# Rapid Prototyping of an FPGA-Based Video Processing System

Zhun Shi

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Engineering

Apr 29th, 2016
Blacksburg, Virginia

Keywords: FPGA, Computer Vision, Video Processing, Rapid Prototyping, High-Level Synthesis

# Rapid Prototyping of an FPGA-Based Video Processing System

Zhun Shi

(ABSTRACT)

Computer vision technology can be seen in a variety of applications ranging from mobile phones to autonomous vehicles. Many computer vision applications such as drones and autonomous vehicles requires real-time processing capability in order to communicate with the control unit for sending commands in real time. Besides real-time processing capability, it is crucial to keep the power consumption low in order to extend the battery life of not only mobile devices, but also drones and autonomous vehicles. FPGAs are desired platforms that can provide high-performance and low-power solutions for real-time video processing. As hardware designs typically are more time consuming than equivalent software designs, this thesis proposes a rapid prototyping flow for FPGA-based video processing system design by taking advantage of the use of high performance AXI interface and a high level synthesis tool, Vivado HLS. Vivado HLS provides the convenience of automatically synthesizing a software implementation to hardware implementation. But the tool is far from being perfect, and users still need embedded hardware knowledge and experience in order to accomplish a successful design. In order to effectively create a stream type video processing system as well as to utilize the fastest memory on an FPGA, a sliding window memory architecture is proposed. This memory architecture can be applied to a series of video processing algorithms while the latency between an input pixel and an output pixel is minimized. By comparing my approach with other works, this optimized memory architecture proves to offer better performance and lower resource usage over what other works could offer. Its reconfigurability also provides better adaptability of other algorithms. In addition, this work includes performance and power analysis among an Intel CPU based design, an ARM based design, and an FPGA-based embedded design.

# Rapid Prototyping of an FPGA-Based Video Processing System

Zhun Shi

(GENERAL AUDIENCE ABSTRACT)

Computer vision technology can be seen in a variety of applications ranging from mobile phones to autonomous vehicles. Many computer vision applications such as drones and autonomous vehicles requires real-time processing capability in order to communicate with the control unit for sending commands in real time. Besides real-time processing capability, it is crucial to keep the power consumption low in order to extend the battery life of not only mobile devices, but also drones and autonomous vehicles. Field Programmable Gate Arrays (FPGAs) are desired platforms that can provide high-performance and low-power solutions for real-time video processing. As hardware designs typically are more time consuming than equivalent software designs, this thesis proposes a rapid prototyping flow for FPGA-based video processing system design. High-level synthesis tools translate a software design into hardware descriptive language, which can be used for configuring hardware devices such as FPGAs. The video processing algorithm design of this thesis takes advantage of  a high-level synthesis tool from one of the major FPGA vendors, Xilinx. However, high-level synthesis tools are far from being perfect. Users still need embedded hardware knowledge and experience in order to accomplish a successful design. This thesis focuses on the effeciency of memory architecture for high-performance video processing system designs using a high-level synthesis tool. The consequent design results in a video processing rate of 60 FPS (frames per second) for videos with a resolution of 1920 x 1080. This thesis shows the possibility of realizing a high-performance hardware specific application using software. By comparing my approach with the approaches in other works, the optimized memory architecture proves to offer better performance and lower resource usage over what other works could offer. Its reconfigurability also provides better adaptability of many computer vision algorithms.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

Computer vision has become increasingly popular in a wide variety of domains in the industry. Some applications of advanced computer vision algorithms include face detection that can be found in modern cameras and many smartphones, license plate recognition used by law enforcement, lane detection for autonomous driving, and object tracking used in military and commercial drones. Although it might not be necessary to have live video processing capability for many applications, some applications such as lane detection and object detection used for autonomous driving system would require an input stream from cameras to be processed at real time in order to send signals back to the powertrain and steering control unit to respond properly. FPGAs are highly desired platforms for real-time video processing because of the potential to extract highly-parallelized computations and its energy efficiency. However, hardware development normally consumes more time and human resources than a similar software development would consume. For a development based on FPGAs, the engineer also needs to have a good knowledge of digital logic circuit as well as some Hardware Description Languages (HDLs) such as Verilog and VHDL for constructing and configuring Register-Transfer Level (RTL) circuits in an FPGA.

In recent years, FPGA development has been moved towards to higher abstraction levels. The move not only helps improve productivity, but also lowers the barrier for more algorithm designers to get access to the tempting FPGA platform. There is a wide selection of tools available in the market that can be used for high-level synthesis. Conventionally algorithm designers prefer using high-level languages such as C/C++ for algorithm developments, and Vivado HLS is one of the tools that is capable for synthesis C/C++ code into RTL for hardware implementation. Nevertheless, most high-level synthesis tools could not translate a high level implementation to a RTL implementation directly, and users must restructure the high level implementations in order to make them synthesizable and suitable for the specific hardware architecture. Therefore, it becomes important to adapt to the high-level synthesis tool and to discover approaches for achieving an efficient design with high performance and low resource usage. The high-level synthesis tool used in this work is Vivado HLS from Xilinx.

This thesis addresses the following issues:

1. How can engineers with limited FPGA experience quickly prototype an FPGA-based SoC design for high performance video processing system?

2. How productive is Vivado HLS? What changes need to be made in order for a software implementation to be synthesized to a hardware implementation?

3. How are the performance and area of video processing algorithms modeled by Vivado HLS compared to that of RTL modeling from related works?

4. How are the performance and power consumption of a FPGA-based video processing system compared to that of an Intel CPU based video processing system?

## 1.2 Research Contribution

This thesis work presents an FPGA-based video processing system rapid prototyping flow that aims to lower the boundary between software and hardware development. The rapid prototyping flow consists of two major parts: 1) the video processing system architecture design, and 2) the video processing algorithms design. By understanding the underlying architecture of Xilinx's Zynq platform, I can quickly assemble a video processing system on the block level with minimum

RTL modifications. The development period can be reduced from months to weeks. In addition, since Vivado HLS does not provide a common structure for domain-specific algorithm designs, this thesis proposed a generalized sliding window memory architecture that can be used for a series of video processing algorithm designs in Vivado HLS. Several optimizations are also done to the proposed sliding window memory architecture so that it not only improves the video processing rate by more than 30%, but also reduces the flip-flop utilization by up to 53% and saves the LUT utilization by up to 75% when comparing with similar works done in [1][2][5][6]. Furthermore, this work also conducts an analysis of the video processing system's power consumption between a FPGA-based SoC platform and an Intel CPU based platform. The video processing system running on the embedded FPGA platform consumes up to 26x less energy comparing to the Intel CPU based platform while can still offer more than 7x more performance. This work demonstrates the possibility of rapid prototyping of a low-power video processing system with more than enough of the real-time processing performance.

## 1.3 Organization

This thesis is organized as the following: Chapter 2 discusses and compares the potential platforms that can be used for video processing acceleration. Also included in Chapter 2 are several related works that were done by others as well as some background information related to the hardware and tools that have been used throughout this work. Moreover, Chapter 3 describes the approaches that I have taken for both the video processing system architecture design and algorithm design. It introduces the generatlized sliding window memory architecture for a series of video algorithm designs. Chapter 3 also discusses the optimizations that have been done to the proposed memory architecture as well as approaches I have taken for evaluating the performance of video processing systems on different platforms. In addition, Chapter 4 presents the results of performance, resource utilization, and power consumption of the system. It includes not only the performance and resource utilization comparison between my work and other works, but also the performance and power consumptions between the embedded FPGA platform and CPU platform. Last but not least, Chapter 5 will conclude this thesis with discussion about the contributions, challenges and future work.

# Chapter 2

# Background and Related Work

## 2.1 High Performance Video Processing Options

The performance improvement of a microprocessor has been largely contributed by the growing number of transistors on chips. Moore's law predicts that the number of transistors will be doubled every two years, which has been proved to be valid and practical in the past and at present. However, unless scientists can find better materials than silicon or can advance the fabrication technology, the growing pace will become slower as the transistor density and power consumption keep pushing to their limits. Furthermore, one of the biggest challenges that prevents a single core CPU from improving its performance is the power wall – the processor's power consumption and temperature. The power wall limits the potential for a processor to increase its frequency in order to improve performance because boosting frequency could bring about severe power consumption and overheating. The power consumption of a processor has become so crucial that mainstream processors still maintain a clock rate (~3GHz) that is similar to that of the year 2007 or earlier [8]. Rather than soaring up the clock rate of a processor or increasing the transistor density on a single core processor alone, the direction of computer architecture in the future should move the focus on multi-core architectures and heterogeneous computing architectures such as multi-core CPU, DSP (Digital Signal Processor), and GPU (Graphics Processing Unit). In addition, custom logics such as FPGA (Field Programmable Gate Array) and ASIC are worth being investigated to become future computer architecture due to their high performance and power efficiency.

In Figure 2.1, it shows the advantages and tradeoffs of each of the platforms mentioned. While custom logic circuits such as ASICs and FPGAs are able to provide the greatest performance and highest power efficiency, the developments on these platforms can be costly. By contrast, since software engineers can extend software developments to DSPs and GPUs, the development cost are lower. As a tradeoff, DSPs and GPUs could not provide the same level of performance and power efficiency of what can be achieved in a custom logic circuit design.

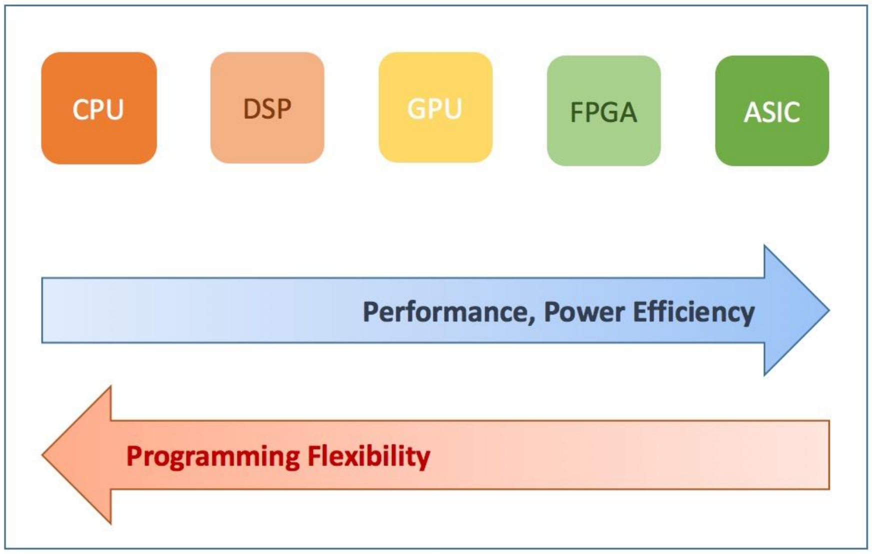


Figure 2.1: Comparison of five different processing platforms

### 2.1.1 DSP

A Digital Signal Processor (DSP) is a kind of customized microprocessor that is able to accelerate applications that involves a great amount of digital signal processing such as audio and video processing. Since a DSP has a specialized architecture that is optimized for digital signal processing, devices with DSPs such as tablets and smartphones can perform better and run processes using less energy in certain categories comparing to what a general purpose microprocessor can offer. DSPs are commonly seen in today’s SoCs. With modern DSPs, mobile devices can deliver outstanding camera and audio performance while saving power to better extend battery performance. For instance, a Hexagon DSP from Qualcomm is capable of executing 29 simple RISC operations within one clock cycle [9]. In addition, normally engineers can program DSPs with C, or with assembly code for enhanced performance. Therefore, although DSPs require a certain amount of hardware knowledge, it will take software engineers less effort to learn programming DSPs than other hardware platforms such as FPGAs, and result in a good rate on time to market.

### 2.1.2 GPU

The Graphical Processing Unit (GPU) has become a popular parallel computer architecture for many computational intensive applications because of its high performance and its nature of parallelism. GPU was used for graphic rendering in the early years, but its application for scientific computation was not explored very widely until CUDA was released by NVIDIA in 2007 [10]. As a simple-many core architecture, a modern GPU typically has significantly more cores than a multi-core CPU, which enable hundreds of computations to be executed concurrently. A typical GPU processor is highly parallelized, and it relies on massive pipelining of operations to achieve high performance. In a typical GPU architecture, it can have hundreds of multiprocessors, and each multiprocessor includes a set of thread processors. Most GPUs use SIMD parallelization, and each thread processor executes the same operation for different data. In addition, GPUs have independent memory from host memory, and they have high memory bandwidth. The multiprocessors are greatly multi-threaded, and they are able to switch quickly between data retrieval and execution. Therefore, GPUs generally can still offer massive throughput though there is a high latency during the memory access between the GPU memory and processing units [11].

### 2.1.3 FPGA

FPGAs have become increasingly popular as a configurable computing platform targets for high performance computing applications. FPGAs typically have the potential to provide notable performance improvement over conventional computing platforms while are still able to consume significantly less energy. FPGAs have been widely adopted in various types of applications such as high performance computing for science, high-speed digital signal processing, software defined radio, low-latency network processing and real-time video processing. The ability for reconfiguration of today's FPGAs allows engineers to flexibly model a hardware specific application. Although the performance of a FPGA implementation can be slower than an ASIC

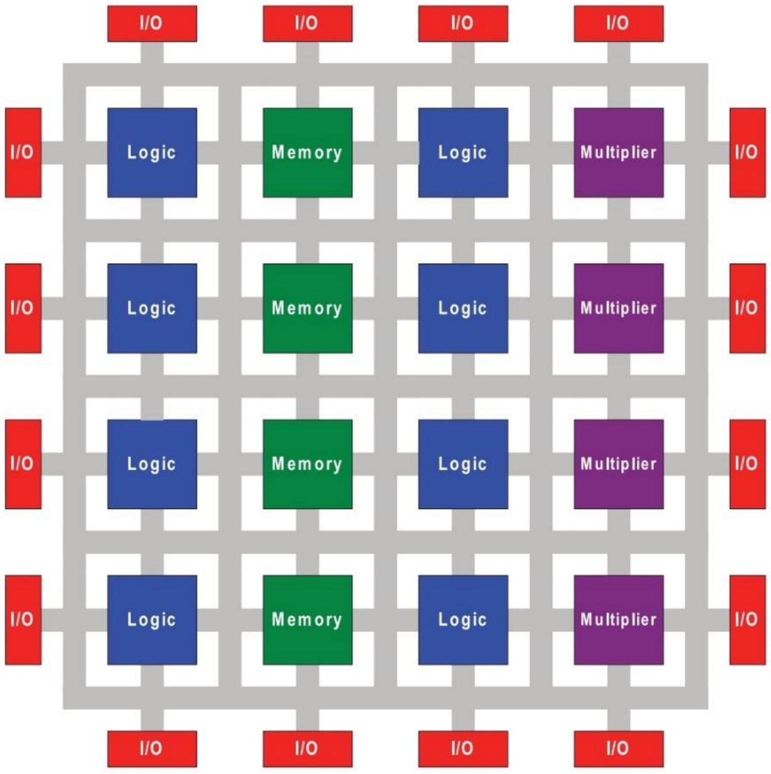


Figure 2.2: FPGA Architecture Overview [7].

(Application Specific Integrated Circuit), and FPGAs consumes more area than an ASIC, FPGAs are significantly cheaper than ASICs, much easier to be fabricated and to be used, and highly customizable. FPGA is composed of thousands of programmable logic blocks and interconnections so that hardware engineers can customize specialized logic functions for their applications. As shown in Figure 2.2, the logic blocks inside an FPGA contain basic general logic blocks, Memory Blocks, Digital Signal Processing blocks (DSP) such as multiplier blocks, Flip Flops (FF), Look-Up Tables (LUT), and many other [12]. Programmers can take control of the connections among the logic blocks and I/O peripherals by configuring the surrounded interconnects in order to create complex digital circuits. Modern FPGAs are typically based on SRAM programming technology because of its ability for reconfiguration and its adoption of standard CMOS manufacturing process [12].

A typical FPGA design flow involves architecture design, HDL coding, behavioral simulation, synthesis, implementation, functional verification and timing and power analysis as shown in Figure 2.3. Tools from two major FPGA vendors, Altera and Xilinx, are highly automated, and they have made FPGA development more efficient than ever before. Although some processes such as synthesis, implementation, and timing and power analysis can all be taken care of by tools automatically, it is crucial for hardware engineers to decide on the architecture design and use Hardware Description Language (HDL) to create designs that can realize desired functionalities and meet timing constraints. VHDL and Verilog are two hardware descriptive languages that have been widely used among FPGA developers. These popular languages allow hardware engineers to create digital circuits ranging from simple and straightforward gate level descriptions to complex algorithmic behavioral models [14]. Although FPGAs can provide better flexibility than fully customizable ASICs, it is still more time-consuming for RTL development than other software-related developments because it requires more skills for engineer to program FPGAs.

