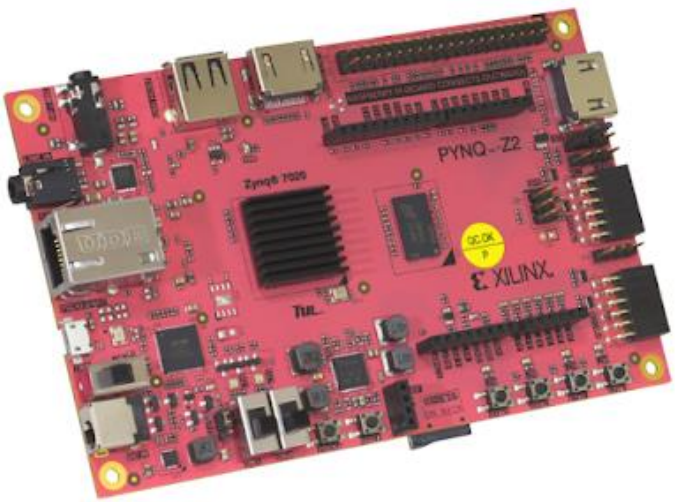
PYNQ FPGA- Canny Edge Detection

Final Research Report submitted for EGH400-2 Semester 2 2022

Student Name: Riley Mcilwain

Supervisor: Dr Jasmine Banks

Submission Date: 5th November 2022



Abstract

The research project was structured around the research question: How do edge detection algorithms perform in hardware implemented on an FPGA, in comparison to a software implemented version in terms of quality, efficiency and speed? An aim was formulated based on this, and the end goal was to implement a chosen edge detection algorithm on an FPGA, that identified the edges of an object. It had to be of high-quality and operate efficiently to further satisfy the aim. Through research and extensive testing, the Canny edge detection algorithm was the finalised choice and implemented on the PYNQ FPGA using C programming language in Vitis HLS and Vivado. The design was simulated and analysed through tests on the PYNQ FPGA to see if it satisfied the aim of the research project. It was concluded that the algorithm, produced high quality detection results, and the execution times were efficient to that of older version designs. As a result, the aim was satisfied.

Table of Contents

Introduction

Purpose

The report is focussed on the research question: How do edge detection algorithms perform in hardware implemented on an FPGA, in comparison to a software implemented version in terms of quality, efficiency and speed?

The aim of the research project is to implement a chosen edge detection algorithm on an FPGA to successfully identify edges of an object while in support of the following objectives:

- The algorithm implemented in hardware is performing significantly more efficient than software
- The implementation of the algorithm considers an FPGA that has parallel capabilities
- The edge detection result of an image is of high-quality standard

Scope

The finalised research project report will cover strictly what had been achieved regarding the timeline and deliverables (refer to Appendix A) that was initially proposed. It will mainly focus on the final chosen design and the methodology approach to achieve the design. Previous designs throughout the project's timeline can also be referred to in the Appendix for more information and understanding. Research in the modern day that was reviewed in literature from the initial project's proposal to the finalised version will be included as it influenced the final design of the edge detection algorithm. The design will consider ethical and sustainability factors from Engineers Australia. Based on the outcome of the design, future recommendations can be communicated to evolve the project further.

Literature Review

Review Background

With rapid advancement of technology over the years, research in Field Programmable Gate Arrays (FPGAs) and Image Processing is one of the areas that had increasingly expanded. FPGAs are used in a wide range of applications such as aerospace and defence, video and image processing, automotive systems, consumer electronics, medical and much more. Image processing had become increasingly more popular in applications with an FPGA. The specific focus of this research project in this area will be the implementation of the image processing technique- edge detection on an FPGA board. Edge detection is important in image processing to define the edges of an object in an image. It acts as an essential pre-processing step for further image analysis techniques. For the implementation of the edge detection technique, adequate research and development is required. Previous technologies and algorithms must be recognised and improved upon where it is feasible. Furthermore, current new technologies in development will be most useful. The purpose of the review will be to collect and evaluate literature to contrast between each other, then conclude on which edge detection technique will be most appropriate.

Review Scope

As this investigation is a research project, it will be used to provide practical solutions for edge detection techniques that can be utilised on FPGA applications based on existing literature. Currently, there is a wide range of literature from the past to now. However, literature that was published within the past decade would be most relevant to the scope of this investigation. As FPGAs and Image Processing are a rapidly developing field, previous technologies will be outdated in terms of implementation and efficiency of edge detection techniques. The main priority of this review will be to pursue current techniques that have been practically tested as a solution in the past decade. These solutions can be further explored and expanded upon to improve overall performance.

FPGAs & Image Processing

General Information on FPGAs

A FPGA is an integrated circuit constructed with Configurable Logic Blocks (CLBs) in an array that are all interconnected. CLBs contain Look Up Tables (LUTs), flip flops and multiplexers that contain logic which are programmed by using a hardware description language such as VHDL or Verilog. Due to FPGAs containing CLBs, they can be reconfigured post manufactured, which allows for rapid prototyping and constant change of design. (Xilinx, 2022) Implementing different types of edge detection algorithms is perfect for this and can be experimented with in numerous ways. FPGAs have parallel capabilities which can allow for heavy computational tasks to be done very efficiently such as that of image processing algorithms. Current algorithms are not completely optimised and will have to be modified accordingly to be supported on an FPGA with parallelism. (Bailey, 2019) Specific types of FPGA boards can also support algorithms that were created in software and transformed into a hardware description language.

PYNQ Technologies

FPGAs designed by Xilinx such as the PYNQ is a part of the ZYNQ7 branch line. PYNQ is an open-source project that consists of a framework to allow developers to use the C, C++ and Python languages, libraries, APIs, and device drivers. Usually, FPGAs require developers to use hardware description languages that logically program a FPGA board. However, developers with a more orientated software perspective can use C and Python languages to implement functions on the PYNQ board. (Lee & Jeon, 2020) The PYNQ board can create highly efficient applications by using parallel computations with a high bandwidth IO, and real-time signal processing. Edge detection techniques deal with algorithms, and the PYNQ can take advantage of this with parallelisation. (Xilinx, 2021)

For a developer to create an application on the PYNQ it must be created in Vitis HLS and Vivado programs. Vitis HLS is a high-level synthesis programming tool that allows C, C++, and OpenCV functions to be optimised and transformed into hardware description language in the form of an Intellectual Property (IP) block. This can be then accessible in Vivado to integrate into a FPGAs design. Vivado is a tool mainly for creating a hardware design to run on an FPGA. It has many features to auto optimise programmable logic for the FPGA to benefit timings, power consumption and resource utilisation. (Xilinx, 2022) For an edge detection application, the PYNQ would be very beneficial as efficiency of algorithms is one of the main factors of success. It also has OpenCV library support, which can assist with image processing techniques. Currently, there is minimal peer reviewed research that involves edge detection with the use of PYNQ. This is one pathway that can be further explored with self-research and from this an implementation of edge detection using the PYNQ.

OpenCV Libraries

OpenCV is an open-source computer vision library that is easily accessible through C, C++, Python, Java and MATLAB programming languages. It can simplify the approach to implementing an edge detection technique through a variety of pre-prepared functions. The library is used all over the globe, including researchers, governments and well-established companies. (Xu, Baojie, & Guoxin, 2017) Xilinx has integrated OpenCV functions into Vitis HLS through C++ programming language. Therefore, these functions can be accelerated on the PYNQ in hardware producing highly efficient results than that of software. (Xilinx, 2021) Using these libraries is a path that can be explored and will be beneficial in implementing a chosen edge detection technique on the PYNQ.

