of the VDMA IP over the AXI DMA is motivated by its optimization for efficient video data transfer between system memory and the PL [39].

The VDMA excels in performing DMA operations on video frame data, facilitating asynchronous transfer of video frames on both read and write channels. This capability proves beneficial in scenarios where the PL needs to buffer video data for different clock domains or wait for the completion of another task. The VDMA can handle up to 32 frame buffers across a 64-bit address space. An incorporated Data Realignment Engine (DRE) enables unaligned access to memory, allowing frame buffers to commence at any address in memory. Figure 12 displays the AXI VDMA block diagram and AXI interfaces [40], including Registers, Control and Status, Data Mover, Line Buffer, AXI4 Memory-Mapped, AXI4-Lite, and AXI4-Stream.

Illustrated in Figure 13 is an example of a frame buffer configuration in the Zynq MPSoC device. This example involves buffering video frames from an incoming High-

Definition Multimedia Interface (HDMI) signal using the AXI VDMA IP Core. Once the video frames are buffered, they are retrieved from system memory and written onto the AXI-Stream (MM2S) channel.

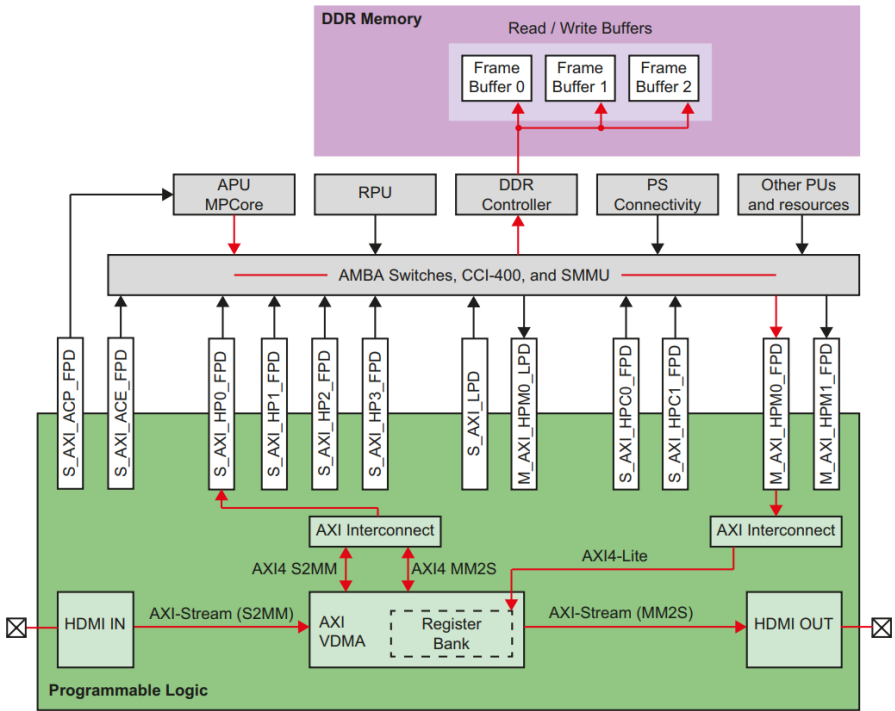


Figure 13. The AXI VDMA video frame buffer example.

2.1.2.5 Open Computer Vision (CV)

A well-known open-source package called OpenCV (Open-Source Computer Vision) offers a complete collection of tools and algorithms for computer vision and image processing jobs. It provides a variety of features and modules to help programmers manage several facets of computer vision applications, including image and video editing, feature identification, object recognition, and machine learning integration [41]. With its support for numerous programming languages, such as C++, Python, and Java, OpenCV is available to a wide range of developers.

For researchers, engineers, and enthusiasts working on computer vision and image analysis projects, OpenCV has emerged as the go-to option thanks to its broad collection

of features and interoperability with a wide range of platforms and operating systems. Its adaptability, effectiveness, and usability have made it a crucial tool in a variety of industries, including robots, augmented reality, surveillance, medical imaging, and more. Some of its features in image processing are:

- Color Space Conversion: Facilitates the conversion between different color spaces.
- Image Filtering: Provides various linear and non-linear filtering techniques, including blurring and sharpening.
- Geometric Image Transformations: Supports scaling, rotation, and affine and perspective transformations.
- Morphological Operations: Includes operations like erosion, dilation, opening, and closing that are particularly useful in image pre-processing.
- Histograms: Functions to compute and manipulate image histograms for tasks like contrast stretching or histogram equalization.
- Feature Detection and Description
- Edge Detection: Implements algorithms like Sobel and Canny for detecting edges in images.
- Contours: Provides functionality to detect and manipulate contours in binary images, which is useful in shape analysis and object detection.

2.2 Methodology

2.2.1 Steps for connecting PYNQ to FPGA

- PYNQ image should be downloaded in the website. In this case PYNQ 3.0 is used.
- The switch should be set to the position to boot from SD Card.
- PYNQ image is flashed to a 128 Gb SD card using balenaEtcher application.
- SD Card is inserted into ZCU104.

- Board is connected to PC using LAN.
- Board should also be connected to PC with a Hdmi cable for video input.
- Another Hdmi cable can be connected to a HD monitor but this is not mandatory since it can display the output on the Jupyter Notebook itself.
- The ZCU104 should be turned on.
- The static IP address of the FPGA is identified (192.168.2.99).
- Assign a static IP for the computer which should be in the same subnet as the FPGA.
- Go to the browser and type the IP address (192.168.2.99).
- Now the Jupyter Notebook is opened containing demos.

2.2.2 Image Processing Filtering

Digital image processing employs various techniques, one of which involves "sectioning" the image data. This segmentation technique, known as neighborhood processing, determines the output pixel value not only from its own data but also from its nearby neighbors. Illustration is shown in the Figure 14.

