Implementing and Comparing Image Convolution Methods on an FPGA at the Register-Transfer Level

by

Anna Celeste Hernandez

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Electrical & Computer Engineering

by

September 2019

APPROVED:

Dr. Zainalabedin Navabi, Thesis Advisor

Dr. William Michalson

Dr. Xinming Huang

Abstract

Whether it's capturing a car's license plate on the highway or detecting someone's facial features to tag friends, computer vision and image processing have found their way into many facets of our lives. Image and video processing algorithms ultimately tailor towards one of two goals: to analyze data and produce output in as close to real-time as possible, or to take in and operate on large swaths of information offline.

Image convolution is a mathematical method with which we can filter an image to highlight or make clearer desired information. The most popular uses of image convolution accentuate edges, corners, and facial features for analysis.

The goal of this project was to investigate various image convolution algorithms and compare them in terms of hardware usage, power utilization, and ability to handle substantial amounts of data in a reasonable amount of time. The algorithms were designed, simulated, and synthesized for the Zynq-7000 FPGA, selected both for its flexibility and low power consumption.

To my late mother, may I live a life she would have been proud of.

Acknowledgements

I would like to express my sincere thanks to my advisor, Zain Navabi. You provided me with the knowledge and tools upon which I grew not only this project, but also myself and my confidence in my craft.

I also wish to thank my wonderful fiance, Chris Jackson, who has supported me in every way imaginable. You have been by my side through thick and thin, something which I can never fully express my gratitude for.

Finally, I acknowledge my friends and family who supported me through this significant chapter of my life. I wish to continue to live up to your expectations of me.

Contents

List of Figures

Chapter 1

Introduction

With the rise of image processing and computer vision, the standard practice of implementing algorithms of Digital Signal Processors (DSPs) and Application Specific Integrated Circuits (ASICs) is losing favor. The shift away from these platforms stems from our desire to balance high performance through parallelism with manageable cost and power requirements.

1.1 Motivation

Whether it's capturing a car's licence plate on the highway or detecting someone's facial features to tag friends, computer vision and image processing have found their way into many facets of our lives. Image and video processing algorithms ultimately tailor towards one of two goals: to analyze data and produce output in as close to real-time as possible, or to take in and operate on large quantities of information offline.

While there are some exceptions, most algorithms require either near real-time computation or the ability to handle high quality images. In real-world scenarios, it is highly unlikely to have the resources to avoid trade-offs. With image and video

sizes growing by the day, the demand to process this massive amount of information becomes more and more demanding.

It is because of these ever-stricter demands we need to take a moment to consider our choice of hardware. When it comes to acceleration, the three platforms to consider are Central Processing Units (CPUs), Graphical Processing Units (GPUs), and Field-Programmable Gate Arrays (FPGAs).

This project examines multiple forms of image convolution and implements each for the Zynq-7000 FPGA. Each algorithm is then analyzed and compared to find out when one is better suited over another for various image compositions.

Chapter 2

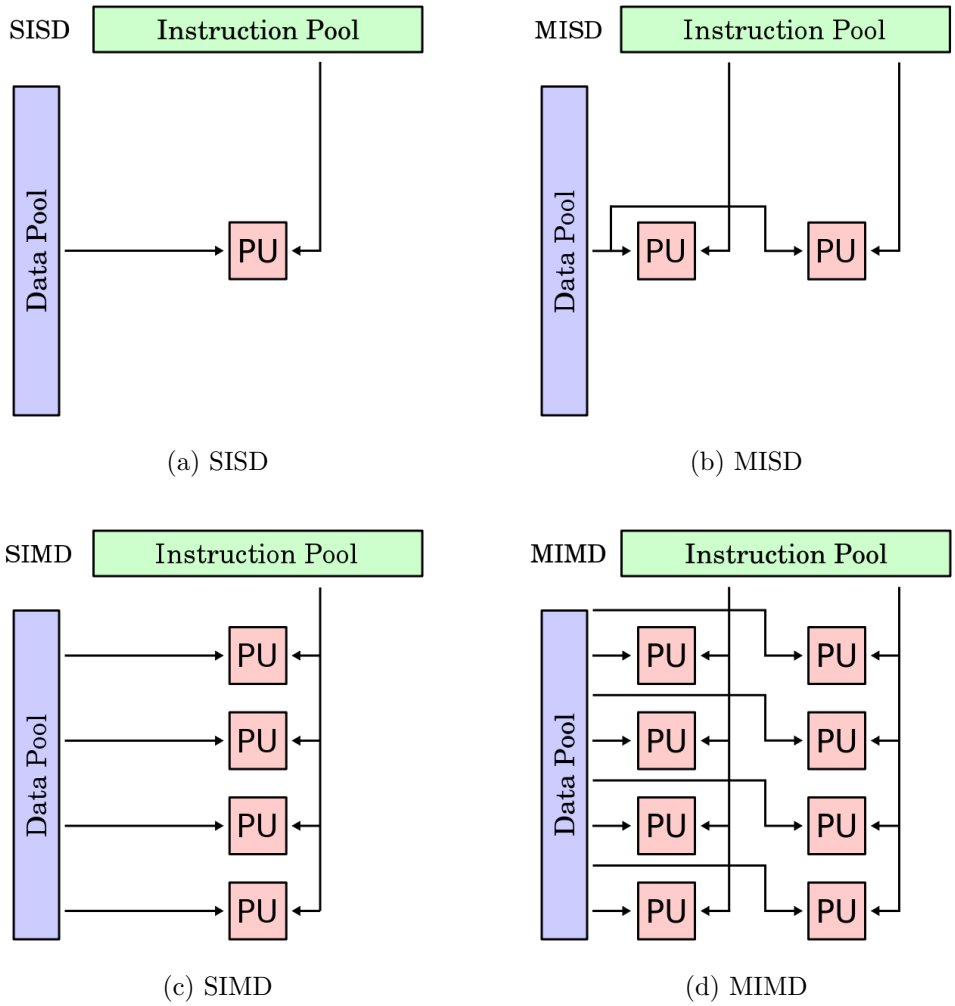
Tools and Environments

When evaluating platforms, we must also examine which methods each one utilizes to accelerate hardware. Parallelism largely exists in three forms: Pipelining, Single-Instruction Multiple-Data (SIMD), and Multiple-Instruction Multiple-Data (MIMD) execution. [11]

2.1 Platform Selection

Typically, given four different instructions to complete, the simplest method would be to take the first, see it through completion, grab the second, see it through completion, and so on. Each instruction has the potential to access various sets of modules through our datapath, considering it way or may not perform arithmetic or access memory blocks.

This methodology, Single-Instruction Single-Data (SISD), is extremely reliable as there is no juggling of memory operations. That being said, if each instruction takes five clock cycles to complete, it will take twenty cycles to perform four instructions. We think to ourselves, what if we could do more at once? Figure 2.1 shows four solutions for increasing efficiency which will be discussed in this chapter.



(a) SISD

(b) MISD

(c) SIMD

(d) MIMD

Figure 2.1: Block diagrams of various taxonomies

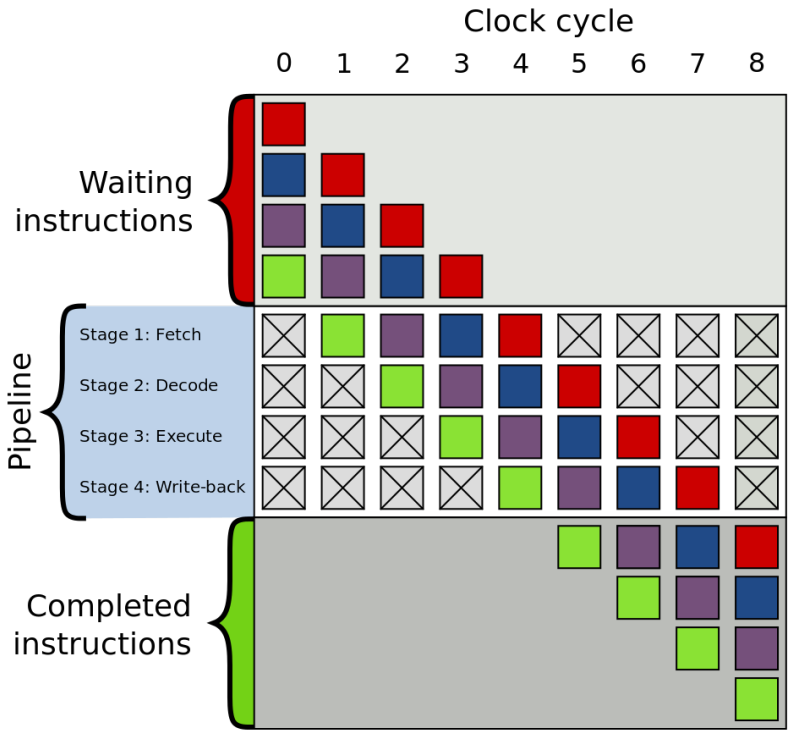
Perhaps the simplest way to speed arithmetic operations would be to simply take your datapath and implement it twice, effectively creating a copy of each RTL element except for the shared memory. By doing this, you could simply feed all the odd-numbered instructions to the first datapath instantiation and all the even instructions to the other, using one controller to control both datapaths. This method, SIMD, is best suited for arithmetic instructions that do not rely on the adjacent instructions. GPUs excel in churning through this form of instruction, but are extremely power hungry and costly as they are designed to be very wide. [13]

While SIMD can be useful for combination and arithmetic operations, it is limited in what instruction types allow for such handling. Rather than make all datapaths depend on the same controller, a second option is to give each datapath its own controller. This Multiple-Instruction Multiple-Data (MIMD) solution is standard in most modern CPUs. [8] With the recent refining of multi-core CPUs and additional/extended instructions, cores can be properly managed to better utilize their ability to multitask. CPUs are less pricey and better suited to general needs than GPUs are. They can, at the same time, be designed to share fewer datapaths and cut down on power usage by properly scheduling their tasks. [15]

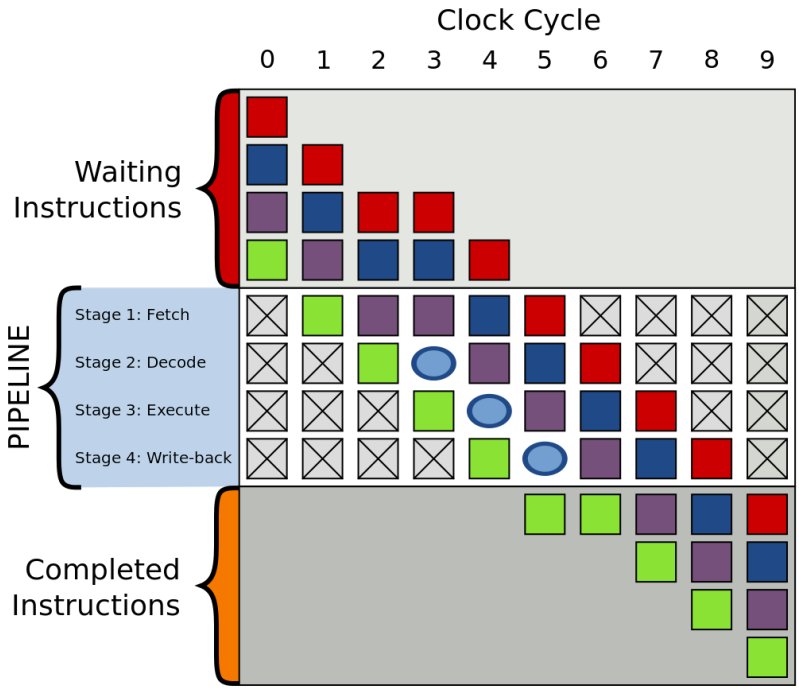
When it comes to cutting down on cost and redundant hardware, out last option is to focus on obtaining the best throughput with a single datapath. This approach, called pipelining, has the goal to keep each RTL element as busy as we can by continuing to fetch new instructions to start as previous instructions are still processing later on in the datapath. This being said, there are sometimes cases where we must wait a clock cycle to avoid conflicts and maintain data integrity. When this "bubble" is inserted tasks are time-delayed one stage as shown in Figure 2.2. [18]

MIMD and Pipelining are similar in that they both attempt to keep each part of the hardware reasonably busy, but the key difference is that pipelining can be done with a singular controller and datapath. [13] While CPUs also pipeline their instructions, they come pre-built with a certain number of cores and a built-in instruction set. Using general solutions may drive up the cost of implementation where slimmer, tailored solutions are available.

With the ability to instantiate a minimalist instantiation of application specific components without incurring the cost of printing chips, field-programmable gate arrays provide this low-cost, tailored solution. With FPGAs we can implement as many copies of each specific RTL component as we want as well as take part in how



(a) Perfect



(b) Bubbled

Figure 2.2: Pipeline execution

to schedule instructions to run simultaneously. It is due to this inherent flexibility to control both timing and power this project is instantiated on an FPGA.

Figure 2.3 shows a visual comparison of SISD, SIMD with a double-wide input, MIMD on a 2-thread CPU, and a single pipelined datapath on FPGA in the context of clock cycles. The figure displays four tasks which each take five stages to complete and how each datapath handles the assigned work.

Clock Cycle	1	2	3	4	5	6	7	8	9	10	11	12
SISD (Default)	1-A	1-B	1-C	1-D	1-E	2-A	2-B	2-C	2-D	2-E	3-A	3-B
SIMD	1-A	1-B	1-C	1-D	1-E	3-A	3-B	3-C	3-D	3-E		
(GPU, 2x wide)	2-A	2-B	2-C	2-D	2-E	4-A	4-B	4-C	4-D	4-E		
MIMD	1-A	1-B	1-C	1-D	1-E	3-A	3-B	3-C	3-D	3-E		
(CPU, 2 threads)		2-A	2-B	2-C	2-D	2-E	4-A	4-B	4-C	4-D	4-E	
	1-A	2-A	3-A	4-A								
Pipelined		1-B	2-B	3-B	4-B							
(FPGA, 1 datapath)			1-C	2-C	3-C	4-C						
				1-D	2-D	3-D	4-D					
					1-E	2-E	3-E	4-E				

Figure 2.3: Timing SISD, SIMD, MIMD, and Pipelining

2.2 Verilog and RTL

Whether a layperson or a hardware designer, is all too easy to forget that each piece of hardware is just a collection transistors and wires. Be it a single not gate, a four-function calculator, or a fully fledged CPU, all share this same composition when broken down to their simplest level. When transistors were large enough to

hold and our computing needs a few dozen transistors complex, we once wired digital logic manually.

As the years have passed and our needs grow ever-more insatiable, we have seen a single CPU possess thousands, millions, and now billions of transistors. Accompanied with shrinking transistor size, it is simply no longer feasible to build, prototype, and test hardware by hand. Even giving ourselves the luxury of working at the gate level or having pre-made registers, the sheer amount of time and labor required to build one version of a circuit by hand is enormous. [9]

It is for this reason Verilog and Verilog Hardware Description Language (VHDL) were developed. Appearing as early as 1984, Prabhu Goel, Phil Moorby, Chi-Lai Huang, and Douglas Warmke developed a language designers could use to describe how they would construct and connect hardware. [2] While often mischaracterized as a programming language, this hardware description language highlights and implements one key property of hardware that no programming language could emulate - concurrent signal assignments. Not only can transistors, gates, registers, and wires be placed by a single line of Verilog code, but multiple components can receive and process signals simultaneously. [9] This is as opposed to having to process assignments line-by-line as you move sequentially through the program, as programming languages are structured to do.

Figure 2.4 below shows an adder module with carry-in and carry-out in Verilog at the RT-Level. This module takes two four-bit numbers and a one-bit carry-in value and produces their sum as a five-bit output (including the carry-out bit) which is resistant to overflow.

At the present day, most hardware designers write in Verilog or VHDL at the Register-Transfer Level (RTL). This scope allows us to make components such as shift registers and RAM blocks out of gates and flip-flops and subsequently tie those

```
module Full_Adder4b(
    input [3:0] a, b,
    input c_in,
    output c_out,
    output [3:0] sum );

  assign {c_out, sum} = a + b + c_in;

endmodule
```

Figure 2.4: Sample code of 4-bit Adder in Verilog

components to one another. In this way we are creating our own building blocks chaining them together to create more complex designs. Should one part need to be expanded on or modified, it can be removed and the architecture rebuilt to accommodate these changes. [3] The attention required is still somewhat significant, but is much less cumbersome now that we have abstracted away worrying about individual transistor placement.

2.3 SystemC and OpenCV

Much like we have abstracted the majority of our work from transistors to gates to registers to arrive at RTL, we can raise our abstraction one level higher. Larger elements such as RAMs, simple CPUs, and I/O are prime candidates for further abstraction as they are large and complex, but generally already solved. Most memories, for example, have a set way they operate and normally only change various parameter sizes.

SystemC is a freely available code repository that provides the tools we will use to design at the electronic system level (ESL). It was first released in 2000 by a collection of companies that banded together to form the Open SystemC Initiative (OSCI), now known as Accellera. Curiously, the SystemC class library is written in C++, a sequential programming language. As discussed earlier, C/C++ was not

designed to handle concurrent signals. Everything, including 4-value logic (0. 1. X. Z), vectors, integers, ports, events, and sensitivity was given its own class in order to emulate concurrency and hardware. [1]

Figure 2.5, shows the SystemC description a four-bit full adder. Its functionality is equivalent to the adder shown in Figure 2.4.

```
#include "systemc.h"

#define WIDTH  4

SC_MODULE(Adder) {
  sc_in<sc_uint<WIDTH>> a, b;
  sc_out<sc_uint<WIDTH>> sum;

  void process() {
    sum.write(a.read() + b.read());
  }

  SC_CTOR(Adder)       {
    SC_METHOD(process);
    sensitive << a << b;
  }
};
```

Figure 2.5: Sample code of 4-bit Adder in SystemC

It should be noted that although commonly referred to as its own language, SystemC is actually a host of C++ classes. While it does define and heavily rely on many of its own classes, it still supports the usage of C++ data types. [5] This allows us to meld it with other C++ libraries, such as Intel's open source computer vision (OpenCV) library.

OpenCV is a collection of C++, Python, and Java functions. While it does provide some framework of its own, it does not try to emulate concurrent functionality like SystemC does. This is an open source library from which we can build up computer vision and deep learning algorithms. While initially designed to accelerate CPU-centric applications, it has since been widely utilized for GPUs and has a few FPGA-friendly functions. [4]

2.4 Vivado HLS

Vivado HLS is an integrated design environment (IDE) in which designers can describe their hardware in C, C++, and SystemC at a high level of abstraction. Because OpenCV and a similar class library, OpenCL, are C-based, they are also accepted by the compiler.

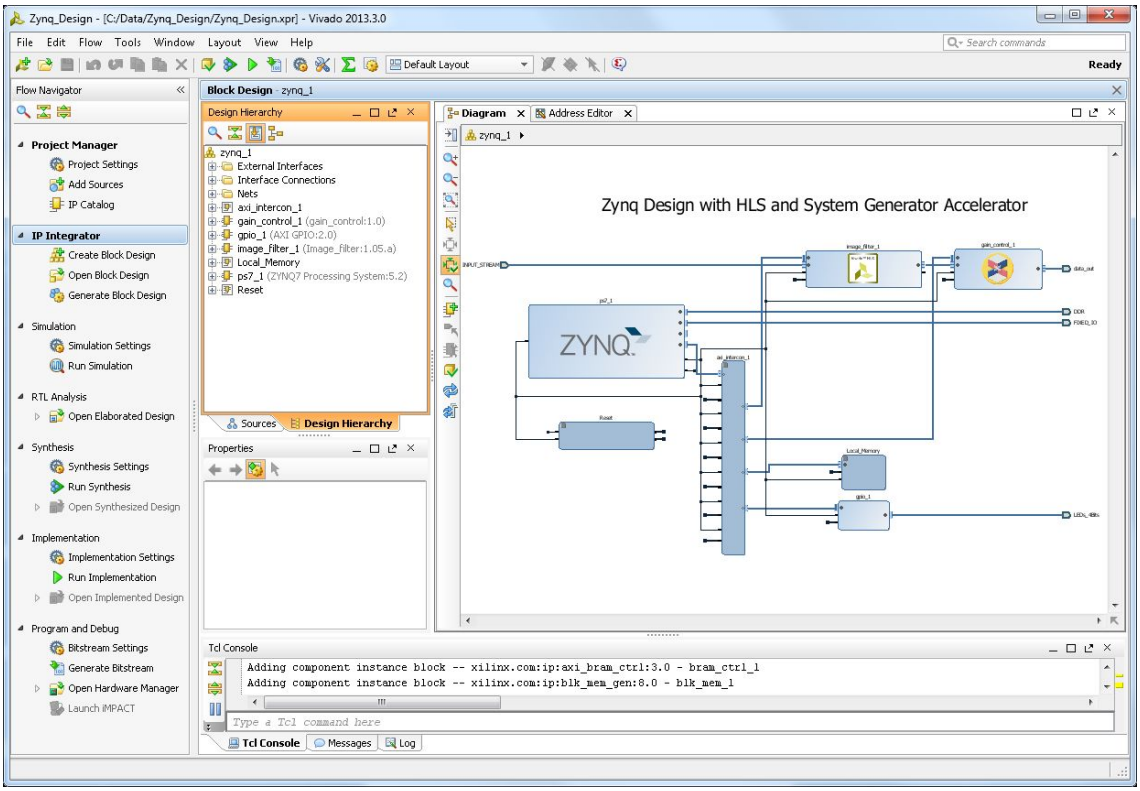


Figure 2.6: A Look at the Vivado HLS Interface

While the IDE provides many helpful functions such as debugging, simulation, and synthesis, it is tightly coupled to Xilinx's chip architecture. Due to this, our high level description and generated IP core must be implemented on an Ultrascale, Virtex, Kinetix, Artix, or Zynq FPGA. This becomes notable later as Xilinx's chipset uses MicroBlaze softcore processors, which highly discourages the implementation of custom instructions (unlike other softcore processors such as NIOS).