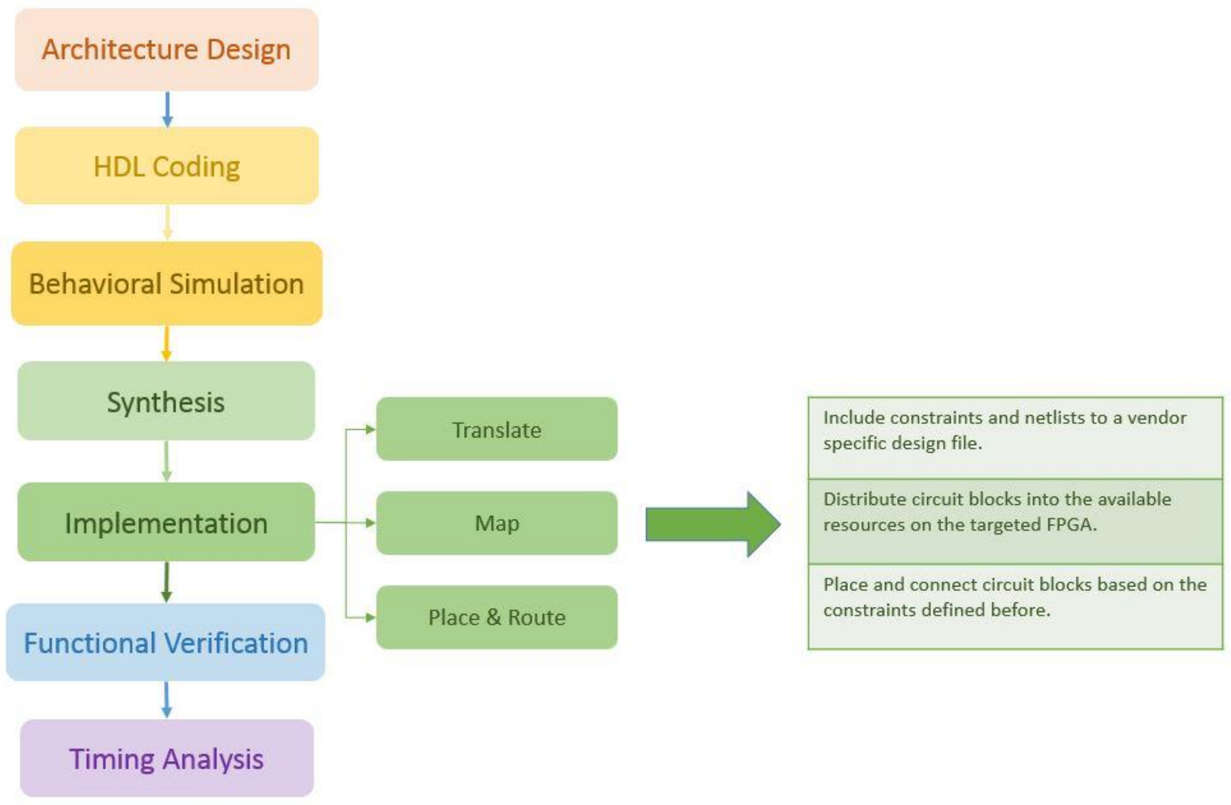

Architecture Design
HDL Coding
Behavioral Simulation
Synthesis
Implementation
Functional Verification
Timing Analysis
Translate
Map
Place & Route
Include constraints and netlists to a vendor specific design file.
Distribute circuit blocks into the available resources on the targeted FPGA.
Place and connect circuit blocks based on the constraints defined before.

Figure 2.3: FPGA Design Flow

### 2.1.4 ASIC

ASICs (Application-Specific Integrated Circuits) are fully customized integrated circuits that are intend for a particular purpose. An ASIC can be customized for a certain application in which high performance and lower power consumption are the main concerns. Since they can be fully customized for a certain application, engineers can take control of the optimization process which will bring about very high performance and low power consumption. Moreover, one of the greatest advantages of choosing ASIC for an application is the cost when the volume is high. Since ASICs are fully customizable, engineers can save on resources which they don't need, and it can cut cost significantly with a high volume.

There are a number of disadvantages of using ASICs as well. Unlike microprocessors and even FPGAs, an ASIC design can hardly be modified once the fabrication process is done. Besides the poor flexibility of an ASIC design, ASIC design flow is also more complex because the processes of floor planning, routing, placement and timing analysis cannot be done by software automatically. Although ASICs can offer even better performance and higher power efficiency than FPGAs, ASICs can be very cost-inefficient for development, and the time to market performance is undesirable. The time for fabricating the design into an ASIC circuit and validation can take several months to many years [15]. Time to market is one of the most crucial reasons that a company will take into account to determine whether it is a good prototyping option. In addition, it takes lots of technical and human resources for an ASIC design, and those factors will likely increase the cost of ASIC development rapidly if the total volume is low.

## 2.2 Hardware

### 2.2.1 Zedboard

Zedboard [26] is chosen for this project because real-time image processing can take advantage of its onboard Xilinx Zynq-7020 All Programmable SoC and its capability for a comprehensive set of port expansions. Zedboard serves as a unique platform for Embedded system design as well as hardware/software co-design. The Zynq 7020 device is composed of a Dual ARM Cortex-A9 CPU based processor system (PS) as well as Xilinx's hardware programming logic (PL), with both on the same chip. The ARM cores are the Cortex A9 MPcore which can run at a peak frequency of 1 GHz. The PS also includes Level-2 cache and memory controllers for DDR2 and DDR3 SDRAM. In addition, users can interface with interrupt controller and various IOs such as SPI, I2C, CAN and UART through software. The Xilinx' AXI Interface also provides high-speed memory access between the PS and the PL. According to Xilinx, the Artix-7 FPGA on the Zynq 7020 device has about 85k Logic Cells, 53,200 Look-Up Tables, 106,400 Flip-Flops, 220 Programmable DSP Slices and 560 KB Block RAM [16]. Xilinx also offers more substantial FPGAs in the Zynq family such as the Zynq 7030 which is equipped with a Kintex-7 FPGA and thus can provide more versatile use cases as well as higher performance.

### 2.2.2 FMC Imageon Module

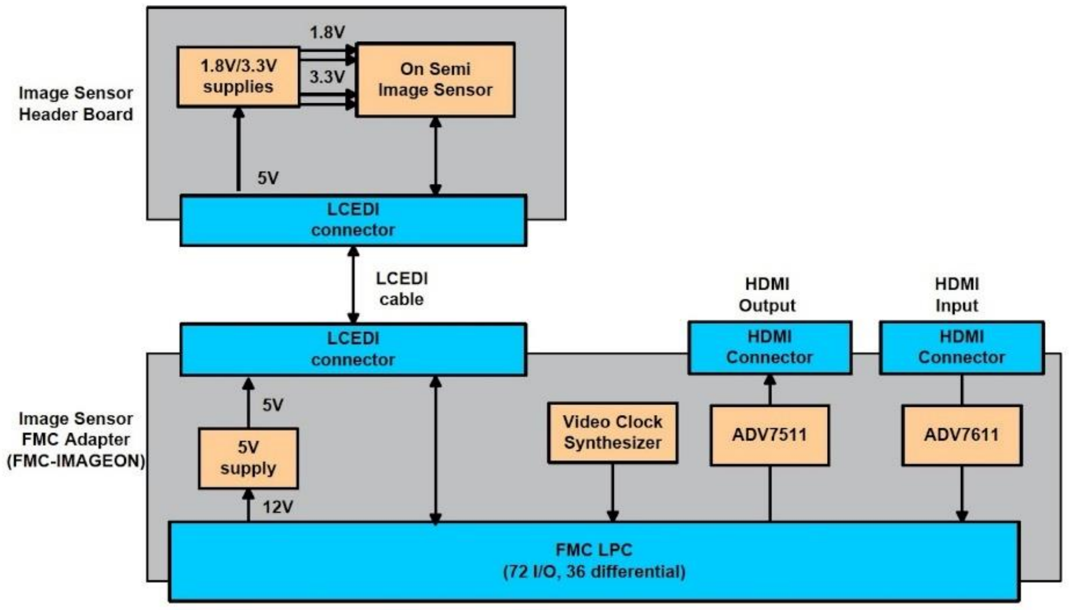


Figure 2.4: FMC IMAGEON module block diagram [17]

Zedboard's capability to attach a FMC module allows a high definition live video source to be streamed in from the HDMI-in port and to be streamed out from the HDMI-out port on the FMC Imageon module. In order to extract the full potential of a FPGA chip, a full HD 1080p video can be streamed in, processed, and outputted to a display at real time thanks to the HDMI input and output on the FMC Imageon module. HDMI has long been the primary interface for high definition video streaming and audio transmitting. Since HDMI contains only digital source, there is no need to apply an A/D converter in the FPGA-based video processing system design. The HDMI input and output ports utilized Analog Devices' ADV7611 and ADV7511 chips, and the module also contains a video clock synthesizer to allow video streams of different definitions and frequencies [17]. A high level block diagram of the FMC IMAGEON module is shown in Figure 2.4.

## 2.3 High-Level Synthesis Tools

Most software engineers will need to spend a significant amount of time to learn to program FPGAs using hardware description language such as Verilog and VHDL because the modeling of a hardware is vastly different than designing a software, and it requires a good knowledge of hardware. Writing sequential software code using languages such as C/C++ corresponds to how a human brain thinks. However, writing hardware description language is a way to model a specific algorithm based on a digital circuit, and it is intrinsically parallel. Also, developing a FPGA application for a given function requires more effort than developing the same function using software due to the time-consuming synthesis as well as simulation and verification. Apparently there is a barrier for software engineers to take on hardware development. Fortunately, there exist a variety of high-level synthesis tools that could help not only software engineers with limited hardware skill prototype a hardware design, but also hardware engineers accelerate the development process.

High-level synthesis tools are able to translate high level programming languages such as C/C++ to Register Transfer Level code in order to program a specific FPGA device. HLSTs are not so new of an idea that the first generation tool was researched in the 1970s. HLSTs were not adopted until the mid-1990s, when the second generation tools were released and being used commercially

[18]. The majority of current generation high-level synthesis tools use general purpose programming languages such as C and C++ for datapath applications synthesis, while the next generation tools are expected to handle to both control path and data path applications [18].

### 2.3.1 Vivado HLS

Vivado high-level synthesis (HLS) provides the flexibility and possibility for software engineers to accelerate designs with computational bottlenecks on the hardware. As to many software engineers, C/C++ is one of the most efficient ways to model algorithms. Even for hardware engineers with experience in writing HDL, HLS frameworks can serve as an alternative solution to hand-coded HDL and thus are able to help engineers improve productivity and save resources. As shown Figure 2.5, the process of HDL coding and behavioral simulation in conventional FPGA design flow can be replaced by the workflow of Vivado HLS. Users can utilize Vivado HLS to create RTL implementations by writing C/C++ code with proper restructurings and to pack the design into IPs for circuit integration. Vivado HLS has become increasingly popular for modeling digital signal processing algorithms which includes image processing algorithms.

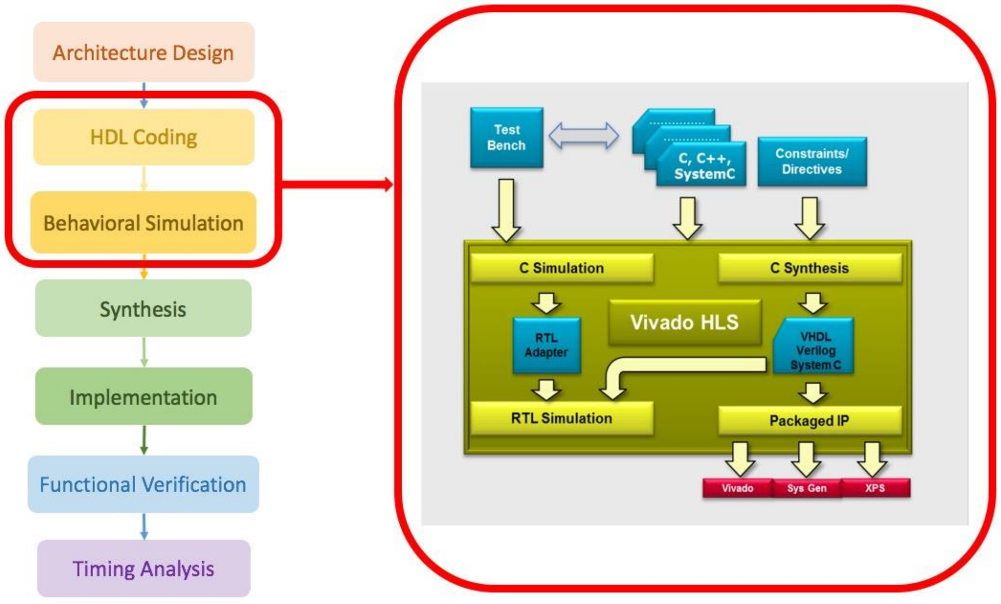


Figure 2.5: FPGA design flow with High-Level Synthesis [20].

Moreover, Vivado HLS allows engineers to verify their design with C/RTL co-simulation for functional correctness. Furthermore, engineers will be able to further improve timing and performance by inserting optimization directives to the code as well as exploiting more parallelisms in the algorithm.

It is beneficial to understand what the tool does on the low level for code translation. Questions may arise that how Vivado HLS will map major software components such as functions, arrays, and loops into hardware. One of the most important concepts in Vivado HLS is the extraction of Control and Data path. Loops and conditional clauses are extracted to control path, and finite state machines are used for control path implementations. With Vivado HLS, variables in the top function declaration will become ports in the hardware block. Some common ports that will be added to the top level blocks during the synthesis include clock, reset, start, done and idle [20]. The control signals are automatically added to the block, and they can be sent to the ARM processor for start control, interrupts, and status readings. These signals can be included into an AXI4-Lite interface, while main data can be under AXI4-Stream interface. Moreover, arrays will be synthesized to memory blocks in the RTL design, and users can choose the type of memory interface such as registers, Block RAMs, DRAMs, and FIFOs [20]. In addition, loops will be automatically mapped to a structure which contains Flip Flops. Some optimizations such as loop unrolling and pipelining can be done by inserting appropriate directives into the code.

### 2.3.2 Other HLSTs

Some examples of recent HLSTs include Bluespec, Catapult C Synthesis, HDL Coder, and Vivado HLS. Many industry leaders such as Ericsson, Nokia, Toshiba and Fujitsu have relied on Catapult C Synthesis, which was released by Cadence, for generating communication related algorithms in various applications [18]. Catapult C Synthesis not only supports generation of HDL from C++, but also from SystemC, which offers the capability to control timing for certain parts of the design. Furthermore, Bluespec SystemVerilog is an innovative hardware description language that can greatly reduce the complexity of modeling a hardware design, and it provides higher abstraction for both behavioral and structural modeling than normal HDLs such as Verilog, VHDL, and

SystemVerilog [19]. In addition, MATLAB has long been proven to be a powerful tool to model image processing and digital signal processing algorithms. Released by the same company, HDL Coder is capable of generating VHDL and Verilog code from MATLAB functions so that engineers can pack the generated code to an IP and integrate it into a FPGA design. The emergence of advanced HLSTs helps software engineers with limited HDL knowledge but still want to accelerate their algorithms on FPGA translate their software code into hardware code with minimum effort compared to writing HDL code.

## 2.4 Related Work

Abdelgawad, Safar, and Wahba have developed a Canny Edge Detection Algorithm on Zynq Platform [1]. They utilized the Targeted Reference Design (TRD) for Zynq from Xilinx as the platform for the experiment of performance comparison between the CPU processor and the hardware accelerator. They declared memory accesses as the major bottleneck for a real-time video processing system. With proper buffering and directive-based optimizations, they were able to achieve a speedup of 100x on Zynq's hardware accelerator. They also provide the utilization estimation of the Canny Edge Detector hardware accelerator, but power analysis is missing. By using the TRD, they couldinspect the performance improvement more directly thanks to the QT GUI interface. However, the TRD design gave rise to less control of the hardware design as well as software development.

Johansson designed a similar Canne Edge Detector using Vivado HLS. His initial algorithm implementation results in only slight performance improvement over the ARM processor and it could not be fitted onto Zynq-7020 because its over resource usage [2]. His final design was able to fit on a Zynq-7020 occupying about 50% of the area on the Zynq-7020. But he claimed the outputs are deviated from all previous implementations, and the performance was not acceptable for a real-time video processing system. One of the reasons that could lead to excessive resource usage is the successive calls of OpenCV functions in Vivado HLS [25]. Although excessive use of OpenCV functions can result in resource overusage and performance downgrade, his result indicates that FPGA still provides over 7x energy savings comparing to what a typical CPU would consume.