Edge Detection Algorithms

Sobel

The Sobel is a popular edge detection algorithm in Image Processing. A main benefit of the Sobel is that it is simple algorithm making it more efficient and implementable than others. The Sobel method is that it will compute the approximate gradient of an image in the horizontal and vertical components, combine them, and as of a result create a gradient edge map. It is accomplished by two Sobel 3 x 3 masks, each containing gradient operators, one for the horizontal and vertical direction to detect its respective edges. (Ravivarma et al., 2021) The Sobel masks are set out by in Equation (1),

$$M_1 = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} and\ M_2 = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} (1),$$

where M_1 is the vertical mask, M_2 is the horizonal mask.

These masks will convolve with the image to be filtered to calculate the approximate gradient of each data point in the horizontal and vertical directions. (Ravivarma et al., 2021) With the two approximate gradients corresponding to their point from the original image, they are combined to calculate the gradient magnitude with Equation (2),

$$g(x,y) = \sqrt{g_1^2(x,y) + g_2^2(x,y)} \quad (2),$$

where $g(x,y)$is the combined gradient of a particular point in the image, g_1 is the vertical gradient of a point, and g_2 is the horizontal gradient of a point.

The gradients direction at a particular pixel is also important as it defines the direction of an edge. The direction can be calculated with Equation (3),

$$\theta(x,y) = \tan^{-1}\left(\frac{g_2(x,y)}{g_1(x,y)}\right) \quad (3),$$

where $\theta(x,y)$ is the calculated angle of the gradient at a particular pixel, g_1 is the vertical gradient of a point, and g_2 is the horizontal gradient of a point.

The gradient magnitudes computed can be stored in an array to form a gradient edge map and can be displayed to see the edges detected in the image. With an understanding of the method, it needs to be implemented onto a FPGA to perform real-time edge detection with a system architecture. Researchers (Singh, S., Saini, A., Saini, R., Mandal, A, et al., 2014) claimed that issues currently exist. These are the optimisation of the resources used and the implementation of the Sobel method on a FPGA. However, they proposed a Sobel-based edge detection architecture that claimed to use "38% less FPGA resources as compared to architectures presented in literature while maintaining real-time constraints." (Singh, S., Saini, A., Saini, R., Mandal, A, et al., 2014) The proposed architecture can be seen in Figure 1.

Block Diagram of The Sobel Edge Detection - *adapted from (Singh et al., 2014)*

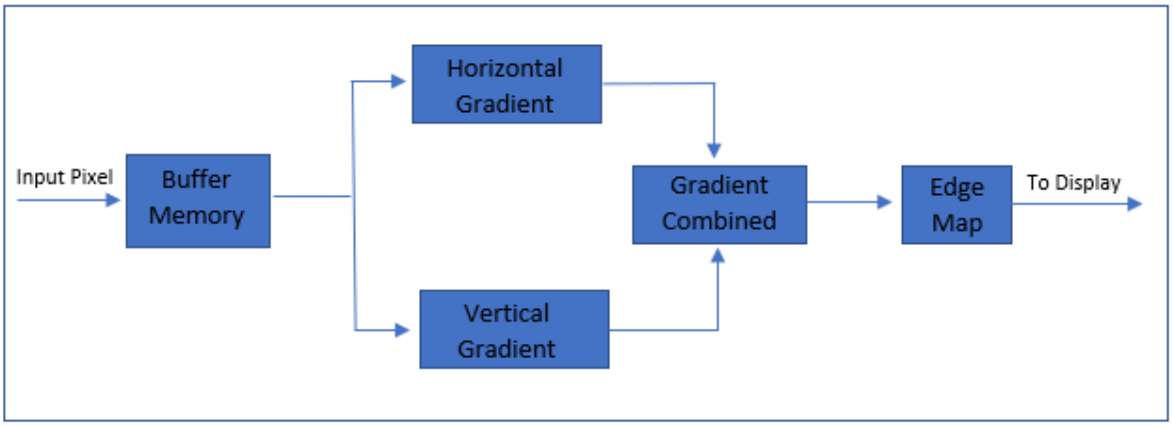


Figure 1

The architecture contains five different blocks with different purposes. Input pixels will be supplied and be stored in a memory buffer until two rows are completely full. The system will begin to filter these pixels and the horizontal and vertical gradients are approximately calculated based on the neighbouring pixels read from the buffer. This type of architecture makes use of the parallel capabilities the FPGA has. Therefore, when the horizontal and vertical gradients are calculated, they are done so simultaneously through parallel hardware, which results in an increased overall efficiency of the system. The gradients are combined based on previous mathematical equations established, and an edge map will be generated based on a threshold value the user can specify. The edge map can be displayed at the output as an image. According to Researchers (Singh, S., Saini, A., Saini, R., Mandal, A, et al., 2014), they claimed that the Sobel algorithm will produce mediocre results based on their practical solution provided and there are different edge detection techniques that can improve upon the Sobel algorithm.

Canny Edge Detection

The Canny edge detection is one of the most popular techniques due to its high-quality detection. It significantly outperforms existing Sobel algorithms in terms producing sharper edges and reduction of noise. (Sangeetha, D., Deepa, P., 2019) However, there are slight drawbacks to the Canny. The algorithm is far more complex and time consuming. It also will require more computational power to use this algorithm which results in a high latency in the system. There are currently multiple research papers dealing with these issues, in particular real-time applications using the Canny on an FPGA. According to (Xu, Varadarajan, Chakrabarti & Karam, 2014), "The Canny algorithm has high latency and cannot be employed in real-time applications." The implemented Canny in this paper executed more efficiently in hardware on the FPGA compared to OpenCV software results. This was due to employing the use of absolute operations which simplified the algorithm. However, the edge detected map produced, was poor quality and not feasible for a real-time application at the time. Although, the paper was produced back in 2014, the system architecture can be discussed for a fundamental understanding of the Canny algorithm. In Figure 2, the Canny algorithm block diagram can be demonstrated.

Block Diagram of The Canny Edge Detection - *adapted from (Xu, Varadarajan, Chakrabarti & Karam, 2014)*

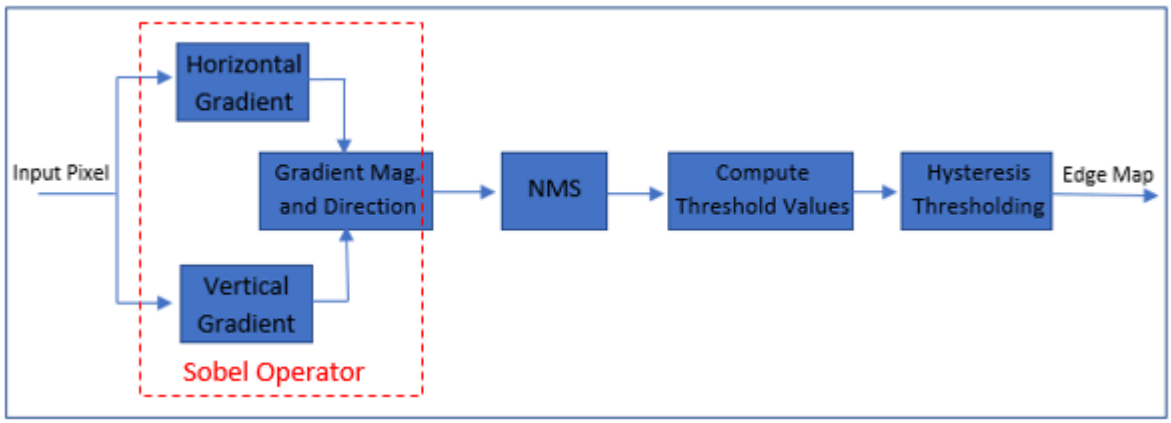


Figure 2

The Canny can be identified as an extension to the Sobel algorithm. It uses the Sobel operator that was discussed previously to produce a basic edge map. However, there are more components with the Canny to produce a high-quality edge of the image. The Non-Maximal Suppression (NMS) component is where each pixel and their respective gradient magnitude can be tested to see if it's at its maximum value in its surrounding pixel field of a 3x3 window. If the pixel is at its maximum, then it can be determined to be an edge. Otherwise, the pixels gradient will be suppressed to a lower intensity. As of result of this technique, the edges will be thinned out and more sharp edges will only be present. (L. Q. Tan and B. He, 2020)