Chapter 3

Image Convolution Algorithms

In this report we implement and explore eight different forms of image manipulation. Our goal is to detect edges in each image frame and benchmark the performance of each algorithm on an FPGA. Edge detection allows us to store the strongest features of each frame and use their location within the image to track them as they move between frames. Before we begin, we must first understand what it means to convolve an image.

3.1 Basic Convolution

Convolution is one of the forms of mathematics we use to blend or merge two datasets into one. This is most simply achieved via point-wise multiplication and summation. [20]

In one dimension the formula for convolution in the continuous domain is as follows:

$$r(i) = (s * k)(i) = \int s(i-n)k(n)dn$$

Computers are discrete by nature, as will our design be. To represent this formula

in the discrete domain would be:

$$r(i) = (s * k)(i) = \sum s(i - n)k(n)$$

Both of these formulae ban be expanded into the second dimension by adding an additional variable. In discrete terms, the two dimensional formula would be:

$$r(i) = (s * k)(i, j) = \sum \sum s(i - n, j - m)k(n, m)$$

To better conceptualize this, consider a 7x7 input matrix of values and a smaller 3x3 matrix that you can superimpose or "slide" over it. While over one section of values, we multiply each overlapping value and store them in an intermediate 3x3 matrix we label our kernel. We can now take every value in the kernel matrix, sum them together, and place this singular value in our output 7x7 matrix. The location it is placed in will correspond to the central cell of where our sliding window matrix was. We can now shift our sliding matrix one space to the right and repeat this process to get the next output cell's value. [6] Figure 3.1 illustrates the process.

Using this method, our first convolution window will center around the pixel in the second column and second row. This results in the lack of data on the border cells of the output matrix, leading our 7x7 input matrix to be truncated to a 5x5 output. To help circumvent this unwanted cropping, we can add in zero padding around our original 7x7 matrix to turn it into a 9x9 matrix. After computation, our output will then be the desired 7x7 size.

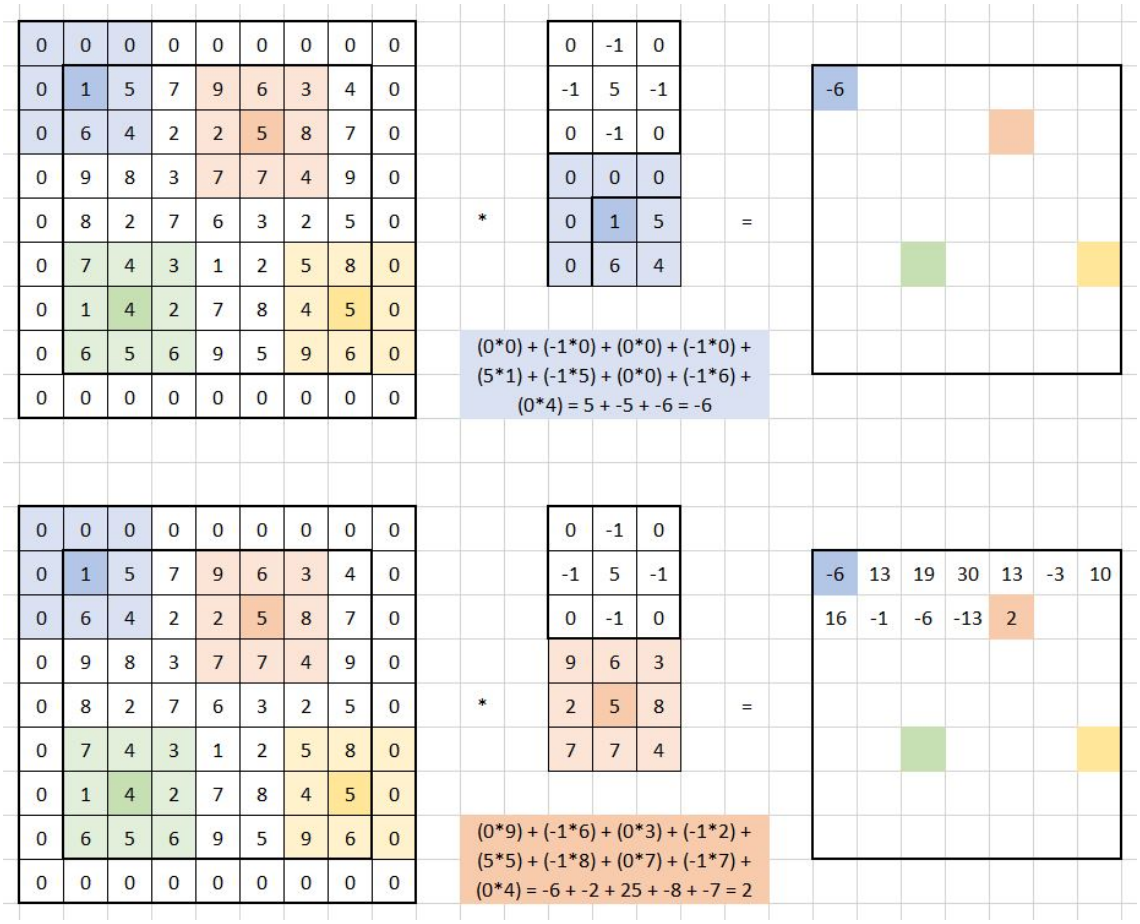


Figure 3.1: Two-dimensional Convolution

3.2 Identity

As to how this applies to images, recall that images have a finite resolution and that a 1920x1080 pixel image is but a 1920x1080 array of color values. While many file formats add on extra information or encode the pixel data to compact their file size, a bitmap (.bmp) or portable graphic format (.png) image are the most straightforward filetypes to work with.

To delve a bit further into the makeup of images, consider what information it takes to represent a color. Color is what we see as a result of various wavelengths of light interacting with a surface. Humans are most sensitive to the spectra around

reds, around greens, and around blues. Modern cameras measure color as the intensity of red light it detects, the intensity of green light, then the intensity of blue light. Each of these intensity values can then be combined into what we perceive as one color. [14]

Rather than store all possible colors as one, continuous set, we choose to keep representing them as triplet of their red, green and blue (RGB) light components. This information is stored as three 8-bit values, meaning each pixel contains 24-bits of data.

A 1920x1080 image is then a 1920x1080 matrix where each cell contains 24-bits of information. In the interest of cutting down on hardware and increasing throughput, most computer vision algorithms will convert a color image to grey-scale, needing only a single 8-bit value to represent each pixel. [7] Our design follows this methodology, though it is worth noting that this is not a hard requirement of convolution, but a design choice that helps us immensely in our search for real-time execution.

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

Figure 3.2: Identity Kernel Window

The simplest example of image convolution is the identity convolution. When choosing the values we wish to convolve our input image with, we can choose to simply null all values except for the center value, which we multiply by one. When the intermediate values are summed, the result to be placed will be identical to the central cell convolved. The result is that the output image is identical to the input image so long as the input image was zero-padded. Figure 3.2 shows the convolution process and padding of the matrix.

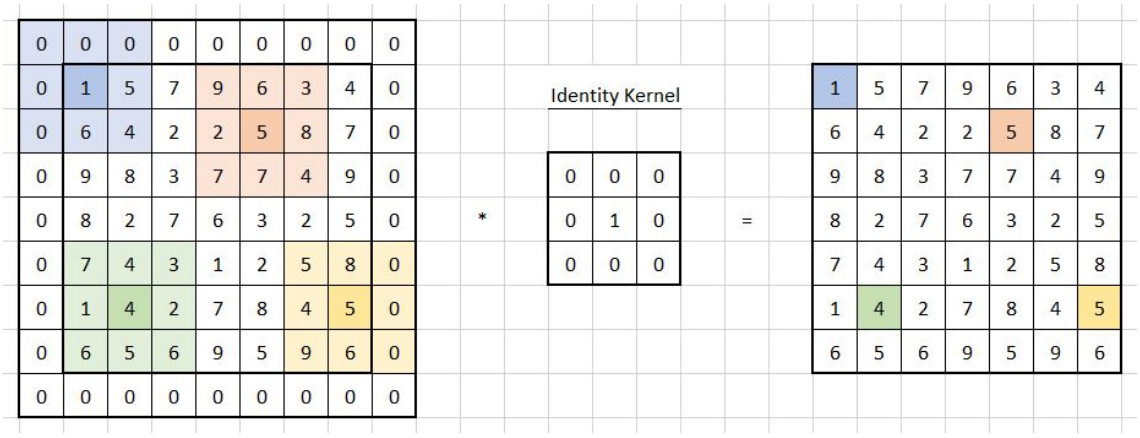


Figure 3.3: Process of 2D convolution with Identity Kernel

3.3 Box Blur

When it comes to algorithms, many will call for the image to be blurred as a preliminary step. The two most common ways to accomplish this are by using a Gaussian blur or by convolving the image with the box blur kernel. While there is a kernel that attempts to emulate the Gaussian blur's results, for this project we opt to use the box blur. [12]

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

Figure 3.4: Box Blur Kernel Window

One major strength this kernel has over the Gaussian approximation kernel in terms of hardware is that it can be represented in hardware as an accumulator, rather than a full convolution kernel, as the results will be equivalent.

This kernel takes the pixels within its window and average them with equal weight. The result is that each output pixel becomes the average of its equivalent input pixel and their neighbors, "smudging" the values together. The purpose of

blurring images is to reduce any noise that may be in the image. This prevents many very sensitive edge detectors from picking up unwanted or "false" edges. [12]

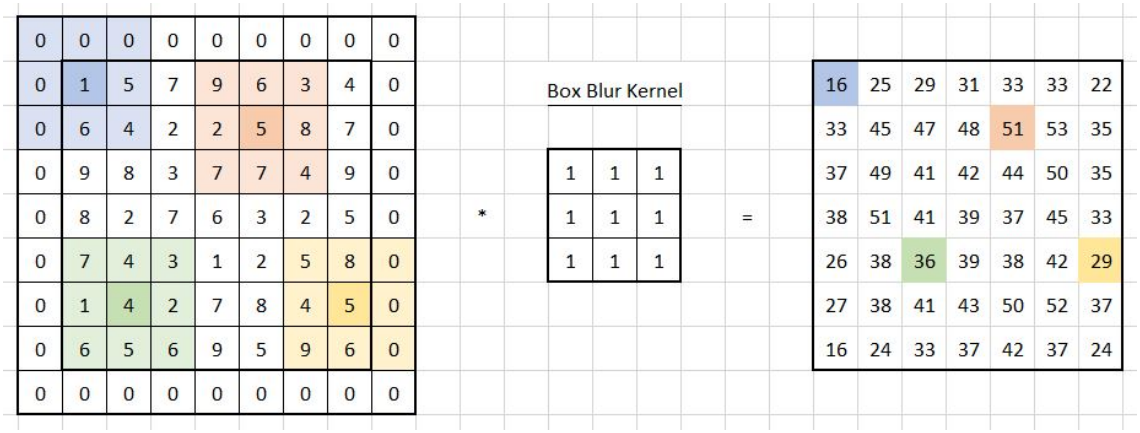


Figure 3.5: Process of 2D convolution with Box Blur Kernel

3.4 Sharpen

Another preliminary step some algorithms may require is to have the input passed through a sharpening kernel. To sharpen an image, as one may expect, is to do the inverse of blurring it. Images that may already be scant in detail should be placed through a kernel that helps accentuate any differences in the image so that less sensitive kernels will pick them up. This kernel will accentuate the central pixel by multiplying it by five, multiply the directly adjacent cells by negative one, and zero out the corner values. The result is that the central cell is accentuated, then tempered by its cardinal neighbors.

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Figure 3.6: Sharpen Kernel Window

This, while not normally called as such, incorporates the discrete, 2D approximation of the Laplace transform (also known as the z-transform). It is, by definition, the sum of the second derivative of the central pixel. Though, since our central value is five and not four it is the Laplace transform added to the identity filter, resulting in our detected edges being overlaid on top of the original image. This is what creates the sharpening or "enhancing" effect. [10]

3.5 Edge

We arrive at what most consider the most simple edge detection kernel. Much like the sharpening kernel, it is another approximation of the Laplace transform. At first glance, it seems quite similar - the central pixel is multiplied by eight while every neighbor is multiplied by negative one. Unlike the sharpening kernel, this does include the diagonal neighbors and does not overlay itself over the identity kernel. Because of this, it is a truer Laplace transform and shows us only the detected edges. [19]

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

Figure 3.7: Edge Kernel Window

3.6 Prewitt

Each kernel we have looked at thus far involves one convolution at a time. While the Laplacian-based kernels rarely call for blurring or sharpening, the Prewitt and subsequent kernels would all benefit greatly from being blurred before input. That

is because the following algorithms will each use two kernels, one to detect edges in the x-direction and another in the y-direction, making them much more susceptible to noise. Using the term noise may be misleading, as few images have literal noise in them; the most practical time to apply smoothing is when images have blades of grass or sandy beaches in the background that you do not wish to capture. [19]

The most naive of the kernels that utilizes this tactic it the Prewitt operator. This operator convolves the input image with a kernel that seeks continuity changes in the x-direction. Simultaneously, it convolves the input image with a y-direction filter. These intermediate results are then taken, squared, summed, and then put under a square root. This gets the Euclidean norm of the x-cell and y-cell to place as the final value in the output matrix.

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

Figure 3.8: Prewitt Kernel X and Y Windows

While it does take two, directional derivatives, it is not a true derivative operator, as you cannot change the weights to detect finer edges. It's strength is that the filters themselves can be simply implemented in hardware with a sign change module and accumulators. Because the Euclidean norm alone can require a lot of hardware to compute, designers may instead take the absolute value of the x-cell and y-cell then sum then for a much rougher magnitude approximation. [21]

3.7 Sobel

The Sobel operator, often called the Sobel-Feldmann operator, was the result Irwin Sobel and Gary Feldmann's doctoral research in 1968. They sought and suc-

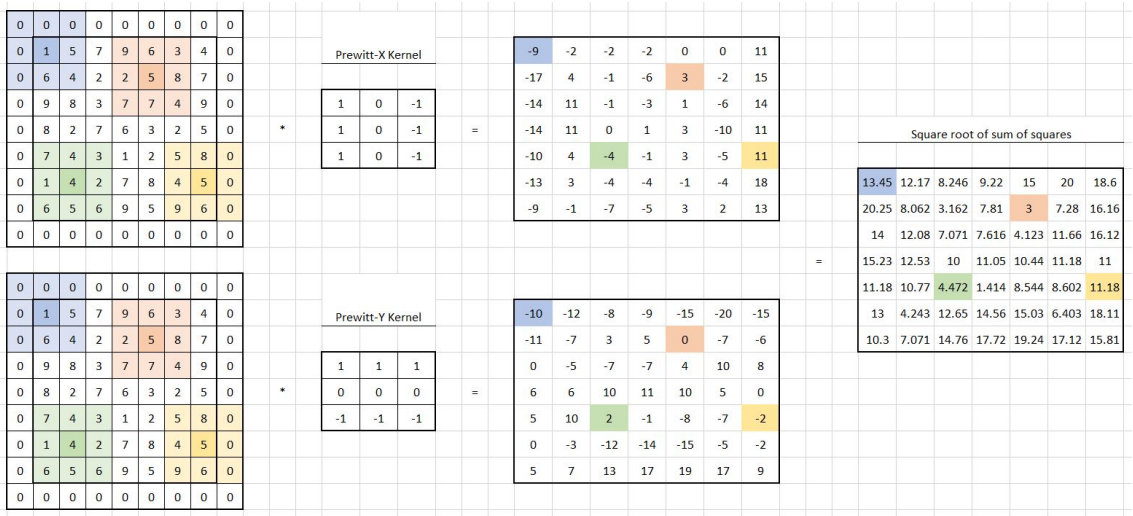


Figure 3.9: Process of 2D convolution with Prewitt Kernel

cessfully derived an image gradient operator that was both computationally efficient and reasonably isotropic. It operates largely like the Prewitt operator though because the coefficients in Sobel and Feldmann's algorithm can be scaled up to produce stronger edges, it is the simplest, true two-dimensional derivative operator.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 3.10: Sobel Kernel X and Y Windows

Sobel and Feldmann did take their computational efficiency seriously when developing this kernel. The values chosen in the windows are all powers of two, allowing for quick and easy binary multiplication via bit-shifting. Similarly, they also recommend using the sum of the absolute values of the x and y gradients to obtain the final gradient. [17]

One can also take the arc-tangent of the y-cell over the x-cell to get the direction of the gradient. A theta of zero, for example indicates a vertical edge which is lighter to its right. Conversely, a theta of 180 would represent a vertical edge where the

lighter side is to the left. While this gradient direction can be extremely valuable in lowering false positives on edge detection, it is also the most sensitive to noise and should never be used on a non-smoothed image. Most interpretations will neglect this component due to its susceptibility to noise.

3.8 Scharr Filters

Hanno Scharr sought to take the Sobel-Feldmann operation as a base and modify it for accuracy. In fact, his paper claims errors are decreased by up to three magnitudes over the standard Sobel parameters. This error reduction is mostly in the estimations involved with combining the two gradients. Figure 3.12 shows the most popular Scharr operands. [16]

$$\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix} \begin{bmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

Figure 3.11: Most Common Scharr Kernel X and Y Windows

Scharr also took it upon himself to find the ideal 8-bit operators, to best maximize the accuracy of systems based on 8-bit registers (as each color value ranges from 0-255). This does require extra hardware, but is a sensical addition if you have the luxury. [16]

$$\begin{bmatrix} 47 & 0 & -47 \\ 162 & 0 & -162 \\ 47 & 0 & -47 \end{bmatrix} \begin{bmatrix} 47 & 162 & 47 \\ 0 & 0 & 0 \\ -47 & -162 & -47 \end{bmatrix}$$

Figure 3.12: Ideal 8-bit Scharr Kernel X and Y Windows

Chapter 4

Design and Implementation

When it comes to computer vision, we must handle the sheer amount of information and hardware involved in convolving a 1920x1080 image first by the smoothing kernel then passed to both the x and y filters, followed by computing the combined result of said images. Add in zero padding and transforming the image to grey-scale, and we have ourselves quite the task.

The design accepts the image as a raster-scan ordered stream of pixel data. The images to be input will be formatted as PNG files which have 24-bits of information per pixel - three 8-bit color channels. The HLS circuit will accept one, 24-bit wide pixel per clock cycle and do the greyscaling process internally. The RTL version of the circuit will not do this and expect a pre-greyscaled PNG file with only 8-bits of information per pixel.

The HLS circuit accepts one 24-bit wide pixel per clock cycle, converts the pixel to greyscale as it enters and stores it in the windowing module (made of line buffers). Once an initial wait period passes that allows the windowing module to fill with enough data to process the first convolution window, the convolution module starts processing the provided information at the rate of one convolution window per clock.

Should the operation require two cores, two convolution modules are implemented, each receiving the same inputs from the windowing module. In the case of single core convolution, the output is then shown. In the case of dual core convolution, a Euclidean magnitude approximation module processes the convolution modules' results and provides the output.

We can design the datapath largely at the register-transfer level, planning out how to best tackle this issue. Our largest concerns will focus around those components most integral to our design. When it comes to image convolution, the trickiest and most involved module will be the one that forms our sliding window.

4.1 Designing the Window Buffer

Because we do not want to store every pixel of every image into memory, we need to consider what information we use at any given window. For our report, all sliding windows will be 3x3 matrices, therefore the simplest conclusion would be that we can simply fetch three full rows of pixel information at a time.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Figure 4.1: Naive Conception of Windowing Module

While nice on paper, this does not reconcile well with how we want to push information through our datapath. Rather than have whole periods of time dedicated to shifting in a row, we devise a better way to handle our inputs. The stress here is less on the number of pixels stored, but the timing in how to capture them effectively.

First, we divert to discuss how the windowing buffer retains and orders the information. As seen earlier, while we retain multiple rows of information at once, the window does not look at one, contiguous block of memory. For this, we must use multiple line buffers that feed in to one another while simultaneously allowing some data to feed out into storage registers for the kernel to access.

Each line buffer acts largely as a shift register or FIFO does, allowing information to shift in a pixel at a time. The width of how many pixels it should store at a time is directly related to how wide the image is. Figure 4.2 below shows the RTL process statements of a single line buffer in Verilog. Breakout delay components are implemented one module higher when the line buffers are tied together.

```
always @(posedge clock, negedge reset)
begin
   if (reset)
      r_reg <= 0;
   else
      r_reg <= r_next;
end

assign r_next = {din, r_reg[(BITS*COLS)-1:BITS]};
assign dout = r_reg[BITS-1:0];
```

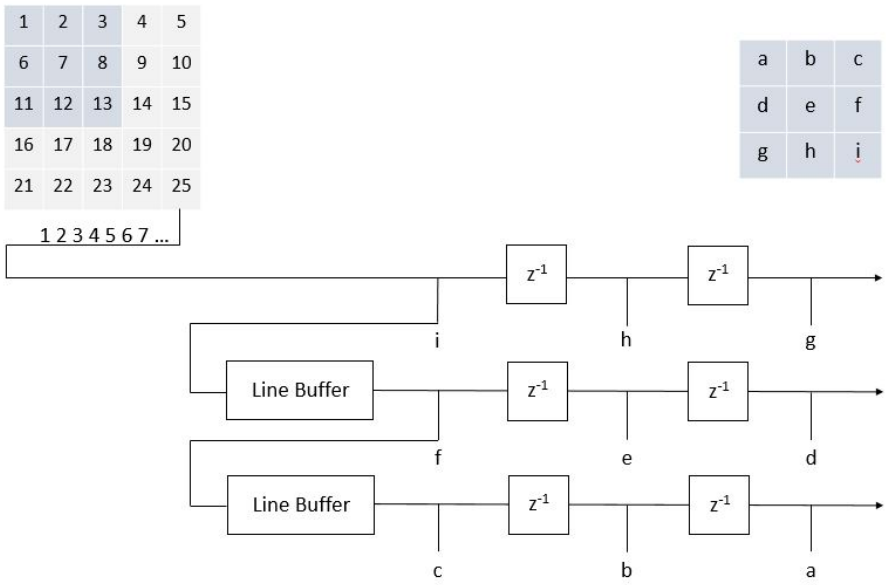
Figure 4.2: Line Buffer Process Statements

The smallest possible amount of information we can have for any frame is as shown in Figure 4.3. We will need the first pixel to sampled by the window, the last pixel to be sampled, and everything in between. This would require two full lines of buffered inputs plus only the width of the kernel (three, in our case) to fill the top line.