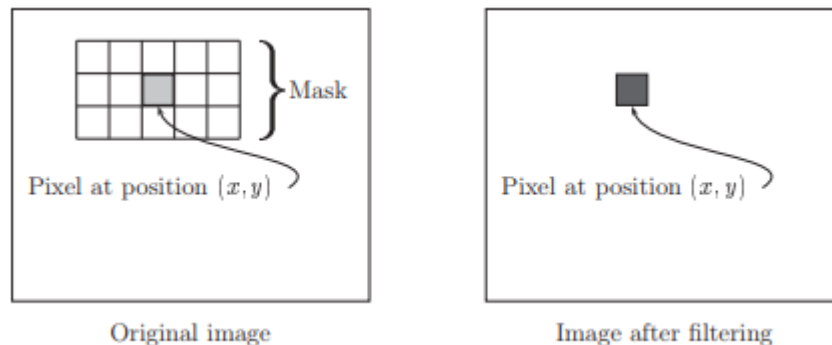


Figure 14. Illustration for processing of pixel through neighborhood operations [42]

2.2.3 2D Convolution

Convolution is a key process in signal and image processing, where it involves combining two functions to produce a third function that reflects how one function influences the other. Specifically, in image processing, spatial convolution is utilized, which involves multiplying each pixel of the image with a value from a flipped kernel mask. Subsequently, the sum of the pixel's immediate neighboring values is calculated to determine the new pixel value. This process is fundamental to 2D convolution and is well-explained by the following formula, which forms the basis of all 2D convolution operations.

$$X[m, n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i, j] * k[m - 1, n - j] \quad \text{Equation 1}$$

where:

- $X_{[m,n]}$ is the output image
- $x_{[i,j]}$ is the input image
- $k_{[m-1,n-1]}$ is the flipped kernel

If the flipped kernel was not used instead the normal one, then this whole operation would be referred to as spatial correlation and the output obtained by it would have been rotated by 180° . Hence, the kernel is flipped first in order to obtain an accurate result.

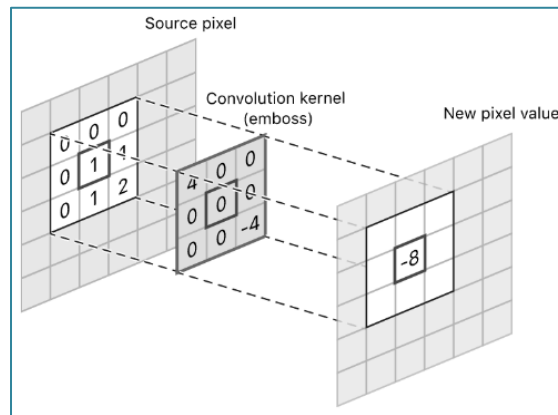


Figure 15. 2D Convolution of Image and Kernel Window [50]

As highlighted in the figure above, the kernel slides collecting data for each pixel and its immediate neighbors (if the kernel is 3x3 there are usually 8 neighbors to the input pixel) and outputs the new pixel in the output image.

2.2.3.1 Sobel Filter

Sobel filter is relatively computationally inexpensive algorithm which uses two window operates, one that detects the discontinuities in the horizontal direction and the other in the vertical direction as shown below. The Sobel filter is a popular edge detection algorithm used in image processing. [43] It is a type of discrete differentiation operator, computing an approximation of the gradient of the image intensity function. The Sobel filter emphasizes regions of high spatial frequency that correspond to edges. Typically, it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image. The Sobel filter uses two 3x3 kernels, one estimating the gradient in the x-direction (horizontal) and the other in the y-direction (vertical). These kernels are convolved with the original image to calculate the gradient approximations. The kernels are as follows:

➤ For the x-direction (Sobel_x): $G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$

➤ For the y-direction (Sobel_y): $G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$

$$G = \sqrt{G_x^2 + G_y^2} \quad \text{Equation 2}$$

In a closer look it can be noticed that if all the coefficients in a window are summed up the value 0 is obtained, therefore meaning that in the areas within the image

that hold a persistent brightness intensity the response would be 0, as it can reflect from the MatLab results below:



Figure 16. The illustration of both sobel operators Gy and Gx for the detction of edges in the horizontal and vertical direction respectively

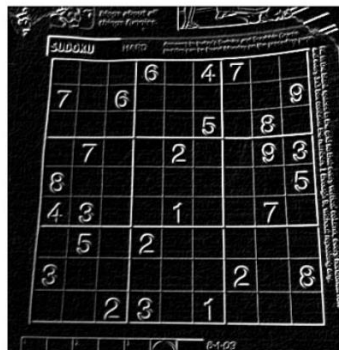


Figure 17. The resulting image from MatLab of the complete Sobel filter as the magnitude of the x and y direction gradients

2.2.3.2 Laplacian Filter

The Laplacian is the 2nd derivative of an image and is given by the equation below:

$$L(x, y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \quad \text{Equation 3}$$

where I represent the pixel *intensity* value on the x and y coordinates. This filter can be calculated using this commonly used kernel window below.

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

However just like in the Canny Filter we might need to apply a gaussian filter before, because the kernel window is much sensitive to noise.

2.2.3.3 Canny Filter

The Canny Filter is rather a composed filter from the Gaussian then the magnitude of two Sobel horizontal and vertical gradient opponents. The Gaussian filter smoothest the input image so when the operators are applied only the general outlines within the image are detected, making it easier for the machine to read information from the filtered image through the located discontinuities in the pixel brightness intensity [44].

2.2.3.4 Gaussian Filter

The Gaussian filter is used to reduce noise from the image thus, creating a blurring effect. The purpose of this operation on edge detection is to help the algorithm distinguish only the main outlines of the higher resolution images and not read every small line of discontinuity within the data as an edge. Especially needed in the developing of autonomous cars applications for example.

$$\frac{1}{159} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

For an illustration of how this filter works the following image¹ is processed in MatLab in a convolution with the above kernel. The input image is converted from RGB to grayscale beforehand in order to compute the 2D convolution.