Some research teams have used and evaluated Vivado HLS for various image processing algorithms to run on FPGAs. For instance, Monson, Wirthlin, and Hutchings attempted to develop an optical-flow algorithm using Vivado HLS [3]. Their work involves restructuring and optimizations of the algorithm code written in C. Since there are certain requirements for the C code to be synthesizable by Vivado HLS, the authors modified the original OpenCV code and include synthesizable memory transactions. In addition, dataflow directives were used to enable parallel process from different groups. Moreover, they also reduced memory bandwidth and computational load by over 50% with integration of Scharr derivative for acceleration. The results showed that the HLS generated RTL code was able to save about 6/7 energy comparing to the version running on a i7 desktop while resulting in similar or higher performance. Besides running on an i7 desktop, they also tested their implementation on the ARM Cortex A9 embedded processor. With most parameters unchanged, the test on ARM gave rise to a processing speed of only 10.1 FPS, which is about 8x slower than the i7 version and FPGA version. Their experiment proved the possibility of developing a computer vision algorithm for FPGA using Vivado HLS. It would be more convincing to the audiences if the authors can integrate the generated RTL to a physical FPGA-based video processing system and test its usability and performance.

Moreover, by the same research team, Monson, Wirthlin, and Hutchings attempted to optimize another popular image processing algorithm, Sobel filter, using Vivado HLS targeting a Zynq-based FPGA [4]. Their first goal was to restructure an existing Sobel filter written in C to a C synthesizable version in Vivado HLS because the original code contains some non-synthesizable portions. Besides the restructuring, the authors discovered and applied three incremental optimizations that can be synthesized in Vivado HLS. The optimizations include replacement of non-synthesizable pointers with fixed-size arrays, the use of FIFO-based I/O for data transfer speed-up, the addition of loop optimization directives, and the use of line buffers and window buffers. The incremental optimization helped their design to achieve a performance of 388 FPS at a resolution of 640x480. However, their research ignored two important factors in hardware designs: timing and resource usage. They concluded that engineers still need to have a decent amount of hardware knowledge as well as knowledge of using the tool properly in order to successfully translate a software algorithm design into an RTL.

There are some related works of SAD algorithm designs and implementations experimented on FPGAs. One of those works is done by S. Wong, S. Vassiliadis, and S. Cotofana [5]. Rather than using any high-level synthesis tools, their design is based on RTL but can apply to a single 16x16 window computation. They performed simulations for results validation. Also, their design utilized 1699 LUTs and resulted in a maximum clock frequency of 197 MHz Similar to the 16x16 SAD implementation, J. Olivares et al. focused on an area-aware SAD design [6]. Their smallest design of a 4x4 SAD processor brings about 240 LUTs utilizations targeting the Virtex-II device from Xilinx, while the 16x16 SAD processor design results in a respectable utilization of 1945 LUTs while having a maximum clock frequency of 424.99 MHz. One of the common weak points of those two works is that they can only be applied to a single window computation, and therefore real-time video processing could not be accomplished.

Cai et al. utilized the capability of Vivado HLS to transform a software face recognition program to a corresponding hardware design based on Zynq platform [7]. Their intention was to improve the face detection performance, and the result indicates the performance was improved by up to 80% after migrating the computation onto the hardware. Their face location algorithm relies on color segmentation to detect human faces. The algorithm involves transforming from RGB color space to YCbCr color space, converting the query image to grayscale, and locating the skin color region after erosion and dilation. This algorithm results in straightforward and fast computations. Using color segmentation can be computationally efficient and it is possible to achieve real-time image processing performance. However, a relatively clean background is required for face detection using color segmentation. Also, misrecognition could occur if hands and arms are exposed in the query image.

# Chapter 3

# Approach

## 3.1 Rapid Prototyping Flow

This thesis presents a video processing system rapid prototyping flow that allows engineers with little to no HDL experience to develop a FPGA based high performance video processing system. Figure 3.1 demonstrates the rapid prototyping flow from a high level point of view, and this thesis focuses on three of the most important steps in the flow:

1. An FPGA-based SoC video processing system architecutre needs to be designed. The system should allow integration of generated IPs from high-level synthesis tools in order to realize real time video processing capability

2. The design enables the adoption of FPGA acceleration kernels developed by high-level synthesis tools so that engineers can quickly reconfigure the functionality of the system. In this thesis, video processing algorithms written in C can be synthesised into RTL by Vivado HLS. The video proceesing IPs can be generated for system level integration.

3. System level communications allow users to use software for initializing and configuring modules that are developed in the hardware system. Besides, users can even interact with the DDR interface for data transmission between hardware and software. In addition, fast reconfiguration to the hardware can be done without hardware synthesis.

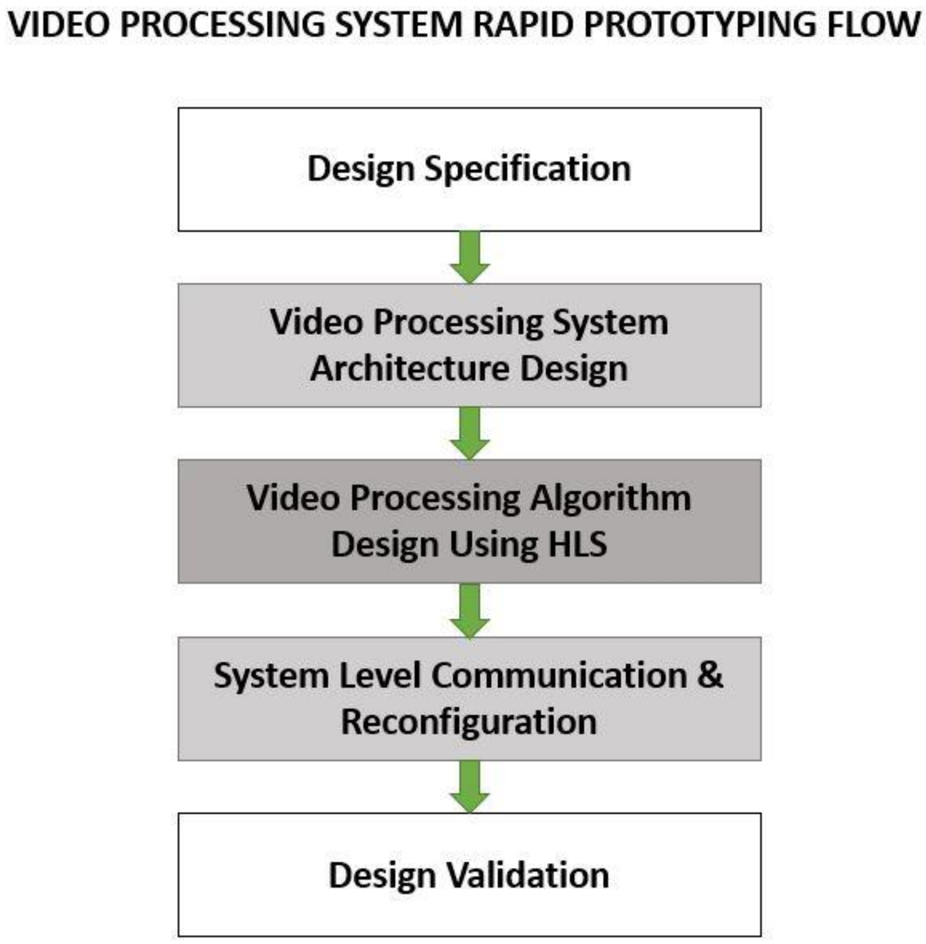

VIDEO PROCESSING SYSTEM RAPID PROTOTYPING FLOW
Design Specification
Video Processing System Architecture Design
Video Processing Algorithm Design Using HLS
System Level Communication & Reconfiguration
Design Validation

Figure 3.1: Video processing system rapid prototyping flow.

The development of the proposed real-time video processing system is divided into two parts: 1) Video processing system architecture design, and 2) Video processing algorithms design. The first part discusses the major components contributed to the video processing system on the Zynq platform including the AXI4 Interfaces used for high throughput data transmissions, while the second part discusses the approaches and optimizations I have taken for building video processing algorithms using Vivado HLS.

## 3.2 Real-Time Video Processing System Architecture Design

The development of the proposed real-time video processing system is divided into two parts: 1) Video processing system architecture design, and 2) Video processing algorithms design. The first part discusses the major components contributed to the video processing system on the Zynq platform including the AXI4 Interfaces used for high throughput data transmissions, while the second part discusses the approaches and optimizations I have taken for building video processing algorithms using Vivado HLS.

Before designing the video processing algorithm, it is essential to build an effective video processing system on the Zynq platform. Thanks to the FMC HDMI Input/output module, a full HD 1080P video source can be streamed into the programmable logic on the Zynq device, and an output video can also be retrieved from the HDMI output. There is also another option to attach a camera module onto the FMC adapter in order to capture live video stream, but the camera module was on back-order and was not available during this research experiment. Figure 3.2 shows a picture of the Zedboard with a FMC module attached:

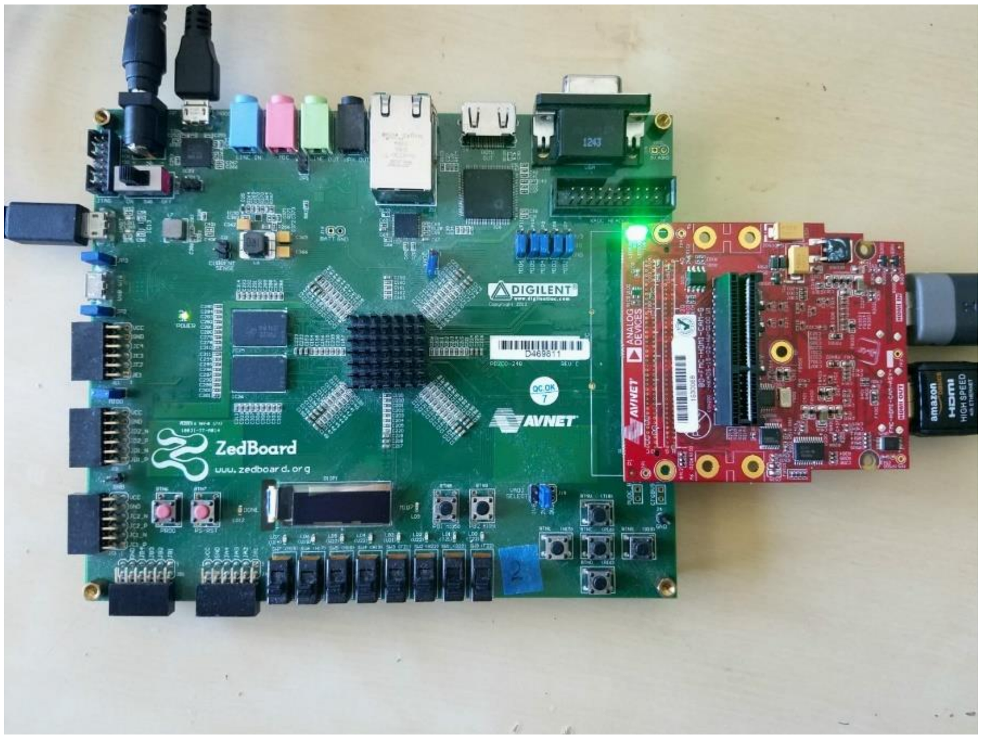


Figure 3.2: Zedboard with FMC IMAGEON HDMI module.

### 3.2.1 Design Overview

One of the main goals of this thesis is to rapidly prototype a video processing system to adapt different video processing modules. Vivado allows users to quickly build an FPGA-based accelerated video processing system by making connections between IP blocks with minimal RTL modifications. Users can also configure IPs by using the GUI interface instead of making changes to HDL code. After a pass-thorough video system was successfully built, users can insert the video processing module into the design for testing. Vivado also allows users to quickly pack their own IPs. But this work focuses on the creation of video processing modules by using Vivado HLS.

The high level block diagram of the video processing system is shown in Figure 3.3.

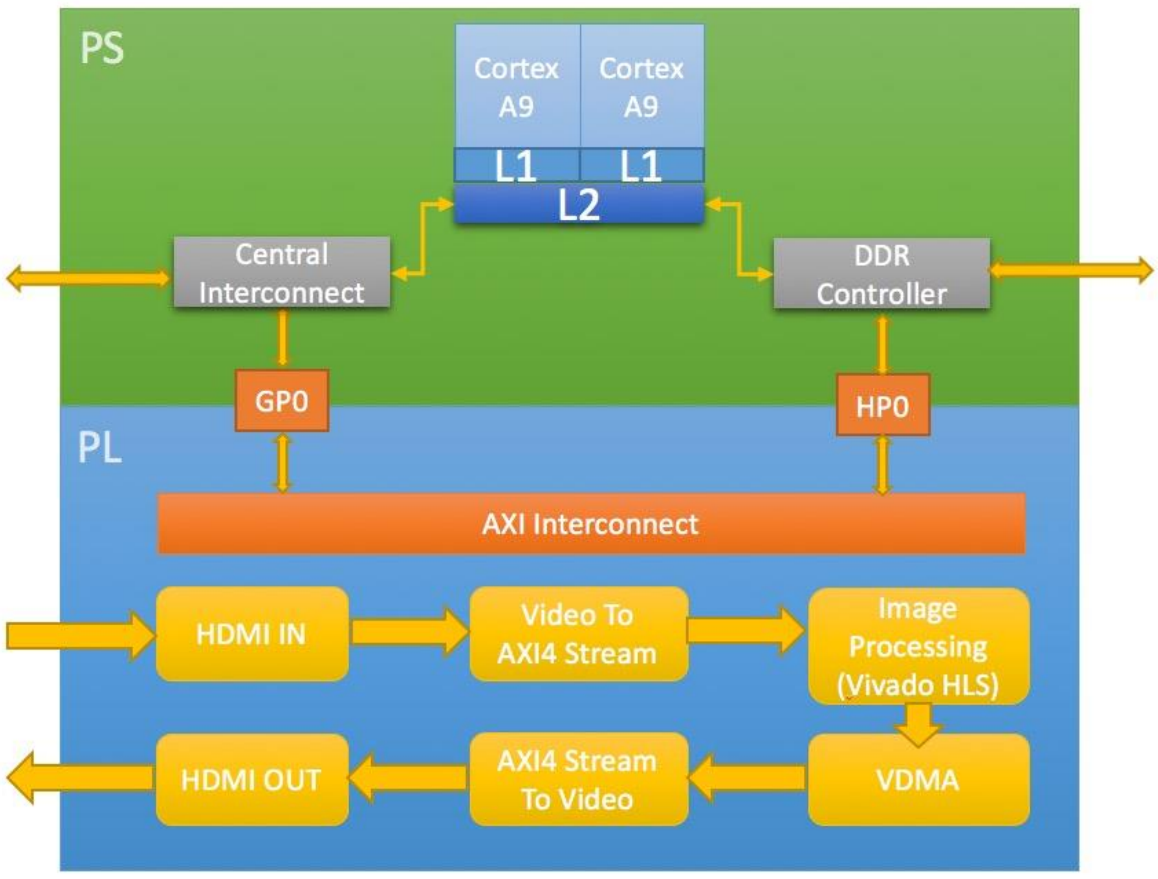


Figure 3.3: Video processing SoC targeting Zynq.

This development flow can save users a great amount of time writing and testing HDL code. By using the ARM processor on the Zynq device, users can access the DDR memory controller and configure all modules with the AXI4-Lite interface. In addition, the actual video processing algorithm can be loaded to the programming logic on the Zynq device for faster parallel computation so that the whole system is more likely to have the real-time video processing capability. The configuration process in SDK requires initializing both VDMA and the video processing IP generated by Vivado HLS. Along with generation of the IP in Vivado HLS, the essential driver files are also generated for initialization and configuration purposes.

The top level block diagram of the proposed video processing system design in Vivado is shown in Figure 3.4.