With each clock comes a new pixel that will push the bottom-right corner's value in and pop the top-left corner's value out. This means the only wait period required for our windowing module is the initial loading of the first set of pixels.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

(a) Pixels Stored



(b) RTL Architecture

Figure 4.3: Ideal Windowing Module

This RTL implementation is shown partly in Figure 4.4 below. Here, the incoming pixel information goes into the first line buffer and the first row of delay

components. The first line buffer cascades into the second while the delay components pass the information amongst themselves.

```
reg [7:0] a = 0;  reg [7:0] b = 0;  reg [7:0] c = 0;
reg [7:0] d = 0;  reg [7:0] e = 0;  reg [7:0] f = 0;
reg [7:0] g = 0;  reg [7:0] h = 0;  reg [7:0] i = 0;


lineBuf #(COLS, BITS) lineBuf0(clk, rst, i, dout0);
lineBuf #(COLS, BITS) lineBuf1(clk, rst, dout0, dout1);
```

(a) Declare Delay Components, Tie Line Buffers

```
else begin
   a <= b;  b <= c;  c <= dout1;
   d <= e;  e <= f;  f <= dout0;
   g <= h;  h <= i;  i <= pixIn;
end
```

(b) Delay Components Pass Information

Figure 4.4: Ideal Windowing Module RTL Code

When drafting our first module at the HLS level, our implementation alters slightly. Vivado HLS has pre-built line buffer and window modules we can utilize for our windowing module, however it implements one line buffer and three delay modules more than our ideal does. The initial wait time will increase a bit due to the additional buffer, but once loaded it will run at the same speed as the ideal windowing module. The implemented module is encapsulated in just two lines of code, shown in Figure 4.5.

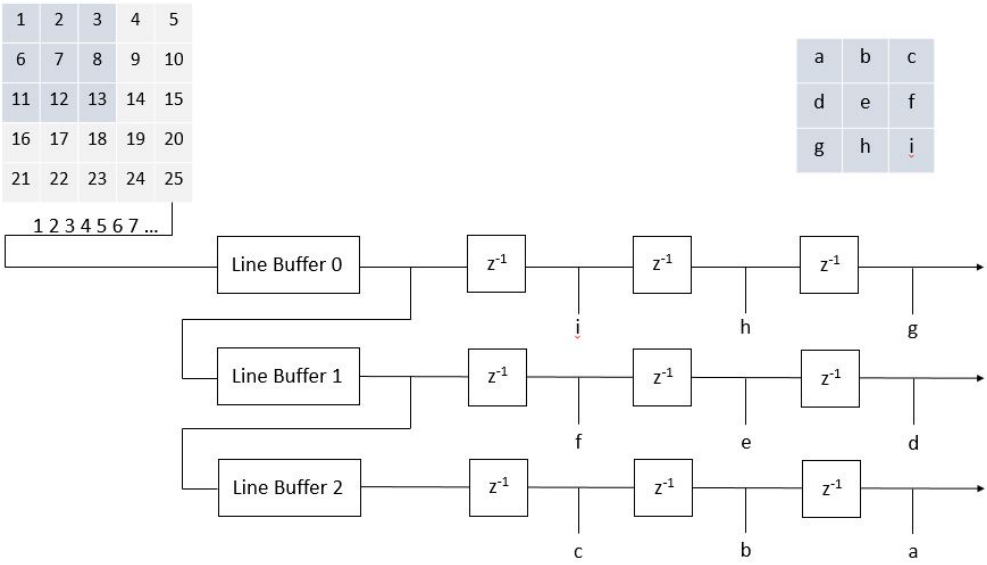
```
hls::LineBuffer<KERNEL_DIM,IMG_WIDTH_OR_COLS,unsigned char> lineBuff;
hls::Window<KERNEL_DIM,KERNEL_DIM,short> window;
```

Figure 4.5: Implemented Windowing Module HLS Code

The architecture change is slight, but has shifted a bit to incorporate the new line buffer and delay components. The block diagram relating to the HLS instantiated hardware is displayed in Figure 4.6.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

(a) Pixels Stored



(b) RTL Architecture

Figure 4.6: Implemented Windowing Module

4.2 Designing the Generalized Convolution Core

Convolution itself is a loop-intensive process, which is not ideal for real-time hardware scenarios. The general method for one-dimensional convolution is as described in the pseudo code in Figure 4.7 below. Note that while 1D convolution

requires a double-nested loop, the main function of this loop is not reliant on its previous iterations. It instead uses the loops to access the input elements one-by-one.

```
1 kSize = 3;                    //Define 3x1 kernel
2 numVals = 12;                 //Define num of input values
3 for (i=0; i<numVals; i++) {   //Output num
4     out[i] = 0;
5     for (j=0; j<kSize; j++) { //Kernel index
6         out[i] += in[i] * ker[j];
7 }
```

Figure 4.7: Psuedo Code for 1D Convolution

We can thusly unroll this loop such that we access each input element at once and perform our computation in parallel as a SIMD instruction. The series used to describe 1D convolution is:

$$y(i) = \sum a(i) * k(j)$$

In which the first output value is expanded as:

$$y(0) = a(0)k(0) + a(0)k(1) + a(0)k(2)$$

Where y is the output value, a is the input value, and k is the kernel weight. This expansion shows that we could take as long as kSize (3) clock cycles per input with one multiplier and full-adder. With SIMD parallelization we can complete the operation in as fast as one clock cycle per pixel, at the cost of hardware - three multipliers and two full-adders.

The case for two-dimensional convolution, furthermore, is generally coded as a quadruple-nested for loop. The pseudo code for standard 2D convolution is shown in Figure 4.8 below.

```
1  kSize = 3;                    //Define 3x3 kernel
2  numCols = 256;                //Define image width
3  numRows = 256;                //Define image height
4
5  for (i=0; i<numRows; i++) {          //Output row
6      for (j=0; j<numCols; j++) {      //Output column
7          out[i][j] = 0;
8          for (m=0; m<kSize; m++) {        //Kernel row
9              for (n=0; n<kSize; n++) { //Kernel column
10                 out[i][j] += in[i][j] * ker[m][n];
11             }
12         }
13     }
14 }
```

Figure 4.8: Psuedo Code for 2D Convolution

The series describing two dimensional convolution is:

$$y(i,j) = \sum\sum a(i,j) * k(m,n)$$

In which the first two output pixels expand as:

$$\begin{aligned} y(0,0) = {} & a(0,0)k(0,0) + a(0,1)k(0,1) + a(0,2)k(0,2) + \\ & a(1,0)k(1,0) + a(1,1)k(1,1) + a(1,2)k(1,2) + \\ & a(2,0)k(2,0) + a(2,1)k(2,1) + a(2,2)k(2,2) \end{aligned}$$

$$\begin{aligned} y(0,1) = {} & a(0,1)k(0,0) + a(0,2)k(0,1) + a(0,3)k(0,2) + \\ & a(1,1)k(1,0) + a(1,2)k(1,1) + a(1,3)k(1,2) + \\ & a(2,1)k(2,0) + a(2,2)k(2,1) + a(2,3)k(2,2) \end{aligned}$$

Which would take kSize*kSize (9) clock cycles per input with one multiplier and full-adder. To do the same computation at the rate of one clock cycle per pixel

would use nine multipliers and eight full-adders.

One issue with the two-dimensional convolution algorithm as presented above is that the first convolution output, while labeled as y(0,0), actually centers around input pixel a(1,1). The background chapter of this report goes further into detail about this unintentional cropping and methods to prevent it. The modified 2D convolution algorithm we use to address this is as shown in Figure 4.9.

```
1  kSize = 3;                    //Define 3x3 kernel
2  kCenter = kSize / 2;          //Center of kernel is (1,1)
3  numCols = 1920;
4  numRows = 1080;
5
6  for (i=0; i<numRows; i++) {          //Output row
7      for (j=0; j<numCols; j++) {      //Output column
8          for (m=0; m<kSize; m++) {        //Kernel row
9              for (n=0; n<kSize; n++) { //Kernel column
10                 ii = i - kCenter;        //Input row
11                 jj = j - kCenter;        //Input column
12
13                 //Make sure input is not out-of-bounds
14                 if (ii>=0 | ii<numRows | jj>=0 | jj<numCols)
15                     out[i][j] += in[ii][jj] * ker[m][n];
16                 //and if it was, zero the result
17                 else
18                     out[i][j] = 0;
19             }
20         }
21     }
```

Figure 4.9: Psuedo Code for 2D Convolution with Boundary Handling

We start at the HLS level to gauge how resource intensive the process will be, how loop unrolling affects our throughput and resource utilization, and what frequency the system achieves. A generalized datapath with little customization between similar operations is implemented in Vivado HLS. Once analyzed and tested, the image processing IP core is translated to RTL and manually tailored to each algorithm.

While based largely on the standard C++ convolution code, SystemC and OpenCV provide a unique spin to the syntax. This is especially true of input and output channels which encapsulate the original data payload and require methods to retrieve. The heart of the convolution code, in OpenCV, is shown in Figure 4.10.

```
#pragma HLS PIPELINE
    //Grab the latest pixel from the input stream
    currPixelSideChannel = inStream.read();

    //Strip the package around it to get just the 8-bit pixel data
    unsigned char pixelIn = currPixelSideChannel.data;

    //Shift data into line buffer
    lineBuff.shift_up(idxCol);
    lineBuff.insert_top(pixelIn,idxCol);

    //These two loops multiply one window's worth of values with the kernel
    for (int idxWinRow = 0; idxWinRow < KERNEL_DIM; idxWinRow++)     //Kernel row
    {
        for (int idxWinCol = 0; idxWinCol < KERNEL_DIM; idxWinCol++) //Kernel column
        {
            //Get one of the windowed values
            short val = (short)lineBuff.getval(idxWinRow,idxWinCol+pixConvolved);

            //Multiply value with appropriate kernel parameter
            val = (short)kernel[(idxWinRow*KERNEL_DIM) + idxWinCol] * val;
            window.insert(val,idxWinRow,idxWinCol);
        }
    }

    //If out-of-bounds, zero the output pixel
    short valOutput = 0;
    if ((idxRow >= KERNEL_DIM-1) && (idxCol >= KERNEL_DIM-1))
    {
        valOutput = sumWindow(&window);  //If in-bounds, sum the intermediate results
        if (valOutput < 0)               //and zero anything that may have overflowed
            valOutput = 0;

        pixConvolved++;
    }
```

Figure 4.10: Core of Convolution Module

In the highlighted code, we see the two innermost loops and out-of-bounds handling are largely as described in the pseudo code. The largest deviation is that the two outermost loops have been merged into one. The output row and column indexing is still tracked internally, however the outer loop now indexes the output pixel. This modification was made to more easily handle the input and output streams.

4.3 Summary of Single-core Systems

Single core systems need little more than what has been covered so far. To summarize, a 24-bit wide pixel arrives at the input and is converted to its 8-bit grey-scale equivalent. The grey-scaled inputs are clocked into the top-most line buffer, which, when full, will cascade into the next line buffer.

While first receiving inputs, there is a period of time in which the line buffers do not yet have enough information to properly fill the first convolution window. Only after we have stored the zeroth row of pixels plus the zeroth pixel of the first row and have the first pixel of the first row will the window contain valid information.

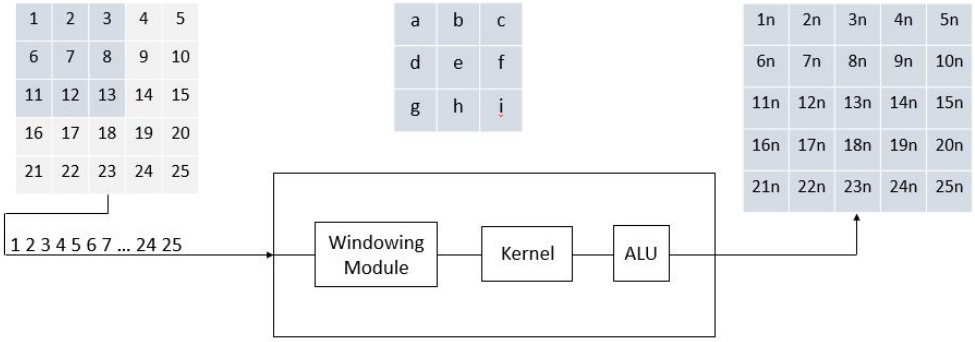
In other words, the initial output pixel (0,0) requires a window which is centered around input pixel (0,0), spanning from (-1,-1) to (1,1). Meanwhile, any preemptive outputs the convolution core may provide are not passed through to the system output (which instead shows zero.)

From the first clock after start (which we shall call $t = 0$) to $t = (imgCols * (kSize - 1) + kSize)/2$ is the initial wait time described just above. From this point in time to $t = imgCols * imgRows$ is standard operation where the windowing module is "sliding" across the image, providing inputs for the convolution core whose outputs (positive values capped at 255) are pushed directly to the system's output.

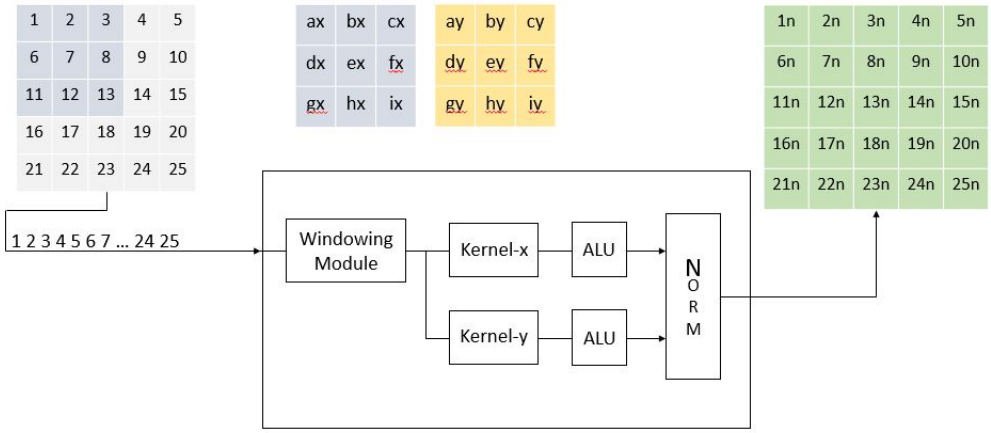
Once at $t = imgCols * imgRows$ there are no new input image pixels for the line buffers to receive or window, concluding our outer-most loop. For the next $(imgCols * (kSize - 1) + kSize)/2$ clock cycles the system output again foregoes the convolution core output to instead pad the end of its transmission with zeroes. Finally, at $t = (imgCols * imgRows) + ((imgCols * (kSize - 1) + kSize)/2)$ the system's output stream formally terminates.

4.4 Additions for Dual-core Systems

Systems that use two cores for convolution such as the Sobel and Scharr build off of and add to the single-core projects. The extent of change at the register-transfer level is as shown below. Note that the diagrams separate the multiplication module and the summation of weighted values into one output pixel as an the kernel and ALU, respectively.



(a) Single-kernel Core



(b) Dual-kernel Core

Figure 4.11: RTL View of IP Cores

The dual-core projects accept 24-bit wide inputs that are grey-scaled upon entry. The line buffers and windowing module remain unchanged, as well as the initial wait period. The first architectural addition is a second convolution cores implemented

in parallel with the first - one with X gradient weights and the other with Y gradient weights. The pixels captured in the windowing module are fanned out to be used as inputs to both cores.

While each individual core remains unchanged, the way we treat their output does. To obtain one value per pixel, we take the Euclidean norm of the two directional gradients. If we wish for accuracy, the HLS code in Figure 4.11 takes the square of each gradient, them takes the square root of their sum. Should resources be limited, the norm approximation shown in Figure 4.11 is less resource intensive, as it just sums the absolute values of the gradients.

```
outputStreamX.read(valOutX);
outputStreamY.read(valOutY);
float xval=valOutX.data*valOutX.data;
float yval=valOutY.data*valOutY.data;
valOut.data= hypot(xval, yval);
outImage[idxRows][idxCols] = valOut.data;
```

(a) Accurate Euclidean norm

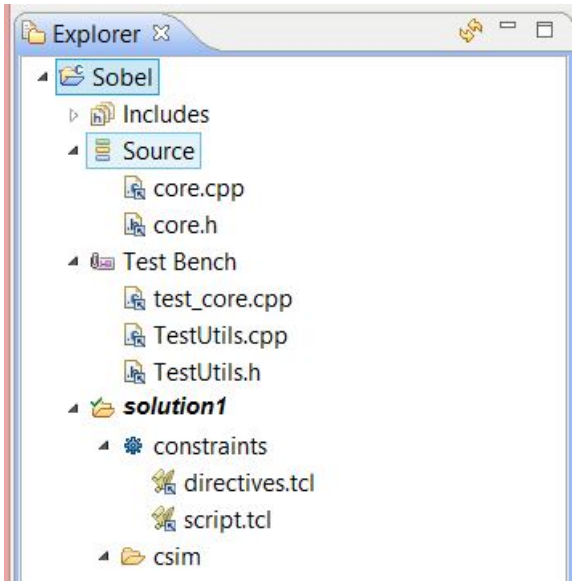
```
assign abs_gx = (gx[10]? ~gx+1 : gx);
assign abs_gy = (gy[10]? ~gy+1 : gy);

assign sum = (abs_gx+abs_gy);
```

(b) Estimated Euclidean norm

Figure 4.12: Norm Calculations in HLS and RTL

4.5 HLS Simulation and Synthesis

Simulation follows the design process, beginning with smaller images, 256x256 and 500x500 wide, and single-kernel processes, such as the Identity kernel or Edge kernel. This provides us a functional yet solid base with which to build off of as we test dual-kernel modules and larger images.



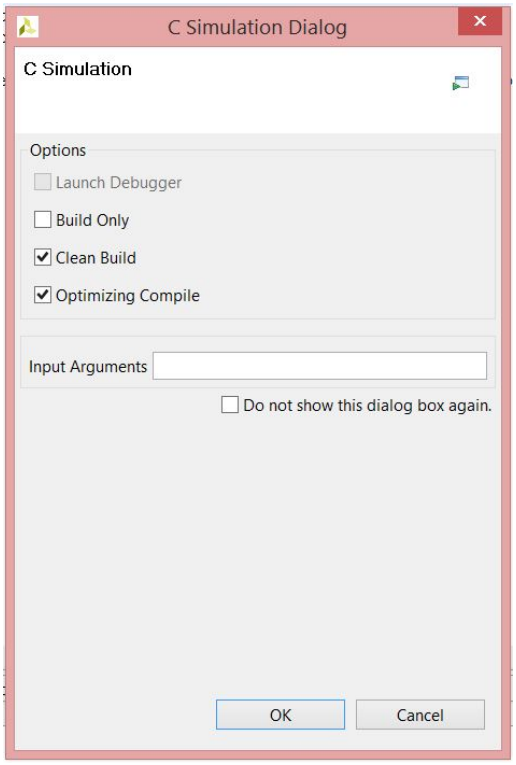
(a) Project Hierarchy

```
1  #include <stdio.h>
2  #include <opencv2/core/core.hpp>
3  #include <hls_opencv.h>
4  #include "core.h"
5  #include "testUtils.h"
6
7  // Image File path
8  char outImage[IMG_HEIGHT_OR_ROWS][IMG_WIDTH_OR_COLS];
9  char outImageRefx[IMG_HEIGHT_OR_ROWS][IMG_WIDTH_OR_COLS];
10 char outImageRefy[IMG_HEIGHT_OR_ROWS][IMG_WIDTH_OR_COLS];
11
12 // Declare kernel variables
13 int kerNum = 0;
14 char *kernelx = NULL;
15 char *kernely = NULL;
16
17 int main() {
18
19     // Read input image
20     printf("Load image %s\n",INPUT_IMAGE_CORE);
21     cv::Mat imageSrc;
22     imageSrc = cv::imread(INPUT_IMAGE_CORE);
23     // Convert to grayscale
24     cv::cvtColor(imageSrc, imageSrc, CV_BGR2GRAY);
25     printf("Image Rows:%d Cols:%d\n",imageSrc.rows, imageSrc.cols);
26
27     // Define streams for input and output
28     hls::stream<uint_8_side_channel> inputStreamX;
29     hls::stream<uint_8_side_channel> inputStreamY;
30     hls::stream<int_8_side_channel> outputStreamX;
31     hls::stream<int_8_side_channel> outputStreamY;
```

(b) SystemC Testbench Code Snippet

Figure 4.13: General View

Figure 4.13 above shows the over-arching layout and a snippet of code in which the core is described in .h and .cpp files. Vivado HLS provides a platform in which from debugging code largely written in System-C to translation and simulation in Verilog or VHDL. Firstly, we confirm the base code compiles and functions properly with a purely C compilation check and simulation run, as shown in Figure 4.14.