Thresholding is another technique in the Canny algorithm to eliminate pixels that contain noise and illumination variation. Potential edges can be identified by high and low threshold values based on the gradient magnitude histogram. The median value of the gradient magnitude can be used to assist in finding the high and low threshold values. (Sangeetha, D., Deepa, P., 2019)

Since 2014, more research and results had been conducted to further improve the Canny algorithm to overcome the latency and quality issues. According to (Sangeetha, D., Deepa, P., 2019), the researchers proposed a parallel implementation of the Canny algorithm on the Xilinx Virtex-5 FPGA board. It would reduce the latency while maintaining the edge detection quality. This more modern approach would work in a real-time application which was superior than other papers explored. Due to this type of FPGA, it supports parallelisation through the method of pipelining functions. It can be demonstrated in Figure 3.

Pipelining Architecture of Canny Algorithm – *adapted from (Sangeetha, D., Deepa, P., 2019)*

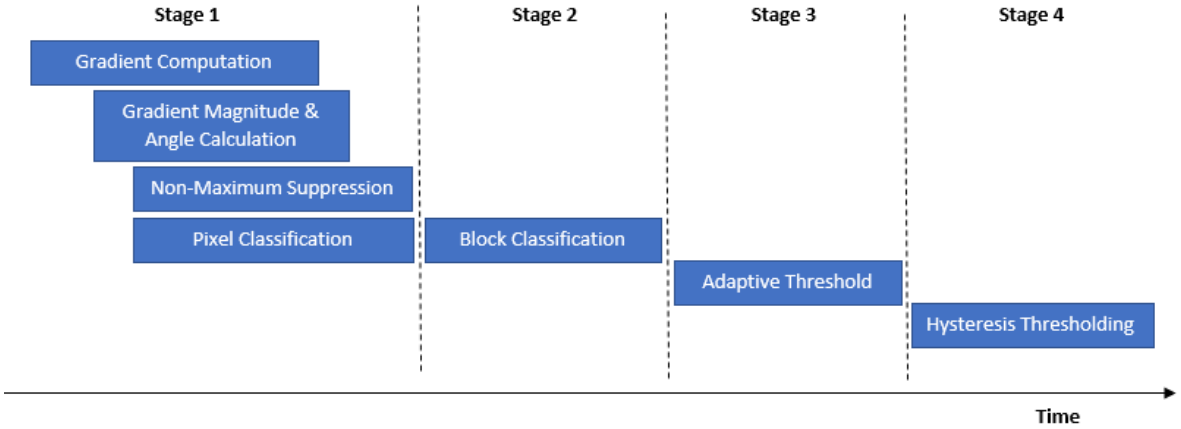


Figure 3

This architecture similarly follows that of the sequential based Canny algorithm discussed previously. However, now some functions can be computed together. In Stage 1, multiple functions can begin computing simultaneously once the data begins processing. The functions in this stage do not require the entire image data to be received all together, due to no data dependencies existing. Stages 2-4 have data dependencies and requires all data to be finished processing in each stage as it relies on the entire data from the previous to do calculations. If pipelining is enabled in these stages, data will result in being corrupt or missing in memory. This architecture system discussed was based on the Xilinx Virtex-5 FPGA which can be difficult to implement using VHDL code to create logic. Other implementation methods can be explored, such as implementing the Canny algorithm on the PYNQ FPGA and use its advantageous features.

Review Conclusion

From gathering primary sources and peer reviewed journals, gaps were identified in the literature review. It was understood that the PYNQ FPGA was a desirable choice due to its parallel capabilities and its support of programming languages and libraries. Especially the use of the OpenCV library integrated into Vitis HLS. It had not been explored yet in literature with edge detection, so persistence and self-learning is required for the challenge. It was also identified that the Canny algorithm was a more superior edge detection method than that of the Sobel. Very few researchers had implemented the Canny on an FPGA in comparison to the Sobel. It was much more complex and required more steps to implement through pipelining multiple functions together. Most practical solutions were implemented on FPGAs that required a lot of resources. However, the Canny did produce much higher quality results. Therefore, the pathway of the research project, implementing the Canny algorithm with C programming language on the PYNQ was the plan. The end goal- creating a practical solution that was efficient and of high-quality.

Methodology

To address the proposed research question, particular methods were chosen to assist with the project. The main project methodology adopted was the Agile approach. The project was destined to continually evolve and change as time progressed due to the nature of software and FPGA boards. The Agile methodology is demonstrated in Figure 4.

Agile Flow Chart

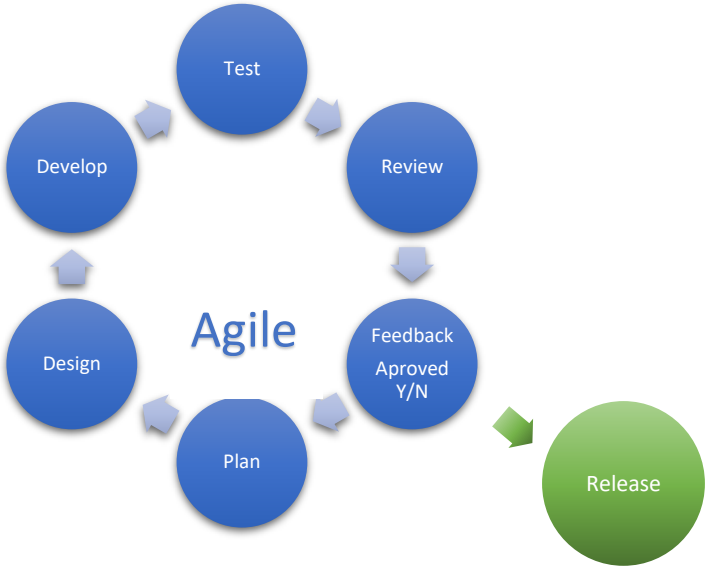


Figure 4.

Initially the project required a planning phase, this is where most of the research will happen. A literature review was required to gain an understanding of different edge detection techniques. To gather the literature, primary sources were important. Peer reviewed journals that provided practical solutions were selected. This ensured that the information was valid, and gaps could be identified accordingly through an analysis of their solutions.