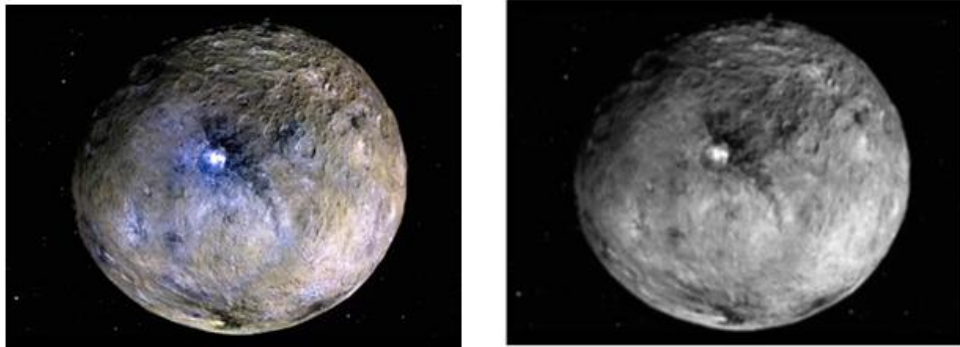


Figure 18. The result from MatLab of the convolution of the image with the Gaussian filter. [45]

To the result of the Gaussian filter convolution the horizontal and vertical operators are separately applied thus obtaining two additional images that are added together in accordance to the equation (2). Finally, the following Canny filtered image is as follows:

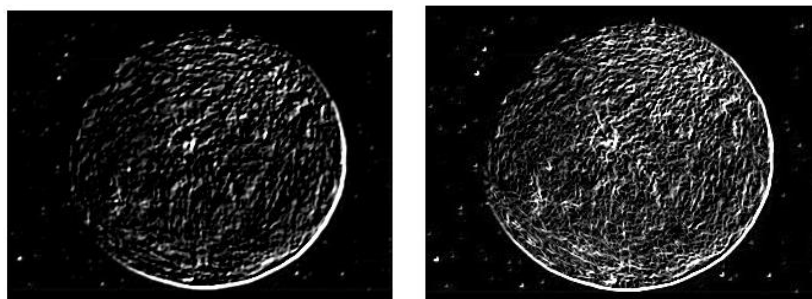


Figure 19. The result from the Canny Filter(left) and Sobel Filter(right) Compared side-by-side

¹The image shown below is that of the dwarf planet Ceres, the “unpredictable” orbit of which became the cause of Gauss approximation, and after that Gauss kept on developing his methods that are now widely used in various mathematics computations, especially engineering. [45]

2.3.1 Steps for implementation of the custom overlay

A custom overlay to be controlled by PYNQ can be developed into two ways, either by editing the existing Base Overlay provided by PYNQ for a specific type of supported board or building one from scratch. In this project the second option is chosen. The same logic as the above examples is followed throughout the designing process of the IP [46]. First, just as it was done before with the grayscale and result images, the data size for each image is allocated using the values of maximum width and height that is expected from the input. Afterwards, each image declared is used to save every transformation the input image goes through. In this case the input image edges need to be located through the Canny filter, so we go through the aforementioned steps of the Canny Filter. Steps for building a custom overlay are mentioned below [47] [48].

Set Up HLS Project

- Install and open Xilinx Vivado HLS.
- Create a new HLS project and target it for the ZCU104 platform.

1. Write HLS Code for each filter

Two quite important steps in designing the IP are the interfaces and the declaration of the TLAST port, the latter serves as signal from the IP of the last bit received. Without them the IP cannot take or receive data from the DMA or PS. Therefore, it is fundamental to declare them correctly.

```
#pragma HLS INTERFACE axis port=in  
#pragma HLS INTERFACE axis port=out\  
#pragma HLS INTERFACE s_axilite port=return bundle=CRTL_BUS  
typedef hls::stream<ap_axiu<24,1,1,1>>IN;  
typedef hls::stream<ap_axiu<24,1,1,1>>OUT;
```

Canny Algorithm implementation: Implement the steps of the Canny algorithm (Gaussian blur, gradient calculation, non-maximum suppression, hysteresis thresholding). They might need to be written in separate functions for each step and call them from your main function.

2. Apply HLS Dataflow Optimization:

In HLS, the `#pragma HLS DATAFLOW` directive can be utilized to enable concurrent execution of functions. This allows pipelining of operations, reducing latency. Algorithm is decomposed into multiple functions or blocks that can run in parallel. The frame is captured from the video stream (in the simulation only one image is used to test the IP). [49] That frame is saved in image 0 and its color space is RGB. Additionally, the data type is 24-bit (3 channels) as the `cvt.h` file suggests the type of the IN data is. The image data of image 0 is converted to grayscale and saved to the destination image 1. The same procedure is applied to the Gaussian filter too, where the kernel dimensions are taken 3x3

```
#pragma HLS dataflow  
hls::AXIvideo2Mat(INPUT_STREAM, img_0);  
hls::CvtColor<HLS_BGR3GRAY>(img_0, img_1);  
hls::GaussianBlur<3,3>(img_1, img_2);  
hls::Duplicate(img_2, img_2a, img_2b);  
hls::Sobel<1,0,3>(img_2a, img_3);  
hls::Sobel<0,1,3>(img_2b, img_4);  
hls::Addweighted(img_4, 0.5, img_3, 0.5, 0.0, img_5);  
hls::CvtColor<HLS_GRAY2RGB>(img_5, img_6)
```

The image data then is duplicated so Sobel gradients in the x direction and y direction respectively may be estimated. The gradient images are added together with the `AddWeighted` function same as before. Only this time the functions are referenced from the “`hls_video.h/ hls/hls_video_imgproc.h`” library. The image data is converted back into the RGB colorspace and sent to stream.

3. Test and Simulate: HLS is used for design simulation. There must be a correct behaviour with the test inputs. After HLS synthesis, RTL code is generated and can analyze performance metrics.
4. Export HLS Design: RTL is generated by exporting the synthesized design (VHDL/Verilog) along with the test bench, and an IP is created by packaging the High-Level Synthesis (HLS) design for seamless integration into the Vivado IP integrator.

- Integrate into Vivado and PYNQ: The HLS-generated IP is imported into a Vivado block design targeting the ZCU104 for Vivado Integration. The bitstream is subsequently generated, and the hardware design, comprising HWH and BIT files, is exported. A PYNQ overlay is then created using the exported files, and Python code is authored to interface with the hardware in the context of PYNQ Overlay creation. The implementation of the HLS IP is depicted within the block design.