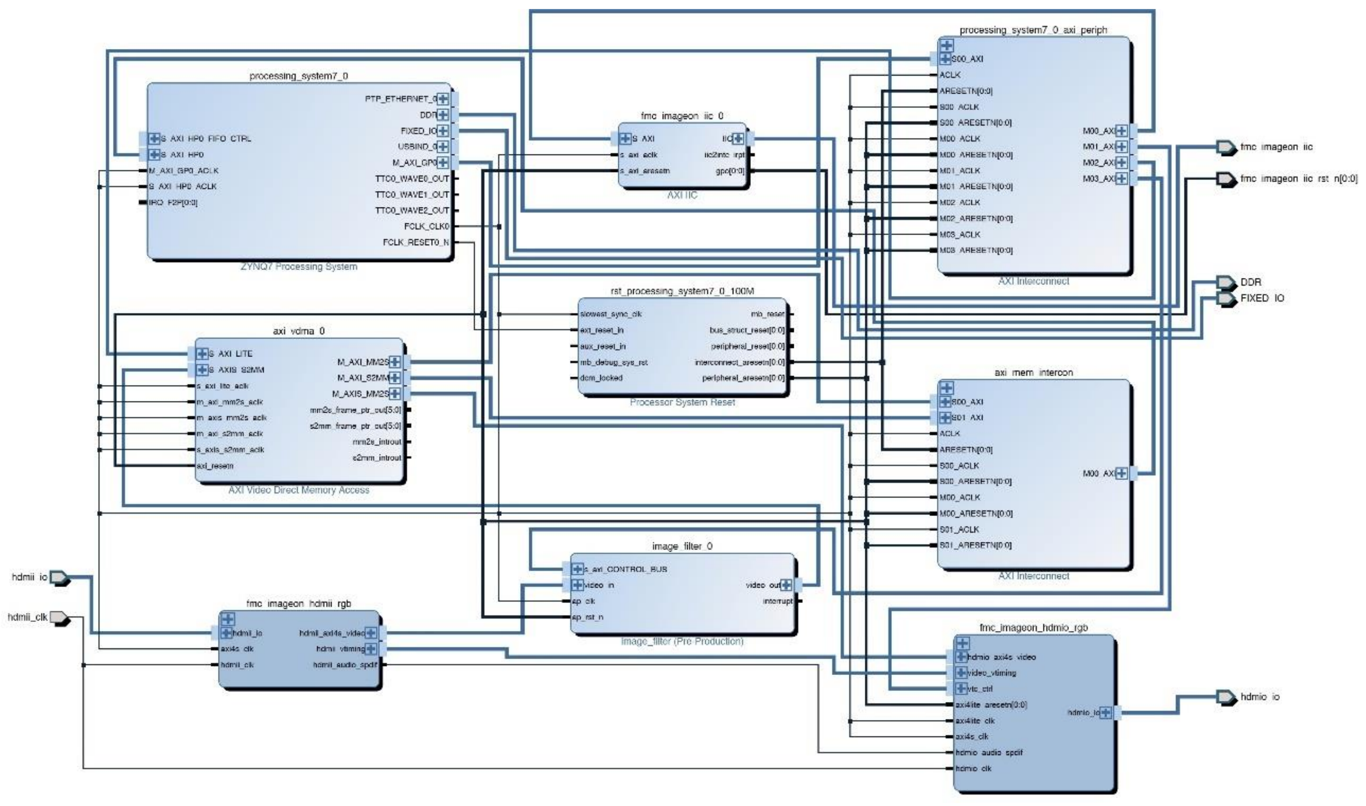


Figure 3.4: Video processing system block design in Vivado.

### 3.2.2 AXI Interface

The AXI bus protocol is used for data transactions between the processing system and programming logic on the Zynq device [21]. The AXI interface allows different blocks to communicate with each other with the same standard, AMBA. The AMBA standard rules the connection between functional blocks, and it allows architecture engineers can effectively prototype SoC designs [22]. It is necessary to understand the protocols behind the AXI4 interface since a wide selection of IPs provided by Xilinx support AXI4 Interface. The most commonly used AXI4 interface includes AXI4-Memory Mapped, AXI4-Lite, and AXI4-Stream. In order to successfully create a video processing system on the Zynq device, it is essential to understand what’s the difference between these three interfaces, and how to use them properly.

AXI4-Memory Mapped interface is the most sophisticated interface of the three. It contains five channels between the AXI slave and AXI master, which are Write Response Channel, Write Data Channel, Write Address Channel, Read Address Channel, and Read Response Channel [21]. Besides address and data, each channel consists a valid and ready signal for the master and slave to behave corresponding to the status of those signals. In order to issue a read/write transaction by the AXI master, the AXI master must send a read/write address along with the data to the AXI slave. By contrast, with AXI4-Stream interface, no read/write address is needed from the AXI master, and the data transfer only goes at one direction for the transaction. In addition, AXI4-Lite interface behaves similarly to the AXI4-Memory Mapped interface, except that the fact it does not support burst of up to 256 data transfer cycles. Therefore, AXI4-Lite interface can be used by the PS to configure hardware parameters, and to access status, control and address registers. AXI4-Lite interface provides the essentials for building software to drive the hardware, as well as the convenience of reconfiguring hardware parameters through software.

### 3.2.3 AXI Interconnect

An AXI4 interconnect is used for connecting various AXI4 masters and AXI4 slaves. It allows slaves to inherit data protocol coming from masters. The cross bars in the center of an AXI4-interconnect can be used for switching and routing streaming data between different slaves and masters. What's more, an AXI4 interconnect can be used to map streaming data into AXI4-memory mapped interface for accessing memory mapped peripherals. An AXI4 Interconnect contains an address decoding table that can be served to the read and write transactions between masters and slaves. Another great feature from the AXI4 interconnect is the flexibility to use clocks from different clock domain. This feature is enabled by integration of an asynchronous FIFO for adaption of different clock domains.

### 3.2.4 AXI VDMA

AXI VDMA is a customized version of the AXI DMA family. On the high level, AXI VDMA is responsible for transmitting data between FPGA and DDR3 memory interface. There are several reasons makes the AXI VDMA an efficient memory interface for a high throughput video processing system design. First, VDMA contains an AXI Data Mover which is capable of AXI Data Mover can convert the data stored in DDR from AXI4-Memory Mapped to AXI4-Stream data type. The video stream coming out from the image processing filter is at AXI4-Stream interface. However, in order to store the image to a shared memory such as the DDR3 SDRAM which the ARM processor can also get access to, it is essential to transfer the AXI4-Stream interface to AXI4-Memory Mapped interface. Likewise, when the data needs to be converted again to the AXI4-Stream data type for video output peripherals such as HDMI and VGA. Figure 3.5 illustrates the data mover implemented in the AXI VDMA module which can handle conversion between AXI4-Stream and AXI4-Memory Mapped Interfaces. MM2S DMA Controller and S2MM Controller are used to issuing control signals to setup the DMA transfer, while the address and number of bytes are store in the Address register and Length register respectively [22].

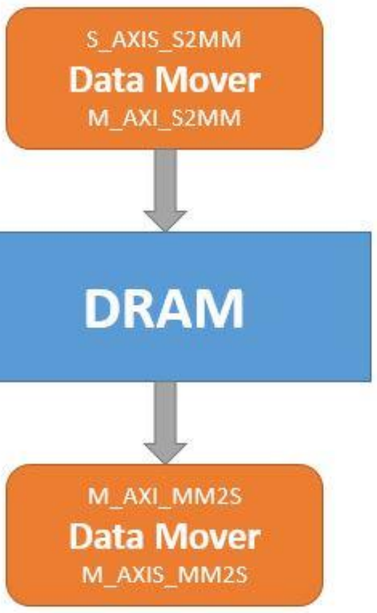


Figure 3.5: AXI VDMA Data Mover.

From the block design's point of view as shown in Figure 3.6, the S_AXIS_S2MM port on the Data Mover reads the AXI4-Stream data, while the M_AXI_S2MM port writes the converted AXI4-Memory Mapped data to external DDR. On the other hand, the M_AXI_MM2S port on the Data Mover reads AXI4-Memory Mapped data from external DDR, and writes the converted AXI4-Stream data to an output peripheral such as HDMI through the M_AXIS_MM2S port.

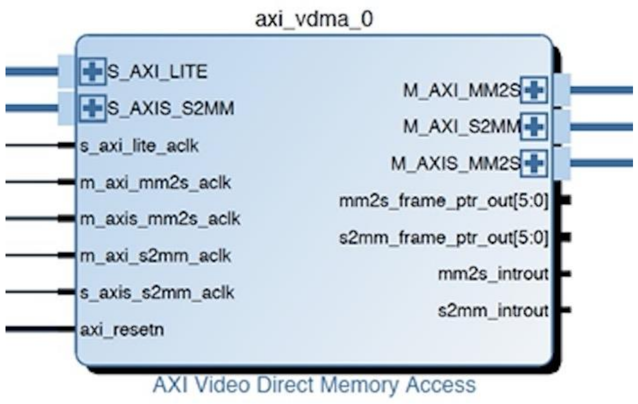


Figure 3.6: AXI VDMA Block in Vivado.

In addition, a frame buffer based video processing system design can be realized with using AXI VDMA. Frame buffers are instantiated in external DDR3, and therefore can be accessed through AXI VDMA. Also, a Genlock mode can be enabled to synchronize the read and write operations in frame buffers so that memory coherence can be effectively prevented. In general, at least two frame buffers need to be allocated when using the Genlock mode. When Genlock mode is enabled, frames can be automatically skipped or repeated if a potential write and read on the same frame at the same time is detected. This can be extremely handy when the incoming video frames are not from the same clock domain. In order to enable Genlock mode, the frame buffer has to be at Park mode instead of circular mode. Also, when using VDMA for high bandwdth memory interface, the VDMA module needs to be connected to the AXI HP ports on the Zynq device.

### 3.2.5 Video Format Conversion

Thanks to the AXI4-Stream interface, the external video signals can be quickly connected to other AXI4 compatible processing blocks such as the video processing IPs generated by Vivado HLS. Therefore, it is necessary to convert the input video stream to AXI4-Stream Interface. According to Analog Devices, the HDMI input device and HDMI output device on the FMC-IMAGEON works in YCbCr 4:2:2 Color Configuration [23]. Computer vision engineers usually prefer to work in YCbCr color space rather than RGB color space because it is more straightforward with only one important channel, Y, the luminance channel. Since human eyes are more sensitive to brightness than color, the Y channel can provide most information a computer vision engineer need for video processing. This work focuses on using the YCbCr color space. Also, since it’s in 4:2:2 format, the video contains 16 bits of data.

In Figure 3.7, it shows the connections being made to allow conversion between the input video data coming from the HDMI input of FMC IMAGEON and AXI4-Stream interface. On the other hand, Figure 3.8 demonstrates the connections for conversion between AXI4-Stream and video out. In the output path, a video timing controller is also connected to the AXI4-Stream to Video Out block so that it can provides timing synchronization signals for a 1080p output video. Users are allowed to quickly configure those IPs in Vivado in order to accommodate for different video formats.

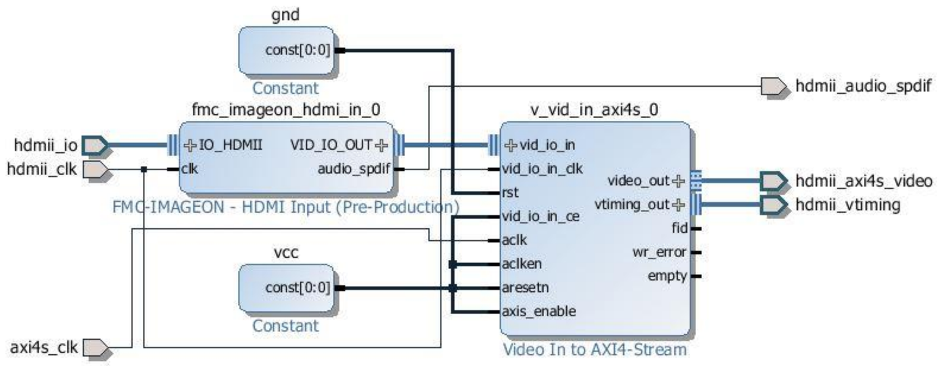


Figure 3.7: Input video conversion in Vivado.

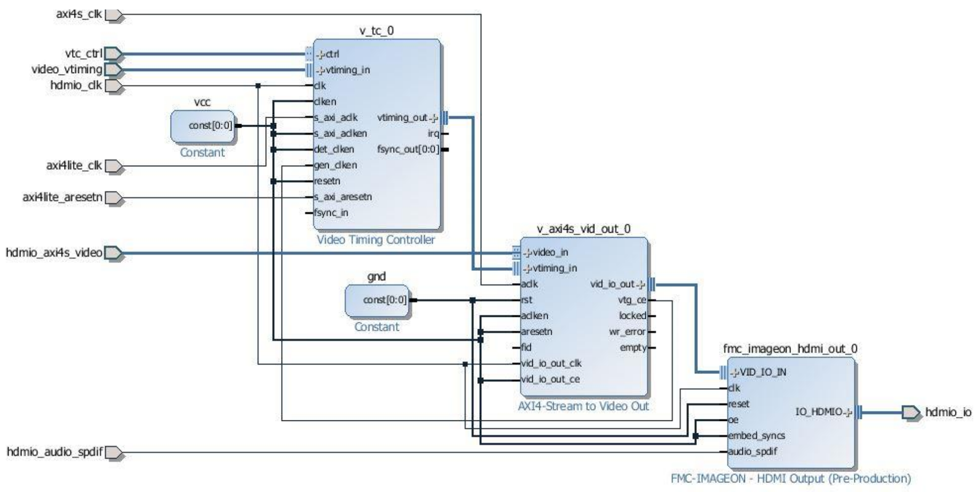


Figure 3.8: Output video conversion in Vivado.

### 3.2.6 AXI VTC

One of the most important components for a video system design is the timing synchronization for each video frame. The timing controller used in design shown in Figure 3.8 is AXI VTC (Video Timing Controller). AXI VTC works in conjunction with the AXI4-Stream to Video Out converter, and it is able to generate essential timing signals for output videos. AXI VTC contains a detector that can be used to detect the horizontal and vertical synchronization signals as well as valid video pixels from an input video frame. Meanwhile, the generator in the AXI VTC can be used to recreate horizontal and vertical synchronization as well as blanking signals that are essential for outputting a video frame. Like many other AXI4 enabled modules, users can access control registers of AXI VTC through AXI4-Lite interface.

## 3.3 Real-Time Video Processing Algorithms Design

The second part of the video processing system development is design of algorithms targeted for the hardware accelerator on Zynq. Most people will think there is not much difference between an image processing algorithm and a video processing system. However, as the resolution and the refreshing rate of a video keep soaring, it becomes very difficult to use a conventional image processing algorithm for video stream processing purposes. The bottleneck from a stream based video processing algorithm usually comes from an inefficient memory management. In order to realize real-time high definition video processing, an efficient sliding window memory architecture is proposed so that it can be applied to a large variety of video processing algorithm designs.

In addition, this section describes the approaches that were used to create two video processing algorithms using Vivado HLS, a Sobel filter, and a Sum of Absolute Difference (SAD) algorithm.

### 3.3.1 Generalized Sliding Window Memory Architecture

The purpose of the proposed sliding window memory architecture is to make it generalized for a variety of video processing algorithms such as image smoothing, image sharpening, edge detection, and even video compression. This memory architecture also contributes to allow stream-data processing with constant fast memory access. The memory architecture involves a line buffer, and a window buffer. One of the greatest advantages of using the line buffer and the window buffer is that normally they only occupy a small amount of memory and they can be synthesized into BRAMs and Flips Flops (FF) on FPGA as shown in Table 3.1, which are notoriously known as the fastest portion of memory that can be accessed in a FPGA. With constantly accessing BRAMs and FFs instead of outside DRAMs can the design result in a lower memory access latency, which will positively contribute to real-time video processing performance.

| **Memory Interface** | **Resource** |
|---|---|
| ***Line Buffer*** | BRAM |
| ***Window Buffer*** | FF |

Table 3.1: Relationship between memroy interface and resource usage.

When using a line buffer in the algorithm design, a fixed number of rows from a video frame are copied into a line buffer. The line buffer keeps updated as the stream data come in, while the window buffer gets updated according to the status of the line buffer. Specifically, the line buffer reads in the stream data one pixel at a time, and horizontally from a video frame as shown in Figure 3.9. The window buffer essentially serves as a sliding window data storage that contains all required data used for the current computation. Before creating any video processing algorithms, it is important to understand the interactions between the line buffer and the window buffer. With proper designs of the interaction between the line buffer and the window buffer, user kernels can access the data stored in the window buffer at every clock cycle for specific computations.