With a plan of the projects' pathway, the design phase began. This was where the agile method truly benefited for the project. Sustainability was considered in the design stage, these were the environmental, social, and economic aspects. With that in mind, an edge detection method and type of FPGA board was chosen. The development stage is where the algorithm is created for an edge detection method. The use of the programs Vitis HLS and Vivado, will assist with the development of the edge detection algorithm being implemented on the FPGA. With the algorithm completed, the testing phase began. Simulation is a vital step before testing the algorithm in real-time hardware. The algorithm design was simulated in Vivado and displayed a list of results on how the FPGA would perform. The data collected was power consumption, timings, and resource utilisation. The design can be tested in hardware, after verifying it works in simulation. Multiple image inputs should be used to gain a range of data where it can be reviewed in the next stage.

The design implemented in hardware and its results produced, can be reviewed with validation and analysis techniques. Using a comparison test between the software and hardware results, it determined which had the superior quality and efficiency. From these results, the stakeholders were informed on the current design and how it was performing. In return, their feedback determined whether the design was a practical solution that can be released, or the planning and design phase will begin once again. The project design was destined to continually evolve as time progressed. Other designs may be also introduced, and the same procedures will be applied. The project should have multiple designs nearing the end of the timeline. This is where a final design will be chosen and implemented successfully, based on the frequent testing and revisions of previous designs.

The risks and ethics of the project should be considered throughout the agile approach. The Engineers Australia Code of Ethics and Guidelines should be continuously viewed, to conduct as a professional Engineer. Risks should also be monitored throughout the project cycle, and they will be identified and reported on in the results section.

Design

Vitis HLS Design

For the Canny algorithm to be implemented on the PYNQ board, a custom Intellectual Property (IP) block must be designed. An IP was designed in Vitis HLS using the programming language C, refer to Appendix B for a full overview of the code. This can make it simpler creating the Canny algorithm without using Verilog or VHDL. The use of C libraries such as OpenCV can be integrated into the design as Vitis HLS has support. Figure 5 demonstrates the Canny algorithm in the form of an IP block.

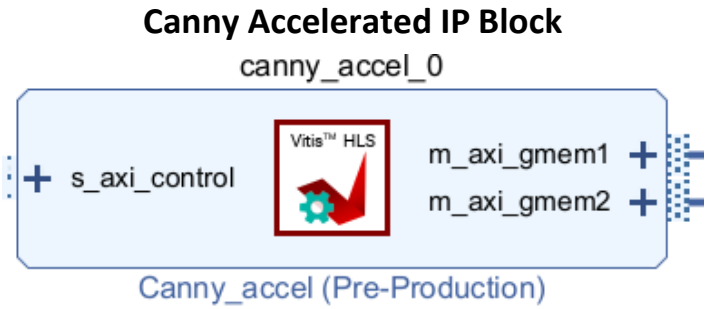


Figure 5.

The IP is designed to consist of an input and output port. These are ***m_axi_gmem1*** and ***m_axi_gmem2*** respectively. They will connect to an Advanced eXtensible Interface (AXI) Interconnect, and they will be responsible for transporting, reading, and writing data between ports on the PYNQ. The ***s_axi_control*** port is connected to a different AXI Interconnect. It will be responsible for controlling the Canny IP block and writing to its register map to modify variables such as the height, width, and threshold values. The data of the selected image will travel into the input of the IP in an AXI format. The input can accept up to a data width of 24 bits divided into three channels. Each channel has a width of 8 bits. This is due to a pixel containing three different values for Red, Blue, and Green (RGB). However, if an image is of Grayscale, then only one channel is required, and the pixel has one colour value. These pixels are then processed through a loop pipeline and stored into a matrix image container acting as an on-board buffer. Pipelining will increase the efficiency of the system dramatically by operating concurrently rather than sequentially. The image container with the data is stored in contiguous physical memory and can be directly passed to the Canny algorithm for the best possible performance. The Canny algorithm is implemented with the assistance of the provided Vitis Vision library. This is a modified form of OpenCV, which Xilinx had integrated into Vitis HLS. Due to the library being modified and the nature of the FPGAs hardware, it can be operated in parallel. This results in much more rapid processing times compared to software. The gradient edge map generated by the accelerated Canny algorithm then must be converted back from an image container to an AXI format. This will allow for the edge map data to be transported from the IPs output. The custom IP is synthesised and exported as a register transfer level (RTL) design which converts the C language to a hardware description language. This IP is readily available to be imported into Vivado where we can see the block diagram.

Vivado Design

The Vivado project is set up initially to support the PYNQ board by using the ZYNQ7 Processing System. Generally, in Vivado, programming in VHDL would be necessary to create logic blocks to produce a desired outcome. However, most of the work was done in Vitis HLS creating the Canny IP. The Canny IP can be implemented with other surrounding pre-made IPs to all connect to the Processing System. Vivado has a feature to assist in routing and connecting the logic blocks together, but knowledge is required to know which specific ones are needed. In Figure 6, it demonstrated the required logic blocks connected to the Canny IP and the ZNYQ7 Processing System.

Block Design Diagram for PYNQ FPGA with Custom IP Canny Accelerated Algorithm

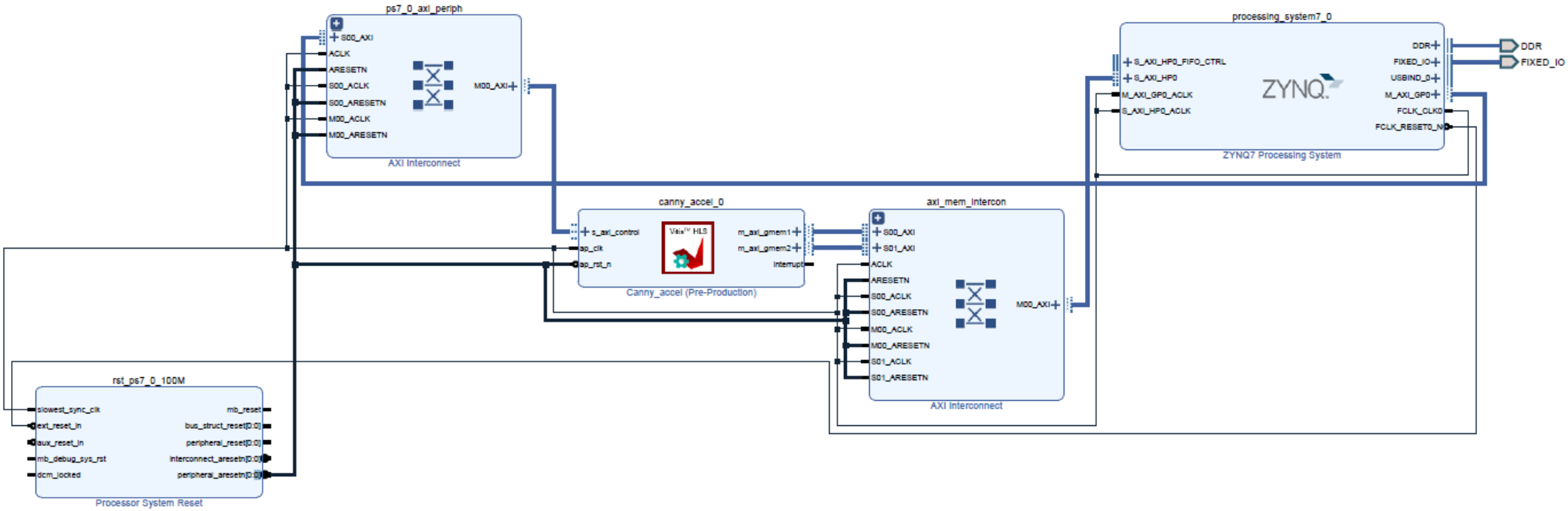


Figure 6. – Use the Zoom feature for finer details*

The ZYNQ7 Processing System is the main module for the entire PYNQ to operate. It contains a Dual-Core processor, a memory controller, communication protocols and high-speed connectivity to Programmable Logic. The ZYNQ7 consists of a System-on-Chip (SoC) style integrated Processing System (PS) that connects to a Programmable Logic (PL) unit where custom IPs such as the Canny can be integrated within. (Xilinx, 2022) AXI blocks in the design are vital in that they will read, write, and transport data on the PYNQ, connecting all peripherals together in a network. AXI Interconnects main purpose is to transport data from the processor to the Canny's IP input, outputs and vice versa. The Canny IP is connected to the Processing Systems clock to be in sync with the entire system. Processors run on a specified clock cycle and instructions are executed within these cycles, therefore synchronisation is crucial for the Canny IP to be functional. A Processor System Reset module is also required in the design to make sure when an asynchronous reset is called and modules are reset, it is re-synchronised with the processors clock.