(a) Simulation Setup

```
1   Compiling ../../../test_core.cpp in release mode
2   Compiling ../../../TestUtils.cpp in release mode
3   Compiling ../../../core.cpp in release mode
4   Generating csim.exe
5 Load image C:\Users\Forte\Desktop\ThesisWork\ThesisHLS\Sobel\Sobel\dogE.bmp
6 Image Rows:1080 Cols:1920
7 Calling Reference function
8 Reference function ended
9 Saving image Ref
10 Calling Core function
11 Core function ended
12 Saving image
13 @I [SIM-1] CSim done with 0 errors.
14
```

(b) Simulation Results

Figure 4.14: C Simulation View

For this process we simply load an image from the computer, as seen in line five of Figure 4.14b. The rows and columns variables in line six of the same figure are set parameters within the module. In the C simulation, images are loaded and saved from the host computer via helper functions that serialize the image data for us. This eliminates any processing steps, another possible source of error, during the initial round of simulations.

The next step of the process is to create a preliminary set of hardware based on SystemC constructs. C instantiation provides a rough estimate of what it believes the hardware will achieve once created. Figure 4.15 displays these initial performance estimates below.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.34	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
18664328	18664328	18664329	18664329	none

Detail

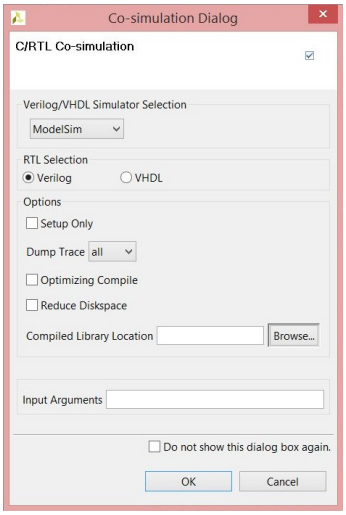
Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	18662405	18662405	15	9	1	2073600	yes
- Loop 2	1921	1921	1	-	-	1921	no

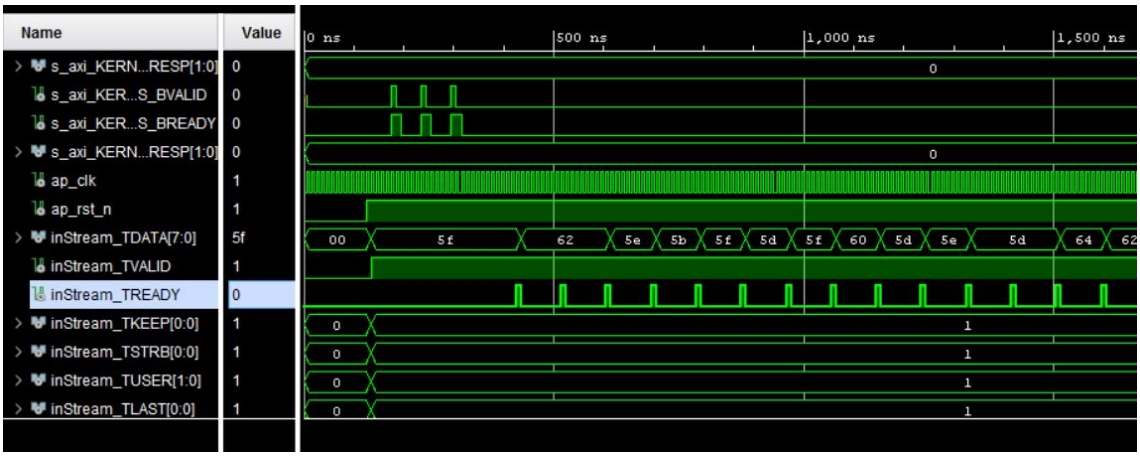
Figure 4.15: C Instantiation

The set of performance estimations shown above are for the pipelined Sobel operation that has yet to be unrolled. Figure 4.15 details that Loop 1, which iterates through the nine kernel multiplications, is labelled as pipelined, yet requires all nine passes (or initiation intervals).



(a) C/RTL Co-sim Setup

(b) Co-simulation Results



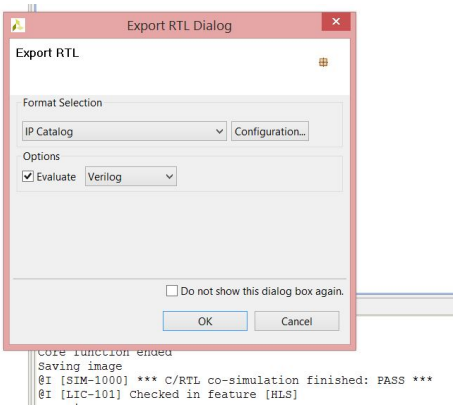
(c) Further Results

Figure 4.16: C/RTL Co-simulation View

We now translate our description to register-transfer level hardware via C/RTL co-simulation. The kernel multiplications and ALU summation are both contained in the function named doImgProc, which is elaborated in Figure 4.16 above.

During C/RTL co-simulation, Vivado HLS brings in RTL development environments such as ModelSim and Vivado to further elaborate and simulate our circuit at RTL. It is through these IDEs we can view both external and internal signals as they change throughout the circuit's operation.

Having validated the co-simulation's interpretation and behavior, we finalize, analyze, and export the RTL code. The setup and break down of resource utilization is shown in Figure 4.17.



(a) Full RTL Elaboration Setup

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	9	0	814
FIFO	-	-	-	-
Instance	2	-	184	214
Memory	3	-	0	0
Multiplexer	-	-	-	358
Register	-	-	891	-
Total	5	9	1075	1386
Available	530	400	157200	78600
Utilization (%)	~0	2	~0	1

(b) RTL Resource Usage

Figure 4.17: RTL Export View

Everything discussed thus far are what we would consider an IP Core, a module where our intellectual property's core functions are encapsulated. This involves the intake, windowing, convolution, post-processing, and output of an image. While this will include our datapath and controller, what me must also consider is how we will interact with memories, I/O interfaces, and timing. The FGPA is the computational heart of the board, but not the whole board.

To take this completed IP core and merge it into a complete design, we must consider where it should be placed. Not only will we need our FPGA and convolution IP core, but we will also use a direct memory access (DMA) module whose purpose is to push the image into the core pixel by pixel, as well as to receive the output of the core. Two interconnects will be required to configure the two possible cores with their kernel weights, and the timer will be the coordinator of the modules as they operate.

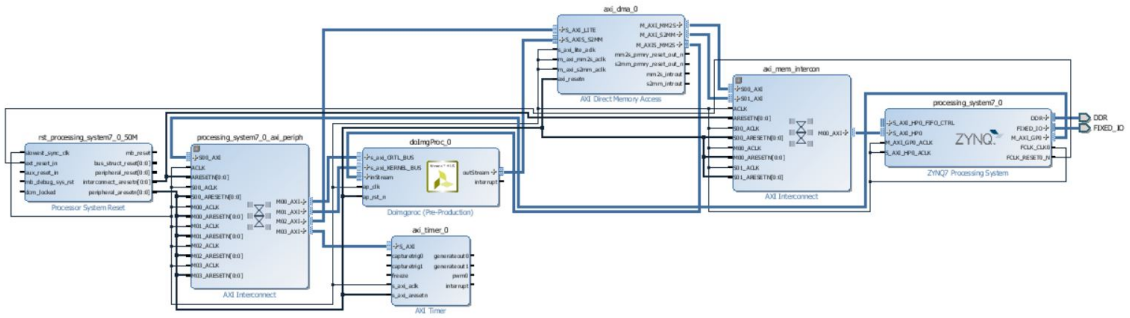


Figure 4.18: A Complete Design

The blocks are organized and connected as shown in Figure 4.18 and are subsequently wrapped into one, cohesive system with one set of inputs and outputs. Figure 4.19 shows the placement and route of the design on the Zynq-7000 FPGA, part number xc7z030fbg676-1.

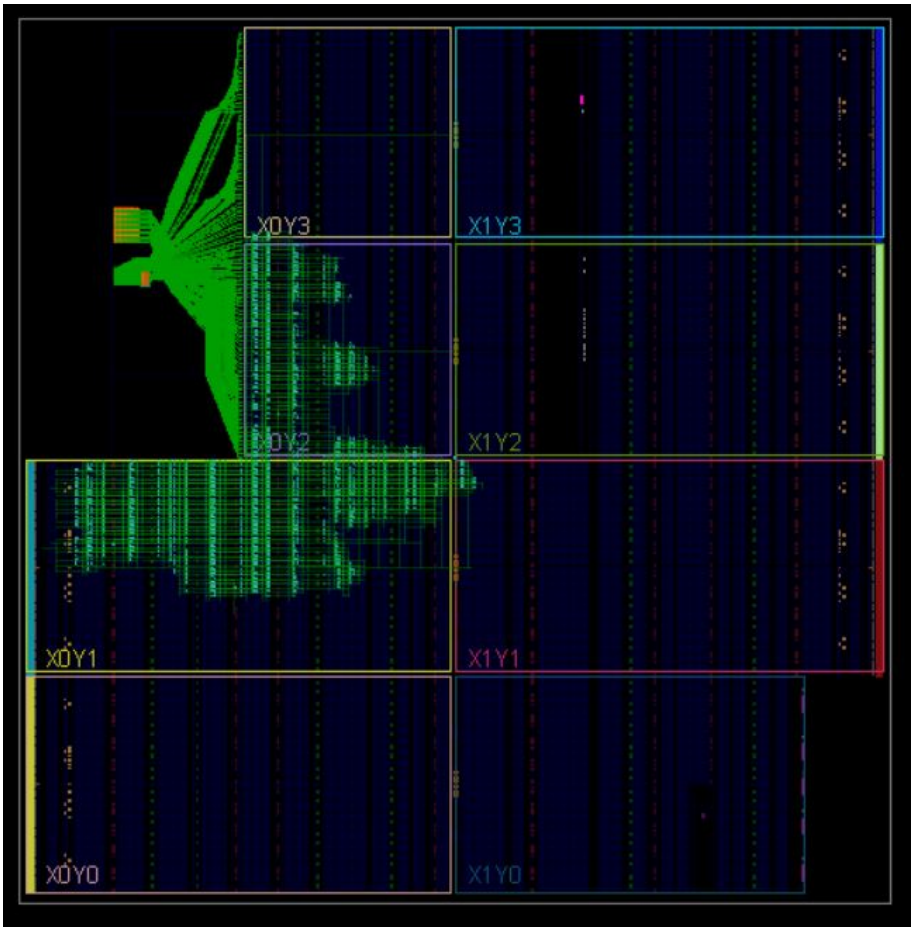


Figure 4.19: Placed Design

4.6 Recreating a Convolution Core

Having tested and finalized a generic module for the single-core and double-core convolution algorithms, we take the following section to tailor make the Sobel-Feldmann module.

We focus on the Sobel algorithm not only because of its popularity, but also showcase how the algorithm took hardware implementation into consideration. Being a double-core operation, it includes the additional step of magnitude approximation. To provide context, Figure 4.20 is an excerpt of the RTL code that Vivado HLS generated from the HLS description.

```
/// assign process. ///
always @(posedge ap_clk)
begin
    if ((exitcond1_reg_1296 == ap_const_lv1_0)) begin
        if ((ap_const_logic_1 == ap_sig_cseq_ST_pp0_stg4_fsm_5)) begin
            reg_533 <= lineBuff_val_0_q1;
        end else if ((ap_const_logic_1 == ap_sig_cseq_ST_pp0_stg3_fsm_4)) begin
            reg_533 <= lineBuff_val_0_q0;
        end
    end
end

/// assign process. ///
always @(posedge ap_clk)
begin
    if ((ap_const_logic_1 == ap_sig_cseq_ST_pp0_stg0_fsm_1)) begin
        ap_reg_ppstg_exitcond1_reg_1296_pp0_it1 <= exitcond1_reg_1296;
        ap_reg_ppstg_tmp_12_reg_1338_pp0_it1 <= tmp_12_reg_1338;
        exitcond1_reg_1296 <= exitcond1_fu_560_p2;
    end
end
```

Figure 4.20: RTL Generated by Vivado HLS

While fully functional, the code becomes very laborious for a human to parse as processes and variables become difficult to track through the code. It is not readily apparent what part of the process or where in an RTL diagram Figure 4.20 relates. Instead of using this as the base, we instead take what we have learned during the HLS implementation and testing phases and hand-interpret the C code to our own RTL.

Near the beginning of this chapter, the implementation of line buffers and an idealized windowing module were presented in block diagram and code. This code can go into our datapath unaltered, especially if we stipulate the input image is to be given in 8-bit per pixel, grey-scaled .PNG format.

We now focus on the algorithm in question and analyze how to best utilize our parameter weights. Sobel and Feldmann intentionally chose kernel weights which are powers of two, allowing us the option to forego multipliers if we represent any signed data values in signed-magnitude format. While the module only inputs and outputs positive 8-bit values, some kernel weights are negative and will require attention.

Each output pixel resultant of a frame convolved in the X direction can be expressed as:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{matrix} (-1*a)+(0*b)+(1*c)+ \\ (-2*d)+(0*e)+(2*f)+ \\ (-1*g)+(0*h)+(1*i) \end{matrix}$$

$$= -a + c - 2d + 2f - g + i$$

$$= (c - a) + 2(f - d) + (i - g)$$

While each frame convolved in the Y direction can be output as:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{matrix} (-1*a)+(-2*b)+(-1*c)+ \\ (0*d)+(0*e)+(0*f)+ \\ (1*g)+(2*h)+(1*i) \end{matrix}$$

$$= -a - 2b - c + g + 2h + i$$

$$= (g - a) + 2(h - b) + (i - c)$$

With the interpretations above, we remove the need for any sequential components within the gradient calculations. To multiply a binary number by two is to left shift the magnitude bits one place leaving the sign bit as-is. While a shift register could be implemented and clocked, this design mimics its functionality without hardware by directly manipulating the data bus. The adders and subtractors are combinational by nature, and will properly handle signed values that are declared as such.

Figure 4.21 incorporates these concepts to obtain the directional gradients gx and gy. They are highlighted in the first two assign statements, were p0 through p8 are windowed pixels numbered in raster scan order.

```
wire signed [10:0] gx,gy;

wire signed [10:0] abs_gx,abs_gy;
wire [10:0] sum;

assign gx=((p2-p0)+((p5-p3)<<1)+(p8-p6));
assign gy=((p6-p0)+((p7-p1)<<1)+(p8-p2));

assign abs_gx = (gx[10]? ~gx+1 : gx);
assign abs_gy = (gy[10]? ~gy+1 : gy);

assign sum = (abs_gx+abs_gy);
assign out = (|sum[10:8])?8'hff : sum[7:0];
```

Figure 4.21: Code of X and Y Sobel Convolutions and Magnitude Approximator

Figure 4.21 also takes the directional gradients to calculate the output value. The signed-magnitude format makes taking the absolute values of the X and Y gradients as easy as zeroing each value's sign bit. One more full-adder is required to sum the absolute values. The result is then capped at a maximum value of 255, which is checked for with two extra or gates.

Chapter 5

Experimental Setup

The HLS design takes advantage of stream constructs and access to the .bmp files on the host computer during simulation. When recreating and tailoring the design as human-readable RTL, we forego said luxuries.

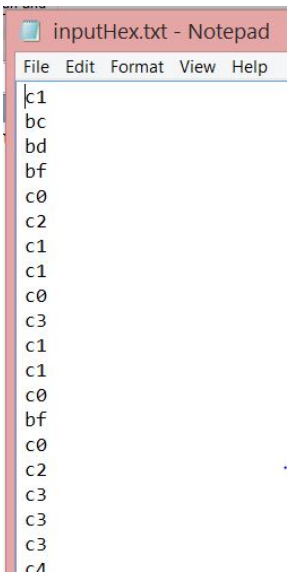
5.1 Input Creation

The RTL datapath is designed to receive one pixel's worth of hex values as one byte of data at a time. Similarly, the output byte of values must be reformatted to .png's expected format. The input and output files used are plain .txt files which contain one pixel's worth of data per line in hex.

As these files will be as many lines long as there are pixels in the input image (plus one new line character), some pre-processing is greatly beneficial. The MATLAB code for such formatting and the first few line of the resultant output file are provided in Figure 5.1 below.

```
% Read the image from the file
[filename, pathname] = uigetfile('*.bmp;*.tif;*.jpg;*.pgm','Pick an M-file');
img = imread(filename);
img = imresize((img),[256 256]);
[ row col p ] =size(img);

if p == 3
    img = rgb2gray(img);
end

% Image Transpose
imgTrans = img';
% iD conversion
img1D = imgTrans(:);
% Decimal to Hex value conversion
imgHex = dec2hex(img1D);
% New txt file creation
fid = fopen('inputHex.txt', 'wt');
% Hex value write to the txt file
fprintf(fid, '%x\n', img1D);
% Close the txt file
fclose(fid)
```

(a) Translate Image to Hex Values



(b) Created Text File

Figure 5.1: MATLAB Code for Input File Creation

The MATLAB script accepts .bmp, .tif, .jpg, and .pgm file formats and exports as a .txt file. As-is the code will convert color images to grayscale and output eight bits of data per pixel. It can be modified to retain color information at 24 bits per pixel (as three 8-bit channels) if desired.

5.2 Top Module's Code

The top module which ties the aforementioned submodules such as the line buffers and convolution core together and coordinates them is as shown in Figure 5.2 below.

Figure 5.2 is split across two pages and contains three subfigures, labeled a, b, and c. Figure 5.2a contains the module declaration as well and defines parameters that keep track of the windowing module's position with respect to the input picture, as well as the initial delay count.

```
module imageProc
  #(parameter IMG_COLS = 1920, IMG_ROWS = 1080) (
    input clk, rst,
    input [7:0] pixIn,
    output [7:0] pixOut,
    output doneFlag
   );

   //Process parameters
   parameter BITS = 8;  parameter KERNEL_DIM = 3;
   reg [15:0] idxCol = 16'd0;    reg [15:0] idxRow = 16'd0;
   reg [32:0] idxPixel = 33'd0;
   reg [7:0] valOut = 8'd0;

   //Delay parameters to fix line-buffer offset
   parameter waitTicks = (IMG_COLS*(KERNEL_DIM-1)+KERNEL_DIM)/2;
   reg [32:0] countWait = 33'd0;
```

(a) Module and Parameter Declarations

```

   //Line buffer delay components
   wire [7:0] dout0, dout1, dout2, sobelOut;
   reg [7:0] a = 8'd0;  reg [7:0] b = 8'd0;  reg [7:0] c = 8'd0;
   reg [7:0] d = 8'd0;  reg [7:0] e = 8'd0;  reg [7:0] f = 8'd0;
   reg [7:0] g = 8'd0;  reg [7:0] h = 8'd0;  reg [7:0] i = 8'd0;

   //Line buffer
   lineBuf #(IMG_COLS, BITS) lineBuf0(clk, rst, pixIn, dout0);
   lineBuf #(IMG_COLS, BITS) lineBuf1(clk, rst, dout0, dout1);
   lineBuf #(IMG_COLS, BITS) lineBuf2(clk, rst, dout1, dout2);
   sobelCore sobel0 (a, b, c, d, e, f, g, h, i, sobelOut);

//Line buffer process statement
always @(posedge clk, posedge rst) begin

  if (rst) begin
     a <= 8'd0;  b <= 8'd0;  c <= 8'd0;
     d <= 8'd0;  e <= 8'd0;  f <= 8'd0;
     g <= 8'd0;  h <= 8'd0;  i <= 8'd0;
  end

  else begin
     a <= b;  b <= c;  c <= dout2;
     d <= e;  e <= f;  f <= dout1;
     g <= h;  h <= i;  i <= dout0;
  end

end
```

(b) Line Buffer Instantiation and Process

```
47
48 //Process through all pixels
49 always @(posedge clk, posedge rst) begin
50
51   //Zero all parameters on reset
52   if (rst) begin
53      idxCol = 16'd0;  idxRow = 16'd0;  idxPixel = 33'd0;
54            valOut = 8'd0;
55            countWait = 16'd0;
56   end
57
58   //Initial wait + accept all input pixels
59   else if (idxPixel < (IMG_ROWS * IMG_COLS)) begin
60
61      //valOut must be an integer between 0-255 inclusive, which sobelOut provides
62      valOut = ((idxRow >= KERNEL_DIM-1) && (idxCol >= KERNEL_DIM-1)) ? sobelOut : 8'd0;
63
64      idxCol = (idxCol + 1'b1) % (IMG_COLS);
65      idxRow = (idxCol == 0) ? (idxRow + 1'b1) : idxRow;
66
67      countWait = countWait + 1'b1;
68      idxPixel = idxPixel + 1'b1;
69
70   end
71
72   //Follow through all input pixels
73   else begin
74      valOut = 8'd0;
75      idxPixel = idxPixel + 1'b1;
76   end
77
78 end
79
80    assign pixOut = (countWait > waitTicks) ? valOut : 8'd0;
81    assign doneFlag = (idxPixel < ((IMG_ROWS * IMG_COLS) + waitTicks)) ? 1'b0 : 1'b1;
82
83 endmodule
```

(c) Process Statements and Output Assigning

Figure 5.2: Top-level RTL Code

Figure 5.2b instantiates the three line buffers as well as the delay components and the Sobel's core module. The first line buffer's output feeds into the second's input, and the second and third buffers are similarly tied. The line buffers and delay components a-f are each initialized to zero to avoid high impedance upon start-up.