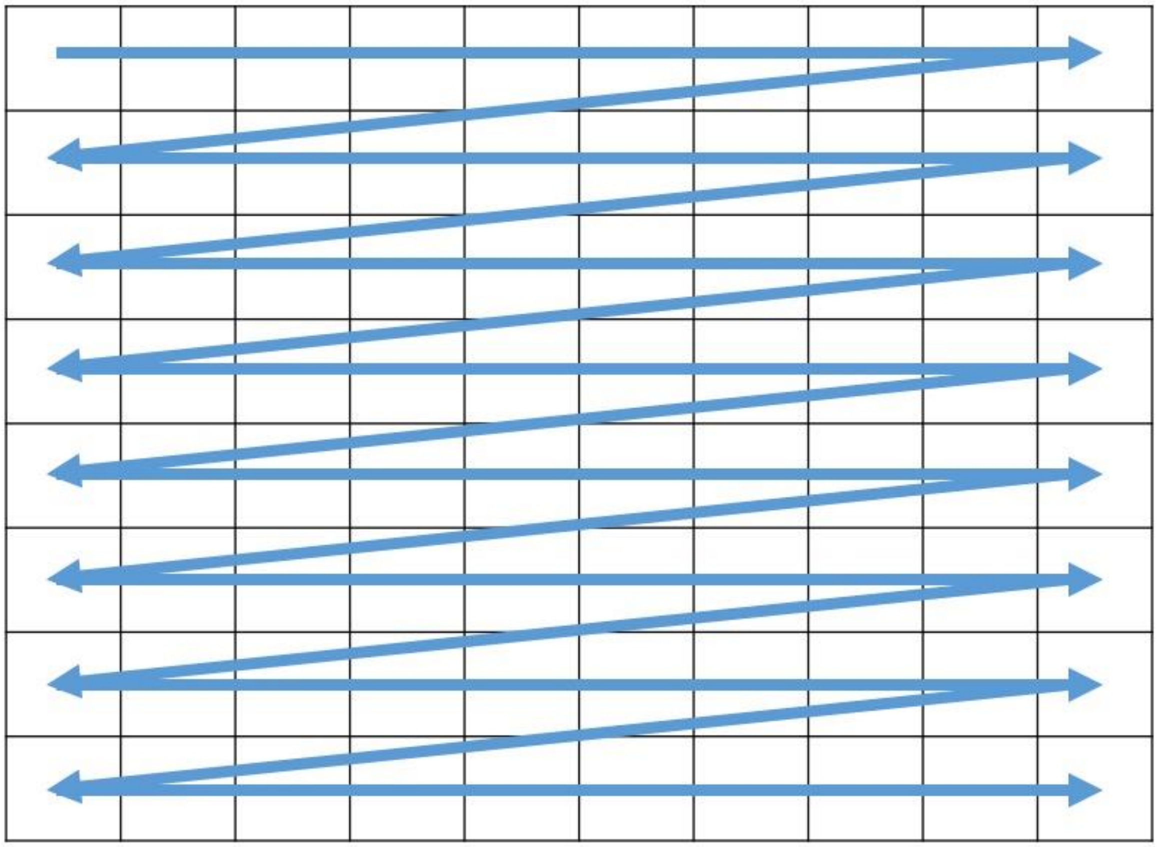


Figure 3.9: Source video stream read order.

The following section introduces an approach for handling the line buffer and the window buffer operations for stream type video processing algorithm designs so that it can bring about efficient memory management and minimum memory access latency. The data being stored in the window buffer are normally used as inputs for a window computation or comparison. For example, in a Sobel filter design, the data contained in the window buffer is convoluted with a fixed Sobel kernel, while in a Median filter design, the data in the window buffer become candidates for a sorting algorithm so that a median value in the window can be computed. No matter which filter is being used, an approach of controlling the interaction between the line buffer and the window buffer can be generalized. Figure 3.10 shows an example of the interaction between the line buffer and the window buffer.

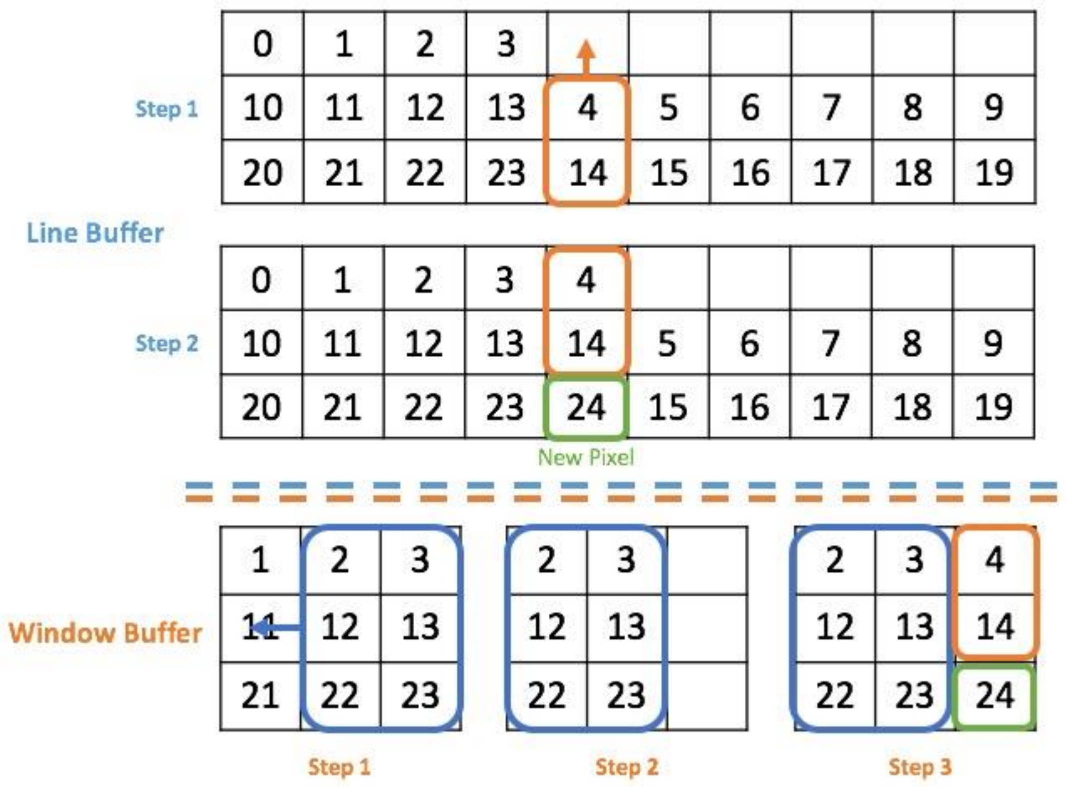


Figure 3.10: Example of the interaction between the line buffer and the window buffer.

In accordance with the example shown in Figure 3.10, the procedures of updating the line buffer and the window buffer are shown in Table 3.2 and Table 3.3. The procedures for updating a line buffer is shown in Table 3.2.

| | |
|---|---|
| **L1** | Declare a *M* x *N* line buffer, where *N* is the number of columns, and *M* is a user defined parameter. |
| **L2** | Shift column ***i*** up. |
| **L3** | Insert the new pixel value at the bottom of column ***i***. |
| **L4** | Enter window buffer operation **W1**. |
| **L5** | Iterates to the next column ***i+1***, repeat from step **L1**. |

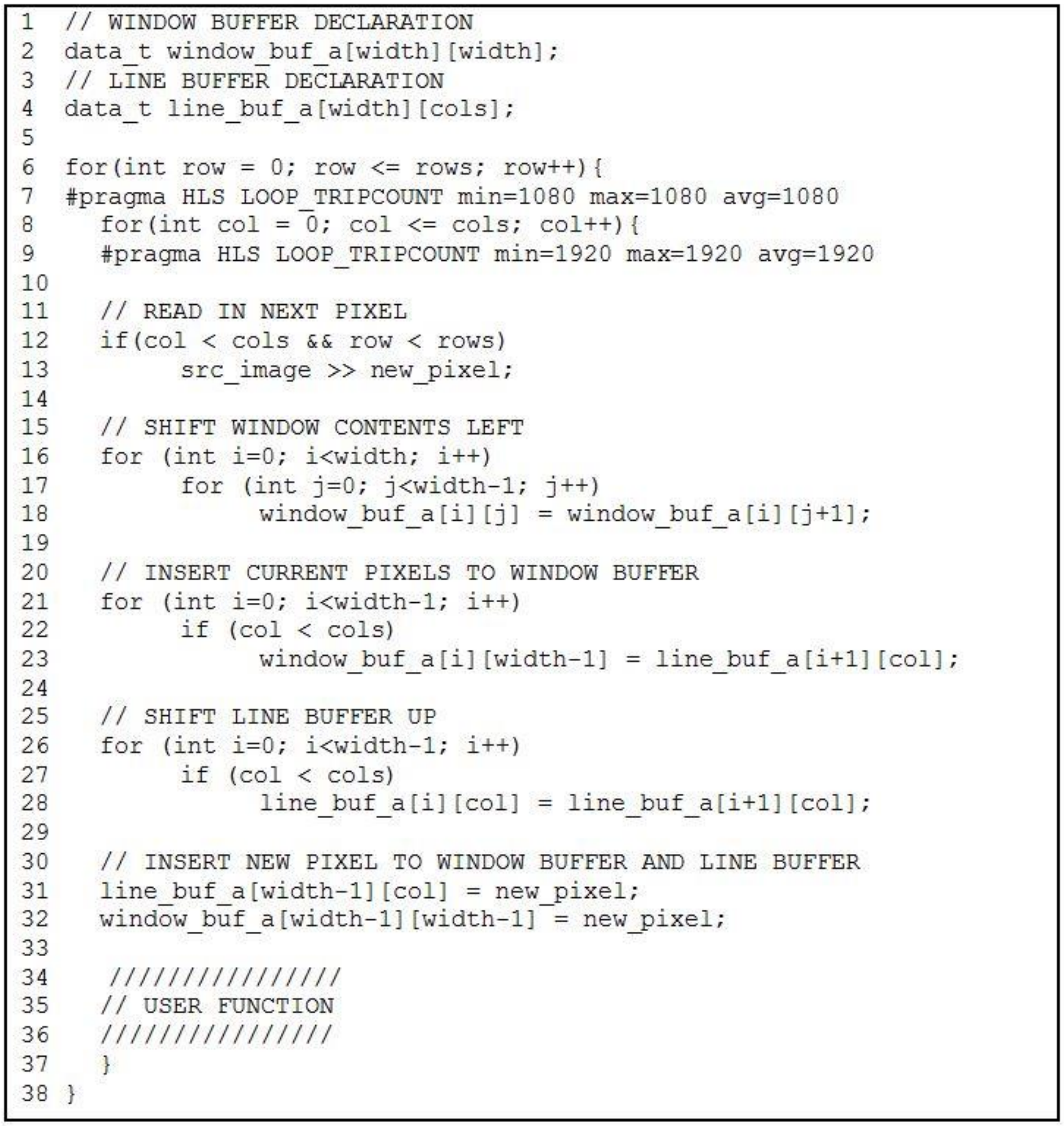
Table 3.2: Line buffer update procedures.

The procedures for updating a window buffer are shown in Table 3.3.

| | |
|---|---|
| **W1** | Declare a *M x M* window buffer, where *M* is a user defined parameter. |
| **W2** | Shift the entire window to the left in order to make room for an empty column on the right. |
| **W3** | Retrieve M-1 pixels from top of column ***i*** in the line buffer and insert them on the top of the rightmost column in the window buffer. |
| **W4** | Retrieve the new pixel from the bottom of column ***i*** in the line buffer and insert it at the bottom of the rightmost column in the window buffer. |
| **W5** | Perform filter computations or comparisons. |
| **W6** | Exit window buffer operation, and enter line buffer operation **L5**. |

Table 3.3: Window buffer update procedures.

Figure 3.11 shows an initial implementation of the line buffer and window buffer interactions for a window size of *width x width*. This version of implementation also allows users to configure the window buffer size and line buffer size simply by altering the *width* variable. With the parameterized module design, users can simply choose the width of the line buffer and the window buffer in order to adapt new algorithm designs. The max *width* value tested in this work is 16. The user function area allows users to insert their specific algorithm that relies on the data in the window buffer. In addition, in order for Vivado HLS to successfully generate a latency estimation report, it is necessary to use *#pragma HLS LOOP_TRIPCOUNT* directive to specify the number of loops that will occur during the actual run. In this work, the max video resolution is 1920 x 1080 pixels.



```
// WINDOW BUFFER DECLARATION
data_t window_buf_a[width][width];
// LINE BUFFER DECLARATION
data_t line_buf_a[width][cols];

for(int row = 0; row <= rows; row++){
#pragma HLS LOOP_TRIPCOUNT min=1080 max=1080 avg=1080
  for(int col = 0; col <= cols; col++){
  #pragma HLS LOOP_TRIPCOUNT min=1920 max=1920 avg=1920

  // READ IN NEXT PIXEL
  if(col < cols && row < rows)
      src_image >> new_pixel;

  // SHIFT WINDOW CONTENTS LEFT
  for (int i=0; i<width; i++)
      for (int j=0; j<width-1; j++)
          window_buf_a[i][j] = window_buf_a[i][j+1];

  // INSERT CURRENT PIXELS TO WINDOW BUFFER
  for (int i=0; i<width-1; i++)
      if (col < cols)
          window_buf_a[i][width-1] = line_buf_a[i+1][col];

  // SHIFT LINE BUFFER UP
  for (int i=0; i<width-1; i++)
      if (col < cols)
          line_buf_a[i][col] = line_buf_a[i+1][col];

  // INSERT NEW PIXEL TO WINDOW BUFFER AND LINE BUFFER
  line_buf_a[width-1][col] = new_pixel;
  window_buf_a[width-1][width-1] = new_pixel;

   ////////////////
  // USER FUNCTION
  ////////////////
  }
}
```

Figure 3.11: Initial implementation of the interaction between the line buffer and the window buffer.

The memory buffer architecture has become an essential component in real-time video processing systems. It is worth noting that several commands to the line buffer and the window buffer are provided by Vivado HLS so that users can incorporate those in an algorithm design. While the tool does not insert the buffers automatically, users need to explicitly define the operations of the line buffer and the window buffer in their algorithm designs. With proper use of the faster BRAMs and

FFs in the design rather than DRAMs, it is highly likely the use of the line buffer and the window buffer can help improve throughput and reduce the latency of the algorithm computation.

In addition, pixels on the border of the input frame must be treated properly in order to prevent memory access violation. Since the convolution is dependent on all eight neighbors of the current pixel, no result can be computed with missing a border pixel. Therefore, my design skips the convolution over the border pixels, and automatically assigns a black pixel value to the border pixels.

### 3.3.2 Sobel Filter

Sobel filter has been widely used in many image processing systems, and it builds up the fundamental for more advanced algorithms in the area of computer vision. Sobel filters are primarily used for edge detections. With a grayscale image, an edge in a given image can be found by identifying the sudden change of pixel intensities.

The prerequisite of using a Sobel filter is that the input image frame needs to be a grayscale image, which has values on a single grayscale channel to represent the intensity of each pixel. This algorithm is developed on a YUV color space instead of conventional RGB color space because the Y channel in a YUV color space essentially represent the luminance of pixels which contains all information needed for a Sobel algorithm.

Sobel filters use convolutions to approximate the derivatives of an image pixel. The main idea of using a Sobel filter is to use two 3 x 3 Sobel kernels in both the horizontal direction (Figure 3.12) and vertical direction (Figure 3.13) for calculating the approximation of gradients, including the magnitude and the orientation of the gradient.

$$\begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline \end{array} * \begin{array}{|c|c|c|}\hline P(i-1,j-1) & P(i-1,j) & P(i-1,j+1) \\ \hline P(i,j-1) & P(i,j) & P(i,j+1) \\ \hline P(i+1,j-1) & P(i+1,j) & P(i+1,j+1) \\ \hline \end{array} = Gx$$

Figure 3.12: Sobel X gradient computation.

$$\begin{array}{|c|c|c|}\hline -1 & -2 & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 2 & 1 \\ \hline \end{array} * \begin{array}{|c|c|c|}\hline P(i-1,j-1) & P(i-1,j) & P(i-1,j+1) \\ \hline P(i,j-1) & P(i,j) & P(i,j+1) \\ \hline P(i+1,j-1) & P(i+1,j) & P(i+1,j+1) \\ \hline \end{array} = Gy$$

Figure 3.13: Sobel Y gradient computation.

Furthermore, the magnitude of total gradient can be approximated by taking the sum of absolute gradient from each direction in order to save calculations as shown in Equation 3.1. One can even approximate the orientation of the total by using the equation shown in Equation 3.2:

$$|G| = |Gx| + |Gy| \tag{3.1}$$

$$\theta = \tan^{-1}\left(\frac{Gy}{Gx}\right) \tag{3.2}$$

As shown in Figure 3.12 and Figure 3.13, when computing the approximation of derivative in both horizontal and vertical directions, each Sobel kernel is convoluted with the pixel values covered by the kernel window, and the magnitude of total gradient is compared with the threshold values for deciding whether the current pixel is an edge pixel.

Zynq is an ideal platform for accelerating Sobel due to its potential high throughput and high memory bandwidth.  The baseline approach is to create a frame buffer which is big enough to receive data from a single 1920x1080 video frame. Then the entire input video frame can be buffered for convolutions over 2,073,600 pixels. This approach can utilize a software Sobel filter implementation with minimum modifications in order to be synthesizable by Vivado HLS.