Results & Analysis

Simulation Results & Analysis

The design can be simulated through synthesis and implementation to see how it will perform before being programmed to the PYNQ. It is essential to do this to prevent issues happening further in real-time. The designs utilisation, timing and power consumption can be analysed in Figure 2, 3 and 4 respectively.

Utilisation of Resources on the PYNQ

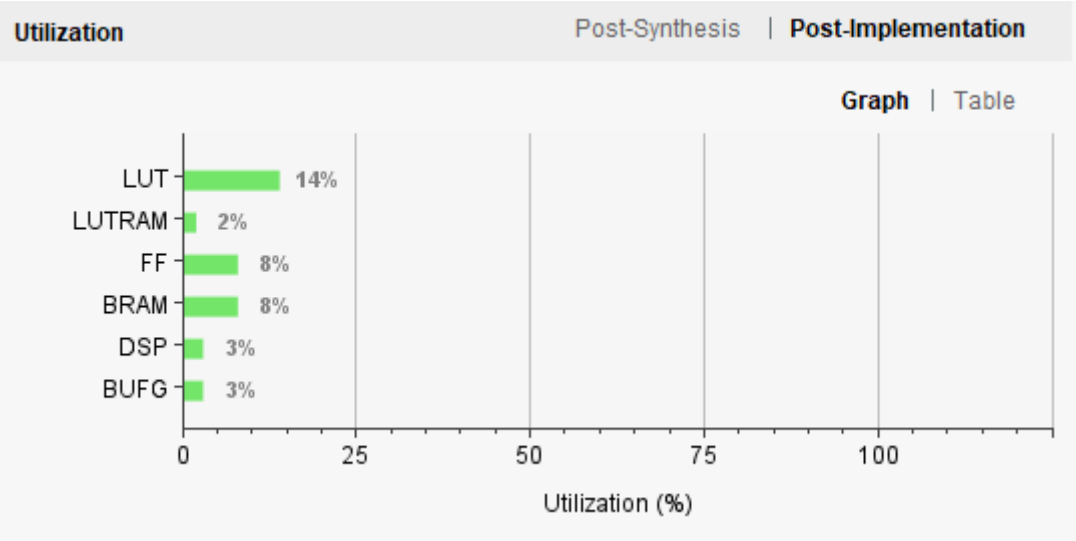


Figure 7.

Through simulation, the designs utilisation of resources can be analysed. Look Up Tables (LUT) are of most important in this analysis. They are essentially truth tables, representing a logic gate. These are all in combination with flip-flops and multiplexers to form a Configurable Logic Block (CLB). CLBs are what makes the FPGA perform functions specified by the user, all through combinational logic. It required 14% of the total amount of Look Up Tables (LUT) on the PYNQ which is a very low amount. The design also used a low amount of RAM to store data on the board. Only 8% Block Random Access Memory (BRAM) and 2% LUT Random Access Memory (LUTRAM) was utilised. Overall, the design did not use a lot of resources and showed how efficient the design is in terms of utilisation.

Summary of the Designs Timing

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.950 ns	Worst Hold Slack (WHS):	0.034 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	22519	Total Number of Endpoints:	22519	Total Number of Endpoints:	9674

All user specified timing constraints are met.

Figure 8.

The design timing is based on the data/ signal propagation time through the system. The worst-case scenario of slack was measured in nanoseconds. It is very minimal and would have little effect on the systems design if this situation did happen. Data would still arrive on time in sync with the clock of the processor. The total slack however in simulation was zero indicating the design operated without any issue. The design had zero failing endpoints

Summary of the Designs Power Consumption

Power

Total On-Chip Power:	**1.46 W**
Junction Temperature:	**41.8 °C**
Thermal Margin:	43.2 °C (3.6 W)
Effective ϑJA:	11.5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium
Implemented Power Report	

Figure 9.

The power consumption of the design can be analysed. Based on these results from Vivado's simulation, it had a medium confidence level that these values produced are fairly accurate. The Total On-Chip Power that the PYNQ operated at was 1.46 Watts. In terms of the sustainability, such as the environmental and economic aspects, this is a highly efficient design that consumed a low amount of power during operation. The upkeep of this design would be of low maintenance.

Real-Time Results & Analysis

With the system architecture designed and analysed through simulation, the bitstream can be generated and flashed to the PYNQ. The PYNQ is connected to the local network and can be accessed by a computer on that same network using Jupyter Notebooks. The code to operate the PYNQ and the Canny accelerated IP block can be referred to in the Appendix D. The Canny is activated by accessing its register map and specifying it to start. The low and high threshold values are set to 100, 200 respectively. The threshold values can be modified, and the results will vary through trial and error.