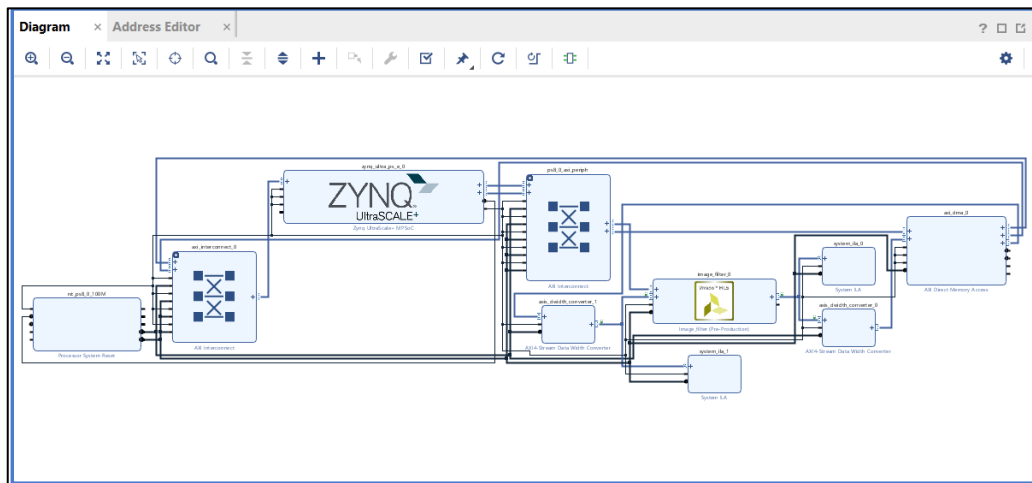


Figure 20. Implementation of the IP in the block design for ZCU104

In the block design it should be made sure that there is no mismatch in the data width or IP I/O interfaces and DMA slave and master, that’s why a Axi Width Converter is used in order to convert the 24-bit wide data stream of the IP to a 32-bit wide data stream at the DMA. Also, the S_AXI_HP0_FPD port is enabled so that the DMA catch is flushed.

The next final steps are the design validation that should be done with no errors and creating an HDL wrapper of the block design. Finally, choosing the option “generate bitstream” a new custom overlay for the board is created.

CHAPTER 3

RESULTS AND DISCUSSION

3.1 Implementation of Real-Time Video through HDMI interface (BASE OVERLAY)

This notebook is run on the “*base overlay.bit*”, and the cells are explained below. First, aliases are created for both HDMI input and output, configuring them based on the format required, which is RGB in this case. Additionally, both HDMI-s are initialized.

```
In [21]: from pynq.lib.video import *  
  
hdmi_in = base.video.hdmi_in  
hdmi_out = base.video.hdmi_out  
hdmi_in.configure(PIXEL_RGB)  
hdmi_out.configure(hdmi_in.mode, PIXEL_RGB)  
hdmi_in.start()  
hdmi_out.start()  
  
Frequency: 148500000
```

Figure 21. Initialization of the HDMI input and HDMI output

The next step would be to import the libraries of PIL Image in order to display the image and time library from python to help measure the rate that the frame is passed through HDMI input to output. This cell measures only the frame rate without applying any filters, just to test the maximum real-time frame rate.

```
In [22]: import PIL.Image
import time
numframes = 200
start = time.time()
for _ in range(numframes):
    frame = hdmi_in.readframe()
    hdmi_out.writeframe(frame)
img = PIL.Image.fromarray(frame)
img.save("/home/xilinx/jupyter_notebooks/base/video/data/opencv_filters_live.jpg")
end = time.time()
print("Frames per second: " + str(numframes / (end - start)))
img
```

Frames per second: 59.137027155731374

Figure 22. Code cell for the real-time video displayed on the HD monitor

Step 3: Show HDMI input frame within notebook using IPython Image

```
In [24]: import PIL.Image
frame = hdmi_in.readframe()
img = PIL.Image.fromarray(frame)
img.save("/home/xilinx/jupyter_notebooks/base/video/data/opencv_filters.jpg")
img
```

Out[24]:

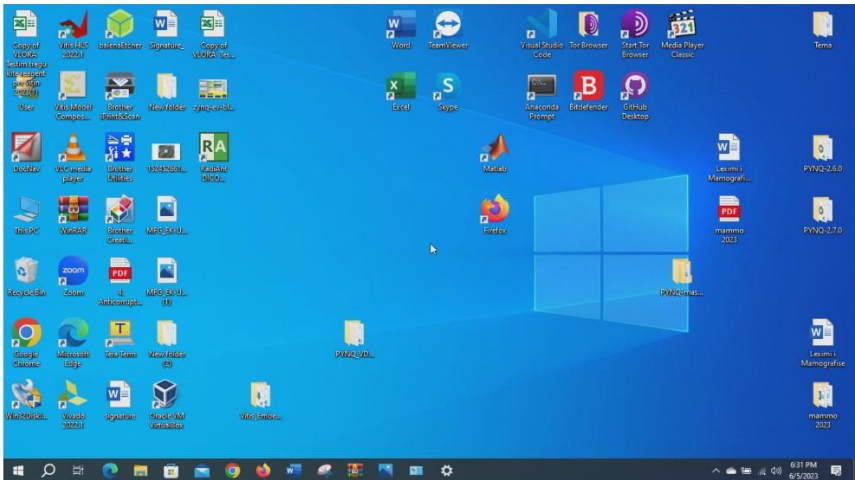


Figure 23. Real time display of Laptop screen on PYNQ


```
In [26]: img = PIL.Image.fromarray(outframe)
img.save("/home/x11inx/jupyter_notebooks/base/video/data/opencv_filters.jpg")

img
```



Figure 24. Real Time Edge Detection Using Laptop Screen as an input

In the following cells implementation of the filter algorithms is made possible through the OpenCV libraries. The code steps are almost similar cell-to-cell. Allocating the space for the grayscale and the result image using the configurations of HDMI_IN in order to ensure there is no mismatch of types. In this case the height and width are that of the Laptop Monitor (1920x1080) and the type is `uint8`, which stands for unsigned 8-bit integer. Inside the loop the video live stream is treated as a stream of images, so each frame is processed one by one by the code. As mentioned above, in order to compute the 2D convolution the input should be only one channel (grayscale) in order to use the `cvtColor()` function to convert each frame to grayscale before starting the convolution. The `cvtColor()` function has an argument format of:

$$cv2.cvtColor(source, code [, destination [, destinationCn]])$$

Where `source` represents the input image, `code` is the code of the color space we wish to convert the image to and the destination image (allocated above) in brackets is the number of channels, in this case being 1. The frame is then filtered with the filter function accordingly:

- The Laplacian filter is similar to the color conversion function.