Although this approach is fairly straightforward to be implemented, during computation it is only possible to use DRAM instead much faster BRAM because of its limited size. Let's assume there is one byte of data contained in a single pixel, an input frame needs 2073.6 Kb of memory space for storing all pixel data, while the available BRAM space on Zynq is less than enough to hold all pixels in from a video frame. Besides the limited size of BRAM, the latency associated with this approach will become excessive compared to what a real-time video processing system requires because there will be no output available until all pixels are iterated twice for Sobel convolution. The second optimized approach utilized the proposed sliding window memory architecture to improve memory access efficiency. In order to apply this sliding window approach, the dimension of the window buffer should be the same as that of the Sobel kernel, which is 3x3. It also indicates that the minimum line buffer depth should hold three lines of data. There are two portions in the restructured code structure. The first portion updates the line buffer while the second portion updates the window buffer accordingly.

Moreover, border treatment is another concern in a sliding window design. In my Sobel design with a 3x3 window buffer, the first result will not be available until the third pixel on the third row is read by the line buffer. This gives rise to a pixel-wide border around the inner frame, and the border pixels are pre-assigned without convolving with the Sobel kernel.

### 3.3.3 Sum of Absolute Difference

Sum of Absolute Difference (SAD) is an area-based matching algorithm that is commonly used for video compression. SAD can also serve to provide some basic feature tracking functionality in a video. For example, it provides a low cost solution for match searching in the stereo vision algorithm when computing the disparity between two images.

Just like the Sobel filter, SAD can take advantage of the sliding window memory architecture in order to achieve real-time processing performance. The fundamental idea behind the SAD algorithm is to compute the sum of absolute difference of the pixel values between a searching window in the target image and a searching template. Instead of having a 3x3 Sobel kernel, SAD usually requires a larger window for improved matching accuracy. The dimensions of the window size are typically 4x4, 8x8 or 16x16. An example of a 4x4 SAD computation is shown in Figure 3.14. For this thesis, a SAD implementation could be accomplished by varying the size of the line buffer and the window buffer, and there is no need to change the line buffer and window buffer operations. Also, instead of performing a convolution between the searching template and the window buffer, SAD only performs matrix subtractions and scalar additions.

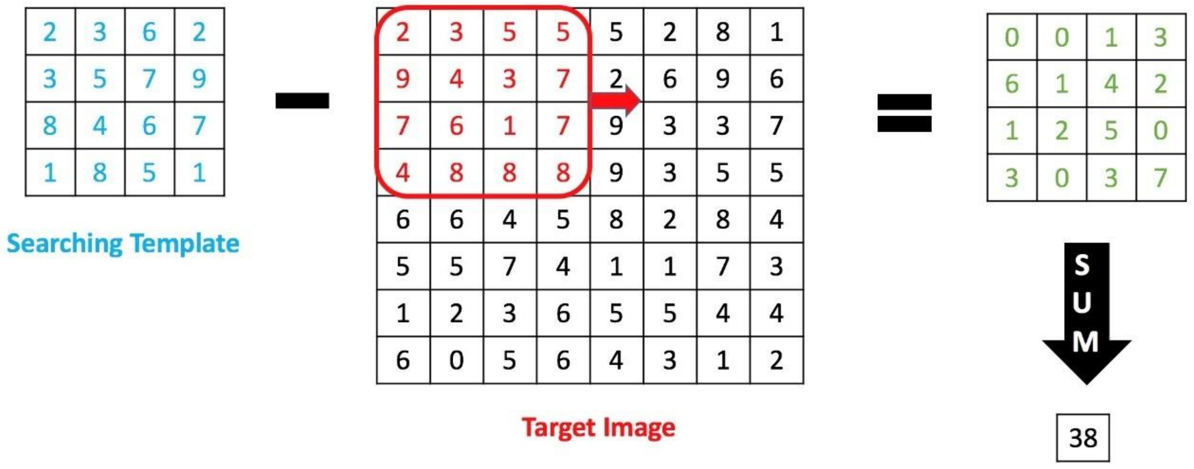
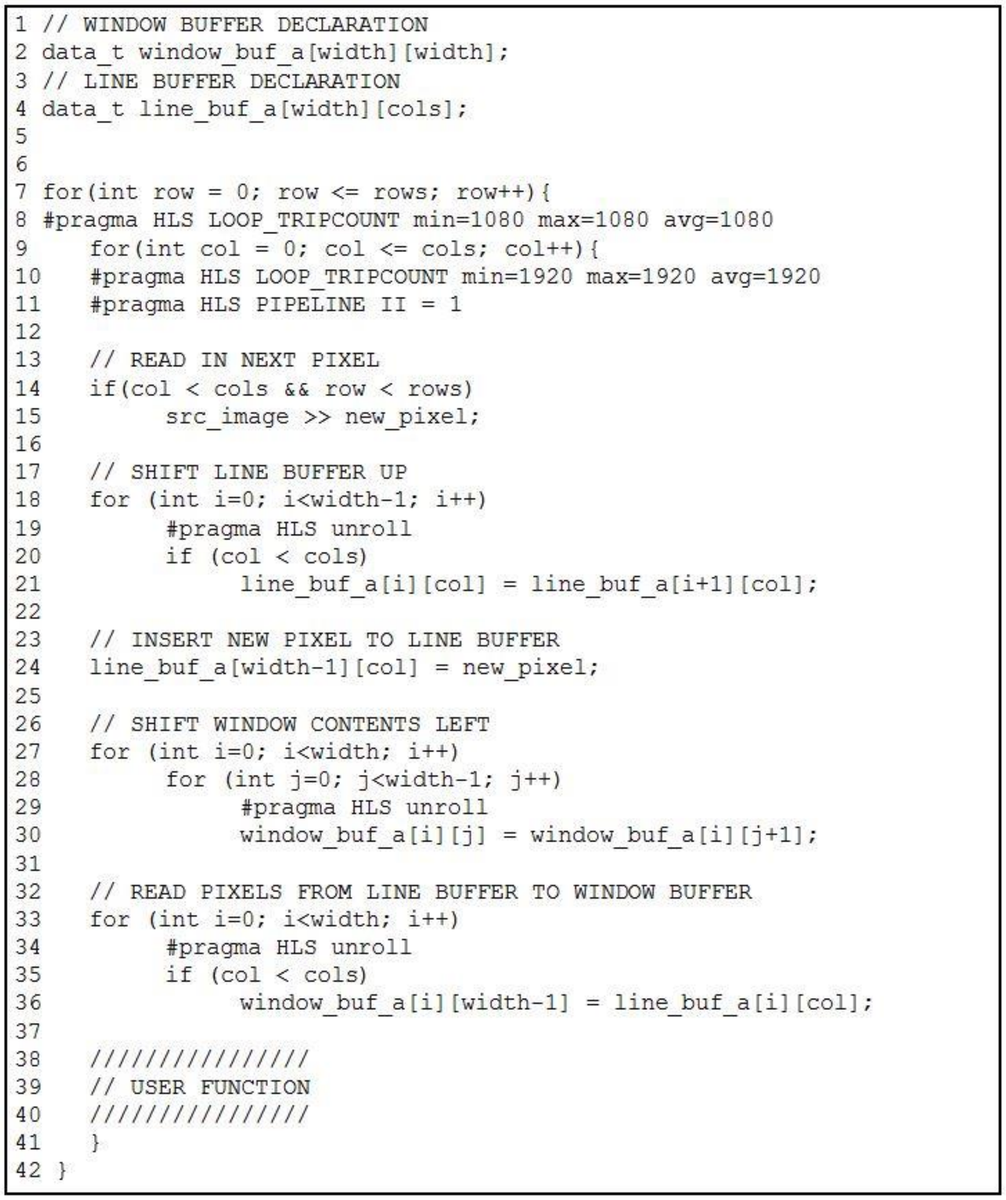


Figure 3.14: Example of a 4x4 SAD computation.

The number of iterations for the SAD design varies as the window size changes. Assume a template window of size *K x K* is slid over an input video frame with *M* rows and *N* columns, the number of row iterations becomes (*M-K+1)*, while the number of column iterations becomes (*N-K+1)*. As a result, a total number of *(M-K+1) x (N-K+1)* of matching results will be computed. Also, the latency is expected to be greater in this design as the window size goes up.

### 3.3.4 Optimizations

Vivado HLS provides the convenience of optimizing the hardware design with insertion of directives in the high level implementations. This work focuses on loop optimziations in order to achieve parallel computations. For instance, the inner loops for computing the Sobel gradients and absolute difference can be fully unrolled because there is no data dependencies between each computation. Unrolling a loop can be enabled by inserting the *#HLS pragma unroll* directive. By using loop unrolling will the tool allocate more resource for multiple operators to achive parallelism. By contrast, the higher level loops can be optimized by using the *#HLS pragma pipeline* directive. Unlike loop unrolling, pipelining will induce operator resource sharing through every iteration of the higher level loop. In addition, Vivado HLS also provides the option to collapse nested loops to a single loop to reduce latency and improve resource sharing. This can be achieved by inserting the *#HLS pragma loop_flatten* directive. Moreover, sequential function calls in a top level function can be automatically pipelined by using the *#HLS pragma dataflow*. It can potentially reduce the latency between execution of functions and improve total throughput if there are multiple function calls in the top level function.



```
// WINDOW BUFFER DECLARATION
data_t window_buf_a[width][width];
// LINE BUFFER DECLARATION
data_t line_buf_a[width][cols];


for(int row = 0; row <= rows; row++){
#pragma HLS LOOP_TRIPCOUNT min=1080 max=1080 avg=1080
    for(int col = 0; col <= cols; col++){
    #pragma HLS LOOP_TRIPCOUNT min=1920 max=1920 avg=1920
    #pragma HLS PIPELINE II = 1

    // READ IN NEXT PIXEL
    if(col < cols && row < rows)
         src_image >> new_pixel;

    // SHIFT LINE BUFFER UP
    for (int i=0; i<width-1; i++)
         #pragma HLS unroll
         if (col < cols)
              line_buf_a[i][col] = line_buf_a[i+1][col];

    // INSERT NEW PIXEL TO LINE BUFFER
    line_buf_a[width-1][col] = new_pixel;

    // SHIFT WINDOW CONTENTS LEFT
    for (int i=0; i<width; i++)
         for (int j=0; j<width-1; j++)
              #pragma HLS unroll
              window_buf_a[i][j] = window_buf_a[i][j+1];

    // READ PIXELS FROM LINE BUFFER TO WINDOW BUFFER
    for (int i=0; i<width; i++)
         #pragma HLS unroll
         if (col < cols)
              window_buf_a[i][width-1] = line_buf_a[i][col];

    ////////////////
    // USER FUNCTION
    ////////////////
    }
}
```

Figure 3.15: Optimized implementation of the interaction between the line buffer and the window buffer.

Figure 3.15 shows an optimized version of the line buffer and window buffer interaction. The main idea behind the optimization is to improve the parallelization level of the design. First, the inner loops are fully unrolled by using the #HLS pragma unroll directive in order to reduce latency. More importantly, the outside loops are parallelized by using the #HLS pragma pipeline directive. Loop pipelining allows the functions from the next iteration to be executed at current iteration along with other functions. Assuming the window dimension is 3x3, and it takes three clock cycles for each line buffer update or each window buffer update. Figure 3.16 illustrates a loop pipelining example that can be done in my optimized implementation. By using loop pipelining, write operations of the next line buffer begin as soon as the window buffer from the last loop iteration finishes its first read operation. In order to realize a high degree of loop pipelining, the read and write operations associated with the line buffer and the loop buffer need to be reordered. In the actual design, the line buffer update and the winodw buffer update can be parallelized to achieve one buffer update per clock cycle rather than taking n clock cycles. Overall, the optimized design with loop pipelining and loop unrolling are expected to improve throughput and to reduce the latency.

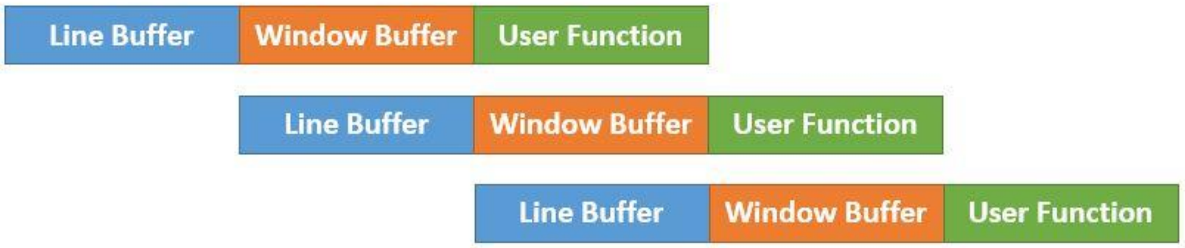


Figure 3.16: An example of loop pipelining.

### 3.3.5 Evaluate Algorithms on ARM

In order to test the performance of Sobel and SAD that I implemented specifically for the ARM processor, I chose the option to cross-compile the two algorithms for the target environment. The first step is to install the ARM GNU Tools for the Zynq device on my desktop, and the installer can be obtained from Xilinx's website. Since my algorithms involves OpenCV functions for image read and write operations, it is necessary to build an OpenCV library for the ARM environment as well. Once the OpenCV library for ARM environment is completed, an image file can be created and is ready to be mounted onto the ramdisk, which is a lightweight and non-persistent file system that can run on ARM.

My approach of measuring the performance on ARM is to measure the execution time for processing an image of a 1920x1080 resolution. The timer start and end clauses are placed around the main portion of the algorithm, and it does not count pre-processing parts such as the conversion from RGB to Grayscale. With regards to the verification of results, since the ramdisk I used is so lightweight that displaying output images is not possible without building and installing QT on the system, I choose to save the result images into the SD card and verify the results on my desktop.

## 3.4 System Level Communication

In order to enable the hardware created in the system, users need to manually write a software driver to interact with the hardware modules using Xilinx Software Development Kit (SDK). The software driver is mainly used to configure three important components, the HDMI interface, the settings for AXI VDMA, and the video processing filter generated from Vivado HLS.

The HDMI port on FMC IMAGEON can be configured through thte IIC interface. Besides that, users need to specify the AXI VTC configuration in the software according to their choice of input and output video formats. When configuring AXI VDMA, it is important to set the mode that is best for target use cases. For my design, both scatter-gather and Genlock are enabled, and a start signal is initiated. Users also need to set the start address of all frame buffers allocated as well as video dimension in order to have successful data transmission between the video stream data and VDMA. Last but not least, the video processing filter needs to be configured according to the register information given in the driver folder generated from Vivado HLS.

## 3.5 Rapid Reconfiguration

Since both Vivado HLS and Vivado would take hours to synthesize, reconfiguring and debugging the video processing system could become extremely time consuming. Although Vivado HLS provides the feature of C/RTL Co-simulation that users can inspect the behavior of the generated code, running the entire system on board is a different story. Therefore, it would be beneficial to enable rapid-reconfiguration to facilitate the design process.

In this thesis, a straightforward and effective approach was applied that can potentially reduce the time spent on reconfiguration process by a large margin. Besides adding physical I/O ports to a RTL design, Vivado HLS also allows users to create ports with AXI4 Slave Lite interface. With AXI4 Slave Lite interface, users can access port addresses in SDK and gain control of them. By taking control of the ports with AXI4 Slave Lite interface can users configure the registers at SDK development phase. Since the SDK development phase comes after both C-synthesis in Vivado HLS and RTL-synthesis in Vivado, users can quickly adjust various register values for configuration and debug wihout waiting for the C-synthesis and RTL-synthesis to finish. This can potentially save users hours of wait for synthesis processes to complete.

For instance, users can bundle the number of rows and columns to the AXI4 Slave Lite interface and control these registers at the SDK development phase according to the resolution of input video stream. Furthermore, users can even configure the parameters used in algorithm computation to achieve different effects. For example, the user can bundle the threshold values to AXI4 Slave

Lite interface and adjust these values in SDK. Users can even adjust the Sobel kernel matrix at SDK development phase in order to realize different functionalities. These configurations can all be taken care of at pre-run time without re-running synthesis design, implementation and routing. By parameterize the C design in Vivado HLS can users re-configure the hardware design more efficiently.

# Chapter 4

# Results and Analysis

This section presents performance, area and power results on the video processing system running on FPGA. Also, a performance comaprison among FPGA, ARM, and a desktop CPU is performed and analyzed. In the following tables, FPGA represents running the video processing algorithm in progammable logic (PL) on the Zynq device, ARM means running the equivelent video processing algorithm in processing system (PS) on the Zynq device, and CPU means running the equivelent video processing algorithm on my desktop PC which is equipped with a 2.5GHz i7-4870HQ quad core processor. The video processing algorithm running on ARM and CPU essentially shares the same implementation with slight modifications in order to execute the kernel on each platform. In addition, the CPU implementation is not optimized for extracting the performance of all four cores in the i7 processor. By contrast, the FPGA implementation has applied memory optimization as well as loop unrollings for performance improvement.