Results of Edge Map - Canny Algorithm in Hardware and Software

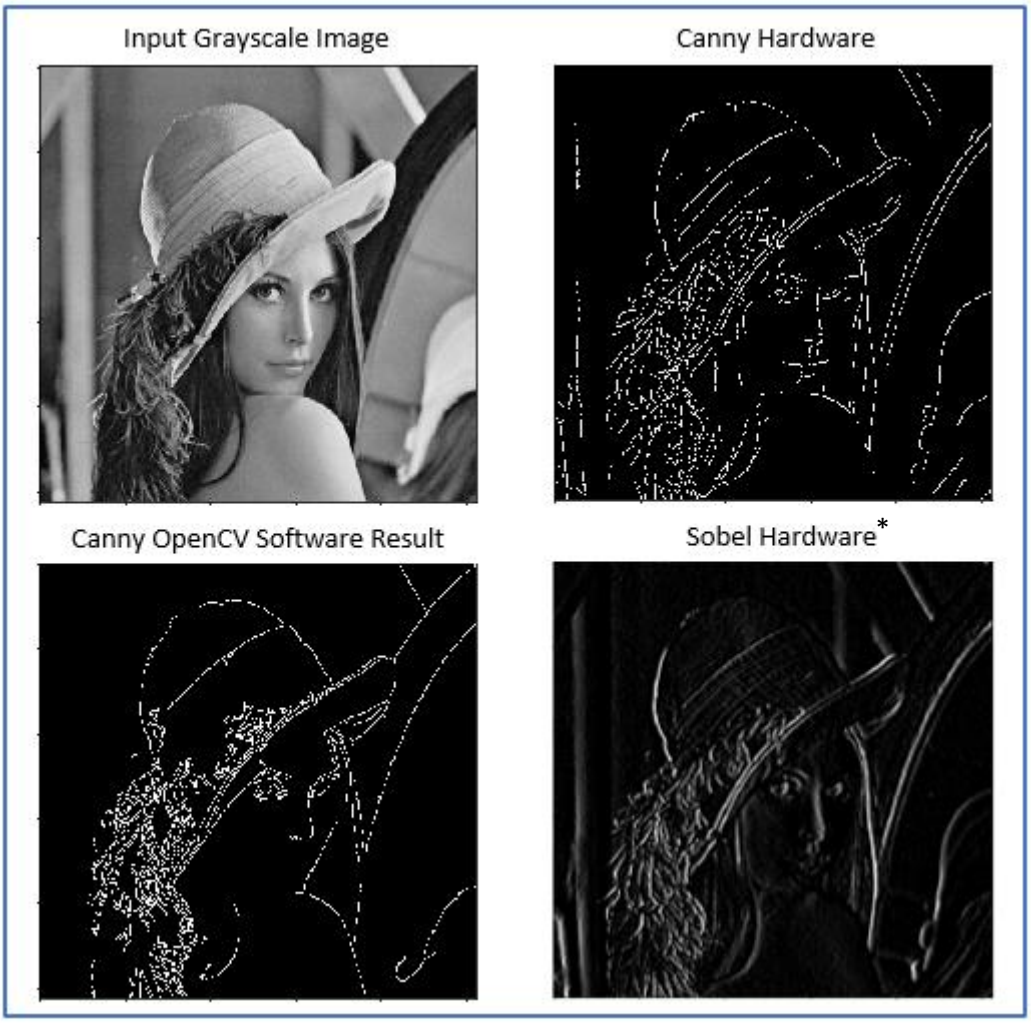


Figure 10. - *Sobel conducted in Progress Report, refer to Appendix B for more info.

Algorithm Execution Timing Results

Implementation Type	Timed Result (10 loops, best of 3)
Sobel Hardware*	6.62 ms / loop
Sobel Software*	145 ms / loop
Canny Hardware	905 µs / loop
Canny Software	33.3 ms / loop

Table 1. – *Sobel conducted in Progress Report, refer to Appendix B for more info.

An analysis of the results in Figure 5, it can be identified that the Canny hardware and software results are somewhat similar in quality. Comparing the Canny hardware with the Sobel hardware result that was produced in the earlier stages of the project progress report (refer to Appendix B for previous results of Sobel). The Canny is much more superior in quality, the grain and the noise is removed. The edges are sharp and clearly defined in white. It is confirmed the project progressed in the right direction to receive higher quality results. However, some of the edges are discontinuous and missing due to thresholding errors. Threshold values do have to be changed for every different type of image edge detected as they have different gradients. More trial and error would be required to get the perfect threshold values of this image. The Canny software result in Python using the OpenCV library, some of the edges are also not continuous due to thresholding. However, it does manage to edge detect further small details within the image. Although, this isn't a major benefit as the main goal is to successfully edge detect the main subject within the image in a high-quality manner. The Canny algorithm in hardware is successfully edge detecting the main subject in the image and can be well recognised. There is some room for improvement, to get the finer details.

Analysing the timing results from Table 1, The Canny hardware algorithm measured to be 905µs per loop, as opposed to the software algorithm measured at 33.3ms per loop. This is a substantial difference, and the hardware version executed was approximately 37 times more efficient than the software version. It clearly had demonstrated the parallel capabilities of the PYNQ FPGA board. The pipelining of the functions was successful as they computed simultaneously. Comparing the hardware execution times between the Canny and the Sobel, the Canny measured to be 7 times more efficient. The final implemented design of the Canny currently meets the objectives that were discussed and satisfies the projects main aim. However, there could be always more room for improvement with thresholding values.

Risks, Ethics, Sustainability & Stakeholders Outcomes

Risks

Originally from the Project's Proposal, minimal risks were identified due to the nature and topic of the project, FPGAs. This was in confirmation agreement with stakeholders. The project would be conducted in a controlled environment on the QUT campus, with few stakeholders involved. The one early risk that was identified was the use of datasets such as images. To minimise any issues involved with this, public datasets were used to avoid breaking copyright integrity. With the development of the research project over the course of the year, more risks were identified. The programs used for the design aspects, Vitis HLS and Vivado; are continuously updated quarterly by Xilinx. There is risk of the design becoming incompatible in the future with the updated programs. Furthermore, there is a risk with the OpenCV libraries used in the project. Vitis HLS currently supports the use of these libraries, however in the past, Xilinx have removed support of different types of libraries. Caution must be taken, and the designs should be updated yearly if there is a continued use required.

Ethics

The Engineers Australia Code of Ethics and Guidelines was demonstrated in the project. Referring to the Code of Ethics 1-4, demonstrating integrity and leadership, practising competently, and promoting sustainability are vital aspects to conduct as a professional Engineer. Over the course of the project, they were demonstrated continuously. Leadership was a major ethic, which led to reasonable decision making and shaped the project to how it is.

Sustainability

Implementing sustainability is important to any project being successful. The environmental and economic, social aspects were considered in the progression of the project. Due to the nature and size of the project involving FPGAs, there are not many factors to consider. Stakeholders were also consulted on this and confirmed. However, the FPGAs design was analysed in terms of the environmental and economic aspects. The FPGA itself consists of a PCB with electrical components and chips on board. The size of it is relatively small and are manufactured with a low cost. The design discussed and analysed that was implemented on the PYNQ FPGA, had a low power consumption and would not be of any concern. The product's end of life can be considered. These components can be easily recycled or reused where necessary in other electrical systems if required. A social aspect is that the project in the long term can be considered in the teaching community. Implementing a method for tutorials of information for the projects design created can be beneficial in the field of work for more exposure of this topic. Currently, there is not a wide range of teaching content regarding the PYNQ FPGA. Self-learning is an important aspect of the PYNQ and reading the limited documentation provided by Xilinx. However, there should be other available methods to learning the PYNQ FPGA.

Stakeholders

Dr Jasmine Banks was the main listed stakeholder on the project. Consultations with stakeholders and their feedback is very beneficial to confirm the vision of the project and making sure it will succeed. Their input had also supported some of the project's progression with ideas of solutions to solve design problems in terms of Image Processing and FPGAs. No other stakeholders or industry partners were involved in the project.

Conclusion

The final report communicated the fundamentals of the PYNQ FPGA and the finalised Canny algorithm in a hardware implementation. Initially at the beginning of the project, the Sobel algorithm was implemented into a design, and it was analysed in the Progress Report. The results showed the Sobel functioning somewhat efficiently on the PYNQ. Although, it could always evolve to produce higher-quality results with the foundation set from the Sobel. Therefore, the Canny was introduced and designed in a similar process which used the parallel capabilities of the PYNQ. The Canny required a more complex understanding of how it operated and was reviewed in literature. With the results the Canny produced, it satisfied the aim of the research project. The Canny was a chosen edge detection method on the PYNQ FPGA to successfully identify edges of an object that was high-quality standard, used the PYNQ parallel capabilities and it performed much more efficiently than software.

