UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Comparative Study between RTL and HLS for Image Processing Applications with FPGAs

A Thesis submitted in partial satisfaction of the requirements
for the degree Master of Science

in

Computer Science (Computer Engineering)

by

Muhsin Gurel

2016

The Thesis of Muhsin Gurel is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2016

DEDICATION

To my family and friends.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to take the opportunity to thank some of the people and organizations that have helped me get to where I am now.

First, I would like to thank my advisor Prof. Ryan Kastner for his guidance during my time as a Master's student at the University of California, San Diego. I also want to thank my thesis committee members, Prof. Ali Irturk and Prof. Steven Swanson for their time and valuable feedback.

I want to especially thank Azeem Ghumman, Janarbek Matai, Dustin Richmond, Dajung Lee and, Alireza Khodamoradi for their support during my research.

In particular, I wish to thank Alireza Khodamoradi for being an awesome friend, colleague, and housemate for the last three years.

Special gratitude to Turkish Petroleum for supporting my studies at the University of California, San Diego.

Last but not least, I would like to express my deepest gratitude to my parents, Fatma and Irfan Gurel, and my brothers Abdurrahim and Abdurrahman Gurel for their unconditional love and support.

ABSTRACT OF THE THESIS

A Comparative Study between RTL and HLS for Image Processing Applications with FPGAs

by

Muhsin Gurel

Master of Science in Computer Science (Computer Engineering)

University of California, San Diego, 2016

Professor Ryan Kastner, Chair

High-level synthesis (HLS) and register transfer level (RTL) are two popular methods to design FPGAs. Both of methods have advantages and disadvantages: HLS provides ease of development, and is less prone to error. Thus, it reduces the development time significantly when designing a hardware system. On the other hand, RTL development enables the developer to make lower level design decisions which can increase the performance and efficiency of the system. Although there have been several studies on comparing HLS and RTL in terms of cost, performance, and area for higher level applications, these studies do not make a one-on-one comparison on the micro-architectural details of a given application. In this thesis, we take further steps to compare HLS and RTL by analyzing the lower level micro-architectural details in HLS and RTL designs for the same applications. We also provide guidelines on how to create these architectures efficiently in both methods. We investigate several image processing algorithms and their designs in RTL and HLS. The algorithms were selected based on their ubiquitous nature and widespread use in image processing applications. We show that if HLS is used to its full potential, it can achieve comparable level of performance as the systems designed using RTL.

Introduction

Technological advancements in the semiconductor industry enable developers to design more complex circuits. As the design complexity grows, the need for higher levels of abstraction grows with it. Higher levels of abstraction allow developers to focus on the system behavior rather than details of the circuit. High-level synthesis is becoming ubiquitous in the digital design industry, similar to the spread of the logic synthesis in the past two decades. A programming language-like environment makes high-level synthesis more attractive compared to the RTL design[11, 14].

With the availability of HLS tools, the barriers between the software developers and the hardware developers began to lower. Hardware acceleration is currently in high demand, and it is likely to maintain its popularity in the future. Many developers from different fields use hardware acceleration, and image processing is one of the primary fields where FPGA acceleration is used.

This thesis aims to compare the implementation of some of the most well-known image processing applications using RTL and HLS methods. We explore the advantages and disadvantages of using HLS or RTL for such applications, and compare two methods based on resource utilization and performance, as well as the development process. We also provide a guideline for optimization techniques for each algorithm in both HLS and RTL to achieve efficient hardware implementations.

We present the rest of this thesis in six chapters:

In Chapter 1, we will discuss the abstraction levels and trade-offs of HLS and

RTL methods. Then we introduce Verilog as the hardware description language (HDL) used for the RTL approach, and Vivado HLS as the tool used for the HLS approach.

Chapter 2 presents the characteristics of image processing applications, as well as different interfaces and integration frameworks. Also, we give a quick overview of the hardware platforms used in our experiments.

Chapters 3, 4 and 5 focus on the detailed description of the design process, optimization and implementation of each algorithm that is studied in HLS and RTL. These chapters also present comparisons between using HLS and RTL for each image processing algorithm in terms of resource, performance, and development process.

Lastly, Chapter 6 presents the summary of the thesis.

Chapter 1

Hardware Development Methods

1.1 Abstraction Levels: Hardware Description Languages and High-Level Synthesis

In early days of digital electronics, engineers used to design integrated circuits (ICs) by doing optimizations with Karnaugh maps, drawing gate-level schematics and transferring them into transistor logic. These principles are still the same, however they are not done manually anymore. The necessity of speeding up this process emerged as the complexity of ICs increased. Also, as the silicon technology advanced, transistor sizes became smaller and smaller. This enabled IC designers to fit more transistors into their circuits, and circuits became too complicated to be designed by hand. Later, electronic design automation tools and hardware description languages were introduced to solve this problem [15].

As the systems grew bigger, people needed higher levels of abstraction to design their hardware. First, developers switched to gate level description from designing ICs by hand. Then, register transfer level description was introduced to as a new design abstraction to describe synchronous digital circuits. Later, logic synthesis increased the abstraction level once again, and a programming-language-like control flow (if/else, case, while, for, etc.) became synthesizable in HDLs.

When logic synthesis tools were first introduced, it took a while for design community to adopt it. Perfecting the synthesis process and optimizing the libraries took years. Logic synthesis tools are currently accepted as one of the most efficient ways to design digital systems. Using pure structural modeling to develop large systems is almost impossible due to the complexity of the modern designs.

Although hardware description languages like Verilog and VHDL are still popular in the digital design process, many will agree that they are not the best way to design electronic circuits. The verbose nature of these languages requires a good amount of expertise to use them efficiently. They are very prone to error, and usually, it is much harder to debug the code when compared to the programming languages. Also, developers still has to go through some of the minor design decisions when using HDLs.

High development cost, long time-to-market, and inefficient design process are some of the main motivations in raising the abstraction level. A programming language like environment is desired for a greater degree of abstraction compared to the HDLs. One of the purposes of the HLS is to isolate the developer from the low-level design decisions such as creating control signals, explicitly declaring port directions and widths, declaring intermediate wires and register, etc. On top of this, HLS tools are expected to generate an efficient hardware. Figure 1.1 demonstrates the abstraction levels in digital design.

Research on developing efficient HLS tools started decades ago and with the introduction of the Field Programmable Gate Arrays (FPGAs) the effort on creating HLS tools increased. The reconfigurable nature of the FPGAs makes them a preferable platform for many developers in different fields and development tools are playing a significant role in their platform decisions. A hardware platform with a high-level design environment will more likely to be selected for development purposes. We can see that the popularity of FPGAs are increasing each day and HLS tools are one of the major

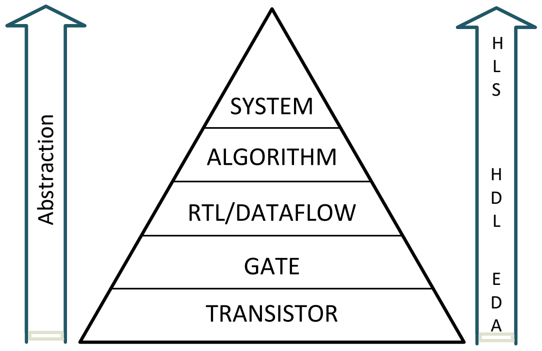


Figure 1.1. Algorithm level is the highest abstraction provided with hardware description languages. High-level synthesis can provide system level abstraction.

factors in this increasing popularity.

Improvements in the levels of abstraction increased the popularity of the FPGAs as well. Nowadays, hardware acceleration with FPGAs is commonly used in scientific and commercial applications. Compared to the hardware acceleration with Application Specic Integrated Circuit (ASICs), FPGAs offer low development costs and flexibility. Also, the design process of the FPGAs is getting less complex with the new tools provided by the major FPGA vendors each year.

In the next section, we will introduce Verilog as the HDL language used in this study. We choose Verilog over VHDL because it is more commonly used both in the industry and in academia while VHDL is more adopted in aviation and defense industry.

1.2 Verilog

With advancements in Very Large Scale Integration(VLSI) field, the complexity of the integrated circuits increased rapidly. The amount of transistors used in a single chip increased from order of hundreds to order of thousands and testing these circuits with breadboards became impossible due to their sizes. In order to verify the functionality of these large designs, logic simulators came into use. However, there wasn't any standard language on how to describe circuits for logic simulations.

Verilog and VHDL languages are introduced in the early 80s to describe hardware for simulation and verification purposes. In the late 80s, logic synthesis enabled the developers to directly design digital circuits by using these languages on RTL level and with the logic synthesis support the popularity of HDLs increased [11].

The Verilog language evolved during the last 30 years and lastly standardized as IEEE Standard 1364-2005. Currently, Verilog is a hardware description language which offers multiple layers of abstraction such as *gate level modeling*, *dataflow modeling*, and *behavioral modeling*.

Gate level modeling is used for a design method which refers to list of gates and their connections without any behavioral description of the module. Listing 1.1 shows a basic multiplexer example described in *gate level modeling* in Verilog.

```
module mux (OUT,A,B,SEL)

input A;
input B;
input SEL;

output OUT;

wire wA;
wire wB;

and (wA, A, !SEL);
and (wB, B, SEL);
or (OUT, wA, wB);

endmodule
```

Listing 1.1. Multiplexer designed using *gate level modeling*. The design is explicitly described gate by gate like a low-level schematic.

The term *dataflow model* is used for designs which are created by defining its output as a function of its inputs with continuous *assign* statements. Generally, the term *RTL model* is used for any synthesizable behavioral code, which contains *always* and *assign* statements. Listing 1.2 shows a basic multiplexer example described in *dataflow modeling* in Verilog.

```
module mux (OUT,A,B,SEL)

input A;
input B;
input SEL;

output OUT;

assign OUT = (!SEL&A) | (SEL&B);

endmodule
```

Listing 1.2. Multiplexer designed using *dataflow modeling*. The design output is described as a function of its inputs. Note that we can also use a ternary operator for this design.

A behavioral level code is used to describe the circuit behavior as it can be understood from its name. *Behavioral modeling* may be used just for simulation purposes since the code doesn't have to be synthesizable. In fact, what is synthesizable and what is not, is defined by the synthesis tool, not by the language. For example, the initial block is not synthesizable in some of the tools, while it is synthesizable and used to initialize registers states in Xilinx FPGAs when it is used with Xilinx Synthesis Technology(XST). Listing 1.3 shows a basic multiplexer example described in *dataflow modeling* in Verilog.

```
module mux (OUT,A,B,SEL)

input A;
input B;
input SEL;

output reg OUT;

always @ *
begin
    if (SEL == 0)
        OUT = A;
    else
        OUT = B;
end
endmodule
```

Listing 1.3. Multiplexer designed using *behavioral modeling*. The module behavior is described, whereas, which gates to be used or the output function is not specified.

Even if we look at the simple code examples above, we will notice that no matter how high the abstraction level is used in Verilog, the verbose nature of the language doesn't allow the developer to have a design experience similar in high-level programming languages. The developer is still in charge of explicitly declaring control signals, clocks, wires, registers, module ports, assignment types, etc. Basically, the developer still needs to focus on the hardware structure rather than the overall system behavior.

The verbose nature of the language also makes Verilog more prone to error compared to high-level languages. The developer must write Verilog code as explicit as possible to make it understandable. Although the language is around for more than 30 years, there is no strong consensus on the coding style. The namings style of the modules, ports, wires, registers, as well as the use of the code structures, shows great variety between developers. One of the factors behind this issue is that the software community is larger and more open while the hardware community is more industry driven and privatized. This cause less source and information circulation in the community.