- The Canny filter function is also similar except the second and fourth arguments are the value of the upper and lower threshold set for the filter.
- The Sobel filter is rather complex. First, the gradient is calculated along the x-axis and afterwards the gradient along the y-axis. The way these two are differentiated from each other are the values of the second than third arguments, respectively the value of dx and dy.

After the input frame is filtered the newframe function is called from the hdmi_out. The filtered frame is converted back to RGB color space in order to be compatible when written in the HDMI output channel, through the calling of the “writeframe()”function

Step 4: Edge detection
 Detecting edges on webcam input and display on HDMI out.

```
In [6]: import time
num_frames = 20
readError = 0

start = time.time()
for i in range (num_frames):
    # read next image
    ret, frame_vga = videoIn.read()
    if (ret):
        outframe = hdmi_out.newframe()
        laplacian_frame = cv2.Laplacian(frame_vga, cv2.CV_8U, dst=outframe)
        hdmi_out.writeframe(outframe)
    else:
        readError += 1
end = time.time()

print("Frames per second: " + str((num_frames-readError) / (end - start)))
print("Number of read errors: " + str(readError))
```

Frames per second: 27.746978335573196
 Number of read errors: 0

Figure 25. Cell code for the real-time video with Laplacian Edge Detection

```

In [7]: num_frames = 20

Mode = VideoMode(640,480,8)
hdmi_out = base.video.hdmi_out
hdmi_out.configure(Mode,PIXEL_GRAY)
hdmi_out.start()

start = time.time()
for i in range (num_frames):
    # read next image
    ret, frame_webcam = videoIn.read()
    if (ret):
        outframe = hdmi_out.newframe()
        cv2.Canny(frame_webcam, 100, 110, edges=outframe)
        hdmi_out.writeframe(outframe)
    else:
        readError += 1
end = time.time()

print("Frames per second: " + str((num_frames-readError) / (end - start)))
print("Number of read errors: " + str(readError))

Frequency: 75600000
Frames per second: 35.42995193566644
Number of read errors: 0

```

Figure 26. Cell code for the real-time video with Canny Edge Detection

Step 7: Sobel Edge detection

```

In [35]: import cv2
import numpy as np
import time
numframes = 200
grayscale = np.ndarray(shape=(hdmi_in.mode.height,
                             hdmi_in.mode.width), dtype=np.uint8)
result = np.ndarray(shape=(hdmi_in.mode.height,
                           hdmi_in.mode.width), dtype=np.uint8)

start = time.time()

for _ in range(numframes):
    inframe = hdmi_in.readframe()
    cv2.cvtColor(inframe,cv2.COLOR_RGB2GRAY,dst=grayscale)
    inframe.freebuffer()
    grad_x = cv2.Sobel(grayscale, cv2.CV_64F, 1, 0, ksize=3, scale=1, delta=0, borderType=cv2.BORDER_DEFAULT)
    grad_y = cv2.Sobel(grayscale, cv2.CV_64F, 0, 1, ksize=3, scale=1, delta=0, borderType=cv2.BORDER_DEFAULT)
    abs_grad_x = cv2.convertScaleAbs(grad_x)
    abs_grad_y = cv2.convertScaleAbs(grad_y)
    result= np.uint8(cv2.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0))

    outframe = hdmi_out.newframe()
    cv2.cvtColor(result, cv2.COLOR_GRAY2RGB,dst=outframe)
    hdmi_out.writeframe(outframe)
end = time.time()
print("Frames per second: " + str(numframes / (end - start)))

Frames per second: 4.963239280237863

```

Figure 27. Cell code for the real-time video with Sobel Edge Detection

Step 6: Show results

Now use matplotlib to show filtered webcam input inside notebook.

```
In [8]: %matplotlib inline
from matplotlib import pyplot as plt
import numpy as np

frame_canny = cv2.Canny(frame_webcam, 100, 110)
plt.figure(1, figsize=(10, 10))
frame_vga = np.zeros((480,640,3)).astype(np.uint8)
frame_vga[:, :, 0] = frame_canny
frame_vga[:, :, 1] = frame_canny
frame_vga[:, :, 2] = frame_canny
plt.imshow(frame_vga)
plt.show()
```

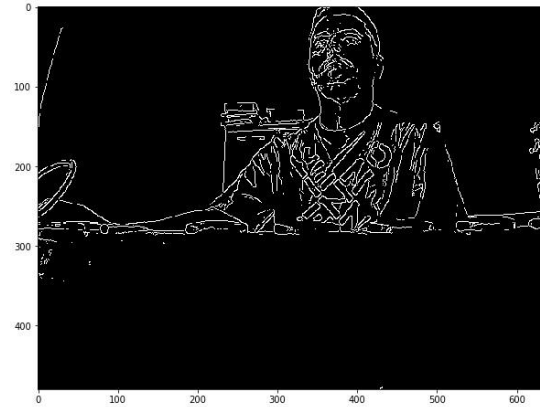
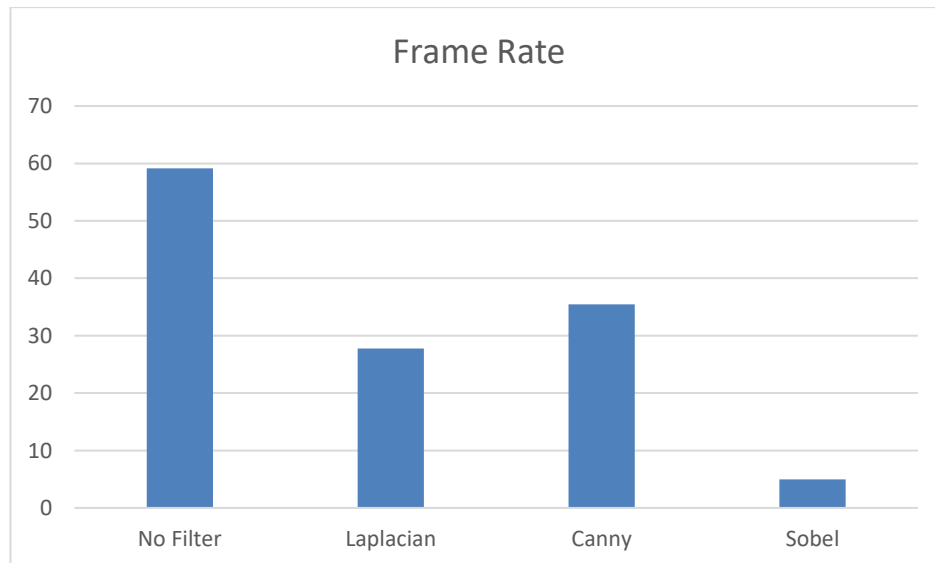