References

Bailey, D. (2019). Image Processing Using FPGAs. Journal Of Imaging, 5(5), 53. doi: 10.3390/jimaging5050053

Lee, H., & Jeon, J. (2020). Accelerating Image Processing on FPGAs using HLS and PYNQ. 2020 IEEE International Conference On Consumer Electronics - Asia (ICCE-Asia). doi: 10.1109/icce-asia49877.2020.9277085

Ravivarma, G., Gavaskar, K., Malathi, D., Asha, K., Ashok, B., & Aarthi, S. (2021). Implementation of Sobel operator based image edge detection on FPGA. Materials Today: Proceedings, 45, 2401-2407. doi: 10.1016/j.matpr.2020.10.825

Xilinx. (2021). PYNQ Introduction — Python productivity for Zynq (Pynq). Retrieved, from https://pynq.readthedocs.io/en/latest/

Xilinx. (2021). Vitis vision library - GitHub Pages. Retrieved, from https://xilinx.github.io/Vitis_Libraries/vision/2021.1/index.html

Xilinx. (2022). Documentation Portal. Retrieved, from https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction-to-Vitis-HLS

Xilinx. (2022). What is an FPGA? Field Programmable Gate Array. Retrieved, from https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

Xilinx. (2022). Zynq-7000 Processing System IP. Retrieved, from https://www.xilinx.com/products/intellectual-property/processing_system7.html

Xu, Q., Varadarajan, S., Chakrabarti, C., & Karam, L. J. (2014). A distributed canny edge detector: Algorithm and FPGA implementation. IEEE Transactions on Image Processing, 23(7), 2944-2960. doi:10.1109/tip.2014.2311656

Xu, Z., Baojie, X., & Guoxin, W. (2017). Canny edge detection based on open CV. 2017 13th IEEE International Conference on Electronic Measurement & Instruments (ICEMI). doi:10.1109/icemi.2017.8265710

Appendices

Appendix A – Timeline and Deliverables from Project Proposal

Timeline & Deliverables

Number	Focus	Deliverable	Dependency	Release Date/ Milestone
1	Literature Review/ Project Proposal	Literature Review Report, Project Proposal		01 May, 2022
2	Conceptual design	Algorithm chosen and PYNQ implementation	1	15 May, 2022
3	Sub project 1			
4	Detailed design	Detailed design of operation of sub project 1	3	25 May. 2022
5	Modelling/ Rapid prototype & experiment	PYNQ FPGA – Edge Detecting Images Sobel Algorithm	4	1 June, 2022
6	Project Progress Report & Oral	Report and Oral Presentation	5	12 June, 2022
7	Sub project 2	Canny Edge Detection – Extension of Sobel		
8	Detailed Design	Detailed design of operation of sub project 2 (Canny)	7	30 September
9	Test & Validate/ Verify	A successful working edge detection system that is efficient, compare with previous sub projects	8	12 October
10	Draft report	Draft	8, 9	24 October
11	Project Delivery Report	Project Final Report	10	28 October, 2022
12	Oral Defence	Oral Presentation		17 Nov, 2022

Figure 11.

Appendix B – Sobel Design from Progress Report

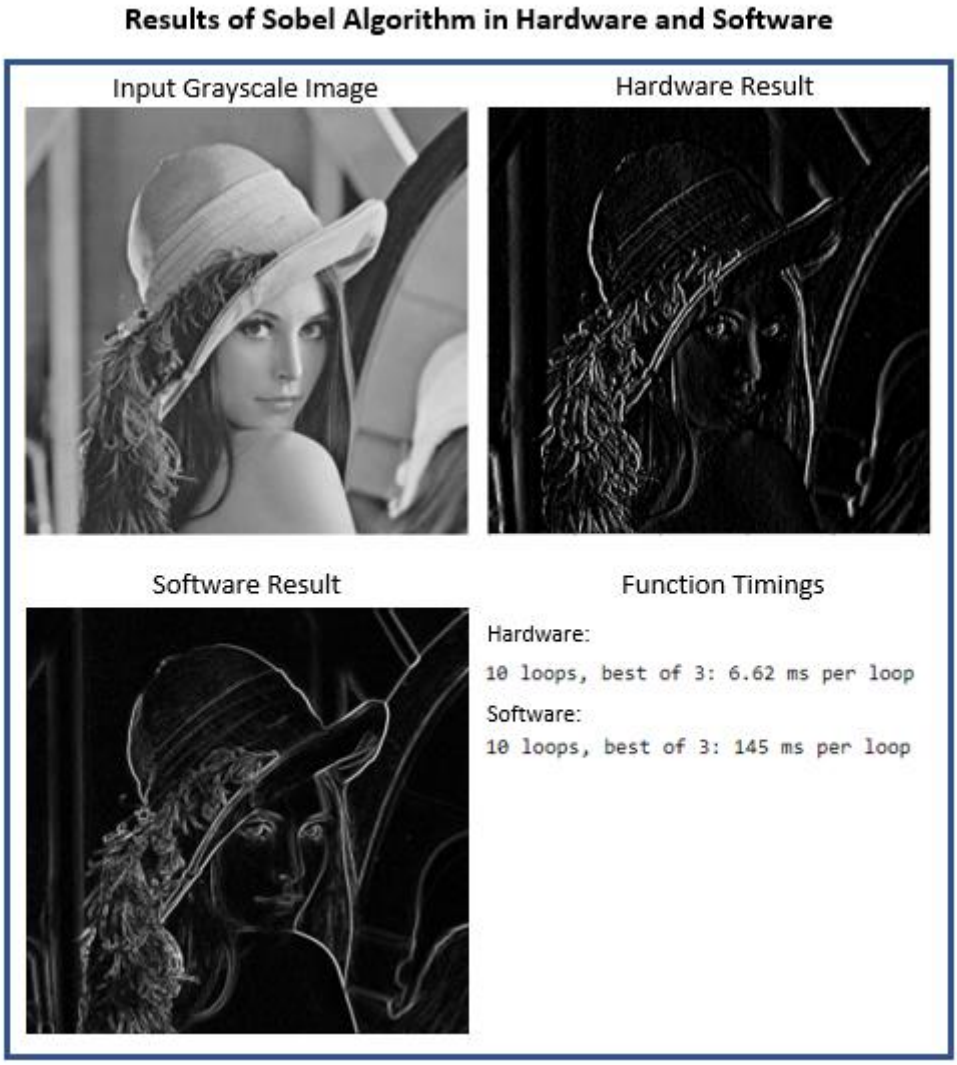


Figure 12.