Using HDL languages efficiently requires a significant amount of expertise and hardware knowledge. For an inexperienced developer, Verilog can be full of traps. Besides from generating a synthesizable code, the developer should also be aware the kind of hardware synthesized at the end.

One of the most common mistakes made in Verilog is the wrong usage of the blocking and non-blocking assignments.The code shown in Listing 1.4 and 1.5 looks very similar from a programmer point of view, but they will result in different architectures and the hardware will generate different results.

```
module blck (IN,CLK,OUT1,OUT2)

input IN;
input CLK;

output reg OUT1;
output reg OUT2;

always @ (posedge CLK) begin
    OUT1 = IN;
    OUT2 = OUT1;
end
endmodule
```

Listing 1.4. A Verilog code with blocking assignments.

```
module non-blck (IN,CLK,OUT1,OUT2)

input IN;
input CLK;

output reg OUT1;
output reg OUT2;

always @ (posedge CLK) begin
    OUT1 <= IN;
    OUT2 <= OUT1;
end
endmodule
```

Listing 1.5. A Verilog code with non-blocking assignments.

Blocking statements in Verilog are evaluated in-order and the next assignment might dependent on the previous evaluation. On the other hand, non-blocking statements are scheduled to happen on exactly at the same time. Many inexperienced RTL designers may confuse this behavior with hardware parallelism; one can think that since non-blocking statements are scheduled to be evaluated at the same time, it will create a parallel hardware. However, the hardware parallelism and Verilog event parallelism are two different concepts, and it should not be confused. Figure 1.2 and Figure 1.3 shows the RTL synthesis results for the two different codes in Listing 1.4 and Listing 1.5. The synthesis result for the code with the blocking assignment shows a hardware which has two registers connected in parallel while the code with the non-blocking statement has two registers connected in series.

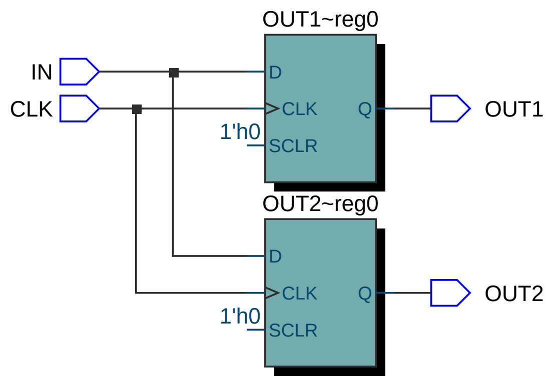


Figure 1.2. The synthesis result of the code with blocking assignments shown in Listing 1.4.

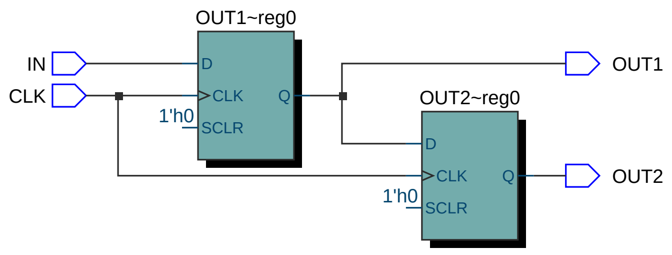


Figure 1.3. The synthesis result of the code with non-blocking assignments shown in Listing 1.5.

Listings 1.6 and 1.7 present another example of how a small change in the code can drastically affect the hardware. Here ”the developer” intention is to select between 3 inputs and transfer one of them to the output. It sounds like an easy task; a 4to1 multiplexer should give us the expected behavior. In order to select between three different inputs we at least need 2-bit wide select input. 2-bit select input will give us four different possibilities; however, we only have three inputs to select and one select combination will not be used. In this case, we can say that the developer doesnt care about the input combination which is not covered in the code shown in Listing 1.6.

```
module mux(OUT,SEL,A,B,C);

input A,B,C;
input [1:0] SEL;

output reg OUT;

always @ (*)
case (SEL)
    0: OUT = A;
    1: OUT = B;
    2: OUT = C;
endcase
endmodule
```

Listing 1.6. A Verilog code with a case statement. It is expected to generate a 4to1 multiplexer. But instead, it generates two multiplexers and one latch.

```
module mux(OUT,SEL,A,B,C);

input A,B,C;
input [1:0] SEL;

output reg OUT;

always @ (*)
case (SEL)
    0: OUT = A;
    1: OUT = B;
    2: OUT = C;
    default: OUT = 1'b0;
```

```
endcase
endmodule
```

Listing 1.7. Same code in Listing 1.6 except this time it has a default state. This code generates 4to1 multiplexer as desired.

Ambiguity is one of the things that should be avoided when designing hardware especially when it is designed with HDLs. It is a good practice to always declare default cases when using case statements especially if the goal is to create a combinational circuit. Figures 1.4 and 1.5 show the difference in resulting hardware when there is an ambiguity in the code.

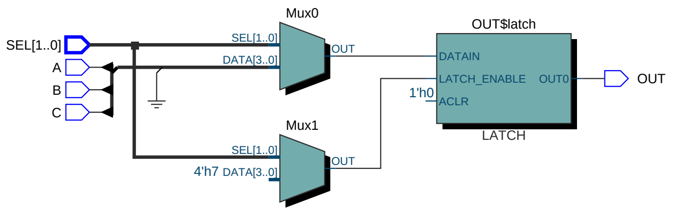


Figure 1.4. Because of the ambiguity in the code in Listing 1.6, synthesis tool generates a hardware with a latch. To create a combinational hardware we should avoid using a latch.

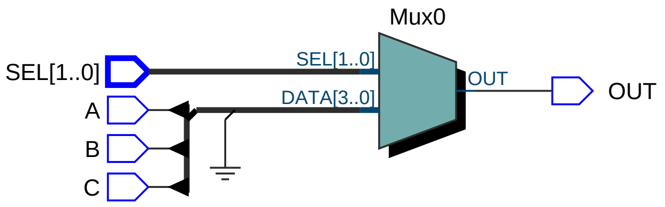


Figure 1.5. The hardware generated when the ambiguity is removed from the code in Listing 1.7.

Verilog is a powerful hardware description language which allows the developer to design the system on a very low level if desired. The flexibility and the low-level design approach of the language can potentially result in better performance. However, the low-level design approach also results in a verbose language and, in consequence, the design process becomes more prone to errors.

In the next section, we will introduce Vivado HLS as the HLS tool used in this study. Currently, Vivado HLS is one of the most popular HLS tools in the market. It allows developers to work on a high-level abstraction and explore the design space easily.

1.3 Vivado HLS

Today, many hardware developers will agree that HDLs which are originally created for simulation and verification purposes are not the best method to design hardware. Yet they are still popular since there was no other strong candidate until recent years. High-Level Synthesis tools introduced in the last few years emerged as a strong alternative to designing hardware with HDLs. The adoption of the high-level synthesis increasing day by day as the vendors offer more powerful and optimized HLS tools. Currently, some of the popular HLS tools include Vivado HLS [13], Catapult High Level Synthesis [5], Synopsys Synphony C Compiler [12], LegUp [4].

An HLS tool transforms the specifications described in a high-level language such as C/C++, Scala, Haskell or MATLAB into a synthesizable register transfer level. HLS tools can be simply described as a bridge which connects the software domain to the hardware domain [17]. The typical flow of the high-level synthesis is shown in Figure 1.6.

Vivado HLS is one of the most popular HLS tools used in FPGA design. It offers a higher abstraction level for algorithmic descriptions as well as for data type specifications and interfaces. It allows the developer to write C/C++ code to describe the algorithmic behaviour of the hardware and generates a synthesizable Verilog or VHDL code as the product. Vivado HLS also allows the user to set directives and constraints in order to generate the desired functionality and the hardware.

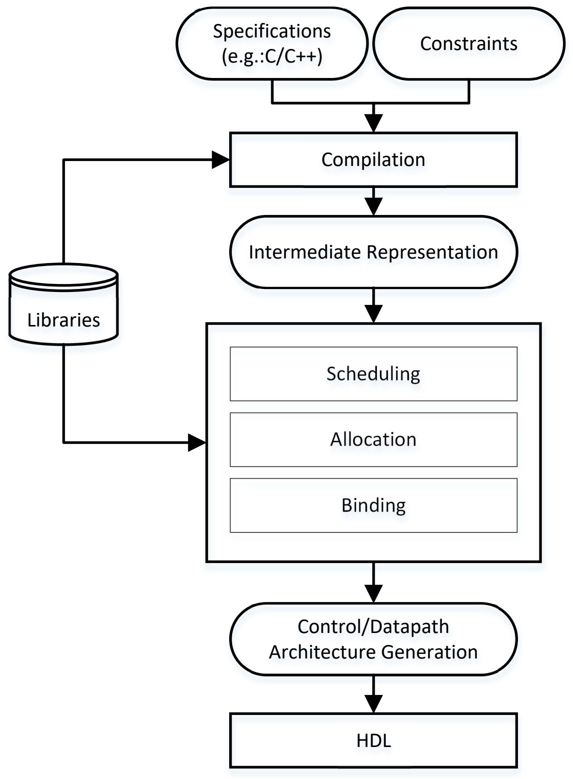

Specifications (e.g.:C/C++)
Constraints
Compilation
Intermediate Representation
Libraries
Scheduling
Allocation
Binding
Control/Datapath Architecture Generation
HDL

Figure 1.6. The typical flow of an HLS tool

Some of the benefits of using high-level synthesis tools like Vivado HLS are:

- Increasing the productivity by working in an environment with a higher level of abstraction.
- Allowing software developers to easily migrate their algorithms to the hardware platform.
- Reducing the debug time.
- Reducing the cost of development by saving the precious developer time.
- Having a design which is less prone to error by isolating the developer from some of the low-level design decisions.

- Exploring the different optimization options and creating different implementations without changing the algorithmic description.
- Automatic use of dedicated hardware resources such as DSP and memory elements.
- Portability and readability of the C code compared to the Verilog or VHDL code.
- Potentially increasing the performance and optimizing the resource utilization by automatically making better design decisions compared to the developer.

Chapter 2

Hardware Accelaration with FPGAs

2.1 Digital Image Processing with FPGAs

Reprogrammability, parallelism, and dedicated hardware elements make FPGAs one of the most preferred platforms to develop image processing applications. FPGAs are currently used for digital image processing in many fields such as medical [9] and automotive industry [3]. The most common characteristic of applications which benefits from FPGAs is that they require a low latency in the critical path. In other applications which are not critical path sensitive, developing on CPU or GPU platforms can be an alternative.

While CPUs and GPUs have higher operation frequencies, FPGAs achieves high performance by providing flexibility and application specific architecture. The high number of on-chip memory blocks in FPGAs enables the developers to exploit the parallelism in many applications. Image processing algorithms are good candidates for FPGA hardware acceleration because of their high inherent parallelism [2].

The inherent reprogrammability of the FPGAs allow the developers to test and develop different settings to optimize their designs. For example, a developer can test a filtering algorithm with two different coefficient matrix and see the trade-offs between the output image quality and the hardware efficiency.

The parallelism can be revealed in many image processing applications by using FPGAs. Most of the kernel operations can be done in parallel by using multiple FPGA resources. Also, the high number of input/output (I/O) pins on the FPGA chips can allow developers to send a part of their image signal from different ports and process different parts of the image at the same time.

Dedicated resources such as DSP48s and Block RAMs (BRAMs) inside the FPGAs facilitate the implementation of image processing algorithms into hardware. Some of the image processing algorithms may require extensive computation and signal processing. DSP48 resources can bu utilized for such computations. BRAMs, on the other hand, can be utilized as buffers or memory elements required for storing pixel data and delivering to the processing elements rapidly.