Figure 5.2b is also displays that the line buffer outputs are fanned out to the inputs of delay components c, f, and i. These delay components are what store the values in the window formation for the Sobel core to process.

Figure 5.3c contains all of the timing and control elements that keep track of the reset signal, current pixel, row, column values, and initial delay as it iterates through the image. The final lines of code assign the output value and a flag that signals the end of the module's calculation. Note lines 61-68 play the integral role of zeroing any results whose window center is out-of-bounds.

The RTL view describes in this top module is as shown in Figure 5.3 below. To reduce clutter, RTL blocks marked with triangles are assumed to share the global clock and reset. Modulo operators paired to counters - for example, idxCol counting from 0 to (IMG_COLS - 1) before rolling over to 0 - are also omitted.

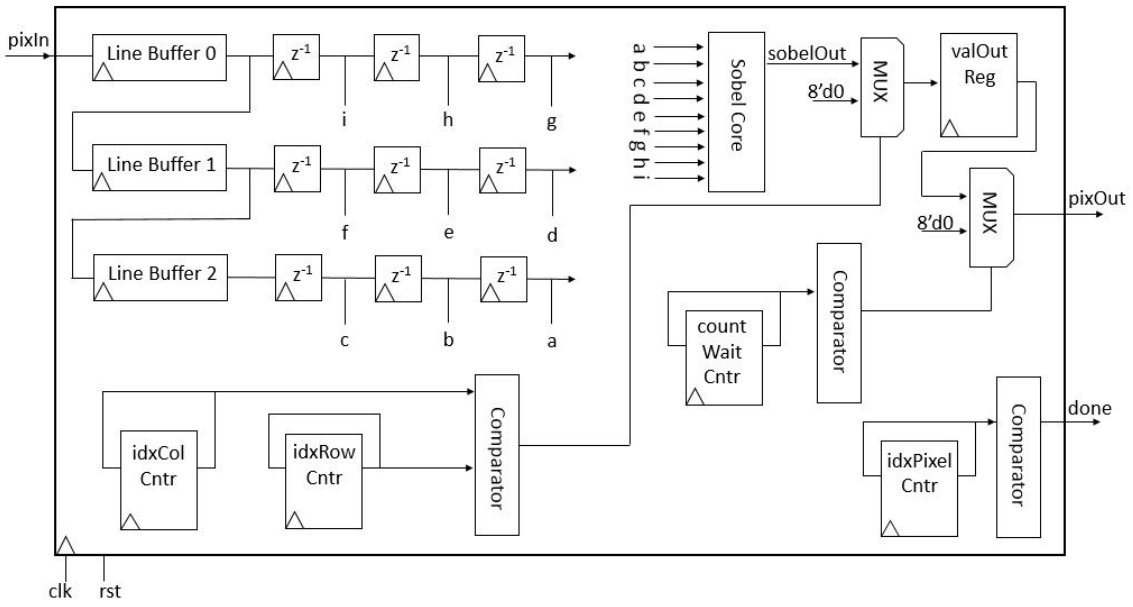


Figure 5.3: Complete RTL Block Diagram

5.3 RTL Testbench

The modules are parameterized such that the user defines the input image dimensions in the testbench. The input file is opened for reading as an output file is created and opened for writing. Inputs are read in and begin processing the rising clock after reset goes low. Inputs are read in and begin populating the line buffers at the rate of one pixel per clock cycle.

```
1 module imageProc_TB();
2
3   reg clk, rst;
4   reg [7:0] pixIn;
5   wire [7:0] pixOut;
6   wire doneFlag;
7
8   parameter IMG_COLS = 256;
9   parameter IMG_ROWS = 256;
10  parameter KERNEL_DIM = 3;
11  parameter OUTPUT_WAIT = (IMG_COLS*(KERNEL_DIM-1)+KERNEL_DIM)/2;
12
13  integer status;
14  integer inputFile;
15  integer outputFile;
16  reg [32:0] countWait = 0;
17
18  imageProc #(IMG_COLS, IMG_ROWS) UUT ( clk, rst, pixIn, pixOut, doneFlag );
19
20 always #6 clk = ~clk;
21
22 initial begin
23  //Open input file
24  inputFile = $fopen("inputHexA.txt", "r");
25  outputFile = $fopen("outputHexA.txt", "w");
26
27  //Initial settings
28  clk = 1'b0; rst = 1'b0; pixIn = 8'h00; #12;
29  rst = 1'b1; #12;
30  rst = 1'b0;
32  while ( doneFlag == 0 ) begin
33
34    status = $fscanf(inputFile, "%h\n", pixIn);
35
36    if (countWait > OUTPUT_WAIT)
37       $fwrite(outputFile, "%h\n", pixOut);
38
39    countWait = countWait + 1'b1;
40    #12;
41
42  end
43
44  $fwrite(outputFile, "%h\n", pixOut);
45  #12;
46
47  $fclose(inputFile);
48  $fclose(outputFile);
49
50  $stop;
51
52 end
53
54 endmodule
```

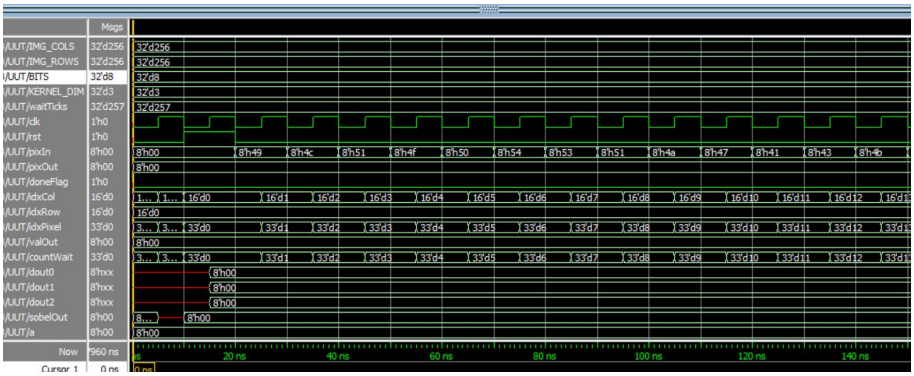
Figure 5.4: RTL Testbench for File I/O

The module, as discussed, has an initial wait period to include consideration for border pixels via zero-padding. This is addressed both within the module and in the testbench in Figure 5.4 above. The module contains an initial delay that zeroes any output before the first valid window.

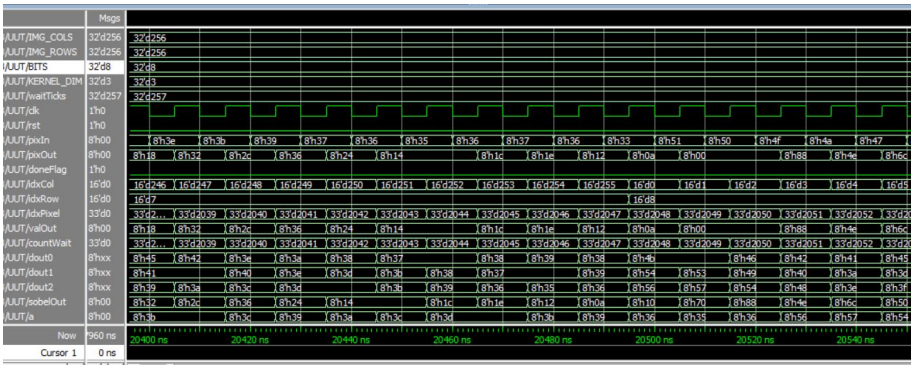
The testbench waits until the initial wait period (as defined by the image dimensions) completes before it writes the output values to the text file. The resultant output file will contain the same amount of information as the defined input dimensions, including both leading and trailing zeros.

5.4 Simulated Waveform

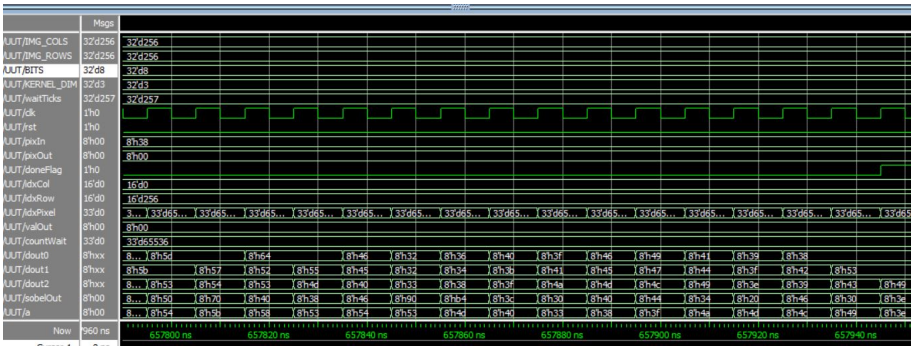
The waveforms in Figure 5.5 below show the start, and middle section, and the end of the simulation. With a clock period of ten nanoseconds, a 256x256 image takes 0.658ms to process one frame. A 1920x1080 image takes 20.753ms to process.



(a) First 150ns



(b) Middle 150ns



(c) Final 150ns

Figure 5.5: Waveform Snippets from Simulation

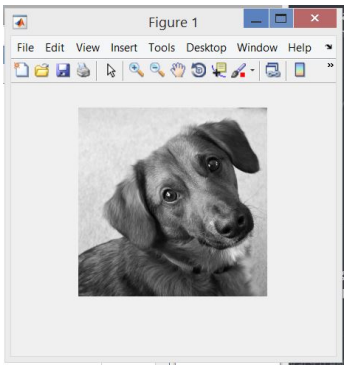
The simulation shown in Figure 5.5 confirms the circuit is in working order and produces the output text file we use for recomposing the filtered image.

5.5 Output Recomposition

The output text file mirrors the text file given at the tie of input in both length and format. Similarly, it is processed by MATLAB so that we can see and retrieve the filtered .bmp picture.

```
% Open the txt file
fid = fopen('inputHex.txt', 'r');
% Scan the txt file
img = fscanf(fid, '%x', [1 inf]);
% Close the txt file
fclose(fid)
% Restore the image
outImg = reshape(img,[256 256]);
outImg = outImg';
% Open image
figure, imshow(outImg,[])
```

(a) Translate Hex Values to Image



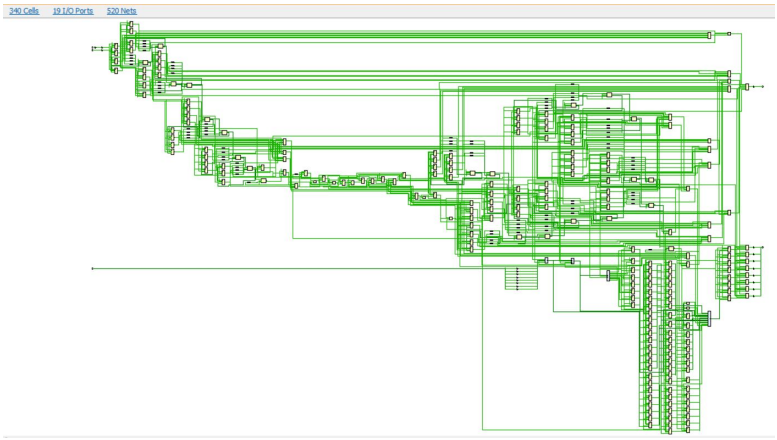
(b) Resultant Image

Figure 5.6: MATLAB Code for Output Image Recreation

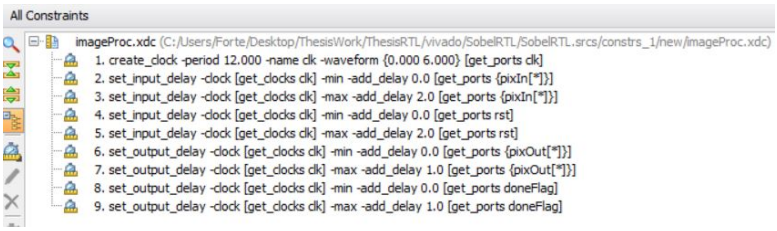
The output text file will have leading and trailing output values of zero at the beginning and end of the file each equal to the initial wait periods calculated. With these, the output file has exactly as many lines as the input file supplied. The image reconstructed from the output text therefore has the same dimensions as the input image. A code snippet and the output are shown in Figure 5.6 above.

5.6 Synthesis, Implementation, and Timing

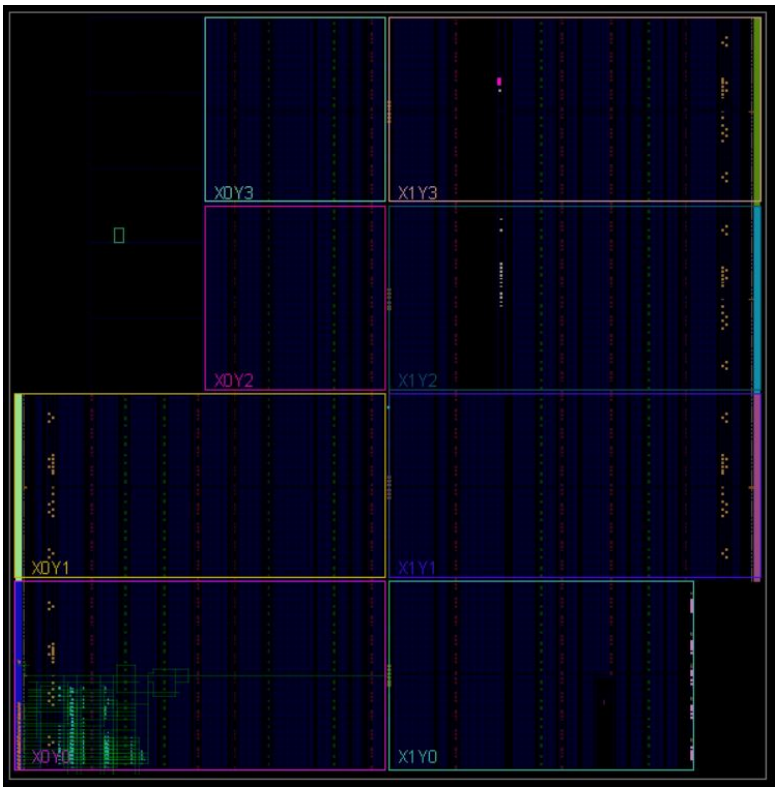
Design synthesis, placement, and routing are all largely automated by the design environment - in out case, Vivado. With valid behavioural simulation, our next step



(a) Synthesized Netlist



(b) User-defined Constraints



(c) Device Placement and Routing

Figure 5.7: Synthesis and Implementation

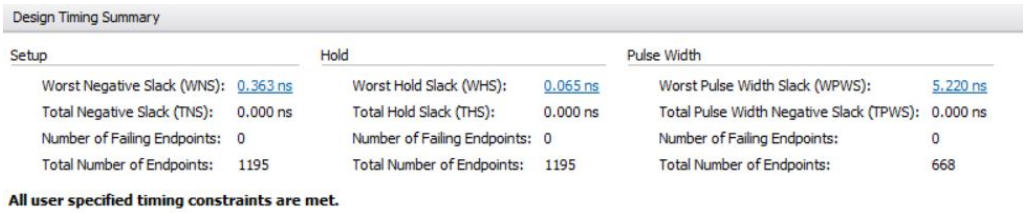
is to define our device and what ports will be used for input and output.

This project sets all inputs to slide switches and all outputs to LEDs. The design can later be modified to interface with outside peripherals that serially feed in and read out the hex values. To assure said peripherals have room to interface with the FPGA, an input delay of 2ns and output delay of 1ns are added as timing constraints. The synthesized netlist, constraints file, and implemented placement and routing are shown in Figure 5.7 above.

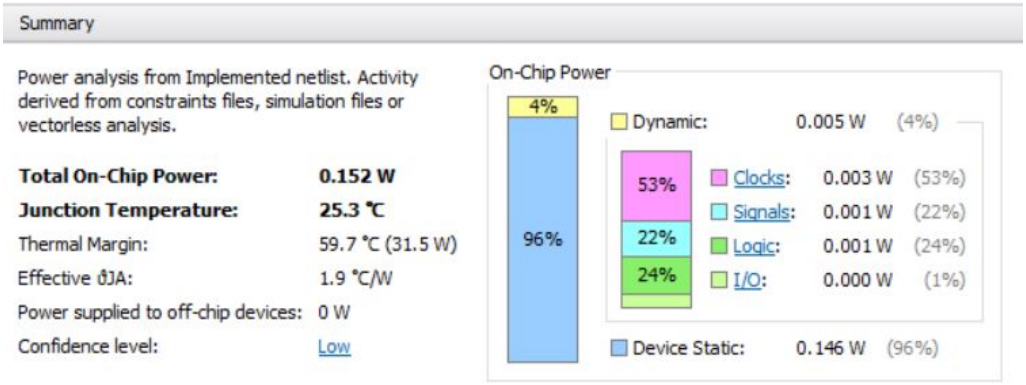
When running the timing, power, and resource utilization reports it was found that to keep the throughput of one pixel a clock cycle, the clock period needed to be increased from 10ns to 12ns. Once readjusted for delays and reimplemented, the timing, power, and resource utilization report summary are as shown below in Figure 5.8.

As expected, the resource utilization has not only remained remarkably low, but has also shrunk further due to the hand tailoring. Block RAMs are no longer utilized for the windowing line buffers, instead implemented as slice registers. Because these designs do not take up many resources, lower cost FPGAs can be used so long as the have the requisite I/O.

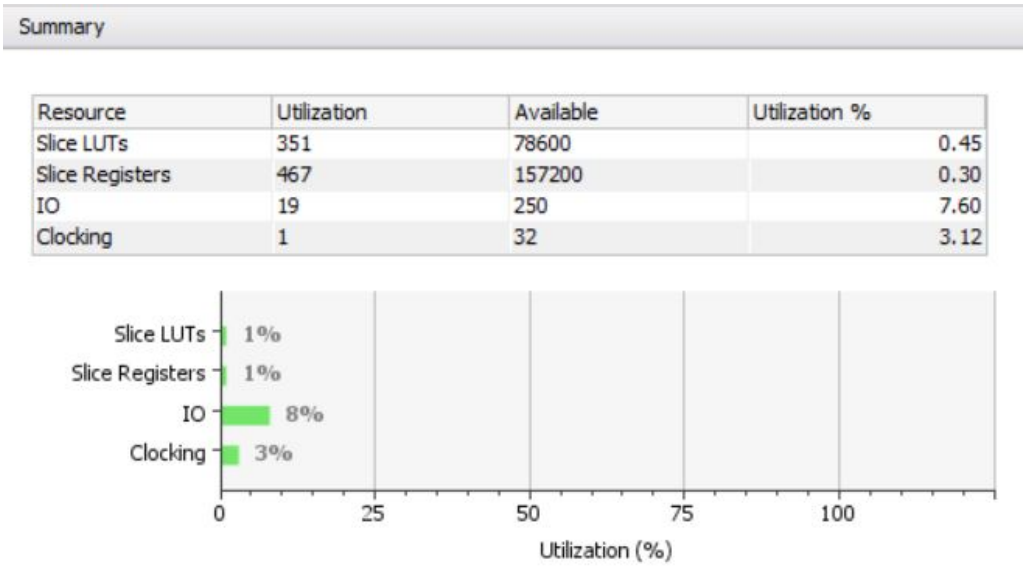
Another possibility for the unused space is to place multiple iterations of the design on one FPGA. This is further discussed in the "Future Work" section of Chapter 7.



(a) Timing Summary



(b) Power Summary

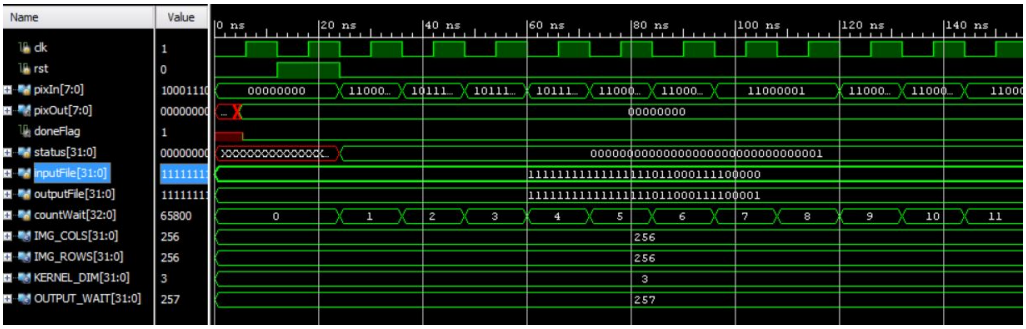


(c) Resource Utilization Summary

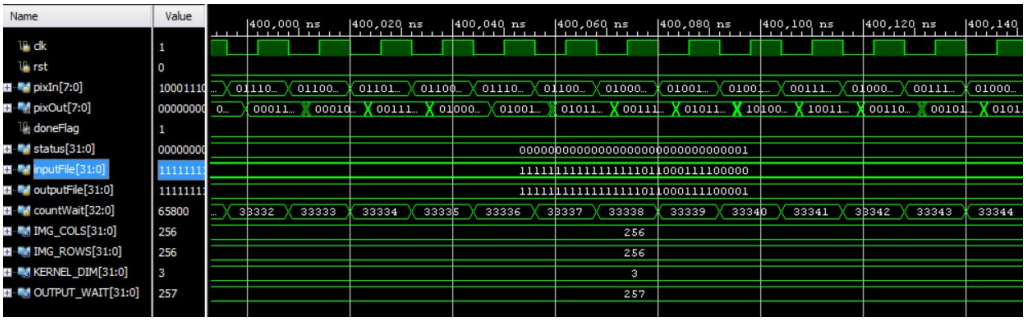
Figure 5.8: Implementation Reports

5.7 Post-Implementation Timing Simulation

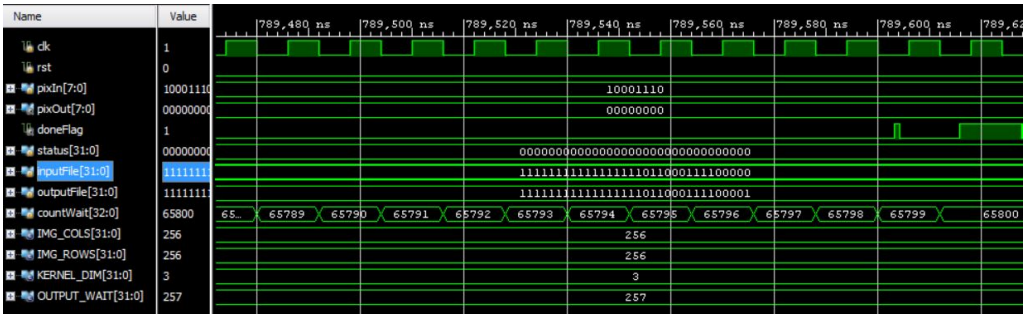
The waveforms in Figure 5.8 below show the start, and middle section, and the end of the simulation. With an updated clock period of twelve nanoseconds, a 256x256 image takes 0.790ms to process one frame. A 1920x1080 image takes 24.906ms to process, as it contains approximately 30 times as many pixels to iterate over.