## 4.1 Sobel Filter Performance and Utilization Comparisons

First, my Sobel implementation is compared with works that were done in [1] and [2], as well as a Sobel function retrieved from OpenCV library for Vivado HLS with regards of the performance as shown in Table 4.1. The comparison metrics are Max Clock Frequency and Max Latency. The Max Latency essentially presents the number of clock cycles it takes for outputting the result. Therefore, the Max FPS can be calculated by Equation 4.1:

$$Max\ FPS = \frac{Max\ Clock\ Frequency}{Max\ Latency} \tag{4.1}$$

|  | Max Clock Frequency (MHz) | Max Latency | Max FPS |
|---|---|---|---|
| ***HLS Sobel [2]*** | 110 | 2538290 | 43.34 |
| ***HLS Sobel [1]*** | / | / | 60.00 |
| ***HLS Sobel (OpenCV)*** | 171.5 | 2100572 | 81.64 |
| ***HLS Sobel (Proposed)*** | 156.7 | 2077924 | 75.41 |

Table 4.1: Performance comparison of HLS syntehsized Sobels [1] [2].

Resource utilization is a crucial metric to determine the area usage of an FPGA design. Typically, a lower resource usage stands for lower cost of the design. Therefore, a better design should result in lower resource usage. The FF and LUT utilization comparisons between my proposed design and [1] [2] are shown in Figure 4.1, while the BRAM and DSP utilization comparisons are shown in Figure 4.2.

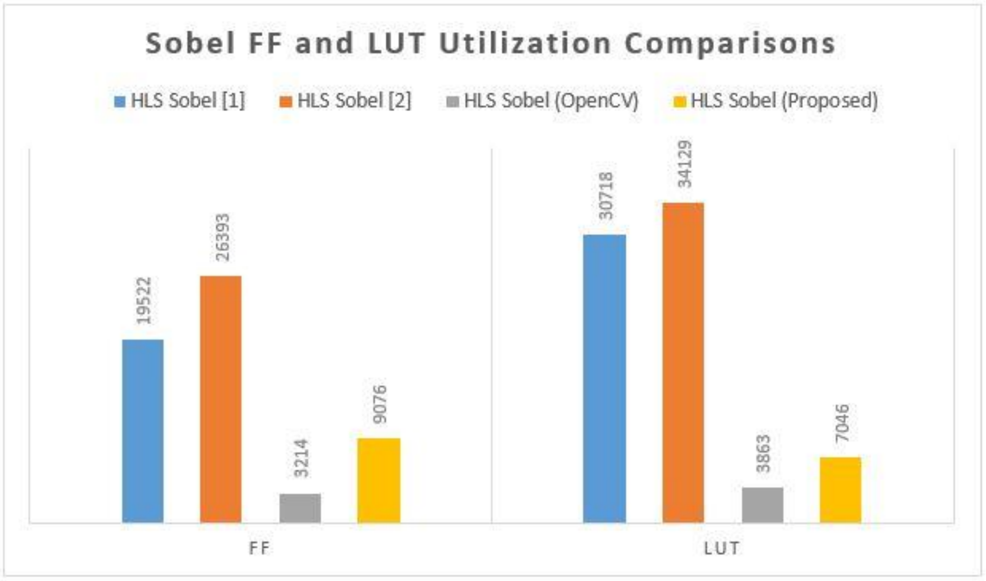


Figure 4.1: FF and LUT utilization comparisons of HLS syntehsized Sobels [1] [2].

Although the OpenCV Sobel function results in the highest performance, the FF and LUT utilization is about 5x greater than that of my Sobel implementation with the optimized sliding window memory architecture. When compared to the work done by Abdelgawad et al., they indicated their implementation is able to achieve 60 FPS. But my proposed design with Sobel function results in a higher 81.64 FPS, which is 36% higher than [1]. Although they also took advantage of using the line buffer and the window buffer, their design is a diretct migration of the TRD reference design provided by Xilinx, which may not be optimized. In addition, my performance result indicates it is about 43% higher than the results seen in [2]. The huge performance difference is likely because of his multiple function calls of the OpenCV Sobel function in his implementation. The payoff of continuous calling OpenCV functions is the tremendous resource usage and performance downgrade regardless of how the functions are pipelined.

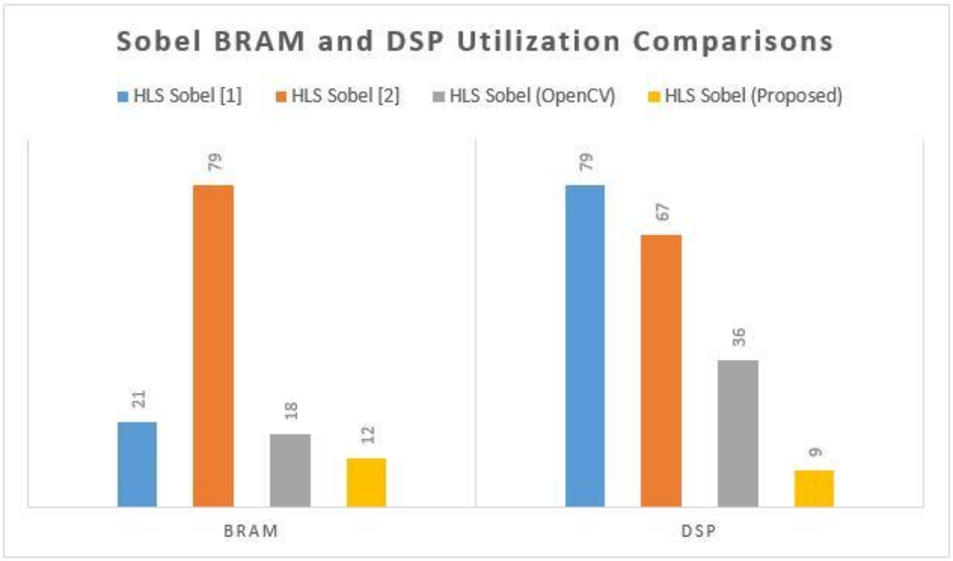


Figure 4.2: BRAM and DSP utilization comparisons of HLS syntehsized Sobels [1] [2].

In addition, the resource utilization comparison results indicate that my design with optimized sliding window memory architecture uses over 50% less FFs, over 3x less LUTs, 2x less BRAMs, and over 6x less DSPs than the lowest resource utilization seen in [1] and [2]. The OpenCV funtion results in 2x lower LUT and over 3x lower FF utilization than the results from my design, but the BRAM and DSP utlization from the OpenCV function exceeds mine significantly.

There is no doubt that my sliding window memory architecture and corresponding implementation method bring about a low latency and low area design of the Sobel filter. Resource sharing is another reason which contributes to the lowest resource usage among the three works [1] [2]. Besides that, the choice of using YUV color space instead of RGB color space also contributes to low resrouce usage and faster performance. Since the YUV422 color space only contains 16 bits of data, and sobel is only dependent on the 8-bit Y channel for computations, very few registers and only a small amount of BRAM space are allocated in order to meet the needs of the design. My design saves the resrouce usage for color space conversion as well while both of the other two designs involve conversions between the color video frames to grayscale video frames. In order to make sure the resource usage estimation is a good representation of the real resource usage, the result is also checked by subtracting the resource utilization of a passthrough design from the resource utilization of a design with the HLS generated filter inserted. The result indicates that the difference in the two resource usage is on par with that of the estimation from Vivado HLS.

## 4.2 SAD Performance and Utilization Comparisons

This sections presents the SAD algorithm's performance and utilization results as well as comparisons against works done in [5] and [6]. Unlike in the previous section, the proposed HLS SAD design is compared against other SAD designs implemented in RTL in order to evaluate the adaptibility of my proposed sliding window memory architectrure and the productivity of Vivado HLS. The performance comparison between HLS synthesized SAD and RTL SADs is shown in Table 4.2. In addition, since BRAM, DSP, and FF utilizations are not provided in [5] and [6], the resource utilization compares only the LUT utilization. All comparisons are based on 16x16 SAD computations. However, one important difference between my design and other two works is that their design only performs one 16x16 window computation, while my design slides the window computation throughout the input video frame of size 1920x1080.

| | Max Clock Frequency (MHz) | Max Latency | Max FPS |
|---|---|---|---|
| ***RTL SAD 16x16 [6]*** | 424.99 | 27 / Window | NA |
| ***RTL SAD 16x16 [5]*** | 197 | 42 / Window | NA |
| ***HLS SAD 16x16 (Proposed)*** | 186.9 | 2073610 | 90.1 |

Table 4.2: Performance comparison between HLS syntehsized SAD and RTL SADs [5] [6].

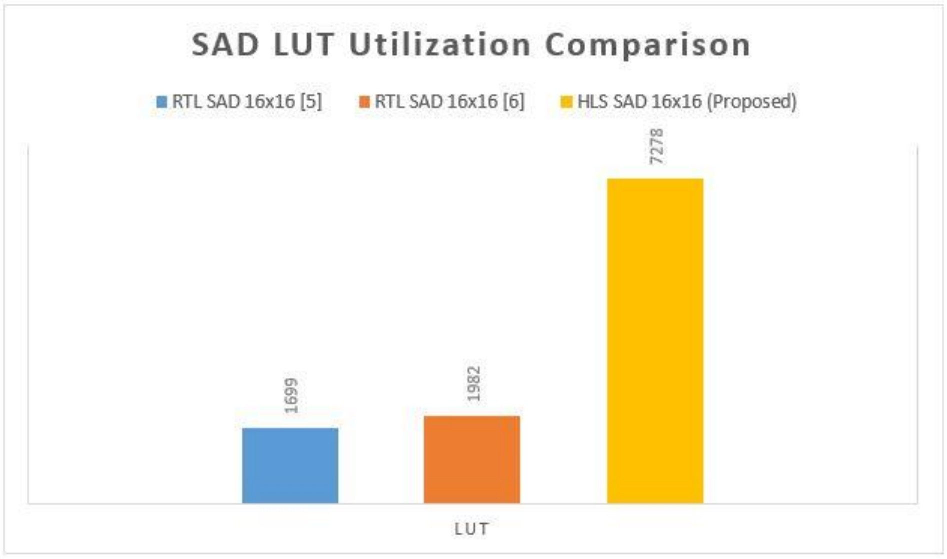


Figure 4.3: LUT utilization comparisons between HLS synthesized SAD and RTL SAD [5] [6].

In [5], the result indicates that their design is able to achieve a max clock frequency of 197 MHz, and they have utilized 1699 LUTs. Their optimized design also indicates a 42 clock cycles latency for a 16x16 window operation. Similarly, in [6], their RTL implementation gives rise to a lower latency of 27 clock cycles for a single 16x16 window operation at a max clock frequency of 424.99 MHz on the Virtex device from Xilinx. Also comes with their design is a repectable LUTs utilization of 1945 LUTs. Although the max clock frequency and max latency indicates very good performance, when approximating the latency for a video frame with a size  of 1920x1080, it results a processing rate at only about 8 FPS while my optimized design results in a much higher 90.1 FPS. With regard to my SAD implementation, it is optimized for real-time processing performance by taking advantage of the sliding window memory architecture. When comparing the LUT utilization between my design and the other two RTL designs, the result shows my design uses about 3.5x more LUTs than the one with higher LUT utilization. Moreover, my design also allocates additional BRAM and FF utilizations, which are not mentioned in [5] and [6]. But considering the fact that my design brings about a video processing rate of 90.1 FPS, the area tradeoff for realizing the performance is not too bad. With a larger window size, my design results in a higher clock rate than the Sobel implementation but with a similar latency.

## 4.3 ARM, x86 and FPGA Performance Comparison with Sobel

Table 4.3 presents the performance comparison of Sobel filters running on the Cortex A9 ARM processors on Zynq, the Intel i7-4850HQ processor, and the FPGA accelerator on Zynq. Specifically, the Sobel filter running on FPGA is synthesized by Vivado HLS with my optimized sliding window memory architecture, while the implementations on ARM and x86 are serial versions with no optiizations. Consequently, the Sobel filter running on ARM processor only results in 1.45 FPS, while running on the i7 CPU gives rise to a fair 9.2 FPS. When moving the sobel kernel to the FPGA accelerator on Zynq, it results a respectable 75.41 FPS.

|  | **Max FPS** | **Speedup Factor** |
|---|---|---|
| ***ARM*** | 1.45 | 1x |
| ***X86*** | 9.20 | 6.3x |
| ***FPGA*** | 75.41 | 52x |

Table 4.3: Performance comparison of Sobel running on Zynq's ARM, Intel i7 CPU, and Zynq's FPGA.

## 4.4 Power Efficiency Analysis

Power efficiency is another very important metric to determine whether the platform is suitable for a certain prototype. Especially when there is a tight power budget for certain applications, engineers should pay greater attention on the power consumption to make sure it meets the goals. This section includes an analysis of the power consumption between two platforms, the Zynq platform and an Intel i7 CPU.

For a FPGA design, power analysis should be conducted during the design phase in order to to meet the power goal and cooling system requirements. Vivado integrates the power estimation tool that can be used for power estimation and thermal analysis. Table 4.3 presents the total on-chip power consumptions as well as the dynamic power consumptions for the video processing system with Sobel and SAD algorithms.

|  | **Total On-Chip Power (W)** | **Dynamic Power (W)** |
|---|---|---|
| ***Sobel*** | 1.852 | 1.687 |
| ***SAD*** | 1.811 | 1.648 |

Table 4.4: FPGA Power Estimation of HLS synthesized Sobel and SAD.

With regards to the power consumption on the i7 CPU, it is difficult to measure with existing tools. Nevertheless, the 2.5GHz i7-4850HQ processor has a TDP (Thermal Design Power) rating at 47W [24]. Although sometimes CPU can reach at a lower power consumption with light loads, TDP still provides an approximate representation of power consumption with taking additional power consumption from other peripherals such as memory and hard drive into considerations.

As the figure indicates that the embedded FPGA based video processing system consumes significantly less power than the Intel i7 quad-core processor. When running the Video Processing System with either video processing algorithm on the Zynq SoC system, the total on-chip power consumption comes at around 1.85 W. To compare this result with the TDP of the i7-4850HQ processor, the Zynq SoC system results  in 26x less power consumption.This result suggests that with proper design and implementations, choosing Zynq platform can result in low-power solutions for real-time video processing systems. Furthermore, the power consumption on FPGA can be further reduced by various optimizations. One of the most straightforward optimizations that can be done is to use two clock domains one for image processing and streaming, and the other one for the AXI-Lite interface. One can use a fast clock for the AXI4-Stream interface and video processing system, and use a slow clock for other interfaces such as the AXI4-Lite interface.

## 4.5 Rapid-Reconfiguration Discussion

Rapid-reconfiguration proved to reduce the reconfiguration and debug process from hours to minutes. It effectively allows reconfiguration of various parameters such as video frame dimensions, kernel paremeters, threshould values, and many other parameters at run-time. Without providing the design with the ability for reconfiguration, users need to wait for the completion of two synthesis processes in Vivado HLS and Vivado. The C synthesis process normally takes between 15-45 minutes, and sometimes the synthesis process never ends without any error indication. In addition, the FPGA synthesis process takes between 20-50 minutes depending on the design complexity. Users can even make intermediate variables configurable such that they can configure those for debugging purpose. By taking advantage of the parameterized module design can users improve productivity significantly.

# Chapter 5

# Conclusion

## 5.1 Summary

This thesis proposed a rapid prototyping flow for high performance video processing system which is consisted of three major steps: 1) Video processing system architecture design using high performance AXI4 interface, 2) Video processing system algorithm design using Vivado HLS and 3) System level communication. This flow allows engineers with limited HDL experience to prototype a domain specific system design within weeks. It substantially enhances hardware development productivity as well as lowers the boundary between software and hardware development. However, Vivado HLS is still far from being a fully-automatic HDL generation tool for non-domain experts. Users still need a decent amount of hardware knowledge as well as domain-specific knowledge in order to successfully create algorithm designs using Vivado HLS.

To address the issue, a generalized sliding window memory architecture is deduced. The purpose of the doing that is to derive a memory framework that can be adopted to a series of video processing algorithms in order to improve the productivity of using Vivado HLS as well as to improve performance. Also, this memory architecture can be configured to have different window dimensions in order to meet the needs for different algorithms. Consequently, my HLS Sobel and SAD designs result in up to 75% less resource utilization and over 30% higher performance than [1][2][5][6]. Furthermore, the proposed FPGA-based video processing SoC outperforms both the ARM only platform and x86 platform by achieving a maximum of 90.1 FPS for 1080P video processing. In addition, the overall power consumption of the FPGA-based video processing SoC design can be over 26x less than that of an active i7 CPU from Intel. It shows the rapid prototyping flow can produce a low-power and high performance video processing system on Zynq SoC.