Block Design Diagram for PYNQ FPGA with Custom IP Sobel Accelerated Algorithm

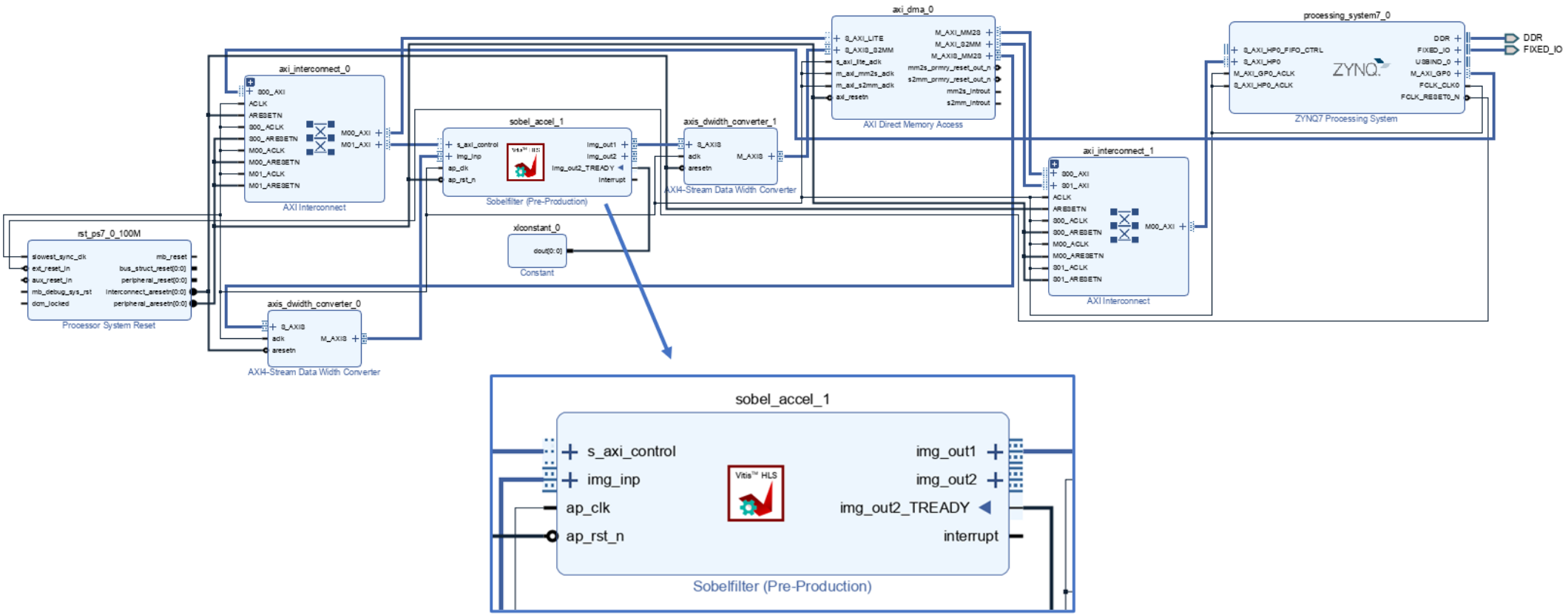


Figure 13.

Appendix C – Vitis HLS code of Canny IP Block

```
#include "xf_canny_config.h"

extern "C" {
void canny_accel(ap_uint<INPUT_PTR_WIDTH>* img_inp,
                 ap_uint<OUTPUT_PTR_WIDTH>* img_out,
                 int rows,
                 int cols,
                 int low_threshold,
                 int high_threshold) {


      //Input and output ports
    #pragma HLS INTERFACE m_axi     port=img_inp  offset=slave bundle=gmem1
    #pragma HLS INTERFACE m_axi     port=img_out  offset=slave bundle=gmem2


      //Variables that can be changed through registers
    #pragma HLS INTERFACE s_axilite port=rows
    #pragma HLS INTERFACE s_axilite port=cols
    #pragma HLS INTERFACE s_axilite port=low_threshold
    #pragma HLS INTERFACE s_axilite port=high_threshold
    #pragma HLS INTERFACE s_axilite port=return


      //Make sure no overflow
    int npcCols = cols;
    int divNum = (int)(cols / 32);
    int npcColsNxt = (divNum + 1) * 32;
    if (cols % 32 != 0) {
        npcCols = npcColsNxt;
    }

    //Create the matrixes to store the image data
    xf::cv::Mat<XF_8UC1, HEIGHT, WIDTH, XF_NPPC1, XF_CV_DEPTH_IN_1> in_mat(rows, cols);
    xf::cv::Mat<XF_2UC1, HEIGHT, WIDTH, XF_NPPC32, XF_CV_DEPTH_OUT_1> dst_mat(rows,
npcCols);

    //Convert Python array data to vitis hls compatiable array data
    xf::cv::Array2xfMat<INPUT_PTR_WIDTH, XF_8UC1, HEIGHT, WIDTH, XF_NPPC1,
XF_CV_DEPTH_IN_1>(img_inp, in_mat);
    //Compute the Canny Algoritm OPENCV
    xf::cv::Canny<FILTER_WIDTH, NORM_TYPE, XF_8UC1, XF_2UC1, HEIGHT, WIDTH, XF_NPPC1,
XF_NPPC32, XF_CV_DEPTH_IN_1,
                  XF_CV_DEPTH_OUT_1, XF_USE_URAM>(in_mat, mid_mat, low_threshold,
high_threshold);
    //Convert vitis hls compatable array data back to python array data
    xf::cv::xfMat2Array<OUTPUT_PTR_WIDTH, XF_8UC1, HEIGHT, WIDTH, XF_NPPC1,
XF_CV_DEPTH_OUT_1>(dst_mat, img_out);
}
}
```

Figure 14.

Appendix D – Jupyter Notebook to operate Canny in Hardware and OpenCV Software

In [ ]:

```
from PIL import Image
import numpy as np
import cv2
from matplotlib import pyplot as plt
%matplotlib inline
from pynq import allocate, Overlay
```

In [ ]:

```
canny_design = Overlay('/home/xilinx/design_1.bit')
canny = canny_design.canny_accel_0
```

In [ ]:

```
#Import the image and convert to grayscale
image_path = "/home/xilinx/jupyter_notebooks/base/video/data/test.jpg"
original_image = Image.open(image_path)

np_array = np.array(original_image)
width, height = original_image.size



grayscale = np.ndarray(shape=(height,
                              width),dtype=np.uint8)

cv2.cvtColor(np_array,cv2.COLOR_RGB2GRAY,dst=grayscale)

plt.figure(1)
plt.imshow(original_image)

plt.figure(2)
plt.imshow(grayscale, cmap='gray')
```

In [ ]:

```
print(canny.register_map)
```

In [ ]:

```
#Allocate buffers for the input and output of the Canny IP Block
in_buffer = allocate(shape=(height, width),
                           dtype=np.uint8, cacheable=1)
```

```
out_buffer = allocate(shape=(height, width // 2), dtype=np.uint8, cacheable=1)
```

In [ ]:

```
#Copy the grayscale image into the input buffer
in_buffer[:] = grayscale
print(grayscale)
```

In [ ]:

```
#Set the threshold values for the Canny
low_t = 100
high_t = 200
```

In [ ]:

```
#Assign values, and assign buffers to the respective input and output
canny.register_map.rows = height
canny.register_map.cols = width
canny.register_map.low_threshold = low_t
canny.register_map.high_threshold = high_t
canny.register_map.img_inp_1 = in_buffer.physical_address
canny.register_map.img_out_1 = out_buffer.physical_address
print(canny.register_map)
```

In [ ]:

```
%%timeit -r 3 -n 10
#Time the Canny Algorithm in hardware
canny.register_map.CTRL.AP_START=1
check = 1
while check > 0:
    check = canny.register_map.CTRL.AP_DONE
print("Canny Done!")
```

In [ ]:

```
plt.figure(figsize=(12, 10));
_ = plt.imshow(out_buffer, cmap = 'gray')
```

In [ ]:

```
print(out_buffer)
```

In [ ]:

```
#Clear the Buffers
del in_buffer
```

```
del out_buffer
```

In [ ]:

```
%%timeit -r 3 -n 10
#Time the OpenCV SOFTWARE CANNY
edges = cv2.Canny(np_array,100,200)
```

In [ ]:

```
#Plot the original image, hardware edge map and software edge map
plt.subplot(131),plt.imshow(np_array,cmap = 'gray')
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
plt.subplot(132),plt.imshow(edges,cmap = 'gray')
plt.title('Edge Image'), plt.xticks([]), plt.yticks([])
plt.subplot(133),plt.imshow(out_buffer,cmap = 'gray')

plt.show()
```

Figure 15.