(a) First 150ns



(b) Middle 150ns



(c) Final 150ns

Figure 5.9: Waveform Snippets from Simulation

Chapter 6

Results

The results displayed here are those run through the HLS project's C/RTL Co-simulation step. The collection of images will be provided here before being summarized in the following chapter. The images convolved are shown in Figure 6.1, where images A, B, and C are each 256x256 and in Figure 6.2, where images D, E, and F are each 1920x1080 pixels.

(a) Image A (b) Image B (c) Image C

Figure 6.1: 256x256 Images Convolved

(a) Image D

(b) Image E

(c) Image F

Figure 6.2: 1920x1080 Images Convolved

6.1 Identity

The identity kernel is straightforward computation-wise and helps to debug the circuit. Figure 6.3 shows each step of the process, where (a) is original, RGB input image, (b) is the grey-scale equivalent of the input, (c) shows the intermediate windowed calculation, and (d) shows the filtered output image. Note that because the identity kernel does not fundamentally alter the image, the only change between (c) and (d) is that the image is correctly re-centered. Figures 6.4 and 6.5 show only the inputs and filtered outputs of the other 256x256 sized images. Figures 6.5 through 6.8 show the same process applied to the 1920x1080 images.

(a) Original Image

(b) Grey-scaled Image

(c) Windowed Image

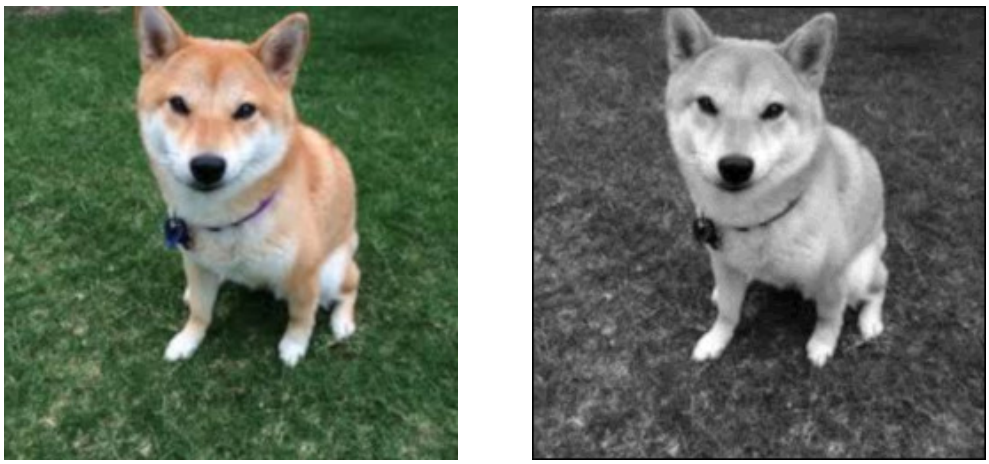
(d) Filtered Image

Figure 6.3: Filtering Image A with the Identity Kernel

(a) Original Image

(b) Filtered Image

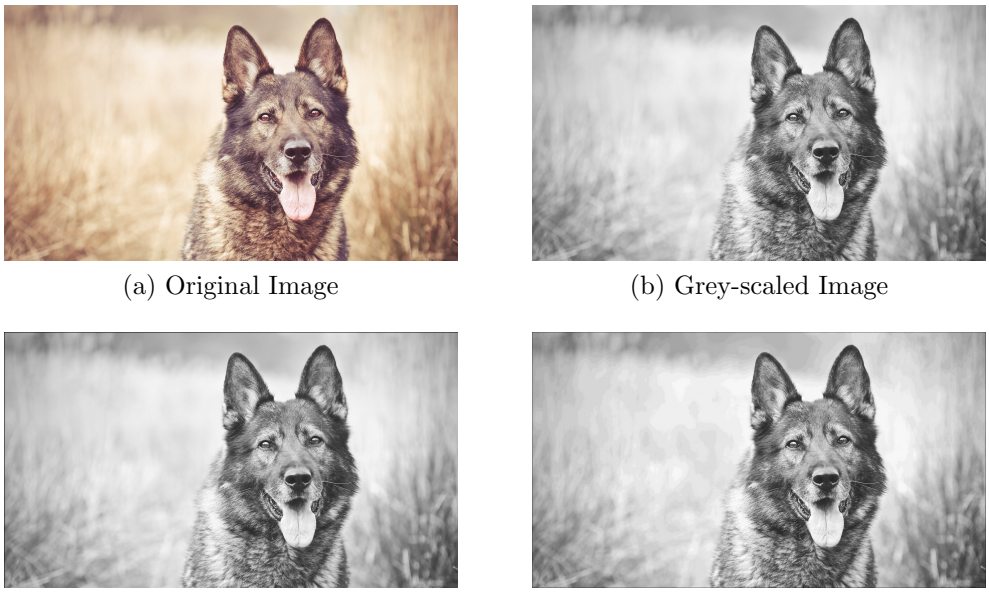
Figure 6.4: Filtering Image B with the Identity Kernel



(a) Original Image (b) Filtered Image

Figure 6.5: Filtering Image C with the Identity Kernel

The identity kernel works flawlessly over every image applied, as expected. There is no classification of image here that is better or worse-suited to be a input to this operation.



(a) Original Image (b) Grey-scaled Image

(c) Windowed Image (d) Filtered Image

Figure 6.6: Filtering Image D with the Identity Kernel

(a) Original Image

(b) Filtered Image

Figure 6.7: Filtering Image E with the Identity Kernel

(a) Original Image

(b) Filtered Image

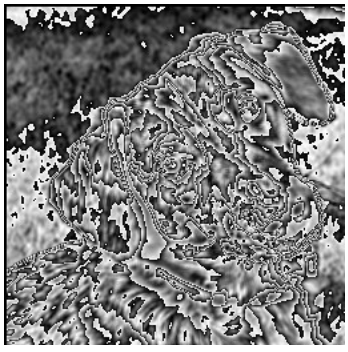
Figure 6.8: Filtering Image F with the Identity Kernel

6.2 Box Blur

The box blur kernel is simplest computation, taking an average of neighboring pixels with even weight. Figure 6.9 shows each step of the process, where (a) is original, RGB input image, (b) is the grey-scale equivalent of the input, (c) shows the intermediate windowed calculation, and (d) shows the filtered output image. Note that we can now see the intermediate change between (c), the post-kernel filter to be applied, and (d) the filter and image combined by the ALU. Figures 6.10 and 6.11 show only the inputs and filtered outputs of the other 256x256 sized images. Figures 6.12 through 6.14 show the same process applied to the 1920x1080 images.

(a) Original Image

(b) Grey-scaled Image



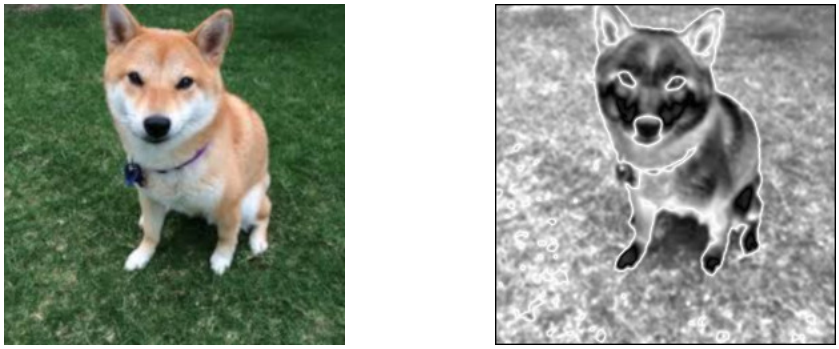
(c) Windowed Image

(d) Filtered Image

Figure 6.9: Filtering Image A with the Box Blur Kernel

(a) Original Image (b) Filtered Image

Figure 6.10: Filtering Image B with the Box Blur Kernel



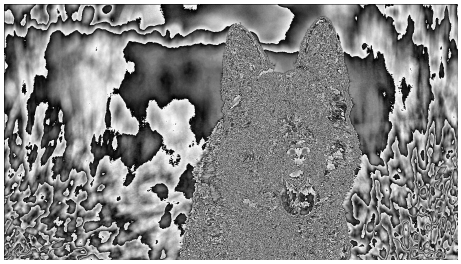
(a) Original Image (b) Filtered Image

Figure 6.11: Filtering Image C with the Box Blur Kernel

The box blur kernel works fairly well on all images, though will lead future kernels to find some unwanted edges. Patches of image that are already very similar to their surroundings are blacked out. Where the image drastically changes color are highlighted in white. This works well in the smaller images, bringing out differences between the dogs and backgrounds with the exception of image B, which does have a clear grass line. On the larger images, this varies wildly, giving few, reasonable false positives on images D and E. Image F, on the other hand, has completely failed to distinguish the dog from its background and only highlights a section of the sky.

(a) Original Image

(b) Grey-scaled Image



(c) Windowed Image

(d) Filtered Image

Figure 6.12: Filtering Image D with the Box Blur Kernel

(a) Original Image

(b) Filtered Image

Figure 6.13: Filtering Image E with the Box Blur Kernel

(a) Original Image

(b) Filtered Image

Figure 6.14: Filtering Image F with the Box Blur Kernel

6.3 Sharpen

The sharpening kernel is as follows. Figure 6.15 shows each step of the process, where (a) is original, RGB input image, (b) is the grey-scale equivalent of the input, (c) shows the intermediate windowed calculation, and (d) shows the filtered output image. Image (c) highlights the points of interest as determined by the kernel, letting us see where the image is most affected. Figures 6.16 and 6.17 show only the inputs and filtered outputs of the other 256x256 sized images. Figures 6.18 through 6.20 show the same process applied to the 1920x1080 images.

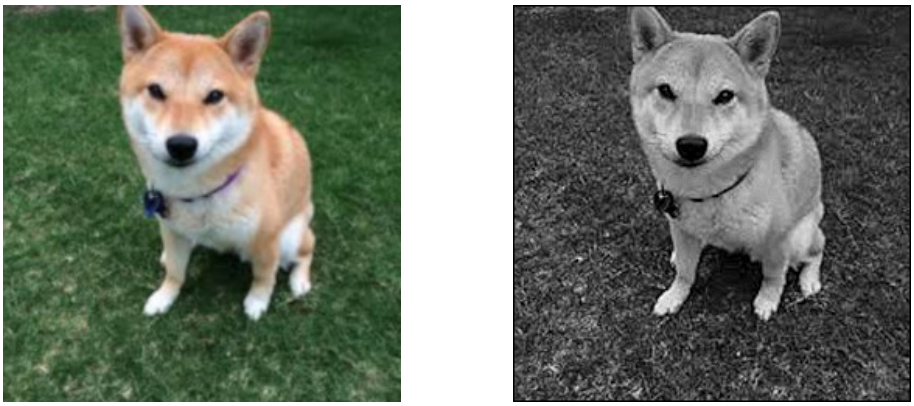
(a) Original Image (b) Grey-scaled Image

(c) Windowed Image (d) Filtered Image

Figure 6.15: Filtering Image A with the Sharpening Kernel

(a) Original Image (b) Filtered Image

Figure 6.16: Filtering Image B with the Sharpening Kernel



(a) Original Image (b) Filtered Image

Figure 6.17: Filtering Image C with the Sharpening Kernel

The sharpening kernel is shown to darken the picture except for where finer details are found. Rather than make the outlines between objects pop out, it has the effect of highlighting intra-object details. The dogs' eyes and noses are all brought out, though smaller background objects, such as the pebbles in image B and the grass in image C are also highlighted. Image D, where the dog has many, small patches of various fur colors, but the field in the background is relatively homogeneous in color is best affected by this filter.

(a) Original Image

(b) Grey-scaled Image

(c) Windowed Image

(d) Filtered Image

Figure 6.18: Filtering Image D with the Sharpening Kernel

(a) Original Image

(b) Filtered Image

Figure 6.19: Filtering Image E with the Sharpening Kernel

(a) Original Image

(b) Filtered Image

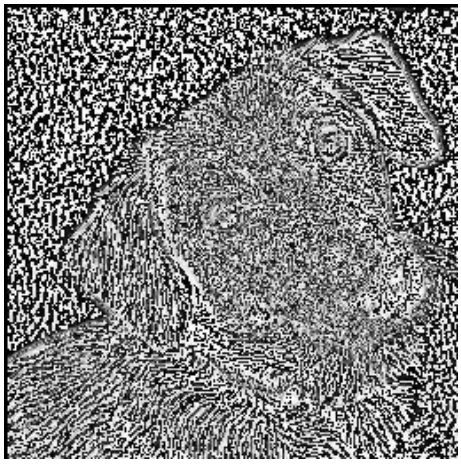
Figure 6.20: Filtering Image F with the Sharpening Kernel

6.4 Edge

The Edge kernel is as follows. Figure 6.21 shows each step of the process, where (a) is original, RGB input image, (b) is the grey-scale equivalent of the input, (c) shows the intermediate windowed calculation, and (d) shows the filtered output image. Image (c) shows the various gradients found by the kernel. Figures 6.22 and 6.23 show only the inputs and filtered outputs of the other 256x256 sized images. Figures 6.24 through 6.26 show the same process applied to the 1920x1080 images.

(a) Original Image

(b) Grey-scaled Image



(c) Windowed Image

(d) Filtered Image

Figure 6.21: Filtering Image A with the Edge Kernel

(a) Original Image

(b) Filtered Image

Figure 6.22: Filtering Image B with the Edge Kernel

(a) Original Image

(b) Filtered Image

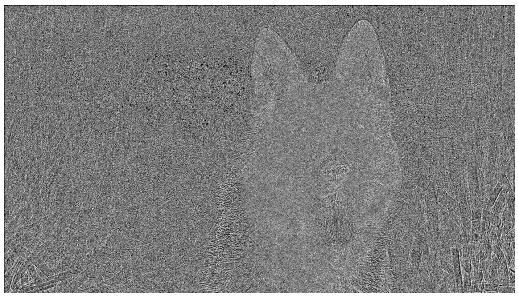
Figure 6.23: Filtering Image C with the Edge Kernel

The filtered images are our first set of outputs that clearly distinguish edges from their objects. Because of this, the images are largely black with only the white edges displayed. The smaller images do clearly pick out the dogs from their backgrounds, with image B's pebbles and image C's grass also detected. Its effects on the larger images were less expected, very reliably removing any and all background objects, and picking out individual hairs on the dogs as edges, best displayed in image D.

This very fine detail detection is a clear result of using a small 3x3 kernel on the large images instead of scaling up the kernel to a 5x5 or 7x7 to catch more general silhouettes. In other words, it is better suited for fine feature detection than gross shape clumping the larger the images become.

(a) Original Image

(b) Grey-scaled Image



(c) Windowed Image

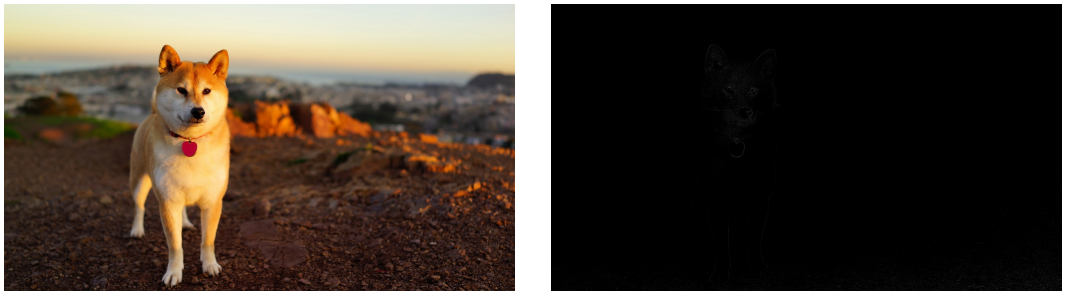
(d) Filtered Image

Figure 6.24: Filtering Image D with the Edge Kernel

(a) Original Image

(b) Filtered Image

Figure 6.25: Filtering Image E with the Edge Kernel

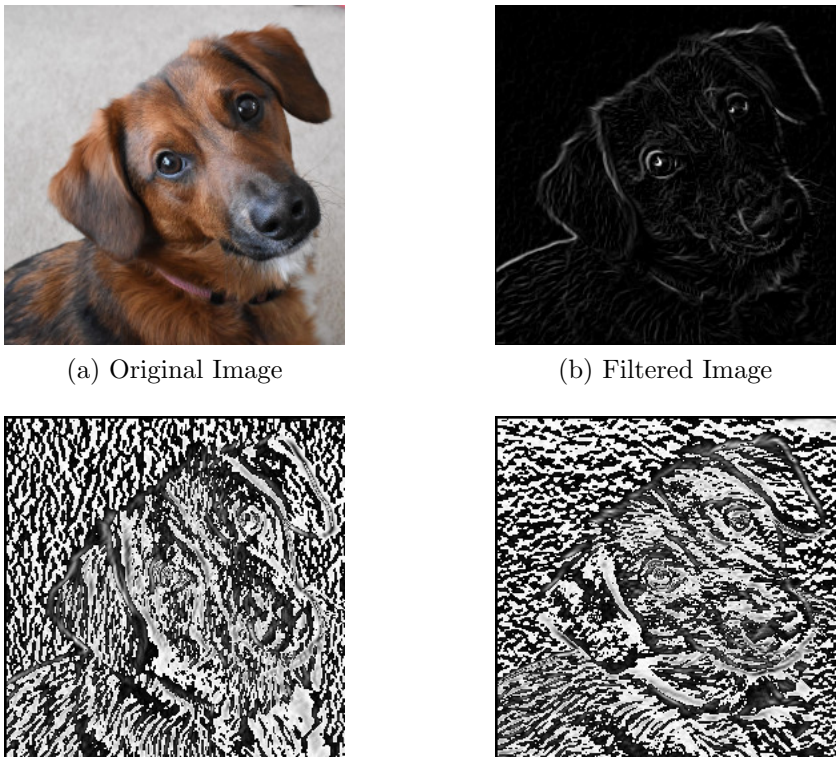


(a) Original Image (b) Filtered Image

Figure 6.26: Filtering Image F with the Edge Kernel

6.5 Prewitt

The Prewitt filter is the first dual-kernel shown so far, making the intermediate results more relevant for each image. Figure 6.27 shows various parts of the process, where (a) is the original, RGB input image and (b) is the filtered output. Sub-figure (c) shows the intermediate calculation regarding the gradient in the x-direction and (d) shows the intermediate y-direction gradient. Each output pixel in sub-figure (b) is calculated by taking the corresponding pixels from the results of sub-figures (c) and (d), passing them through the ALU, and taking their Euclidean norm. Figures 6.28 and 6.29 maintain the same, four sub-figure format in regards to the other 256x256 sized images. Figures 6.30 through 6.32 similarly display these steps for the 1920x1080 images.