Figure 28. Implementation of real time canny filter on USB CAMERA

3.2 Results of the Real time video in the Jupyter Notebook

3.2.1 The Frame Rate Comparison:

Table 3. Side to Side Comparison of Filters Implementation on Software



From the chart above it can clearly be seen that the highest frame per second achieved is 60fps. And that is only achieved when no filter is applied. This frame rate is decreased when filters like Canny, Sobel, and Laplacian are applied. This happens because this is the maximum capacity of the software domain.

3.2 Results from of Real-Time Video from USB camera and pre-saved Video)

3.2.1 USB Camera Input (1280x720)

Table 4. Comparison of Hardware and Software with the input coming from the USB Camera Input

	SOFTWARE	HARDWARE
DILATE	16.96 fps	79.04 fps
LIVE STREAM	7.905 fps	9.03 fps

3.2.2 Pre-saved video Input (768x576)

Table 5. Comparison of Hardware and Software with the input coming from the pre=saved video "vtest.mp4"

	SOFTWARE	HARDWARE
DILATE	34.8 fps	165.42 fps
LIVE STREAM	16.31 fps	18.6 fps

The empirical findings presented herein substantiate the assertion that computational efficiency is notably enhanced when convolution occurs through the Programmable Logic (PL) in comparison to exclusive programming utilization of the processing system (PS) within the board. Furthermore, it is observed that at reduced resolution rates, exemplified by the video saved, a heightened frame rate is achieved due to the diminished size of the image data.

3.2.3 Power Consumption Comparison:

Table 6. Power Comparison 1240x720(Watts)

	SOFTWARE	HARDWARE
DILATE	10.325	10.275
LIVE STREAM	10.325	10.290

These tables present a comparison of power consumption between the two methods based on the average of measurements taken from three instances. The results clearly indicate that hardware implementation is more energy-efficient than software implementation, making it the preferable choice for image processing tasks.

3.2.4 Data read from disk:

This is a very important parameter because it affects how quickly video frames can be processed, the strain on system's memory and overall performance. The data read from the disk for the for the two resolutions throughout the hardware implementation is

shown in the table below. As it can be seen the data read from disk grows with the resolution of the video.

Table 7. Data read from disk during hardware implementation

	Data read from disc(bytes/s)
1280x720	2764800
768x576	1658880

3.3 Results from the HLS IP C simulation for Gaussian Filter 5x5:

	VERILOG
Slice	0
LUT	753
FF	221
DSP	12
URAM	0

These results suggest that the design does not utilize any Block RAM (BRAM_18K) or UltraRAM (URAM) resources. It employs 12 DSP slices, which are specialized hardware units for complex arithmetic operations. The design uses 221 flip-flops (FF) for storing

state information and control logic. The majority of resources are LUTs (Lookup Tables), with 753 LUTs utilized. LUTs are essential for implementing combinational and sequential logic in FPGA designs. These resource utilization numbers provide insights into the design's logic complexity, resource requirements, and potential areas for optimization.