Finally, the other major advantage of using FPGAs for image processing is the power efficiency. CPUs and GPUs have fixed designs, and different instructions use the same datapath; thus the number of clock cycles required for an instruction is fixed. On the other hand, FPGAs can be customized based on the operation. The same algorithm can be completed in fewer clock cycles compared to the CPUs and GPUs.

```
unsigned int reverseBits(register unsigned int x)
{
    x = (((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1));
    x = (((x & 0xcccccccc) >> 2) | ((x & 0x33333333) << 2));
    x = (((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4));
    x = (((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8));
    return((x >> 16) | (x << 16));

}
```

Listing 2.1. Bit reversal operation in C/C++ code. A CPU or a GPU will complete this function in multiple clock cycles while an FPGA can complete the same function in one clock cycle.

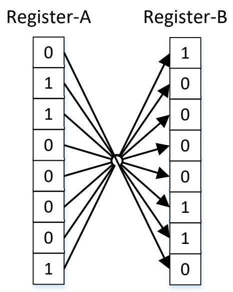


Figure 2.1. Wiring Register-A to Register-B in a reversal order to reverse 8-bit numbers.

The C code shown in Listing 2.1 is a bit reversing function. It takes a 32-bit number and reverse the bits of the number (i.e. The order of bits reversed as LSB:MSB). To do this operation, a typical CPU or a GPU will definitely need multiple clock cycles since they do not have ALU resources just for bit reverse operations. On the other hand, we can do this just by storing the number in a register and wiring the digits in reverse order into another register in an FPGA. Figure 2.1 presents an bit reversal architecture.

2.2 Interfaces and Integration Frameworks

The flexible nature of the FPGAs allows developers to integrate their designs in many different ways. An FPGA can be used as a stand-alone chip connected to I/O devices and utilized as the control and computation core of the system. But usually, in hardware acceleration applications, FPGAs are used as co-processors connected to a processing system (PS). In a typical FPGA accelerator application, the FPGA takes the data from the PS, processes the data and send the processed data back to the PS. All of this communication requires an interface to handle the data flowing between the PS and the FPGA. Although one may prefer to create a custom interface, it usually requires a substantial amount of work to create a robust and fast interface capable of transferring data between system elements. Other option would be to use a standard interface to transfer data between the CPU and the FPGA. These standard interfaces can also be used between the different Intellectual Property (IP) cores in the FPGA.

The Advanced eXtensible Interface (AXI) protocol [1] is currently adopted by the Xilinx FPGAs. The AXI protocol offers a standard interface for developers. There are three types of AXI4 protocol for different applications:

- AXI4 : This protocol should be preferred if the design requires a memory mapped interface.
- AXI4-Lite : This protocol is a subset of AXI4 protocol and designed for simpler interface requirements. Some of the signals are removed from AXI4 interface to reduce the design logic. AXI4-Lite is a memory mapped interface but it doesn't support data burst, it is designed for single transactions.
- AXI4-Stream : This protocol is designed to support data streaming, and it is a subset of AXI4 interface. The address signals are removed from the AXI4 interface since the data will be streamed and not mapped on to the memory. This protocol is preferred for data streaming applications such as image processing applications.

Using the AXI4 interfaces could be a difficult task if the design is developed purely with an HDL. On the other hand, using AXI4 interfaces is very simple with Vivado HLS.

Xilinx provides Verilog templates for AXI4 interfaces, but it is the developer's responsibility to correctly connect all the buses and generate correct signals for the interface. Usually, a state machine is required to control the interface. The same task can be accomplished in Vivado HLS just by adding a line of interface pragma. The code shown in Listing 2.2 presents a simple example on how to use AXI4 streaming interface in Vivado HLS.

```
void dividePixels (BYTE input_pixel[76800], BYTE
    output_pixel[76800] ){
    #pragma HLS INTERFACE axis port=output_pixel
    #pragma HLS INTERFACE axis port=input_pixel

    int i;
    for (i=0; i<76800; i++){
        #pragma HLS PIPELINE
        output_pixel[i]=input_pixel[i]/2;
    }
}
```

Listing 2.2. An example code in Vivado HLS to divide the pixel values of a streaming QVGA size gray-scale image by two.

Connecting a host CPU to an FPGA requires both hardware and software interfaces for FPGA acceleration applications. On the hardware side, the interface should be capable of transferring large amount of data with high bandwidth rates. On the software side, user functions should be simple enough to provide a good abstraction from the hardware interface.

As an alternative to the licenced solutions for CPU-FPGA connection, RIFFA (Reusable Integration Framework for FPGA Accelerators)[8] provides all the characteristic we mentioned above. It is an open source framework which supports both Xilinx and Intel (Altera) FPGAs. The connection between the host CPU and the FPGA is provided via a PCI Express bus. On the software side of the framework, APIs are provided in C/C++, Java, MATLAB and Python. The framework can be used both in Windows and Linux machines. The hardware interface of the RIFFA framework is similar to the standard FIFO interfaces.

2.3 Hardware Platforms

The experiments conducted in this thesis are implemented mainly on the ZedBoard [19] platform. The ZedBoard is a development kit for Xilinx Zynq-7000 with a variety of peripherals support. The good documentation support and the large variety of examples ease the use of the ZedBoard platform. Also, the Vivado IP Integrator facilitates the design process with ZedBoard significantly. VC707 development board selected as the target hardware for some of the other experiments with the RIFFA interface.

The Zynq-7000 is a System on Chip (SoC) which combines a CPU and an FPGA inside the same chip. This combination brings the flexibility of an FPGA and the performance benefits of a CPU together. The Zynq architecture has two main elements; a processor system (PS) and a programmable logic (PL). The PS and the PL part connected through a set of interfaces such as AXI and Interrupt channels. Figure 2.2 illustrates the architecture of the Zynq-7000 SoC.

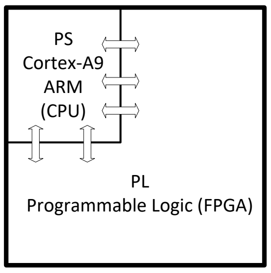


Figure 2.2. Simplified Zynq-7000 SoC architecture.

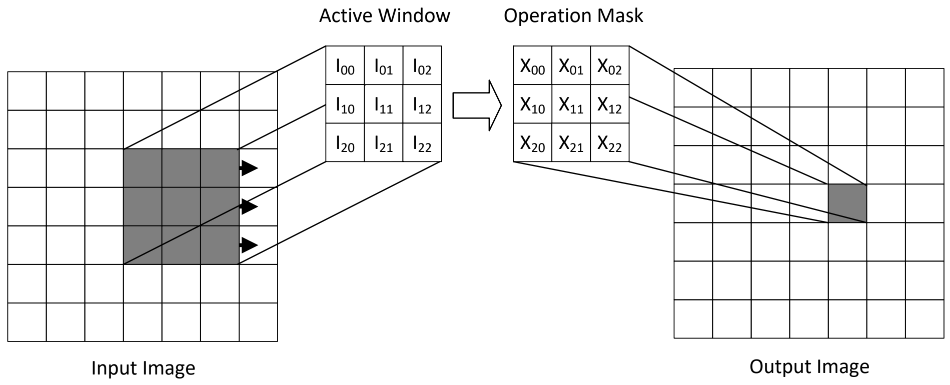
The combination of an FPGA with a CPU in the same IC brings some advantages with it. Some of these advantages are; power and area efficiency, reduced cost, and high performance. Bit manipulation is one of the operations which can significantly benefit from a SoC which combines a CPU with an FPGA. Now, if we take a look again on the code shown in Listing 2.1, we can notice that an application which consists of a vast number of such operations, can be accelerated by processing bit manipulation operations in the PL part of the chip.

Chapter 3

Line and Window Buffers

Line and window buffers are key elements in any kernel based image processing application. They are allowing a significant speed up by storing and shifting the previous pixel values needed for the kernel operation. This saves the kernel from accessing the local memory more than once for every pixel it covers.

The theoretical approach to an image processing algorithm with a window operation is to slide the window through the image, and write the operation mask result back to the position corresponding the center of the window. This approach is shown in Figure 3.1



Active Window
Operation Mask
I_{00} I_{01} I_{02}
I_{10} I_{11} I_{12}
I_{20} I_{21} I_{22}
X_{00} X_{01} X_{02}
X_{10} X_{11} X_{12}
X_{20} X_{21} X_{22}
Input Image
Output Image

Figure 3.1. The active window slides through the image. The operation mask gets the current pixels from the active window.

The sliding window, or in other words, the active window slides through the image starting from the top-left corner of the image. The window slides to the right, and when it reaches to the right border of the image, the sliding window moves to the next row and starts sliding from the left side of the image again. One of the corner cases we need to consider is the pixels on the boundary of the image. If the active window places any of the border pixels on its center, some of the elements in the active window will be empty in this case. One way to avoid this matter would be filling up the empty elements in the active window with zeros. The other way is to put the center pixels value into empty elements. However, this approach adds extra complexity to the algorithm. Another solution is to limit the active window operation area in a way that the active window never gets the empty values.

The size of the window may vary between the different image processing applications. As one can expect, the scale of the window is directly related to the computational complexity of the algorithm. If we assume that every pixel is processed in a square window, any increase in the size of the square will increase the computation requirement exponentially.

One of the main bottlenecks in an FPGA application is the data access time. Usually, It is not feasible to store the entire image inside the FPGA. If the image is stored outside of the FPGA, the main goal should be to reduce the transfer amount as much as possible. If we consider the theoretical sliding window approach, it requires access to the same pixel three times per row if a 3x3 window size is selected. Similarly, if a 5x5 window size is selected, the same pixel needs to be accessed five times per row. Also, the window requires a group of neighboring pixels, however only the pixels at the same rows are neighbors to each other in the array representation of the image. For this reason, this approach requires addressable access for every pixel in the image.

Streaming the image data into the FPGA is a faster solution and it removes the logic overhead created by the addressable access. In a data stream, the boundaries of the data are defined with the control signals such as data valid, data ready, start, etc. In order to prevent the redundant access to the same pixel multiple times, we can temporarily store all the required rows for the window operation, into the line buffers, also know as row buffers. The characteristic of the line buffers is simple; every new incoming pixel will push the previous ones to the other end of the buffer. This is a typical FIFO behavior. We provide the sliding functionality by placing the window buffer to the output of the line buffers. The high-level representation of this architecture is shown in Figure 3.2.

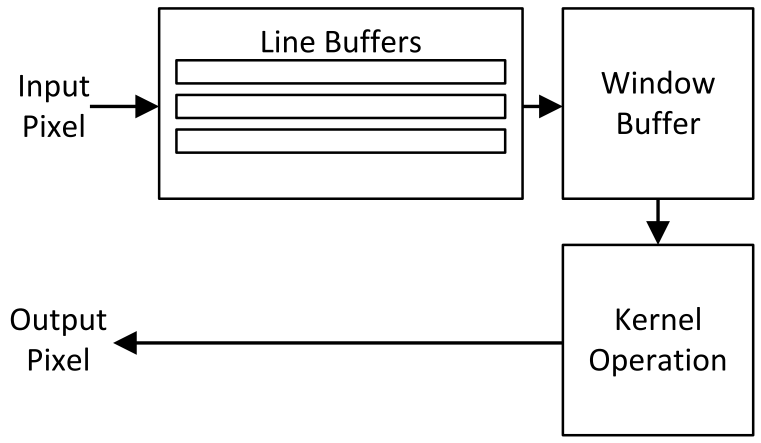


Figure 3.2. Line and window buffers.