(a) Original Image

(b) Filtered Image

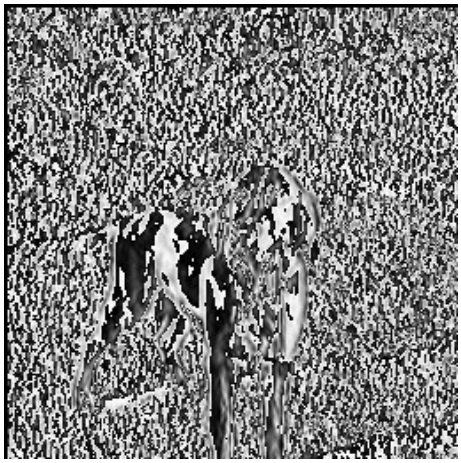
(c) X-Windowed Image

(d) Y-Windowed Image

Figure 6.27: Filtering Image A with the Prewitt Kernel

(a) Original Image

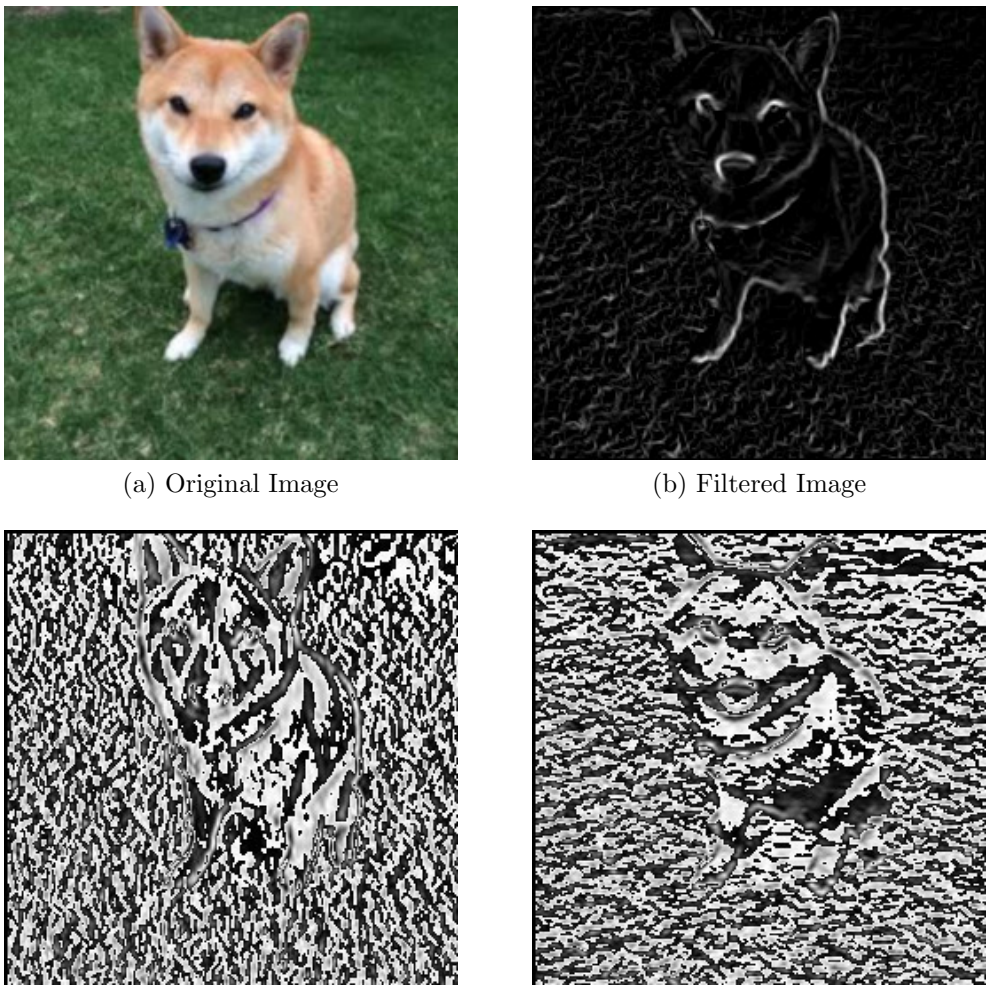
(b) Filtered Image



(c) X-Windowed Image

(d) Y-Windowed Image

Figure 6.28: Filtering Image B with the Prewitt Kernel



(a) Original Image

(b) Filtered Image

(c) X-Windowed Image

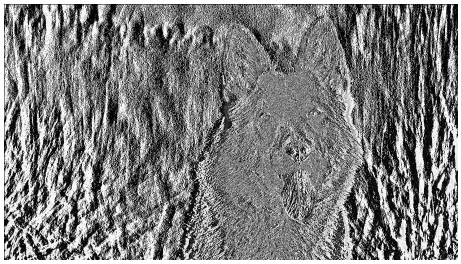
(d) Y-Windowed Image

Figure 6.29: Filtering Image C with the Prewitt Kernel

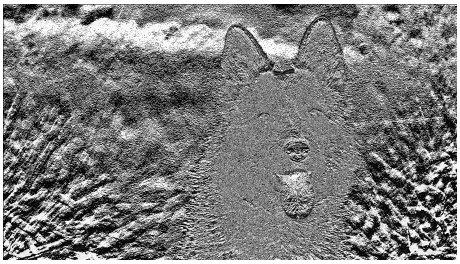
The Prewitt operation works fairly well in capturing the silhouettes of the dogs in the smaller images. Out of these, image B is the least favorably affected, having the had shadows and pebbles included as edges. While the grass in image C is not removed entirely, the dog's outline is definitely much clearer to us. When applied to the larger images, we again find features, rather than outlines are clearly highlighted, working best on image D. While image E is not as clear as the others, it maintains a high level of detail.

(a) Original Image

(b) Filtered Image



(c) X-Windowed Image

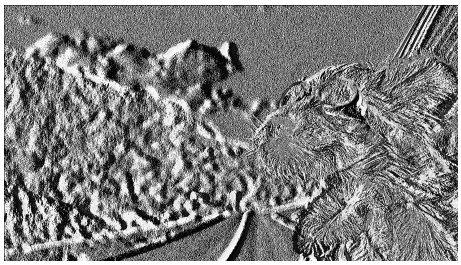


(d) Y-Windowed Image

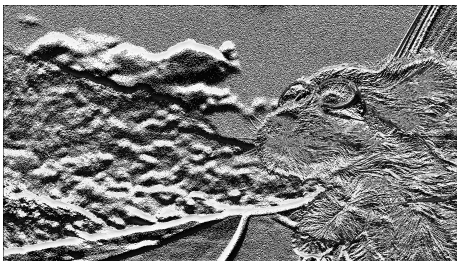
Figure 6.30: Filtering Image D with the Prewitt Kernel

(a) Original Image

(b) Filtered Image



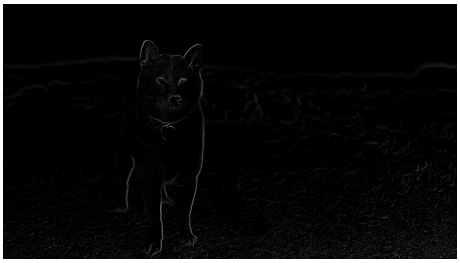
(c) X-Windowed Image



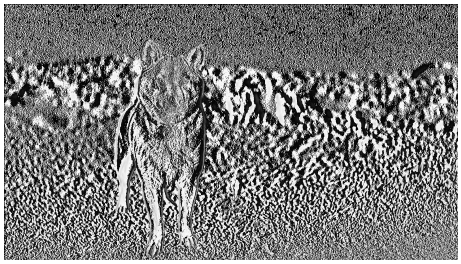
(d) Y-Windowed Image

Figure 6.31: Filtering Image E with the Prewitt Kernel

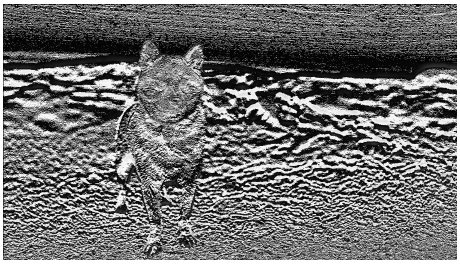
(a) Original Image



(b) Filtered Image



(c) X-Windowed Image



(d) Y-Windowed Image

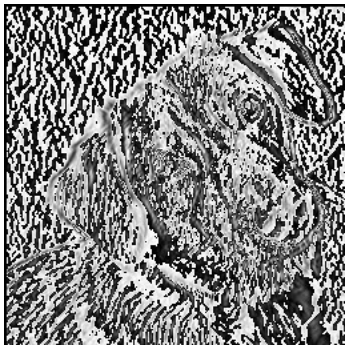
Figure 6.32: Filtering Image F with the Prewitt Kernel

6.6 Sobel

The Sobel-Feldmann filter is the most popular dual-kernel in image processing, giving a very good result for notably compact hardware. Figure 6.33 shows various parts of the process, where (a) is the original, RGB input image and (b) is the filtered output. Sub-figure (c) shows the intermediate calculation regarding the gradient in the x-direction and (d) shows the intermediate y-direction gradient. Each output pixel in sub-figure (b) is calculated by taking the corresponding pixels from the results of sub-figures (c) and (d), passing them through the ALU, and taking their Euclidean norm. Figures 6.34 and 6.35 maintain the same, four sub-figure format in regards to the other 256x256 sized images. Figures 6.36 through 6.38 similarly display these steps for the 1920x1080 images.

(a) Original Image

(b) Filtered Image



(c) X-Windowed Image

(d) Y-Windowed Image

Figure 6.33: Filtering Image A with the Sobel Kernel

(a) Original Image

(b) Filtered Image

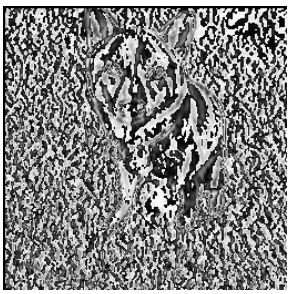
(c) X-Windowed Image

(d) Y-Windowed Image

Figure 6.34: Filtering Image B with the Sobel Kernel

(a) Original Image

(b) Filtered Image



(c) X-Windowed Image

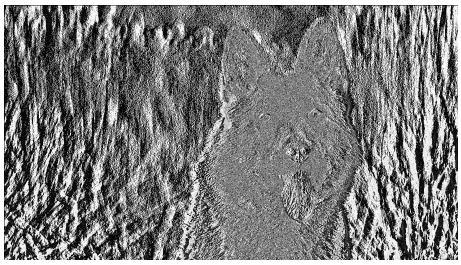
(d) Y-Windowed Image

Figure 6.35: Filtering Image C with the Sobel Kernel

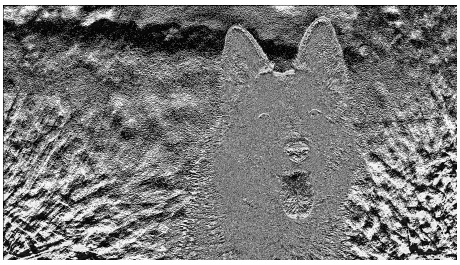
The Sobel-Feldmann operation's results are quite similar to the Prewitt operation's, catching approximately the same amount of detail in the smaller image set. Image B, in particular, seems to have a better silhouette of the dog but still has detects the shadows and pebbles as edges. When applied to the larger images, the Sobel filter maintains the level of detail in the dogs but does minimize the detail in unwanted background objects, such as a the wheat in image D and the horizon in image F.

(a) Original Image

(b) Filtered Image



(c) X-Windowed Image

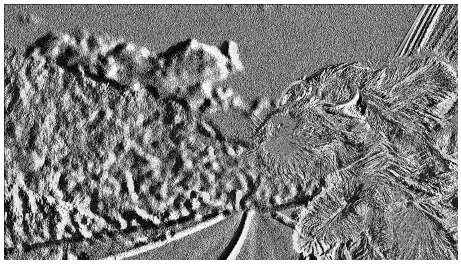


(d) Y-Windowed Image

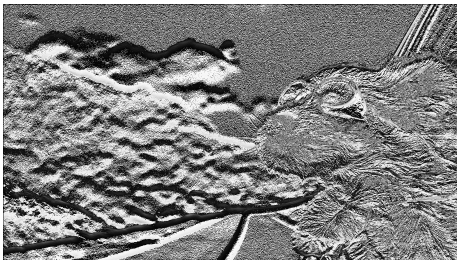
Figure 6.36: Filtering Image D with the Sobel Kernel

(a) Original Image

(b) Filtered Image



(c) X-Windowed Image

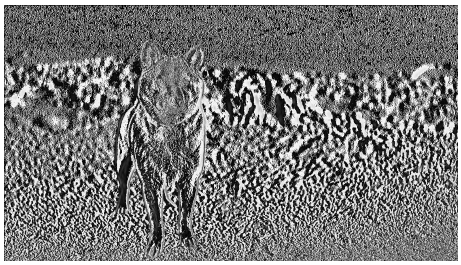


(d) Y-Windowed Image

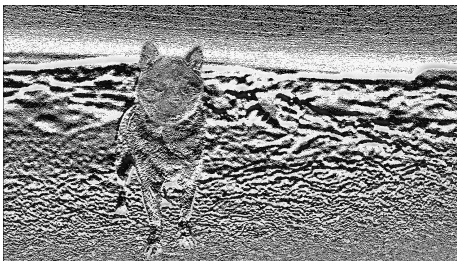
Figure 6.37: Filtering Image E with the Sobel Kernel

(a) Original Image

(b) Filtered Image



(c) X-Windowed Image



(d) Y-Windowed Image

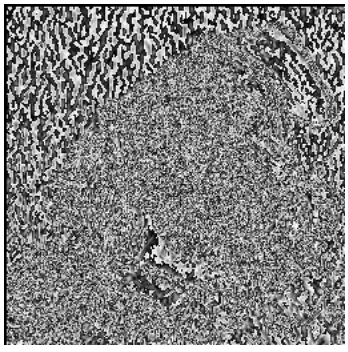
Figure 6.38: Filtering Image F with the Sobel Kernel

6.7 Popular Scharr

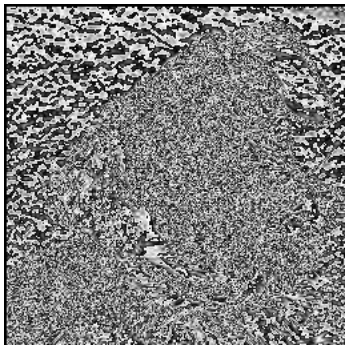
The Scharr operation with its most common set of parameters is as follows. Figure 6.39 shows various parts of the process, where (a) is the original, RGB input image and (b) is the filtered output. Sub-figure (c) shows the intermediate calculation regarding the gradient in the x-direction and (d) shows the intermediate y-direction gradient. Each output pixel in sub-figure (b) is calculated by taking the corresponding pixels from the results of sub-figures (c) and (d), passing them through the ALU, and taking their Euclidean norm. Figures 6.40 and 6.41 maintain the same, four sub-figure format in regards to the other 256x256 sized images. Figures 6.42 through 6.44 similarly display these steps for the 1920x1080 images.

(a) Original Image

(b) Filtered Image



(c) X-Windowed Image

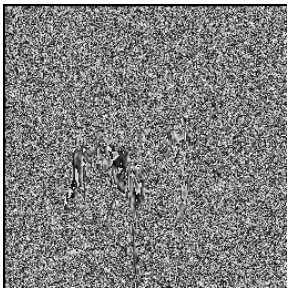


(d) Y-Windowed Image

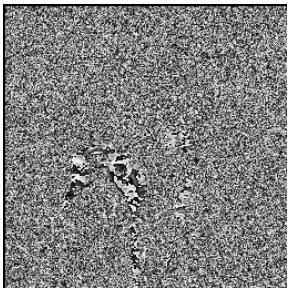
Figure 6.39: Filtering Image A with the most popular Scharr Kernel

(a) Original Image

(b) Filtered Image



(c) X-Windowed Image

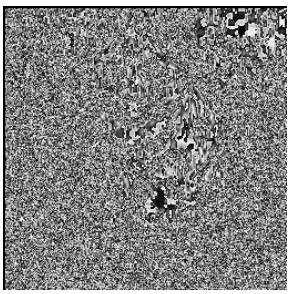


(d) Y-Windowed Image

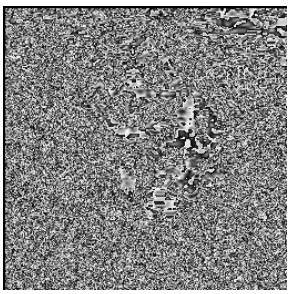
Figure 6.40: Filtering Image B with the most popular Scharr Kernel

(a) Original Image

(b) Filtered Image



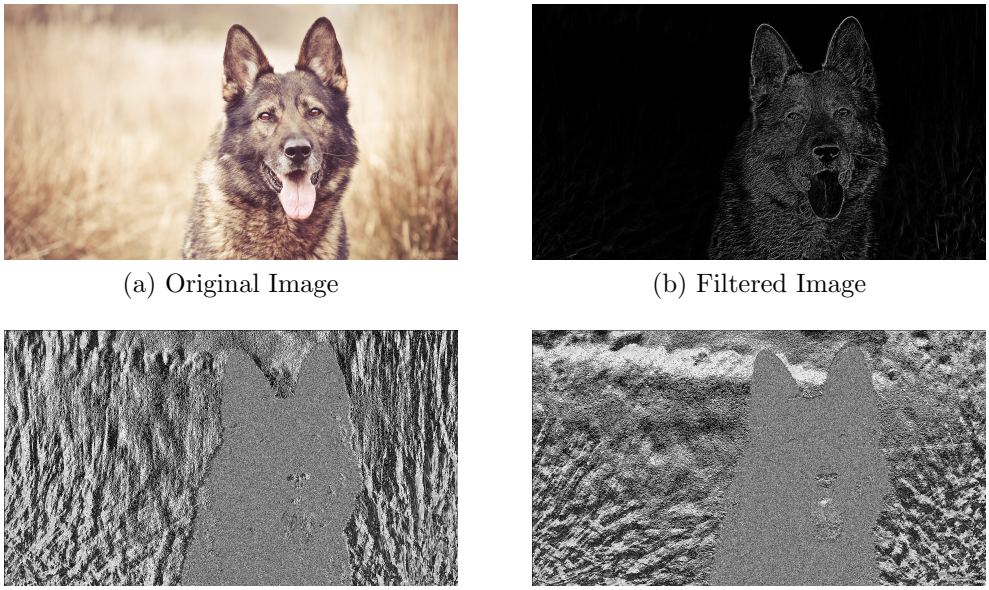
(c) X-Windowed Image



(d) Y-Windowed Image

Figure 6.41: Filtering Image C with the most popular Scharr Kernel

The Scharr operator with common parameters catches more detail than the Prewitt and Sobel do by an insubstantial amount. While the dog is always clear in the smaller images, the background detail is also very high. When against a simple background as in image A, the kernel operates very well, and does quite well with image C's grass as well. Image B, however, remains difficult to correctly separate. In the larger images it catches features clearly and sharply, but maintains more background detail than the Sobel-Feldmann operator did. This is most clear in image E, where the foliage and road are detected.



(a) Original Image

(b) Filtered Image

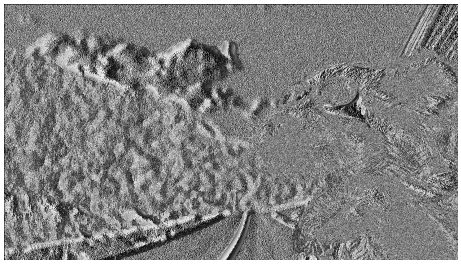
(c) X-Windowed Image

(d) Y-Windowed Image

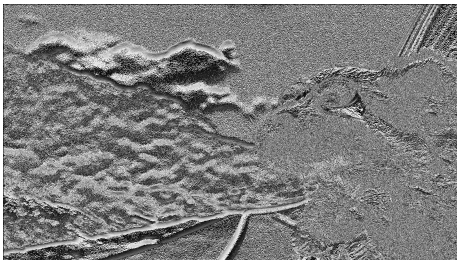
Figure 6.42: Filtering Image D with the most popular Scharr Kernel

(a) Original Image

(b) Filtered Image



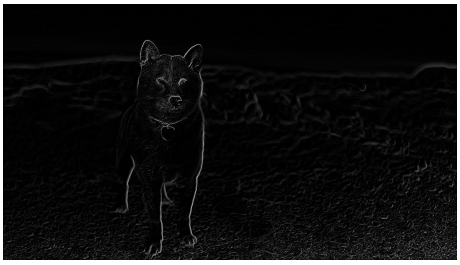
(c) X-Windowed Image



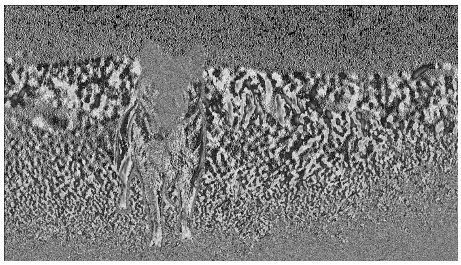
(d) Y-Windowed Image

Figure 6.43: Filtering Image E with the most popular Scharr Kernel

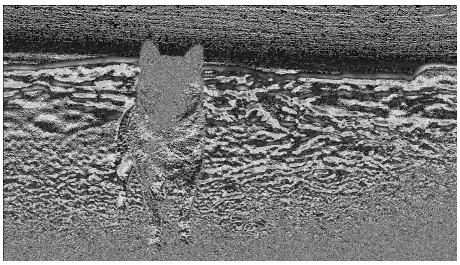
(a) Original Image



(b) Filtered Image



(c) X-Windowed Image



(d) Y-Windowed Image

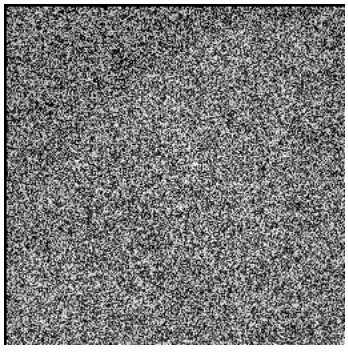
Figure 6.44: Filtering Image F with the most popular Scharr Kernel

6.8 Ideal 8-bit Scharr

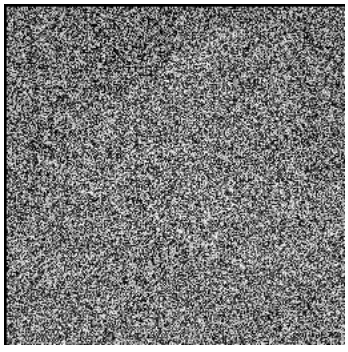
The Scharr operation with its idealized 8-bit parameters is as follows. Figure 6.45 shows various parts of the process, where (a) is the original, RGB input image and (b) is the filtered output. Sub-figure (c) shows the intermediate calculation regarding the gradient in the x-direction and (d) shows the intermediate y-direction gradient. Each output pixel in sub-figure (b) is calculated by taking the corresponding pixels from the results of sub-figures (c) and (d), passing them through the ALU, and taking their Euclidean norm. Figures 6.46 and 6.47 maintain the same, four sub-figure format in regards to the other 256x256 sized images. Figures 6.48 through 6.50 similarly display these steps for the 1920x1080 images.