3.4 Results from the HLS IP C simulation for Sobel Filter:

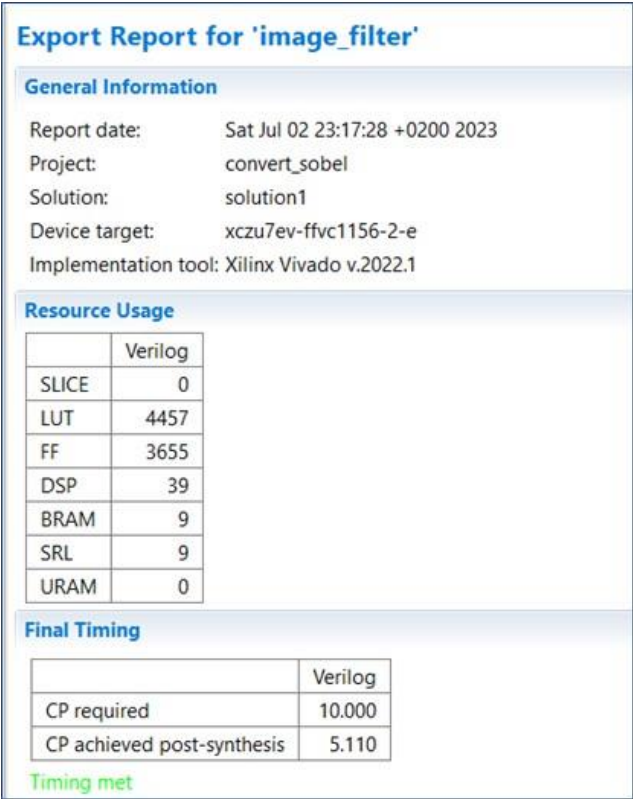


Figure 29. The simulation results of the HLS IP (1280x720) image

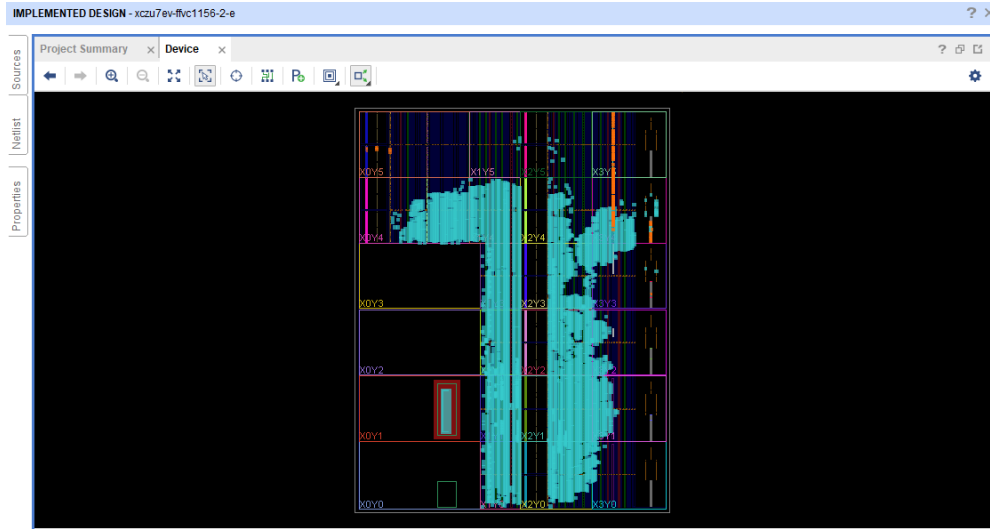


Figure 30. Graphical representation of the device and placed logic resources

A graphical representation of the device and placed logic resources in FPGA design provides a visual depiction of how various logic elements and resources are physically arranged and utilized within the FPGA chip. This representation includes the device structure, showing logic blocks, routing resources, I/O pins, clocking elements, memory blocks like BRAM and URAM, DSP slices, and other specialized components specific to the FPGA architecture. Within this representation, placed logic resources are visualized based on their physical placement and utilization, such as LUTs, flip-flops, DSP slices, BRAM, and URAM. Graphical elements like squares, rectangles, icons, or symbols are used to represent these resources, often with color coding to indicate utilization levels. This graphical view aids designers in understanding how their design is mapped onto the FPGA chip, identifying resource allocation, optimizing utilization, and ensuring efficient hardware implementation.

CHAPTER 4

CONCLUSION

In conclusion, this thesis has explored the ongoing evolution of technology, which continually raises the bar for digital media processing in terms of quality and processing speed. It has become evident that keeping up with this rapid evolution demands more than just software enhancements; the real power lies in programmable hardware. Through our utilization of the ZCU104 board equipped with the ZYNQ Ultrascale+ MPSoC EV, we have unequivocally demonstrated that direct programming on the Programmable Logic (PL) portion of the board, combined with executing computations through peripherals and custom Integrated Circuits (ICs), surpasses the efficiency of implementing algorithms solely through the processor. A substantial increase in performance, yielding a frame rate up to 4.7 times higher, has been observed in hardware-based video processing compared to software implementations despite having similar power consumption.

These findings not only present a compelling demonstration of the project but also offer an intriguing glimpse into future possibilities. They suggest that with the appropriate investment of effort, hardware will transcend its traditional role as a bottleneck in data processing. Instead, it will emerge as a pivotal element in driving forward a more advanced and efficient future in data processing.

REFERENCES

- [1] <https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html>. [Online].
- [2] [Online]. Available: <https://www.xilinx.com/products/silicon-devices/resources/programming-an-fpga-an-introduction-to-how-it-works.html>.
- [3] R. K. Prasad, P. V. Subbaiah and S. S. Rao, "FPGA-based high-speed data transmission system for space applications," *International Journal of Satellite Communications and Networking*, vol. 37, pp. 189-198, 2019.
- [4] H. Kim, J. Kim and a. J. Lee, "Design of a high-speed data transmission system based on FPGA and Ethernet," *IEEE Transactions on Industrial Informatics*, vol. 11, pp. 416-423, 2015.
- [5] L. W. a. Y. Z. C. Wang, "Design and implementation of a high-speed data transmission system based on FPGA," *Journal of Networks*, vol. 9, pp. 873-879, 2014.
- [6] X. W. a. J. Zheng, "Design and implementation of a high-speed data transmission system based on FPGA," *Journal of Computer and Communications*, vol. 11, pp. 101-109, 2015.
- [7] K. Murai and a. K. N. T. Ohnishi, "High-speed data transmission using FPGA and optical fiber," in *Proceedings of the 9th International Conference on Advanced Communication Technology (ICACT)*, Pyeongchang, Korea, 2017.
- [8] H. C. a. H. Chen, "Design and implementation of a high-speed data transmission system using FPGA," in *Proceedings of the 6th International Conference on Wireless Communications and Signal Processing (WCSP)*, Hangzhou, China, 2014.
- [9] M. A. Hossain, S. A. Khan and M. A. Rahman, "FPGA-based high-speed data transmission system for digital communication," in *Proceedings of the 5th International Conference on Electrical and Computer Engineering (ICECE)*, Dhaka, Bangladesh, 2018.
- [10] A. J. R. a. J. J. Sanchez, "Design of a high-speed data transmission system based on FPGA for IoT applications," in *Proceedings of the 13th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, Abu Dhabi, UAE, 2017.
- [11] Z. Yu, W. Wang Guang Yi and a. Y. S. Ying, "The image pipeline for color video surveillance system," in *2010 5th IEEE Conference on Industrial Electronics and Applications*, 2010.

- [12] N.Sudha, "Enabling seamless video processing in smart surveillance cameras with multi-core," in *International Conference on Advanced Computing and Communications*, 2015.
- [13] A. E. Wilson and M. Wirthlin, "Reconfigurable Real-Time Video Pipelines on SRAM-based FPGAs,," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, 2019.
- [14] P. Greisen, S. Heinzle, M. Gross and A. Burg, "An FPGA-based processing pipeline for high-definition stereo video," *EURASIP Journal on Image and Video Processing*, 2011/12/01.
- [15] E. Onat, "FPGA implementation of real time video signal processing using Sobel, Robert, Prewitt and Laplacian filters," in *2017 25th Signal Processing and Communications Applications Conference (SIU)*, 2017.
- [16] D. R. Menaka, D. R. Janarthanan and Dr. K. Deeba, " FPGA implementation of low power and high speed image edge detection algorithm," *Microprocessors and Microsystems*, vol. Volume 75, no. 0141-9331, 2020.
- [17] X. Guo, X. Wei and Y. Liu, "An FPGA implementation of multi-channel video processing and 4K real-time display system," in *2017 10th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*, Oct 2017.
- [18] Y. Gong and F. Yu, "Design of high-speed real-time sensor image processing based on FPGA and DDR3," in *2017 3rd IEEE International Conference on Computer and Communications*, 2017.
- [19] G. a. C. Xijun, W. a. Qiang and W. Fu and Wei, "Design of high-speed real-time processing platform faced on video tracking," in *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*, 2011.
- [20] Xilinx, "High Level Synthesis," February 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf. [Accessed April 2023].
- [21] A. E. Guzel, V. E. Levent, M. Tosun and M. A. Özkan, "Using high-level synthesis for rapid design of video processing pipes," in *2016 IEEE East-West Design Test Symposium (EWDTS)*, 2016.
- [22] E. K. a. I. Hamzaoglu, "FPGA implementations of HEVC inverse DCT using high-level synthesis," in *2015 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2015.