3.1 Window Buffer Design

In order to temporarily store the pixels needed for the kernel computation, a window buffer is required. As we mentioned previously, the kernel needs access to the same pixel multiple times as the algorithm iterates. Instead of reading the same pixel multiple times, the window buffer can shift all the pixels, dump the ones which are not required anymore, and place the new pixels.

The output number of the window buffer may vary based on the kernel operation. Some kernel computations may not require all the pixels inside the active window. However, in order to create a design which can be used with any kernel, we need to output every pixel stored in the window buffer. This means if the window buffer size is 3x3 the number of outputs should be 9; likewise, for a 5x5 window buffer, the number of outputs should be 25.

The number of inputs of the window buffer depends on the number of rows it contains. The window buffer should store the data coming from the row buffers at every iteration. To process one pixel on every clock cycle, the window buffer should do shift, store and discard operations at the same clock cycle. This characteristic is very similar to the shift registers. For every row of the window buffer, we need to connect the registers back to back so that the data shifts through the window. We also need to output every register value since the kernel requires all of the pixel data inside the window buffer. Figure 3.3 shows the RTL schematic of a 3x3 window buffer.

To create a 3x3 window buffer architecture in Verilog, we need to declare nine registers and connect them like a shift register for every row. This is a relatively simple task, however even in a task like this, there are some pitfalls for inexperienced developers. In section 1.2 we talked about the difference of blocking and non-blocking assignments in Verilog. If a blocking assignment used in this example, the code would not be synthesized to the desired architecture.

Xilinx Synthesis Technology supports *for loops* and multidimensional arrays in Verilog, both of them are playing a significant role in creating parameterizable modules. *For loops* in Verilog should not be considered as a sequential loop. The distinction between a programming language and a hardware description language should be understood correctly. Listing 3.1 shows the *for loop* usage to create the RTL design shown in Figure 3.3. The size of the window buffer can easily be adjusted based on the parameters.

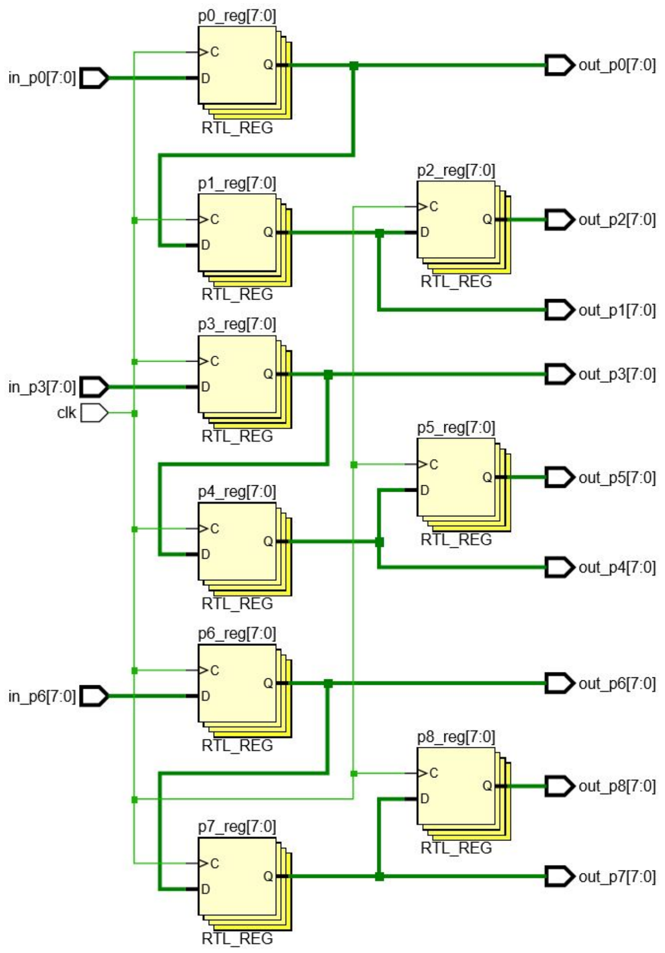


Figure 3.3. The RTL schematic of a 3x3 window buffer. The selected data width is 8-bit for a gray scale image.

```
for (i=0; i< row_size; i=i+1)
     begin
     for (j=0; j< column_size -1 ; j=j+1)
         begin
             p[i][j+1] <= p[i][j];
         end
     p[i][0] <= wire_in_p[i];
     end
```

Listing 3.1. Using a for loop in Verilog to connect the registers as shown in Figure 3.3.

Another important point in designing the window buffer is the amount of the control signals. Generally, the more control signals we include to the system, the more area is utilized. But in some cases, adding extra control signals may not add additional hardware depending on the FPGA architecture. Nevertheless , the performance will be negatively affected by these control signals due to the extra wiring and setup times of these signals. Figure 3.4 shows the flip-flop design inside the *Slices* used in 7 series FPGAs. Xilinx recommends using these control signals only when necessary [18] . For the sake of simplicity and performance, we can exclude these control signals from the window buffer design.

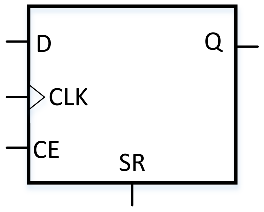


Figure 3.4. D-type flip-flops used in Xilinx 7 series slices. The ports for control signals such as clock enable(CE) and Set/Reset (SR) are provided within the flip-flop used in Xilinx 7 series FPGAs.

Designing a window buffer in Vivado HLS is an easier task compared to the RTL. Yet, understanding the underlying architecture is crucial for a successful design. Again, we use a two-dimensional array to create the window buffer, just as we did in Verilog. In Vivado HLS, arrays are considered as memory and they are mapped into the random access memory (RAM) structures. However, RAM structures do not have access ports for every memory cell it contains. Thus, with a RAM structure, it is impossible to read all the array elements simultaneously at the same clock cycle.

To achieve the same architecture shown in Figure 3.3, we need to give some directives to Vivado HLS about the desired hardware. Array partitioning is a pragma in Vivado HLS, which tells the tool how the arrays should be mapped to the FPGA. The array partition pragma comes with several options; type, factor, and dimension. In order to place every array elements into individual registers, type option should be set as complete. Since the array is going to be completely partitioned, the factor option is not valid. One important detail should not be forgotten in this case. Even though the partitioning type is selected as complete, the Vivado HLS will only partition the first dimension of the array completely by default. To partition every element of a multidimensional array, the dimension option should be set to zero. Lastly, the unroll directive is necessary to create a parallelism between the loop iterations. The code in Listing 3.2 shows how to shift the pixel values in the window buffer with Vivado HLS.

```
#pragma HLS ARRAY_PARTITION variable=windowBuffer complete
    dim=0

for    (int k = 0; k < 3; k++) {
#pragma HLS unroll
                windowBuffer[k][2] = windowBuffer[k][1];
                windowBuffer[k][1] = windowBuffer[k][0];
                windowBuffer[k][0] = lineBuffer [k][col];
            }
```

Listing 3.2. Creating window buffers in Vivado HLS. Partitioning the windowBuffer array will force the HLS tool to map the array into individual registers, not into a memory.

3.2 Line Buffer Design

Line buffers are temporary storage units which can store several lines of image data. From a higher level perspective, a line buffer is a shift register which shifts all the pixels stored, through the window buffer.

A naive approach to design line buffers in hardware is to apply the shift register architecture directly to the line buffers. This approach is very similar to the window buffer model in which every row is a shift register. Listing 3.3 shows the Verilog code which will produce this naive approach. Figure 3.5 shows the simplified RTL schematic generated from the code shown in Listing 3.3.

```
for (i=rowBufferSize-1; i>0; i=i-1)
    begin
        rowBuffer0[i] <= rowBuffer0 [i-1];
    end
rowBuffer0[0] <= wire_in_pixel_in;
```

Listing 3.3. A naive way to create a line buffer in Verilog.

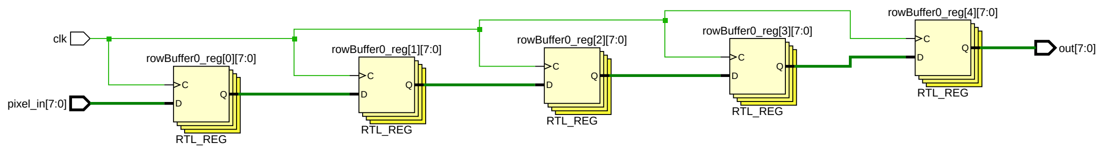


Figure 3.5. The simplified RTL schematic for a line buffer with the size of five registers. When the size of the line buffer gets larger, this design becomes unfeasible.

The main problem with the naive approach is the scalability. This design is feasible only for the line buffers with small sizes. For example, if we implement a 640 wide line buffer with the naive approach the total number of slices required for is going to be 160. When the usage report examined closely, we can notice that all 160 slices are SLICEM type and they are used as shift registers.

There are two types of slices in Xilinx 7 series FPGAs; SLICEM and SLICEL. Typically the number of SLICEMs are less than half of the number of SLICELs. SLICEMs have all the functionality of SLICELs, additionally can be utilized as 32 bit shift registers and distributed RAMs, while SLICEL can not. For this reason, developers should not wastefully utilize SLICEMs resources, especially for large designs.

A 640 wide line buffer with 8-bit data width will require 5120 bits storage in total. The naive approach will map this architecture into the shift registers in SLICEM. Thus the total number of SLICEM can be found as 160 by dividing 5120 to 32. The other problem of using a large amount of slices is the delays caused by wire routings. If we consider the clock signal, the path of the clock signal will increase when more logic slices are used in the design; thus the performance will be affected negatively.

A better approach to implement line buffers is to use memory structures in FPGAs. There are two types of memory in Xilinx 7 series the FPGAs; distributed RAM and BRAM. A typical RAM element has two main types of ports, data ports and address ports. The address port carries the information of which cell in the RAM will be accessed, and the data port transmits the addressed cell data. There are different variations of RAM structures such as single port and dual port RAMs. A true dual port RAM can independently write and read data with two different address ports. While a simple single port RAM uses the same address port for read and write operations. We will discuss the different RAM variations in detail later in the histogram design (Section 6).

To create a line buffer from a RAM structure, there are couple additions required in the design. Since we are no longer technically shifting the data, we need to index the next cell whenever we receive a new pixel data. To create such functionality, a counter should be connected to the RAMs address data. Since the input and the output channels of the line buffer can process data at the same cycle, the RAM kind should be a dual port RAM. Figure 3.6 illustrates the architecture with line buffers built from RAMs integrated with the window buffer and the kernel. There is built-in FIFO support in Xilinx 7 series FPGAs, The architecture in Figure 3.6 is a generic design which can be used in other FPGAs as well.