(a) Original Image

(b) Filtered Image



(c) X-Windowed Image

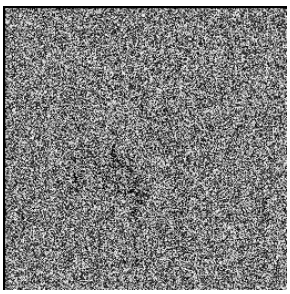


(d) Y-Windowed Image

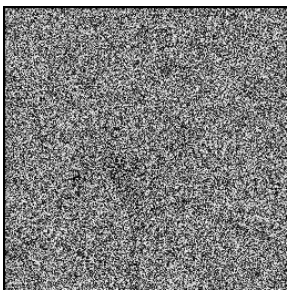
Figure 6.45: Filtering Image A with the ideal 8-bit Scharr Kernel

(a) Original Image

(b) Filtered Image



(c) X-Windowed Image

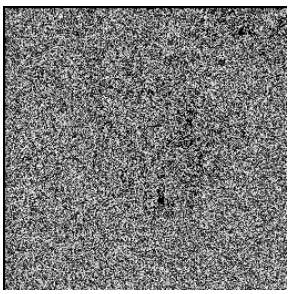


(d) Y-Windowed Image

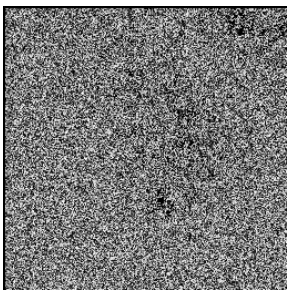
Figure 6.46: Filtering Image B with the ideal 8-bit Scharr Kernel

(a) Original Image

(b) Filtered Image



(c) X-Windowed Image



(d) Y-Windowed Image

Figure 6.47: Filtering Image C with the ideal 8-bit Scharr Kernel

While this kernels parameters are said to be idealized for 8-bit inputs, in practice this seems to be sub-optimal. While image A can be re-filtered to block together patches of black and white for a decent result, the same cannot be said for images B and C. The same seems to be true of the larger inputs, with image D being workable, but images E and F of little use for this application. While unfortunate, I have confirmed this filter is properly operating and is not the victim of integer overflow.

(a) Original Image

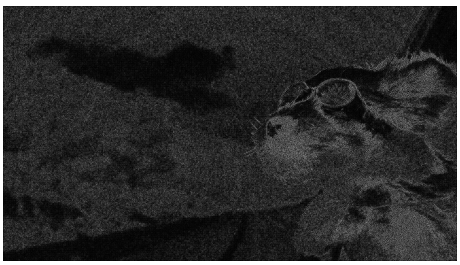
(b) Filtered Image

(c) X-Windowed Image

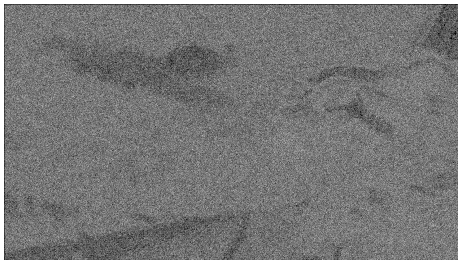
(d) Y-Windowed Image

Figure 6.48: Filtering Image D with the ideal 8-bit Scharr Kernel

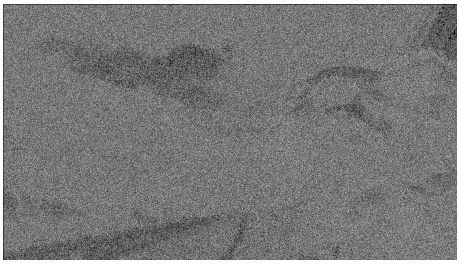
(a) Original Image



(b) Filtered Image



(c) X-Windowed Image

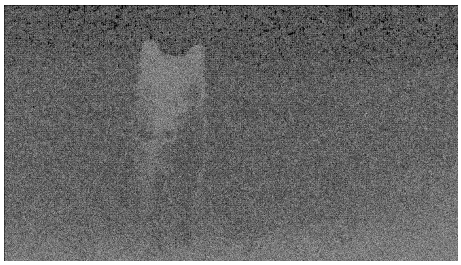


(d) Y-Windowed Image

Figure 6.49: Filtering Image E with the ideal 8-bit Scharr Kernel

(a) Original Image

(b) Filtered Image



(c) X-Windowed Image

(d) Y-Windowed Image

Figure 6.50: Filtering Image F with the ideal 8-bit Scharr Kernel

Chapter 7

Discussion

We begin the discussion by comparing the results of the RTL project generated by Vivado HLS and the RTL tailored by hand. For brevity, only the Sobel kernel's results will be reviewed in this manner.

7.1 Comparing the Generated and Tailored RTL

Below are Figures 7.1 through 7.6, which compare side-by-side the original image, the Vivado HLS C/RTL co-simulation results, and the tailored RTL results.

What becomes immediately apparent is that, while it finds the same edges, the tailored RTL is more sensitive than the Vivado generated code. This can be remedied or modified by shifting the intermediate results one to three bits to the right, effectively decreasing the number of edges detected. As to what sensitivity level is optimal lies in the application, as previously discussed, though for most intents and purposes fewer, major edges are more valuable than more, minor edges.

(a) Original Image (b) Generated RTL Filtered (c) Tailored RTL Filtered

Figure 7.1: Filtering Image A with the Sobel Kernels

(a) Original Image (b) Generated RTL Filtered (c) Tailored RTL Filtered

Figure 7.2: Filtering Image B with the Sobel Kernels

(a) Original Image (b) Generated RTL Filtered (c) Tailored RTL Filtered

Figure 7.3: Filtering Image C with the Sobel Kernels

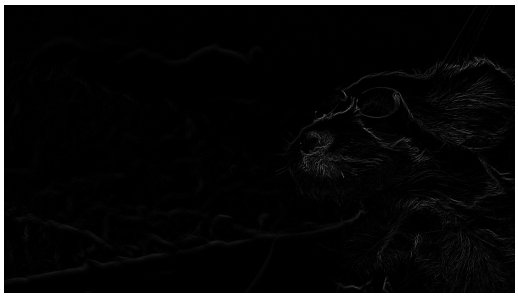
(a) Original Image

(b) Generated RTL Filtered Image

(c) Tailored RTL Filtered Image

Figure 7.4: Filtering Image D with the Sobel Kernels

(a) Original Image



(b) Generated RTL Filtered Image

(c) Tailored RTL Filtered Image

Figure 7.5: Filtering Image E with the Sobel Kernels

(a) Original Image

(b) Generated RTL Filtered Image

(c) Tailored RTL Filtered Image

Figure 7.6: Filtering Image F with the Sobel Kernels

7.2 Resource Usage Comparison

For single-kernel algorithms, there was little to no worry about space requirements, allowing us to focus on performance. When implemented with a 256x256 image, resource utilization across the board was in the single digits, unrolled or not. The HLS code only allows us to unroll loops, not to instantiate multiple instances of the kernel. The unrolled datapath does not present any bottlenecks, allowing each pixel calculation to be completed in two clock cycles. It is pipelined to accommodate a new input pixel each cycle without disturbing in-process results. Due to the single datapath, we can only operate on one pixel at a time. In a 256x256 image, this results in 65,536 pixels to process.

When implemented with a 1920x1080 image, the resources used, even unrolled, were still extremely low. The kernel and ALU remain the same size, making the line

buffers used for windowing the only component that scales with image size. This being said, our maximum throughput of one pixel a clock cycle quickly catches up to us, as 65,536 pixels becomes 2,073,600 pixels.

For dual-kernel algorithms, more hardware is required. The results from the line buffer are fanned-out to avoid doubling the hardware count of the windowing module. The kernel and ALU are both cloned and a module that approximates the Euclidean norm of the squares is added. This approximation, taking the absolute value of the x and y values and summing them, typically requires an absolute value module and a full adder. Due to our sign convention, the sign bit can be dropped without taking up more hardware or clock cycles. Since we cannot reuse our adder from earlier in the circuit without throwing a wrench in the pipeline, the additional 8-bit full adder is a mandatory addition. Again, these components will not have to scale up with the image size and can be fully unrolled without worry of resource usage. Even when implemented for a 1920x1080 image and fully unrolled, a single datapath still does not exceed single digit resource utilization in any category.

The FPGA selected and targeted for this project is the Zynq-7000 xc7z030fbg676-1. This device's maximum frequency is indicated in the user's manual as 667 MHz, as it is speed grade -1. There are FPGAs in this family that have higher maximum attainable frequencies, but as we will see in Figure 7.7, it would not benefit us.

Unrolled Datapath	BRAM (18 Kb/block)	DSP Slices	Flip-Flops	LUTs	Max frequency
Single-kernel 256x256	5 Blocks	9	1051	1343	177.336 MHz
Single-kernel 1920x1080	5 Blocks	9	1067	1386	177.336 MHz
Dual-kernel 256x256	7 Blocks	18	2044	2686	173.370 MHz
Dual-kernel 1920x1080	7 Blocks	18	2076	2772	173.370 MHz
Total FPGA Resources	530 Blocks	400	157,200	78,600	677 MHz

Figure 7.7: Resource Usage

What one may notice here is that single-kernel cores operate over same-sized images with equal time and resources, regardless of operator given. The same is

true for the dual-kernel cores as well. Predictably, larger images require only slightly more resources over their counterparts as the largest growing components, the line buffers, are still contained in the same number of BRAM blocks.

The dual-kernel cores require roughly double the resources as the single-kernel cores. Most notably, we save three BRAM blocks with the aforementioned fan-out of the windowing core. While the additional module approximating the Euclidean norm of the squares does add some hardware, the impact on resources has been successfully minimized.

While the lack of diversity single-kernels and dual-kernels have amongst themselves has been abstracted out, this can be hand-adjusted if desired. Depending on the algorithm used, multipliers that are dedicated to powers of two can be replaced for shift registers or bus manipulation at the RT Level.

As each tailored RTL datapath may have slight differences, we continue to focus on the tailored Sobel kernel. The generated RTL utilized Block RAMs for the line buffers and windowing module. BRAMs, while valuable resources we have aplenty, are also very expensive when it comes to both power and space. The tailored RTL implementation instead opts to use slices (half a Combinational Logic Block) for these submodules. The usage comparisons can be seen below in Figure 7.8.

Datapaths	BRAM (18 Kb/block)	Slices	Flip-Flops	Slice LUTs	Power Usage
Generated RTL 256x256	7 Blocks	18 DSP	2,044	2,686	1,703 mW
Generated RTL 1920x1080	7 Blocks	18 DSP	2,076	2,772	1,707 mW
Tailored RTL 256x256	0	187	467	351	152 mW
Tailored RTL 1920x1080	0	828	2,134	1,664	164 mW
Total FPGA Resources	530 Blocks	400 DSP / 19,650	157,200	78,600	-

Figure 7.8: Resource Usage

Figure 7.8 confirms both the smaller footprint and the power reduction of the tailored designs. The Block RAMs and DSP slices utilized by the generated RTL are both more power intensive than the slice LUTs and slices of the tailored design. The bulk of the power utilized by the generated design, 1,527 mW of the approximately

1,700 mW, is due to the IP requiring the PS7 block (the Zynq-7000's processing system) for implementation. Even discounting the power used by the PS7, both the 256x256 and 1920x1080 tailored RTL datapaths save power over their generated counterparts.

7.3 Future work

Further research includes the implementation of multiple cores at once so that the current bottleneck of one new pixel per clock can be overcome. One method involves splitting the input image into parts pre-FPGA, filtering them, then reconstituting them either in or outside of the FPGA. A more sensible method would be to feed in the whole image and store it in internal memory before accessing four pixels worth of information at once with four image processing module instantiations. An example datapath is shown below in Figure 7.9.

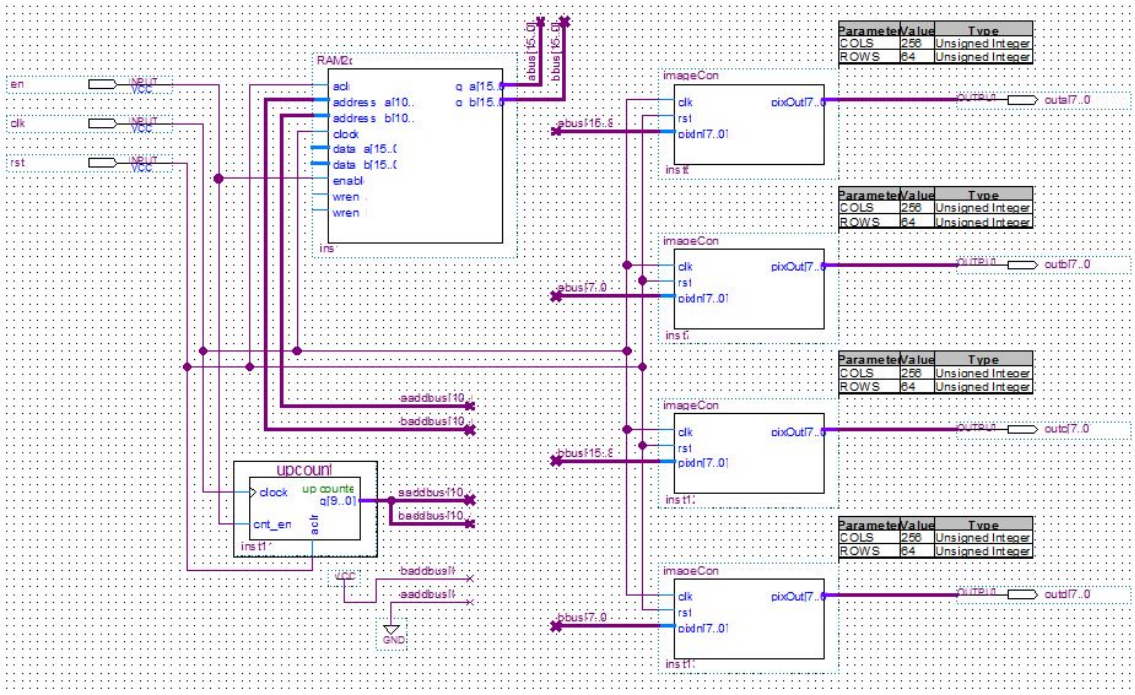


Figure 7.9: RTL Diagram of four Cores in parallel

This, of course, entails the addition of a large RAM that can scale with and store the entire image's data. It also turns a real-time system to an offline system that requires the RAM to refresh with new data between each image frame.

Instantiating multiple copies of the datapath on the same FPGA is entirely reasonable, knowing the resources one datapath requires, so long as the on-chip memory is used for the additional RAM for image storage. The issue arises in the storage RAM's rapid growth with image size and in the necessity to operate offline to re-initialize its entire contents with each successive frame.

The example setup in Figure 7.9 above shows four image convolution cores (be they singles-kernel or double-kernel cores) which share two, 16-bit buses containing four pixels' worth of information. Another choice, especially for larger images, is to share one, 32-bit bus with four pixels' worth of data.

```
:02000000c1bc80
:02000100bdbf80
:02000200c0c279
:02000300c1c178
:02000400c0c376
:02000500c1c176
```

Figure 7.10: Section of RAM initializer

Regardless, the image must now be interpreted as four, narrow image strips. Herein lies the pre-processing required to properly format the RAM initializer file. Figure 7.10 shows the the initialization of the first six RAM addresses in the format Quartus II expects. The first byte indicates the how many bytes of data will be loaded, the next two bytes are the address to store in, followed by two bytes of zeroes, our telegraphed four bytes of data, ended with a check-sum byte. In Figure 7.10's case, the first line stores two bytes of data into address 0000: c1 and bc. Line

two stores data bd and bf into address 0001, and so on.

Whether or not this form of offline parallelism merits future research varies with the module's intended application. This project was designed to intake images a stream of pixel data in raster-scan order and to work in real time. That being said, a datapath with n image convolution cores paired with a MATLAB script that can cut the image into n strips and correctly order them in the RAM initializer hex file can be a powerful tool for convolving large images offline. On the other hand, feature/edge tracking would be better served by a single core that can operate in real time, as this project provides.

Appendix A

Pseudo Code for 1D Convolution

```
1 kSize = 3;
2 for (i=0; i<kSize; i++) {
3     out[i] = 0;
4     for (j=0; j<kSize; j++) {
5         out[i] += in[i-j] * ker[j];
6     }
7 }
```

Appendix B

Pseudo Code for 2D Convolution

```
1  kSize = 3;                    //Define 3x3 kernel
2  kCenter = kSize / 2;          //Center of 3x3 kernel is (1,1)
3  for (i=0; i<imgRows; i++) {             //As you process rows
4      for (j=0; j<imgCols; j++) {         //At each column
5          for (m=0; m<kSize; m++) {       //Convolve through
6              for (n=0; n<kSize; n++) {   //the 2D kernel
7                  iin = i - kCenter;      //Find the first pixel
8                  jin = j - kCenter;      //to be convolved
9
10                 //Make sure it's not out-of-bounds
11                 if (iin>=0 | iin<imgRows | jin>=0 | jin<imgCols)
12                     out[i][j] += in[iin][jin] * ker[m][n];
13                 //and if it was, zero the result
14                 else
15                     out[i][j] = 0;
16 }   }   }   }
```

Appendix C

Separable 2D Convolution

Noting that the 2D convolution takes many more nested loops to compile, this appendix highlights that certain kernels are separable. If a 3x3 matrix can be divided into a 3x1 times a 1x3 matrix, such as the Sobel operators in the figures below:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

Figure C.1: Sobel Kernel X Separability

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

Figure C.2: Sobel Kernel Y Separability

then we can perform two, 1D convolutions and multiply their results to obtain the same results as the 2D convolution. Whether the 1D convolutions are done serially or in parallel and then multiplied does not affect the result. Note that most kernels implemented in this paper are separable by design, but not all are, such as the Edge operand.

Bibliography

[1] System-level modeling in systemc. https://m.esa.int/Our_Activities/Space_Engineering_Technology/Microelectronics/System-Level_Modeling_in_SystemC. (Accessed on 04/01/2019).

[2] Verilog hdl background and history. https://reference.digilentinc.com/learn/fundamentals/digital-logic/verilog-hdl-background-and-history/start. (Accessed on 04/01/2019).

[3] Verilog: A brief history. http://syllabus.cs.manchester.ac.uk/ugt/2018/COMP32212/01_Verilog.pdf, 2015. (Accessed on 04/01/2019).

[4] About. https://opencv.org/about.html, 2019. (Accessed on 04/01/2019).

[5] Systemc. https://www.accellera.org/downloads/standards/systemc, 2019. (Accessed on 04/01/2019).

[6] Song Ho Ahn. Convolution. http://www.songho.ca/dsp/convolution/convolution.html, 2018. (Accessed on 04/01/2019).

[7] Song Ho Ahn. Luminance. http://www.songho.ca/dsp/luminance/luminance.html, 2018. (Accessed on 04/01/2019).

[8] Blaise Barney. Introduction to parallel computing. https://computing.llnl.gov/tutorials/parallel_comp, June 2018. (Accessed on 04/01/2019).

[9] Stephen Edwards. Verilog language. http://web.cecs.pdx.edu/~mperkows/CLASS_VHDL_99/verilog.pdf, 2001. (Accessed on 04/01/2019).

[10] Cornelia Fermuller and Marc Pollefeys. Edge detection. https://www.ics.uci.edu/~majumder/DIP/classes/EdgeDetect.pdf. (Accessed on 04/01/2019).

[11] Michael Flynn. Some computer organizations and their effectiveness. https://ieeexplore.ieee.org/document/5009071, September 1972. (Accessed on 04/01/2019).

[12] Kris Kitani. Image gradients and gradient filtering. http://www.cs.cmu.edu/~16385/s17/Slides/4.0_Image_Gradients_and_Gradient_Filtering.pdf, 2017. (Accessed on 04/01/2019).

[13] Vijay Nagarajan. lecture02-types. http://www.inf.ed.ac.uk/teaching/courses/pa/Notes/lecture02-types.pdf, 2017. (Accessed on 04/01/2019).

[14] Jay Neitz and Gerald Jacobs. Polymorphism of the long-wavelength cone in normal human colour vision. https://www.nature.com/articles/323623a0, October 1986. (Accessed on 04/01/2019).

[15] Marek Perkowski. Introduction to parallel computing. http://web.cecs.pdx.edu/~mperkows/CLASS_VHDL_99/050.Introduction-to-Parallel-Computing.pdf, 2001. (Accessed on 04/01/2019).

[16] Hanno Scharr. Optimal operators in digital image processing. http://archiv.ub.uni-heidelberg.de/volltextserver/962/1/Diss.pdf, May 2000. (Accessed on 04/01/2019).

[17] Irwin Sobel. An isotropic 3x3 image gradient operator. https://www.researchgate.net/publication/239398674_An_Isotropic_3x3_Image_Gradient_Operator, February 2014. (Accessed on 04/01/2019).

[18] Jon Squire. Pipelining data forwarding. https://www.csee.umbc.edu/~squire/cs411_l19.html, January 2019. (Accessed on 04/01/2019).

[19] Michal Strzelecki. Image filtering. http://mstrzel.eletel.p.lodz.pl/mstrzel/pattern_rec/filtering.pdf, 2002. (Accessed on 04/01/2019).

[20] Eric Weisstein. Convolution. http://mathworld.wolfram.com/Convolution.html. (Accessed on 04/01/2019).

[21] Matt Zucker. Filters. http://www.swarthmore.edu/NatSci/mzucker1/e27/filter-slides.pdf, February 2013. (Accessed on 04/01/2019).