- [23] W. Ahmad, J. Iqbal, M. Martina and G. Masera, "High level synthesis based FPGA implementation of H.264/AVC sub-pixel luma interpolation filters," in *2016 European Modelling Symposium (EMS)*, 2016.
- [24] E. F. E. R. Roberto Millon, "A Comparative Study between HLS and HDL on SoC for Image Processing Applications," *Elektron*, vol. 4, 2020.
- [25] A. B. Amara, E. Pissaloux and M. Atri, "Sobel edge detection system design and integration on an FPGA based HD video streaming architecture," in *2016 11th International Design Test Symposium (IDT)*, Dec 2016.
- [26] M. Kowalczyk, D. Przewlocka and T. Krvjak, "Real-time implementation of contextual image processing operations for 4k video stream in Zynq UltraScale+ MPSoC," in *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2018.
- [27] PYNQ, "What is PYNQ?," 2022. [Online]. Available: <http://www.pynq.io/>. [Accessed 20 May 2022].
- [28] C. Tănase, "Dynamic scheduler implementation used for load distribution between hardware accelerators (RTL) and software tasks (CPU) in heterogeneous systems," *Journal of Supercomputing*, p. 10122–10139, 13 March 2020.
- [29] "Jupyter Webpages," [Online]. Available: <https://jupyter.org/>. [Accessed 27 May 2023].
- [30] L. Crockett, D. Northcote, C. Ramsay, F. Robinson and B. Stewart, *Exploring Zynq MPSoC With Pynq and Machine Learning Applications*, Xilinx, University of Strathclyde, 2019.
- [31] H. Shen, *Interactive Notebooks: Sharing the Code*, vol. Vol. 515, Nature, 6th November 2014.
- [32] Xilinx, "Xilinx Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit," Xilinx, Inc, [Online]. Available: <https://www.xilinx.com/products/boards-and-kits/zcu104.html>.
- [33] I. Xilinx, "Zynq UltraScale+ MPSoC Overview: Advance Product Specification," november 2018 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.
- [34] K. Stokke, "An FPGA-Based Hardware Accelerator For The Digital Image Correlation Engine," 2018.
- [35] Xilinx Revision, "PYNQ SD Card," 29 January 2021. [Online]. Available: <https://pynq.readthedocs.io/en/v2.6.1/index.html>. [Accessed 27 May 2022].

- [36] Xilinx Revision, "Base Overlay," January 2018. [Online]. Available: https://pynq.readthedocs.io/en/v2.4/pynq_overlays/pynqz2/pynqz2_base_overlay.html. [Accessed 29 May 2022].
- [37] PYNQ, "The Jupyter Notebook" documentation page ("Browser Compatibility" section), [Online]. Available: <https://jupyter-notebook.readthedocs.io/en/latest/notebook.html#browser-compatibility>.
- [38] PYNQ, "PYNQ Libraries" documentation page, [Online]. Available: https://pynq.readthedocs.io/en/latest/pynq_libraries.html.
- [39] "Python Package Library," [Online]. Available: <https://pypi.org/>.
- [40] P. P. R. webpages, "How to Package Your Python Code," [Online]. Available: <https://python-packaging.readthedocs.io/en/latest/index.html>.
- [41] OpenCV, "Image Processing in OpenCV," [Online]. Available: https://docs.opencv.org/4.x/d2/d96/tutorial_py_table_of_contents_imgproc.html.
- [42] R. E. Rafael C. Gonzales, Digital Image, Upper Saddle River, New Jersey: Pearson Education, Inc., 2008.
- [43] H.-A. T. a. M. R. Z. a. Z. S. M. A. a. M. R. A. a. A. M. R. a. T. A. W. Abdullah, "FPGA-Based Three Edge Detection Algorithms (Sobel, Prewitt and Roberts) Implementation for Image Processing.," *Przeglad Elektrotechniczny*, vol. 2024, 2024.
- [44] Z. Tan and J. S. Smith., "Real-time Canny Edge Detection on FPGAs using High-level Synthesis," in *7th International Conference on Information Science and Control Engineering (ICISCE)*, Changsha, China, 2020.
- [45] NASA, "Ceres," 5 April 2018. [Online]. Available: https://solarsystem.nasa.gov/resources/846/ceres-rotation-and-occator-crater/?category=planets/dwarf-planets_ceres.
- [46] J. Kalomiros, J. Vourvoulakis and S. Vologiannidis, "A Workflow for Designing Video Processing Pipelines with PYNQ," in *11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, Cracow, Poland, 2021 .
- [47] H. S. Lee and J. W. Jeon, "Accelerating Image Processing on FPGAs using HLS and PYNQ," *2020 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*, 2020.

- [48] H.-A. T. Abdullah, R. Z. Mahmood, S. M. A. Zber, R. A. Mohammed, M. R. Ahmed and A. W. Talab, "Hardware implementation of Sobel edge detection system for blood cells images-based field programmable gate array.," *Indonesian Journal of Electrical Engineering and Computer Science*, 2022.
- [49] M. Hagara, R. Stojanović, T. Bagala, P. Kubinec and O. Ondráček., "Grayscale image formats for edge detection and for its FPGA implementation," *Microprocessors and Microsystems*, vol. 75, no. 0141-9331, 2020.
- [50] Apple Inc., "Blurring an Image," Apple Inc., 10 01 2021. [Online]. Available: https://developer.apple.com/documentation/accelerate/blurring_an_image. [Accessed 21 05 2022].