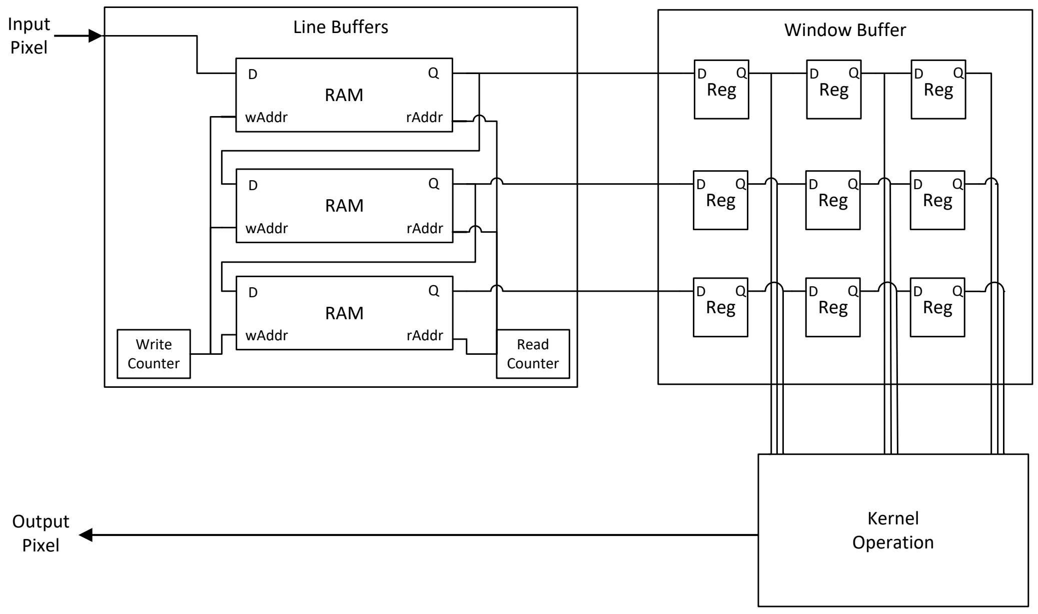


Figure 3.6. The hardware architecture of line and window buffers. Read and write counters provide a FIFO capability to the RAMs.

Vivado HLS can create a line buffer from RAM structures easily. Unlike the line buffer implementation in Verilog, Vivado HLS generates all the necessary control logic to use a RAM structure as a line buffer. One important detail needs to be pointed at is the partitioning the line buffer array. We need to completely partition the first dimension of the line buffer array in order to create multiple line buffers; otherwise, the array will be implemented as a whole and accessing to the data from every dimension will take multiple clock cycles. The piece of code in Listing 3.4 shows how the line buffers can be implemented in Vivado HLS. This technique is based on [10].

```
#pragma HLS ARRAY_PARTITION variable=lineBuffer complete dim
    =1
...
lineBuffer [0][j] = lineBuffer [1][j] ;
lineBuffer [1][j] = lineBuffer [2][j] ;
lineBuffer [2][j] = input_image[i][j] ;
```

Listing 3.4. Creating line buffers in Vivado HLS

Look up tables (LUT), also know as function generators in FPGAs, can also be implemented as RAM elements. In Xilinx 7 series, LUTs have six inputs; therefore each LUT can store 64-bit. Each slice contains four LUT, which means each slice can be implemented as a 256-bit memory. It is important to remember that only SLICEMs can be used as distributed RAM elements.

If we implement a line buffer with the same size we used in the naive approach, this time we will only need 20 SLICEM elements, compared to the 160. We can formularise the number of SLICEM required for a distributed RAM as follows.

$$SLICEM\ number = \frac{Memory\ Requirement}{LUT\ capacity\ LUT\ number\ in\ a\ slice} \tag{3.1}$$

Block RAMs are dedicated hardware elements designed to meet the on-chip memory requirements of the FPGAs. In Xilinx 7 series FPGAs, each block RAMs module can store up to 36 Kbits of data or can be used as two independent 18 Kbits RAM. They can provide fast memory solutions for FPGA applications with their dedicated designs.

The line buffer design we previously mentioned requires 5120 bits of memory. Clearly, a half block RAM tile would be enough to implement this line buffer. We can find the resource utilization for each approach in Table 3.1

Table 3.1. Resource utilizations for different line buffer implementations.

Implementation Approach	Resource Required
Shift Registers (Naive Approach)	160 SLICEMs
Distributed RAM	20 SLICEMs
Block RAM	1 BRAM (18 Kbits)

There are couple criteria when making the decision between the BRAM and the distributed RAM. If the size of the buffer is small, distributed RAM can be a better option. Distributed RAMs are implemented with the same slices used for the computational part implementation, a small size distributed RAM can be placed right next to slices where the computation happens. This will reduce the delays by placing the memory and the computational element side by side. However, as the size of the memory grows, distributed RAMs will become slower due to the routing delays. BRAMs are usually a better option to implement large line buffers in FPGAs; they can provide very fast memory implementations if utilized correctly.

In Vivado HLS, the type of RAM is selected automatically based on the size of line buffer required. If the memory requirement is less than the threshold, distributed RAM is selected. Otherwise BRAM is used by default. For the XC7Z020 chip used on the ZedBoard this threshold is 1024 bits. However the developer can override this decision in Vivado HLS. Sometimes what is optimal may depend on the developer and the tool. Table 3.2 shows the different implementation results of the same design with distributed RAMs and BRAMs. The buffer size used is 3x640 for this design, which is mapped to BRAMs by default. The distributed RAMs may be preferred with a tradeoff of more slice usage for a slightly higher clock frequency. By adding resource pragmas, we can explore the different RAM implementation results in Vivado HLS. Listing 3.5 shows some of the pragmas that can be used to select the memory resources.

```
#pragma HLS RESOURCE variable=lineBuffer core=RAM_S2P_LUTRAM
#pragma HLS RESOURCE variable=lineBuffer core=RAM_T2P_BRAM
```

Listing 3.5. Vivado HLS resource pragmas for Distributed RAM and BRAM

Table 3.2. The implementation results of a kernel operation with line buffers, using different RAMs.

	RAM Type	
	Distributed RAM	BRAM
Clock Period Achieved	6.688 ns	6.968 ns
SLICE	172	83
LUT	496	183
FF	218	194
BRAM	0	2

When designing with HDLs, correctly implementing a RAM is more challenging than the Vivado HLS design. Adding unnecessary control signals such as the reset signal may result in inefficient and undesired memory implementations. The reset signal for a random access memory is an expensive control signal and should be avoided in the design. For instance, none of the memory blocks (BRAMs) in the FPGA have a reset signal to reset the memory cells. Instead there is an optional flip-flop that can be activated on the output of the BRAM with a reset signal, which will only reset the output but not the data stored in the cell. Listing 3.6 shows the code to implement a ram structure in Verilog.

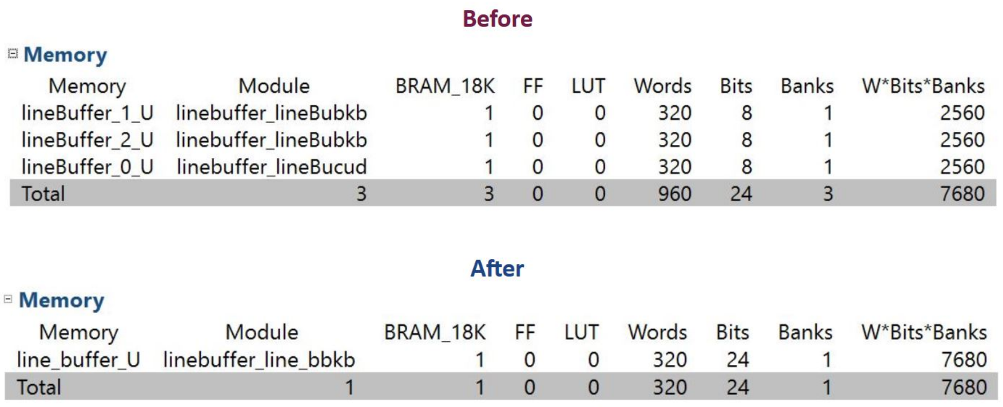
```
//(* ram_style = "distributed" *)
//(* ram_style = "block" *)

reg [DWIDTH-1:0] ram[MEM_SIZE-1:0];

always @(posedge clk)
begin
    if (ce0)
    begin
        q0 <= ram[addr0];
    end
end

always @(posedge clk)
begin
    if (ce1)
    begin
        if (we1)
        begin
            ram[addr1] <= d1;
        end
    end
end
```

Listing 3.6. A verilog code to implement a dual port ram with two clocks, the same clock signal can be used for read and write operations. The XST can infer the RAM style automatically; even so, the ram style directive can override this behavior. The ram style directive can be set as distributed or block.

In an efficient line buffer implementation, we can minimize the number of BRAMs used in the system. We can customize the BRAMs in a way that the three line buffers is placed on the same BRAM. To do that, we can concatenate the lines and store 24-bit data instead of 8-bit data in each memory cell. On the RTL level, we need to modify our RAM structure and change the data width from 8-bit to 24-bit to achieve this. Then we can take the buses used in individual RAM data ports and concatenate them into a single bus. This way we will use only one BRAM instead of three.

Depending on the coding style Vivado HLS can achieve this optimization automatically. The other way to do this in Vivado HLS is to use arbitrary precision data types.

We can do bit-slicing and concatenation operations using these data types. Including the *ap_int.h* header file to our code will allow us to use arbitrary data types and achieve this optimizations. Figure 3.7 shows the difference in memory usage when this optimization is used.



Before

Memory

Memory	Module	BRAM_18K	FF	LUT	Words	Bits	Banks	W*Bits*Banks
lineBuffer_1_U	linebuffer_lineBubkb	1	0	0	320	8	1	2560
lineBuffer_2_U	linebuffer_lineBubkb	1	0	0	320	8	1	2560
lineBuffer_0_U	linebuffer_lineBucud	1	0	0	320	8	1	2560
Total	3	3	0	0	960	24	3	7680

After

Memory

Memory	Module	BRAM_18K	FF	LUT	Words	Bits	Banks	W*Bits*Banks
line_buffer_U	linebuffer_line_bbkb	1	0	0	320	24	1	7680
Total	1	1	0	0	320	24	1	7680

Figure 3.7. BRAM usage of the same application. Concatenating data lines will create this optimization.

3.3 Conclusion

In this chapter, we presented the implementation details of window and line buffers both in RTL and HLS methods. First, we discussed the theoretical approach on how line and window buffers work and why they are necessary for doing fast kernel operations. Then, we introduced the hardware structure that implements this theoretical approach. We explained possible pitfalls on the hardware implementation and showed ways on how to avoid them. These pitfalls will only happen on the RTL design since HLS infers these memory structures more elegantly.

We presented how different optimizations can be applied to achieve an efficient hardware implementation of window and line buffers. To apply these optimizations we showed what type of low-level design changes need to be made in RTL design. We also showed that Vivado HLS can generate all the optimizations we covered in this chapter without the necessity of doing low-level design changes. Having a fair amount of hardware knowledge and understanding the underlying architecture is sufficient to design an efficient implementation with Vivado HLS. On the other hand, a significant amount of hardware knowledge and an HDL expertise is required for an efficient implementation of these buffers when RTL method is used.

Chapter 4

Image Processing Kernels

A kernel in image processing is a matrix of operations which is applied to an active window to create different effects on the output image. This operation is also known as convolution. Although there is no limit on what types of kernel we can create, some of them can produce certain effects which can be used in image processing applications. For example, we can use a Sobel filter to emphasize the edges in an image. Likewise, we can use a Gaussian filter to blur an image.

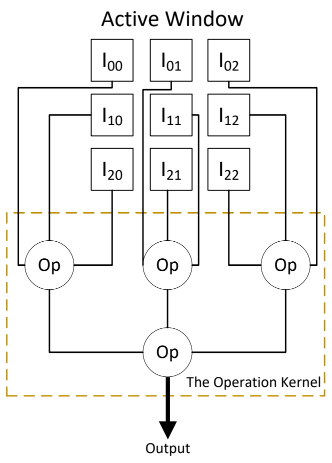


Figure 4.1. An operation kernel from the hardware perspective. Operation kernels vary between different image processing algorithms.