## 5.2 Future Work

With the proposed memory architecture, users can realize a series of video processing algorithms such as Sobel filter, SAD and Median filter by integration of the user functions. The research could be further extended to discovery of advanced computer vision algorithm designs using high-level synthesis tools such as Vivado HLS. Since learning how to use a high-level synthesis tool can cost as much as the cost of learning a new programming language, it is important to derive generalized algorithm frameworks and effective memory architecture in order to make developments of domain specific applications more productive.

# Bibliography

[1] H. M. Abdelgawas, M. Safar, A. M. Wahba, “High Level Synthesis of Canny Edge Detection Algorithm on Zynq Platform,” *International Journal of Computer, Electrical, Automation, Control and Information Engineering Vol:9, No:1*, 2015, pp. 148-152

[2] H. Johansson (2015). Evaluation Vivado High-Level Synthesis on OpenCV Functions for the Zynq-7000 FPGA.

[3] J. Monson, M. Wirthlin and B. L. Hutchings, "Implementing high-performance, low-power FPGA-based optical flow accelerators in C," *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, Washington, DC, 2013, pp. 363-369.

[4] J. Monson, M. Wirthlin and B. L. Hutchings, "Optimization techniques for a high level synthesis implementation of the Sobel filter," *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*, Cancun, 2013, pp. 1-6.

[5] S. Wong, S. Vassiliadis and S. Cotofana, “A sum of absolute differences implementation in FPGA hardware,” *Euromicro Conference, 2002. Proceedings. 28th*, 2002, pp. 183-188.

[6] J. Olivares, J. Hormigo, J. Villalba, I. Benavides, “Minimum sum of absolute differences implementation in a single FPGA device,” *14th International Conference, FPL 2004, Leuven, Belgium,* 2004, pp. 986-990.

[7] T. Han, G. W. Liu, H. Cai and B. Wang, "The face detection and location system based on Zynq," *Fuzzy Systems and Knowledge Discovery (FSKD), 2014 11th International Conference on*, Xiamen, 2014, pp. 835-839.

[8] K. Asanovic, R. Bodik, K. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, K. Telick, "A View of the Parallel Computing Landscape", *Communications of the ACM*, Vol. 52 No. 10, P56-67.

[9] Qualcomm. What's So Special About the Digital Signal Processor? Available: https://www.qualcomm.com/news/onq/2013/12/06/whats-so-special-about-digital-signal-processor [Online; accessed 13 Mar 2016].

[10] Oak Ridge National Laboratory. Accelerated Computing Guide. Available: https://www.olcf.ornl.gov/support/system-user-guides/accelerated-computing-guide/#3056 [Online; accessed 13 Mar 2016].

[11] PGI Insider. The AGI Accelerator Programming Model on NVIDIA GPUs. Available: https://www.pgroup.com/lit/articles/insider/v1n1a1.htm [Online; accessed 13 Mar 2016].

[12] Kuon, R. Tessier, and J. Rose, "FPGA Architecture: Survey and Challenges," *Foundations and Trends in Electronic Design Automation*, Vol. 2, No. 2 (20087) pp. 135-253.

[13] VLSI-WORLD. FPGA Design Flow. Available: http://www.vlsi-world.com/content/view/28/47/1/4/ [Online; accessed 15 Mar 2016].

[14] N. Calazans, E. Moreno, F. Hessel, V. Rosa, F. Moraes, and E. Carara. "From VHDL register transfer level to SystemC transaction level modeling: A comparative case study", Proceedings of the 16th Symposium on Integrated Circuits and Systems Design, p 335, 2003.

[15] L. Adams, "Choosing the Right Architecture for Real-Time Signal Processing Designs." *Texas Instruments*. 2002 Nov. Available: http://www.ti.com/lit/wp/spra879/spra879.pdf.

[16] Xilinx. Zynq-7000 All Programmable SoC Overview. Available: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf [Online; accessed 2 Apr 2016].

[17] AVNET. FMC-IMAGEON HDMI Display Controller Tutorial. Available: http://www.em.avnet.com [Online; accessed 4 Feb 2016]

[18] G. Martin and G. Smith. High-level synthesis: Past, present, and future. Design

Test of Computers, IEEE, 26(4):18 25, 2009.

[19] Xilinx, Bluespec. Available: http://www.xilinx.com/products/design_tools/logic_design/advanced/esl/bluespec.htm [Online; accessed 16 Mar 2016]

[20] Xilinx. Vivado Design Suite User Guide, High-Level Synthesis. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf [Online; accessed 19 Feb 2016]

[21] Xilinx. AXI Reference Guide. Available: http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf [Online; accessed 2 Mar 2016]

[22] Xilinx. A Zynq Accelerator for Floating Point Matrix Multiplication Designed with Vivado HLS. Available: http://www.xilinx.com/support/documentation/application_notes/xapp1170-zynq-hls.pdf [Online; accessed 9 Mar 2016]

[23] Analog Devices. ADV7511 Low-Power HDMI Transmitter with Audio Return Channel, Hardware User's Guide. Available: [Online; accessed 21 Feb 2016]

[24] Intel. Intel Core i7-4850HQ Processor. Available: http://ark.intel.com/products/76086/Intel-Core-i7-4850HQ-Processor-6M-Cache-up-to-3_50-GHz [Online; accessed 11 Apr 2016]

[25] Xilinx. Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries. Available: http://www.xilinx.com/support/documentation/application_notes/xapp1167.pdf [Online; accessed 25 Mar 2016]

[26] Avnet. Zedboard Product Brief. Available: http://products.avnet.com/opasdata/d120001/medias/docus/1/Xilinx_ZedBoard-pb.pdf [Online; accessed 20 Feb 2016]

# Appendix A

## Image Filter Implementation in Vivado HLS

Listing A.1: image_filter.h

```
#ifndef _TOP_H_
#define _TOP_H_

#include "hls_video.h"

#define MAX_WIDTH  1920
#define MAX_HEIGHT 1080

typedef hls::stream<ap_axiu<16,1,1,1> >               AXI_STREAM;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC2>     IMAGE_16;
typedef hls::Scalar<2, unsigned char>                 PIXEL_16;
typedef unsigned char data_t;

void image_filter(AXI_STREAM& INPUT_STREAM, AXI_STREAM& OUTPUT_STREAM, int rows, int cols);

#endif
```

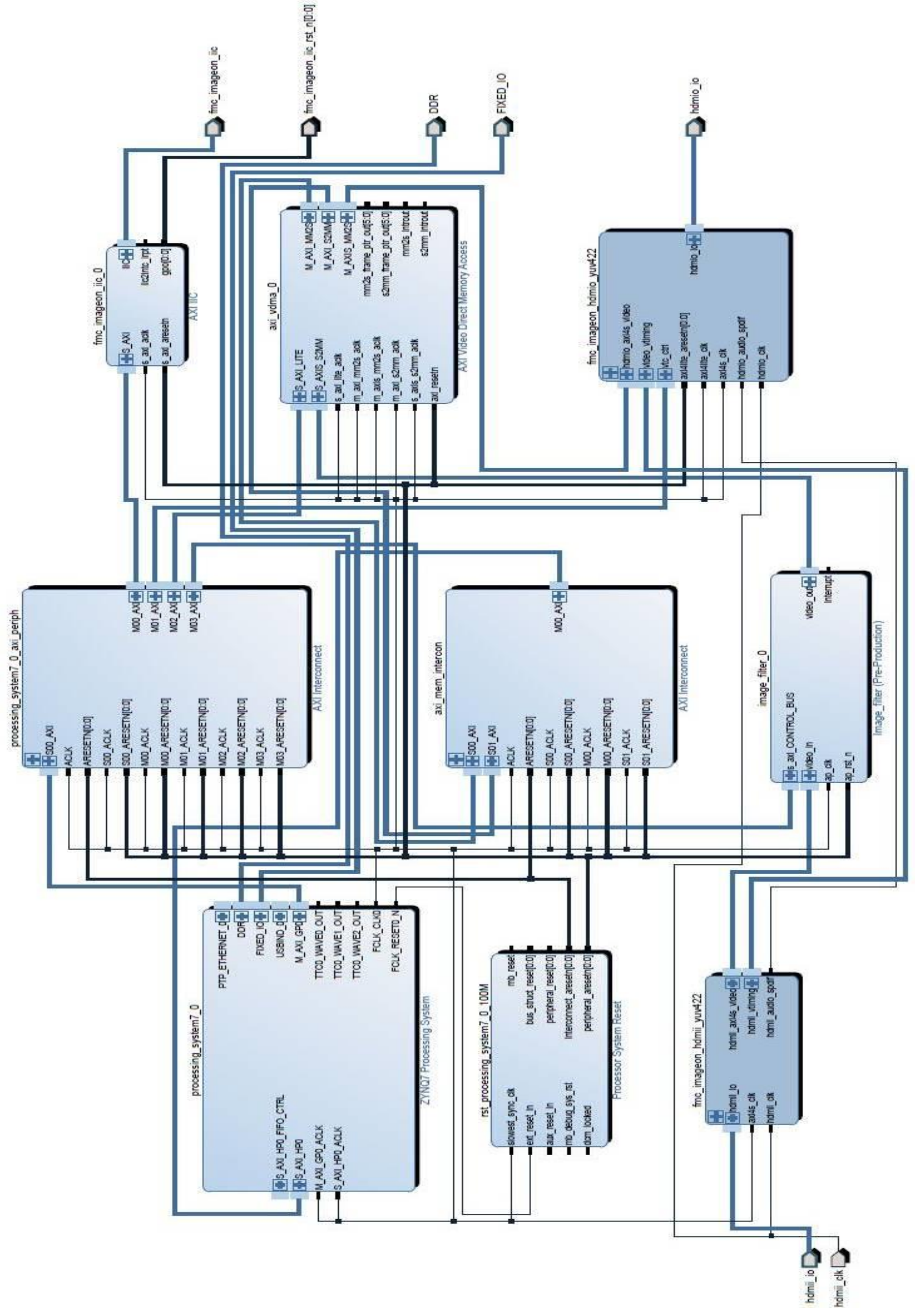
## Listing A.2: image_filter.cpp

```
#include "image_filter.h"
#define WIDTH 3
#define H_THRESH 225
#define L_THRESH 175

data_t usr_filter(data_t *window[WIDTH*WIDTH])
{
   short x_weight = 0;
   short y_weight = 0;
   short new_pixel_weight;
   data_t new_pixel_val;

   const char sobel_x[3][3] ={{-1,0,1},{-2,0,2},{-1,0,1}};
   const char sobel_y[3][3] = {{1,2,1},{0,0,0},{-1,-2,-1}};

   Convolution_Loop_ROW: for(int i=0; i < 3; i++){
           Convolution_Loop_COL: for (int j=0; j<3; j++){
           #pragma HLS UNROLL
                   x_weight = x_weight + (window[i][j] * sobel_x[i][j]);
                   y_weight = y_weight + (window[i][j] * sobel_y[i][j]);
           }
   }

   x_weight = ((x_weight>0)? x_weight : -x_weight);
   y_weight = ((y_weight>0)? y_weight : -y_weight);
   new_pixel_weight = x_weight + y_weight;
   new_pixel_val = ((new_pixel_weight < 255) ? (255-(unsigned char)(new_pixel_weight)) : 0);

   if(new_pixel_val > H_THRESH)
           new_pixel_val = 255;
   else if(new_pixel_val < L_THRESH)
           new_pixel_val = 0;
   else
           new_pixel_val = 0;

   return new_pixel_val;
}

void image_filter(IMAGE_16& src, IMAGE_16& dst, int rows, int cols)
{
   PIXEL_16 pixel_in;
   PIXEL_16 pixel_out;
   data_t new_pixel;
   // WINDOW BUFFER DECLARATION
   data_t window_buf_a[WIDTH][WIDTH];
   // LINE BUFFER DECLARATION
   data_t line_buf_a[WIDTH][cols];

   for(int row = 0; row <= rows; row++){
   #pragma HLS LOOP_TRIPCOUNT min=1080 max=1080 avg=1080
           for(int col = 0; col <= cols; col++){
           #pragma HLS LOOP_TRIPCOUNT min=1920 max=1920 avg=1920
           #pragma HLS loop_flatten
           #pragma HLS dependence variable=&buff_A false
           #pragma HLS PIPELINE II = 1

```

```
                    // READ IN NEXT PIXEL
                    if(col < cols && row < rows)
                            src_image >> pixel_in;
                            new_pixel = pixel_in.val[0];

                    // SHIFT LINE BUFFER UP
                    for (int i=0; i<WIDTH-1; i++)
                    #pragma HLS UNROLL
                            if (col < cols)
                                    line_buf_a[i][col] = line_buf_a[i+1][col];

                    // INSERT NEW PIXEL TO LINE BUFFER
                    line_buf_a[WIDTH-1][col] = new_pixel;

                    // SHIFT WINDOW CONTENTS LEFT
                    for (int i=0; i<WIDTH; i++)
                            for (int j=0; j<WIDTH-1; j++)
                            #pragma HLS UNROLL
                                    window_buf_a[i][j] = window_buf_a[i][j+1];

                    // READ PIXELS FROM LINE BUFFER TO WINDOW BUFFER
                    for (int i=0; i<WIDTH; i++)
                    #pragma HLS UNROLL
                            if (col < cols)
                            window_buf_a[i][WIDTH-1] = line_buf_a[i][col];

                    ////////////////
                    // USER FUNCTION
                    if((row < (WIDTH-1)) || (col < (WIDTH-1)) ||
                            (row >= (rows-(WIDTH-1))) || (col >= (cols-(WIDTH-1)))){
                            new_pixel = 0;
                    }
                    else
                            new_pixel = usr_filter(&window_buf_a);
                    ////////////////
                    pixel_out.val[0] = new_pixel;
                    pixel_out.val[1] = 0;

                    if(row > 0 && col > 0)
                            dst << pixel_out;
            }
    }
}

image_filter(AXI_STREAM& video_in, AXI_STREAM& video_out, int rows, int cols)
{
#pragma HLS INTERFACE axis port=video_in bundle=INPUT_STREAM
#pragma HLS INTERFACE axis port=video_out bundle=OUTPUT_STREAM
#pragma HLS INTERFACE s_axilite port=rows bundle=CONTROL_BUS offset=0x14
#pragma HLS INTERFACE s_axilite port=cols bundle=CONTROL_BUS offset=0x1c
#pragma HLS INTERFACE s_axilite port=return bundle=CONTROL_BUS
    IMAGE_16 img_0(rows, cols);
    IMAGE_16 img_1(rows, cols);
#pragma HLS dataflow
    hls::AXIvideo2Mat(video_in, img_0);
    image_filter(img_0, img_1, rows, cols);
    hls::Mat2AXIvideo(img_1, video_out);
}
```

# Appendix B

## Video Processing System Top Level Block Design in Vivado



# Appendix C

## Video Processing System Configuration in SDK

Listing C.1: video_config.c

```c
#include <stdio.h>
#include "platform.h"
#include "fmc_imageon_hdmi_passthrough.h"
#include "ximage_filter.h"

void print(const char *str);
fmc_imageon_hdmi_passthrough_t demo;
XImage_filter filter;

int main()
{
    init_platform();

    demo.uBaseAddr_IIC_FmcImageon = XPAR_FMC_IMAGEON_IIC_0_BASEADDR;
    demo.uBaseAddr_VTC_Axi4sTiming = XPAR_FMC_IMAGEON_HDMIO_YUV422_V_TC_0_BASEADDR;

    filter.Control_bus_BaseAddress = 0x43C10000;
    filter.IsReady = XIL_COMPONENT_IS_READY;

    XImage_filter_EnableAutoRestart(&filter);
    XImage_filter_Set_rows(&filter, 1080);
    XImage_filter_Set_cols(&filter, 1920);

    XImage_filter_InterruptEnable(&filter, 0x 0);
    XImage_filter_InterruptGlobalEnable(&filter);
    XImage_filter_Start(&filter);

    while(!XImage_filter_IsDone(&filter))
     {
            printf("Waiting......\n");
        sleep(1);
     }
      printf("Done!\n");

    // VDMA Configuration
    Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR + 0x00, 0x008B);
     Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR + 0x5C, 0x01);
     Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR + 0x60, 0x02);
     Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR + 0x64, 0x03);
     Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR + 0x58, 0x1680);     // 1920x3
     Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR + 0x54, 0x1680);     // 1920x3
     Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR + 0x50, 0x0438);     // 1080

     Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR + 0x30, 0x108B);
     Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR + 0xAC, 0x01);
     Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR + 0xB0, 0x02);
     Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR + 0xB4, 0x03);
     Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR + 0xA8, 0x1680);     // 1920x3
     Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR + 0xA4, 0x1680);     // 1920x3
     Xil_Out32(XPAR_AXI_VDMA_0_BASEADDR + 0xA0, 0x0438);     // 1080

    fmc_imageon_hdmi_passthrough_init(&demo);
    cleanup_platform();
    return 0;
}
```