From the hardware perspective, these kernels are set of computation units that take the pixel data from the window buffer and generate an output after the process. Figure 4.1 illustrates the kernel operation from the hardware perspective. All the computations in the operation kernel will have a certain complexity. However, some computations will require more time and resources. In this thesis, we use the hardware friendly term for operations that does not require complex computations.

In this chapter, we will discuss different image processing kernels and their optimizations both in HLS and RTL designs. These kernels are commonly used in image processing applications. Having an efficient implementation of these kernels is crucial in many designs.

4.1 Sobel Filter

The Sobel filter is a commonly used kernel in image processing for edge detection. Edges are important features in an image that are used for separating objects from each other. To detect the edges in an image, we need to look at the difference (gradient) between the neighboring pixels. Of course, we need to do this in a way that we can differentiate the continuous differences between pixels which represent edges in the image. The Sobel operator uses two filters to detect horizontal and vertical edges in the image. Equation 4.1 shows the two kernels used for horizontal and vertical edge detection.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * WB \qquad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * WB \tag{4.1}$$

After getting the results from each filter, we can get the total gradient magnitude by:

$$Edge\ Weight = \sqrt{{G_x}^2 + {G_y}^2} \tag{4.2}$$

Overall, we can categorize the Sobel algorithm as a hardware friendly algorithm except the part shown in Equation 4.2. The power of two and the square root operations are complex operations that require multiple cycles and a dedicated hardware in FPGA. Especially the square root operation requires a floating point operator core like DSP48 in FPGA implementation. In order to avoid these complex operations, we sacrifice some accuracy and can transform the Equation 4.2 into:

$$Edge\ Weight = G_x + G_y \tag{4.3}$$

Multiply and divide operations are considered as complex operations. If we look at a CPU architecture, these operations take more clock cycles compared to the addition or subtraction operations. But there are some exceptions in both operations. If we consider dividing or multiplying values with numbers that can be represented with 2^a ($a \in \mathbb{W}$), these operations become nothing but shift operations. Also, with the flexibility of the FPGA we can simply hardwire the corresponding bits instead of actually shifting them. In fact, most of the synthesis tools will recognize the shift operations and map them into hardwired registers. Figure 4.2 demonstrate the multiply by two operations from the FPGA perspective. Fortunately, the Sobel operator only requires multiplication with two, which can be done easily.

Now we can write the HDL code for Sobel filter using the characteristic we described above. We can separate this task into four steps:

- Find the gradients in horizontal and vertical filters.

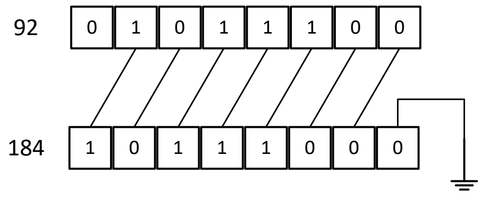


Figure 4.2. Multiplication with two in hardware. We need to fill empty bits with zero. Likewise, we can do division or modulo operation by wiring registers directly.

- Take the absolute value of these gradients.
- Add the absolute value of gradients.
- Limit the Sum to the max value (Prevent the overflow).

As we emphasized in the previous chapters, the developer should be careful when designing with the RTL method. We can generate the correct functionality in many different ways, but if we want to have a fast and efficient design, we need to take more things into account rather than just the functionality. One of the key factors in creating a fast design is pipelining the long processes. If we consider the Sobel filter, we can pipeline the steps we mentioned above instead of creating a long datapath that combine all the steps in a single clock cycle. Figure 4.3 demonstrates the pipelining of the Sobel filter operation.

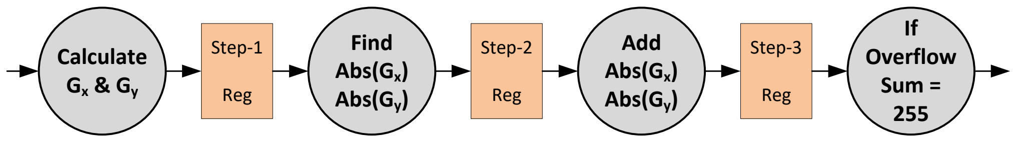


Figure 4.3. Pipelining the Sobel filter operation. We divide the opreation into four different steps and place registers between them.

We need to create this pipeline structure manually when writing the HDL code. To do so, we need to declare the data types as registers and use non-blocking assignments to assign each step's result in every clock cycle. Listing 4.1 shows the Verilog code constructed in a pipelined manner.

```
always @ (posedge clk)begin
    // Step One
    Gy<=((pixel0-pixel6)+((pixel1-pixel7)<<1)+(pixel2-pixel8
   ));
    Gx<=((pixel2-pixel0)+((pixel5-pixel3)<<1)+(pixel8-pixel6
   ));

    // Step Two
    absGy <= (Gy[data_size+2]? (~Gy+ 1'b1) : Gy);
    absGx <= (Gx[data_size+2]? (~Gx+ 1'b1) : Gx);

    // Step Three
    edgeWeight<= absGy+absGx;

    // Step Four
    pixelOut <= (|edgeWeight[data_size+2:data_size])?
   max_pixel_val : edgeWeight[data_size-1:0];
end
```

Listing 4.1. The Verilog code for pipelining the Sobel operations as demonstrated in Figure 4.3.

Designing the Sobel filter in HLS is simpler than the RTL design as expected. Vivado HLS is providing practical user directives to optimize the hardware. To create a pipeline architecture in Vivado HLS, we need to add a pipeline pragma into the code. Listing 4.2 shows the HLS code that creates a pipelined architecture. This code is based on [16] and [10].

```
for (int i=1; i<IMG_H-2; i++){
    for (int j=1; j<=(IMG_W-2); j++){

        #pragma HLS PIPELINE

        // Line and Window Buffers here...

        // Calculating Gx and Gy for pixel[i][j]
        Gx = (wB[2][0]+wB[2][1] + wB[2][1] +wB[2][2])-(wB
    [0][0]+wB[0][1]  +wB[0][1] +wB[0][2]);
        Gy = (wB[0][2]+wB[1][2] + wB[1][2] +wB[2][2])-(wB
    [0][0]+wB[1][0]  + wB[1][0]+wB[2][0]);

        output_image[i][j] = (abs(Gx) + abs(Gy));
    }
}
```

Listing 4.2. The HLS code for the Sobel filter.

We can notice that in Vivado HLS, we dont need to take care of the overflow case where the result of the $|G_x| + |G_y|$ is greater than the max value. The C/C++ synthesis will automatically take care of such corner cases. Adding the *PIPELINE* directive into the code will automatically create necessary pipeline stages for the Sobel operation. Noticed that both in RTL and HLS approaches, we created parameterizable designs, so we can adapt the hardware for a different image size just by changing these parameters.

4.1.1 Results

Table 4.1 shows performance and resource utilization results for both HLS and RTL approach. HLS and RTL designs are very similar regarding the resource utilization and the throughput. The RTL design uses slightly fewer resources, and it is slightly faster than the HLS design. Creating an efficient Sobel filter with RTL is challenging, we can achieve very similar results with HLS. In this design, we also applied all the optimizations we discussed in the previous chapter. In the RTL design, we manually pipelined the process by placing registers between each step. In the HLS design, we just gave a pipeline directive to Vivado HLS to create an efficient pipeline architecture.

Table 4.1. Resource and Performance comparison of both methods for Sobel filter. HLS can generate very similar results without going to the details of the hardware.

	FF	LUT	BRAM	DSP48	Throughput
RTL	153	202	1	0	2.350 kHz
HLS	172	252	1	0	2.213 kHz

4.2 Gaussian Smoothing Filter

Gaussian smoothing is a widely used filter in image processing. It creates a blur effect and reduces details in an image. This feature is mostly used for noise suppression. Equation 4.4 shows the two-dimensional Gaussian function.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{4.4}$$

We can create a convolution matrix by using discrete values that represent a Gaussian function by sampling Equation 4.4. At this point, it is important to select discrete values that can facilitate the computation of the filter. One of the most commonly used masks for Gaussian filtering is shown below. According to [7], the matrix below and the matrix used for mean filtering probably constitute over 80% of all discrete approximations to a Gaussian.

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{4.5}$$

In the previous chapter, we discussed the advantages of using discrete values that can be represented with 2^a ($a \in \mathbb{W}$). We also discussed that the division operation could also be made by hardwiring the registers. To apply the convolution matrix shown in Equation 4.5, we need five multiplications, eight additions, and one division.

Now, if we turn our attention to the multiplication operation, we will notice that if we multiply a pixel (8-bit data) by four, the product would be 10-bit. In this case, we need to use an 11-bit adder to add this product with another pixel. Likewise, to add the product of the previous addition, we will require 12-bit adders.

In Chapter 1, we mentioned that the way the code is written is directly related with the resulting hardware. We need to transform the convolution matrix in a way that it will reduce the bit-width of the operations in the code. Instead of multiplying the pixels with the coefficients in the convolution matrix and dividing the final product, we can modify the matrix to remove the multiplications. This way, we can reduce the size of the adders used in the kernel computation. The convolution matrix shown in Equation 4.6 is a different representation of the same Gaussian function.

$$\begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix} \tag{4.6}$$

The advantage of the new Gaussian matrix is that it doesnt require multiplications and it will not increase the data width for add operations. With the new coefficients, the largest adder needed in the kernel computation is an 8-bit adder. As expected, the longest path in an 8-bit adder is shorter than a 12-bit adder. This allows us to operate at higher clock frequencies with less number of slices. We also manually pipelined the process similar to what we did in the previous section. Listing 4.3 shows the optimized code in Verilog which reduce the data width of the adders used in kernel computation.

```
always @ (posedge clk) begin

    line1 <=((pixel0[7:4])+(pizel1[7:3])+(pixel2[7:4]));
    line2 <=((pixel3[7:3])+(pixel4[7:2])+(pixel5[7:3]));
    line3 <=((pixel6[7:4])+(pixel7[7:3])+(pixel8[7:4]));

    out <= (line1+line2+line3);

end
```

Listing 4.3. The Verilog code with efficient Gaussian filter implementation.

We can apply the same optimization technique to the HLS version of our implementation. Listing 4.4 shows the Vivado HLS code written with the same approach we applied to the RTL design.

```
for (int i=1; i<IMG_H-2; i++){
    for (int j=1; j<=(IMG_W-2); j++){

        #pragma HLS PIPELINE
        // Line and Window Buffers here...

        // Gaussian Kernel
        X = (wB[2][0]>>4 + wB[2][1]>>3 + wB[2][2]>>4);
        Y = (wB[1][0]>>3 + wB[1][1]>>2 + wB[1][2]>>3);
        Z = (wB[0][0]>>4 + wB[0][1]>>3 + wB[0][2]>>4);

        output_image[i][j] = (X+Y+Z);
    }
}
```

Listing 4.4. The Vivado HLS code with efficient Gaussian filter implementation.

Lastly, we demonstrate a similar optimization that can speed up the multiplication operation. In some cases, we may require a Gaussian matrix with coefficients that are not representable with 2^a ($a \in \mathbb{W}$). However, we can still use bit-wiring techniques to avoid usage of DSP48 cores. Equation 4.7 shows an example of a multiplication that can be separated into two parts.

$$A \times 6 = (A \times 4) + (A \times 2) \tag{4.7}$$

Using the characteristic shown in the equation above, we can implement this operation in hardware as show in Figure 4.4

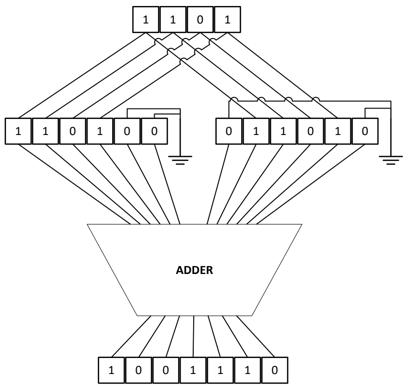


Figure 4.4. An illustration for multiplication by six in hardware without using a DSP48 core.

Vivado HLS will automatically recognize such multiplications that can be generated with up to two level adders/subtractors. For the sake of performance, Vivado HLS will map other multiplications which require more than two level adders/subtractors into DSP48 cores.

4.2.1 Results

Table 4.2 shows performance and resource utilization results of both RTL and HLS implementations with different convolution matrices. Again, we applied all the previous optimizations we discussed previously. Our results show that the described optimization improves the performance and efficiency in both in approach. By avoiding the multiplication, we decreased the adder size used for the kernel computation. Also, our results show that the Vivado HLS can generate an efficient implementation of Gaussian filter if the same coding practice is applied as in the RTL approach. Our HLS implementation is performing better than our RTL implementation in terms of throughput. This is probably because of Vivado HLS can create a more balanced pipeline than our manual implementation.

Table 4.2. Resource and Performance comparison of both methods for Gaussian filter. Optimized HLS outperforms Optimized RTL design in terms of throughput.

	FF	LUT	BRAM	DSP48	Throughput
RTL	98	148	1	0	2.228 kHz
Optimized RTL	63	122	1	0	2.788 kHz
HLS	128	174	1	0	2.118 kHz
Optimized HLS	86	152	1	0	2.890 kHz

4.3 Morphological Operations on Binary Images

Morphological operations are often used in analysis of binary images. Various morphological operations can be applied for noise reduction, object identification and image enhancement. For example, in [9] , morphological operations are used for enhancing cell walls in a real-time cell sorting system.

In a binary digital image, a pixel can only be represented with two values; 0 or 1. Thus, the computation complexity of morphological operations is lower when compared to previous kernel operations we discussed. Instead of using 8-bit registers to store a pixel value, now we can use a single flip-flop.

Dilation and erosion are two main operators used in morphology. The primary effect of the dilation is to grow objects in size. This operation is useful to remove salt noise from an image. It also has an effect of fixing the cracks in objects. On the other hand, erosion is used for shrinking objects in size. It removes the pepper noise and eliminates thin objects from an image.

Although dilation and erosion operations seem like they are opposite of each other, they do not cancel each other's effects. Closing and Opening effects can be created by using erosion and dilation operations consecutively. Figure 4.5 demonstrates the combination of dilation and erosion kernels to create a closing operator. Likewise, to create an opening operator, we need to switch the places of erosion and dilation kernels.

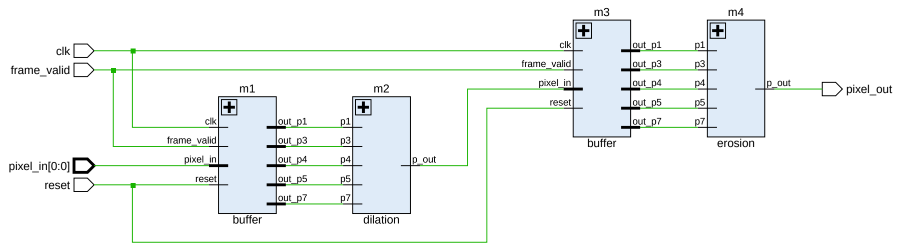


Figure 4.5. Combining erosion and dilation kernels to create a closing operator.

In this section, we use the term *structuring element* to refer the matrix that represents the shape of the set operations. For all the morphological operations we mentioned above, their effect is based on the structural element. In this experiment, we use the structuring element shown below.

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{4.8}$$

In dilation operation, if any pixel masked by the structuring element is 1, the output pixel will be set to 1. In contrast, the erosion operation sets the output pixel to 0 if any pixel masked by the structuring element is 0. Listing 4.5 shows how this two morphological operations can be described in Verilog. Notice that, this time we are not manually pipelining the kernel operations since they are not complex enough to create a bottleneck in our design.

```
// Dilation
assign p_out = p1|p3|p4|p5|p7;

// Erosion
assign p_out = p1&p3&p4&p5&p7;
```

Listing 4.5. Dilation and Erosion operations in Verilog.

To apply the framework we created for kernel operations, we need to make two modifications. Apart from changing the kernel operation, we also need to change the line and window buffer widths to 1-bit in order to make our buffer architectures compatible with binary images. Here we use the advantage of creating parameterizable components, and just change our data size to 1-bit.

Vivado HLS can handle binary operations effectively. In Chapter 3, we introduced arbitrary precision data types on Vivado HLS. Instead of using an *integer* or a *char* data type, we can create a custom data type which is only 1-bit wide. The Vivado HLS code for erosion and dilation kernels can be seen in Listing 4.6.

```
// Erosion Kernel
ap_uint<1> erosion(ap_uint<1> wB[3][3]){
    return (wB[0][1]&wB[1][0]&wB[1][1]&wB[1][2]&wB[2][1]);
}
// Dilation Kernel
ap_uint<1> dilation(ap_uint<1> wB[3][3]){
    return (wB[0][1]|wB[1][0]|wB[1][1]|wB[1][2]|wB[2][1]);
}
```

Listing 4.6. Dilation and Erosion operations in Vivado HLS.

4.3.1 Results

Table 4.3 presents performance and resource utilization results of both RTL and HLS implementations of morphological operators. Notice that dilation and erosion operators have same results. The reason behind this, is that these kernels are not the longest delay elements in the pipeline and they can be both implemented using the same number of resources. Our results show that Vivado HLS can efficiently handle binary operations if correct data type used.

Table 4.3. Resource and Performance comparison of both methods for morphological operations on binary images.

	FF	LUT	BRAM	DSP48	Throughput
RTL (Erosion and Dilation)	80	77	0	0	5.571 kHz
HLS (Erosion and Dilation)	119	123	0	0	5.261 kHz

In Chapter 3, we discussed BRAM and distributed RAM preferences. Since our data type is eight times smaller when processing binary images, the total capacity of our buffers is eight times smaller as well. As we discussed before, in XC7Z020 FPGAs, the threshold memory capacity for BRAM preference is 1024-bit. In this design, the total size of line buffers is 960-bit, which is automatically mapped to the disturbed RAMs.

4.4 Conclusion

In this chapter, we presented the hardware implementation details of commonly used image processing kernels. Respectively; we analyzed Sobel filter, Gaussian filter, and morphological operators.

We explained the underlying architecture and explored optimizations for each kernel. We applied these optimizations both on our RTL and HLS implementations. We demonstrated that to create an efficient design, understanding the architecture is critical both in HLS and RTL approach. We also showed that HLS tools are able to generate efficient hardware when correct optimizations are applied. Although HLS tools are getting better each day on mapping the code into the hardware, code restructuring is still necessary.

In our Gaussian filter implementation, our results show that the Vivado HLS can generate a faster implementation than our RTL design. The RTL method allows us to make low-level design decisions for efficient hardware implementations. On the other hand, this creates a major challenge since the efficiency of the design heavily depends on the correctness of the developers decisions.

Chapter 5

Histogram Design

Histogram is a commonly used algorithm in many image processing applications. An image histogram refers to the frequency distribution of pixels in an image. Histogram can be used in applications such as thresholding, image analysis, contrast adjustment, and so on. It is important to have a fast and robust implementation of this algorithm since it has a potential to be a bottleneck for the rest of the system if not designed carefully. The equation below represents the basic histogram equation.

$$histogram[pixel] = histogram[pixel] + 1; \tag{5.1}$$

Although the histogram implementation seems like a straightforward task from the software perspective, it has many pitfalls on the hardware implementation. Listing 5.1 shows a software approach to RTL implementation of the histogram algorithm.

```
reg [7:0] histogramReg [255:0]

always @(posedge clk) begin
// Control Logic ...
    histogramReg[pixel_in] <= histogramReg[pixel_in] + 1'b1;
end
```

Listing 5.1. Software approach to histogram design in RTL

Table 5.1 shows the naive RTL implementation results. We observe that the initial RTL implementation uses uncommonly high amount of FPGA resources. To understand this unexpected resource utilization, we need to look at the Equation 5.1 again. We can notice that there is a dependency between the previous and the next value of the histogram array. This forces the synthesis tool to map every element of the histogram array into individual registers. As expected this creates a massive logic and wiring overhead.

Table 5.1. Inefficient histogram implementation of the code shown in Listing 5.1.

	FF	LUT	BRAM	DSP48	Throughput
Initial RTL Design	3607	5794	0	0	1.558 kHz

We can apply the same approach on Vivado HLS with the code shown in Listing 5.2

```
for (int i=0; i<IMG_H; i++){
     for (int j=0; j<=IMG_W; j++){
     #pragma HLS PIPELINE II=1
        hist[pixel_in]= hist[pixel_in] + 1;
       }
}
```

Listing 5.2. Vivado HLS histogram code with the software approach.

When the code shown in Listing 5.2 used in Vivado HLS, we will notice that the generated system requires two clock cycles to process an input pixel. The reason for this behavior is the same reason why the RTL design uses a massive amount of resources. However, instead of forcing the system to process one pixel at every clock cycle with a large area overhead, Vivado HLS generates a control logic that process one pixel at every 2 clock cycles. We show the implementation results in Table 5.2

As we demonstrated above, the dependency in the histogram algorithm prevents

Table 5.2. Implementation results of the code shown in Listing 5.2.

	FF	LUT	BRAM	DSP48	Throughput
Initial HLS Design	81	80	1	0	1.198 kHz

us from achieving desired results both in RTL and HSL designs. To create a more efficient hardware implementation, a closer look on the memory design is necessary. Figure 5.1 illustrates the architecture we are trying to create.

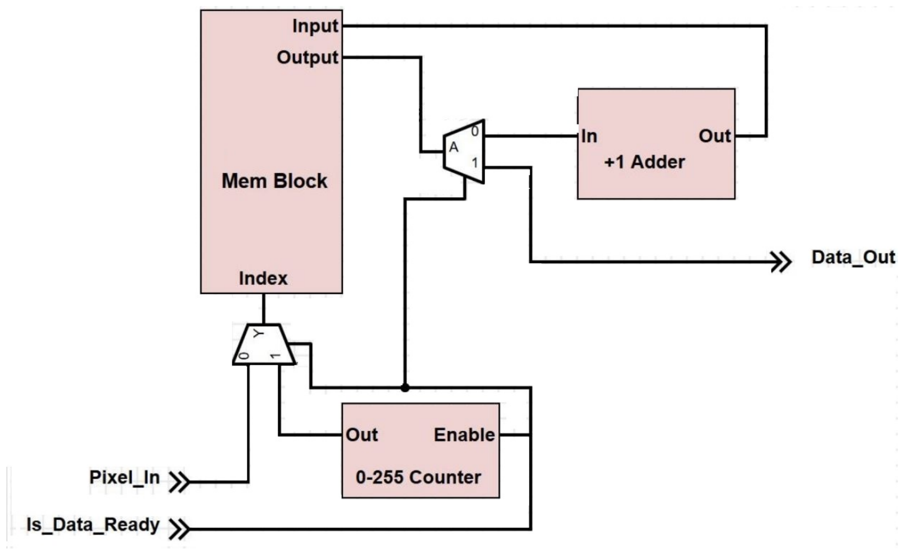


Figure 5.1. Illustration of the histogram architecture.

In Chapter 3, we discussed various memory types such as single-port memory and dual-port memory. In the histogram design, a single-port memory is out of option since we need to write and read at the same clock cycle to achieve an initiation interval equal to one.

The memory operating mode is another parameter we need to consider in the histogram design. There are three primitive operating modes: WRITE FIRST, READ FIRST, and NO CHANGE. Since the data hazard type on the histogram algorithm is read-after-write (RAW), we need to select read first operating mode for the memory. Distributed RAMs on Xilinx FPGAs only operates on write-first mode. This leaves

BRAMs as our only memory option since they support all the primitive operating modes we mentioned above. Another aspect to consider is that the read-first mode operates slower than the write-first mode. At this point, we need to sacrifice some performance in order to get an initiation interval equal to one.

In the BRAM architecture, there is an optional register connected to the data output to increase the memory performance. Although we can achieve higher clock frequencies by enabling this register, we also add one clock cycle latency to the memory access. This optional register is enabled by default, and we need to disable it in order to do read and write operations at the same clock cycle. Listing 5.3 shows how these setting can be applied to a BRAM macro instantiation.

```
BRAM_SDP_MACRO #(
    .BRAM_SIZE("18Kb"), // Target BRAM, "18Kb" or "36Kb"
    .DEVICE("7SERIES"), // Target device: "7SERIES"
    .WRITE_WIDTH($clog2(C_INPUT_COUNT)),
    .READ_WIDTH($clog2(C_INPUT_COUNT)),
    .DO_REG(0),  // Optional output register (0 or 1)
    .INIT_FILE ("NONE"),
    .SIM_COLLISION_CHECK ("ALL"),
    .SRVAL(72'h000000000000000000),
    .INIT(72'h000000000000000000),
    .WRITE_MODE("READ_FIRST"),  // Operation Mode
```

Listing 5.3. Setting BRAM parameter for histogram design. Using a device macro is a safer choice for mapping the histogram array into BRAMs.

All the settings we discussed above are low-level RTL design decisions, and it is very likely to make mistakes during the design process. Creating an efficient and correctly functioning histogram design in RTL requires a hardware expertise. Table 5.3 demonstrates the implementation results of the RTL design with the correct functionality.

Table 5.3. Reconstructed RTL Design.

	FF	LUT	BRAM	DSP48	Throughput
Reconstructed RTL Design	176	201	1	0	1.758 kHz

As we discussed above, in the RTL design, we were able to eliminate the data hazard by changing the memory operating mode and by disabling the register at the output of the memory. But these are low-level design choices, and HLS isolate the developer from these low-level design details.

In order to eliminate the dependency in the HLS design, we can reconstruct the code in a way that reading and writing to the same memory cell never occurs. To do this, we can use an accumulator to store the histogram value of a pixel temporarily; and instead of accessing to the BRAM consecutively for the same value, we can just increase the accumulator's value. Whenever the system gets a different pixel, the histogram value accumulated in the register will be written to the BRAM. Listing 5.4 shows the reconstructed C code for Vivado HLS. This design is based on [10].

```
for (int i=0; i<IMG_H; i++){
   for (int j=0; j<=IMG_W; j++){
       #pragma HLS PIPELINE
       new_pixel = pixel_in;
       if(old_pixel ==new_pixel ){
           accu_reg = accu_reg + 1 ;
       }
       else {
           hist_reg[old_pixel] = accu_reg ;
           accu_reg = hist_reg[new_pixel]+1;
       }
       old_pixel=new_pixel ;
   }
   hist_reg[old_pixel] = accu_reg ;
```

```

    for(int i=0;i<256;i++){
        hist[i]=hist_reg[i] ;
    }
}
```

Listing 5.4. Using temporary accumulator to resolve the data dependency.

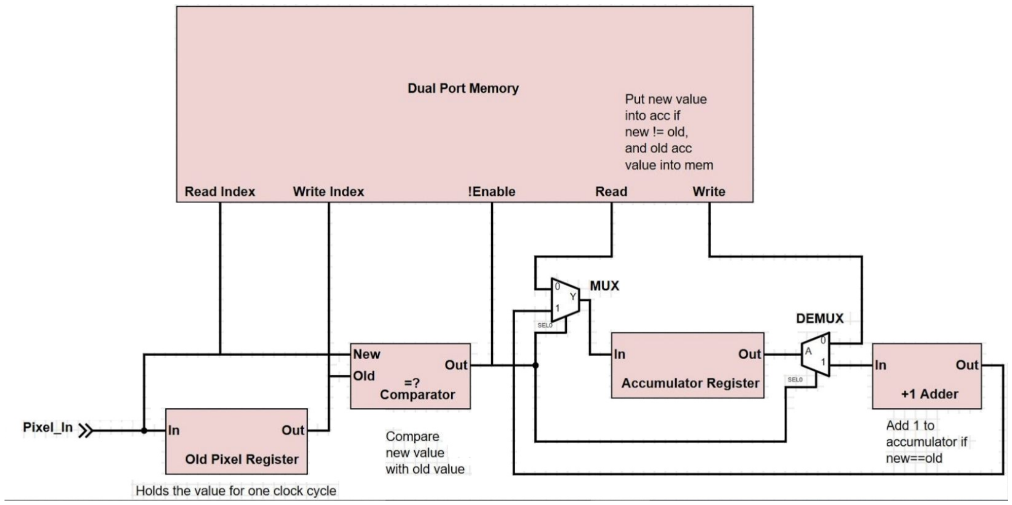

Dual Port Memory
Put new value into acc if new != old, and old acc value into mem
Read Index
Write Index
!Enable
Read
Write
MUX
DEMUX
New
Old
Out
=? Comparator
In
Out
Accumulator Register
In
Out
+1 Adder
Pixel_In
In
Out
Old Pixel Register
Compare new value with old value
Add 1 to accumulator if new==old
Holds the value for one clock cycle

Figure 5.2. Illustration of the architecture created by the reconstructed HLS code.

In HLS, we solved the dependency issue algorithmically without going into the details of the RTL design. Table 5.4 shows the implementation results of the reconstructed histogram design in Vivado HLS. In our RTL design, changing the BRAM operating behavior negatively affected our performance. Solving the dependency issue algorithmically in Vivado HLS created a faster histogram implementation.

Table 5.4. Reconstructed HLS Design.

	FF	LUT	BRAM	DSP48	Throughput
Reconstructed HLS Design	141	214	1	0	1.819 kHz

5.1 Conclusion

Histogram implementation is a potential bottleneck for the rest of the system if not designed carefully. In the RTL design, it is very likely to generate a functionally incorrect implementation if the memory behavior is not considered cautiously. In the case of the HLS design, even though if the developer is not aware of the memory behavior, HLS will automatically recognize the data hazard and will generate a state machine that divides the histogram process into two stages.

In order to achieve an initiation interval of one, we demonstrated the necessary modifications on the memory behavior. While these modifications allowed us to achieve our goal, we ended up having a slower memory structure. By removing the data dependency algorithmically in Vivado HLS, we generated a faster implementation than our RTL implementation. Table 5.5 presents the results of each approach we discussed in this chapter.

Table 5.5. Histogram implementation results.

	FF	LUT	BRAM	DSP48	Throughput
Initial RTL Design	3607	5794	0	0	1.558 kHz
Initial HLS Design	81	80	1	0	1.198 kHz
Reconstructed RTL Design	176	201	1	0	1.758 kHz
Reconstructed HLS Design	141	214	1	0	1.819 kHz

Chapter 6

Summary

In this thesis, we compared implementation details of well-known image processing algorithms using HLS and RTL methods. We explored the advantages and disadvantages of using HLS and RTL for implementing image processing algorithms, and compared them based on resource utilization, performance, and the development process. We also provided guidelines on optimization techniques for each algorithm we studied. While there have been several studies such as [6] about comparing HLS and RTL methods, they do not present details about the implementation differences of both methods.

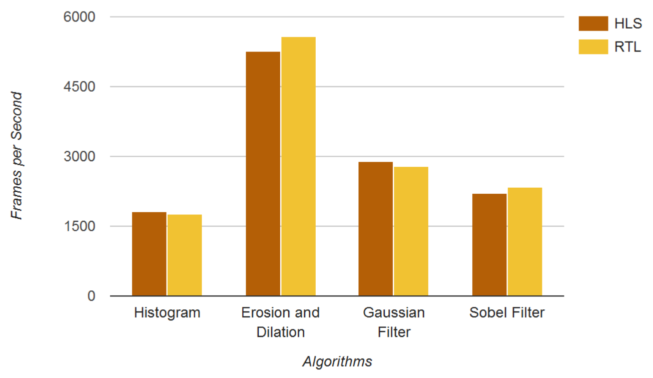


Figure 6.1. Throughput Comparison of HLS and RTL methods used in different algoriths.

We demonstrated that understanding the underlying architecture is critical in both HLS and RTL. While a significant hardware knowledge is required to implement an efficient RTL design, we showed that with the help of HLS tools, developers can generate comparable results without going too deep into the architectural details. In Figure 6.1, we present the performance comparison of the algorithms we explored in this thesis. In the case of the Gaussian filter and the histogram, our Vivado HLS implementations performed better than the RTL implementations.

RTL method provides flexibility and a low-level design approach that can provide the developer with a higher-performance platform. However, the efficiency of the design depends heavily on the developer's level of proficiency in hardware design. Thus, the RTL method has liabilities of potential human errors that can produce systems with lower performance and efficiency than the HLS method.

In fact, technological advancements showed us that under specific circumstances, machines can make better judgments than humans. By isolating the developer from making low-level design decisions, HLS tools have a potential to create more efficient designs.

Bibliography

[1] ARM. *AMBA AXI and ACE Protocol Specification*, d edition, October 2011.

[2] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *2009 international conference on field programmable logic and applications*, pages 126–131. IEEE, 2009.

[3] Christian Banz, Sebastian Hesselbarth, Holger Flatt, Holger Blume, and Peter Pirsch. Real-time stereo vision system using semi-global matching disparity estimation: Architecture and fpga-implementation. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 93–101. IEEE, 2010.

[4] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[5] Catapult high level synthesis. https://www.mentor.com/.

[6] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[7] E Roy Davies. *Computer and machine vision: theory, algorithms, practicalities*. Academic Press, 2012.

[8] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. Riffa 2.1: A reusable integration framework for fpga accelerators. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(4):22, 2015.

[9] Dajung Lee, Pingfan Meng, Matthew Jacobsen, Henry Tse, Dino Di Carlo, and Ryan Kastner. A hardware accelerated approach for imaging flow cytometry. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–8. IEEE, 2013.

[10] Janarbek Matai. *Templates and Patterns : Augmenting High-Level Synthesis for Domain-Specific Computing.* . PhD thesis, UC San Diego, 2015.

[11] Samir Palnitkar. *Verilog HDL: a guide to digital design and synthesis*, volume 1. Prentice Hall Professional, 2003.

[12] Synphony c compiler. http://www.synopsys.com/.

[13] Vivado hls. https://www.xilinx.com/.

[14] Robert A Walker and Raul Camposano. *A survey of high-level synthesis systems*, volume 135. Springer Science & Business Media, 2012.

[15] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Tim Cheng. *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009.

[16] Xilinx. *Zynq All Programmable SoC Sobel Filter Implementation Using the Vivado HLS Tool*, xapp890 v1.0 edition, September 2012.

[17] Xilinx. *Vivado design suite user guide: High-level synthesis*, ug902 v2014.1 edition, May 2014.

[18] Xilinx. *7 Series FPGAs Configurable Logic Block: User Guide*, ug474 v1.8 edition, September 2016.

[19] Zedboard development board. http://zedboard.org/.