# Image Filtering in VHDL

Utilizing the Zybo-7000

Austin Copeman, Azam Tayyebi
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: akcopema@oakland.edu, atayyebi@oakland.edu

***Abstract*—This project involves the use of a FPGA and microprocessor to successfully implement image filters on the FPGA and control them using the microprocessor. Utilizing the Zybo-7000 developmental board, we were successfully able to create two simple filters, Gaussian and Grayscale.**

## I. Introduction

The use of image filtering is an important and broad area of study. By filtering images, we are able to; sharpen edges, remove noise, line detection and many more applications. These considerations are important when considering which filter to use.

On modern day computers most image filtering is done by the microprocessor using complex algorithms. These algorithms, depending on their complexity, require a powerful processor that is capable of performing all of these calculations. Even if you have a powerful enough processor to perform these calculations, the algorithms take time to complete due to the physical limitations of the microprocessor. A microprocessor, being remarkably fast, is only able to compute one task at a time. This is where the use of dedicated hardware comes into play, in our case, a FPGA.

By using dedicated hardware, we are able to lift the burden of these heavy calculations off of the processor and perform what would be daunting and time consuming formulas onto the FPGA. The FPGA, being a reprogrammable piece of hardware, is able to perform complex calculations in a timely manner compared to that of a microprocessor due to its ability to perform multiple tasks and calculations at a single time.

In our report we will go over the Gaussian filter and Grayscale filter along with their implementation. These two filters are part of the most basic filters but they are also some of the most important and fundamental filters.

## II. Methodology

Our project was designed and implemented utilizing the Xilinx Vivado IDE and Xilinx SDK. The filters and control path were all created on the Xilinx Vivado IDE using VHDL as the programming language. The microprocessor was programmed using Xilinx SDK and was programmed in C.

Modern day computers use upwards of 64bits of color for images. For simplicity, we will be using only 8bit color for our images. In hardware, images are composed of pixels. An image size depends on the amount of pixels it is composed of Each pixel is composed of three main colors; red, green blue (RGB).

With 8bit color we are able to distinguish 256 shades of color. We are able to control the shades of color by varying the intensity of RGB. To achieve the 8bit color we define the following for the color scale; 3bits for red ($b_7$ to $b_5$) and green ($b_4$ to $b_2$) and 2 bits for blue ($b_1$ to $b_0$). Blue is a more dominate color and is part of the reason for it receiving less bits compared to red and green.

To convert the images, we used a MATLAB script that would take an image and convert it to an 8bit hex text file. This file was then copied and pasted into an array in our C code.

### *A. Gaussian Filter*

In this project a filter is designed to smoothen the given grayscale image based on Gaussian blur technique figure I-I. Blurring of an image is a technique of taking a pixel as the average value of its surrounding pixels to reduce image noise and sharpness at the edges. Noise reduction is one of the major concerns in the image processing. The main goal of noise reduction is to remove information that may corrupt the image. This can be achieved by many different techniques, such as Median/Mean filtering, Gaussian filtering, applying Fourier transformation and many more.

Edges in an image are the outline that details the structure of an object in the image. Blurring is in fact a technique which is used to reduce the edges and making the transition from one color to the next color in a smooth manner. To achieve this goal in this project, a filter kernel is used. The filter kernel coefficients change the pixel's values of the image, which is to be smoothened but still preserves the valuable features of the image. Mathematically, applying a Gaussian blur to an image is the same as convolving the image with a Gaussian function.

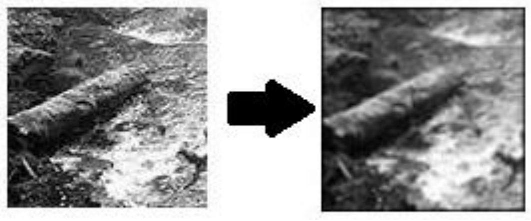

*Figure I-1: desired blur image*

To approximate the Gaussian filter. One 3*3 kernel mask is used, shown on Figure I-II. The integrals are not integers; we rescaled the array so that they are integers. This matter put us in trouble because we didn't know how to display properly in Matlab; the output was just black.

| | | |
|---|---|---|
| 1 | 2 | 1 |
| 2 | 4 | 2 |
| 1 | 2 | 1 |

*Figure I-2I: 3*3 Kernel(e=1)*

By changing the coefficients, we are able to blur input image at different degrees.

Once a suitable kernel has been calculated, then the Gaussian smoothing can be performed using standard convolution method which is multiply 9 pixels of image in the given kernel and then add them together. The method that is used to iterate the image shown in Figure I-III.A zero-padding is used for the first row and last row of the image.

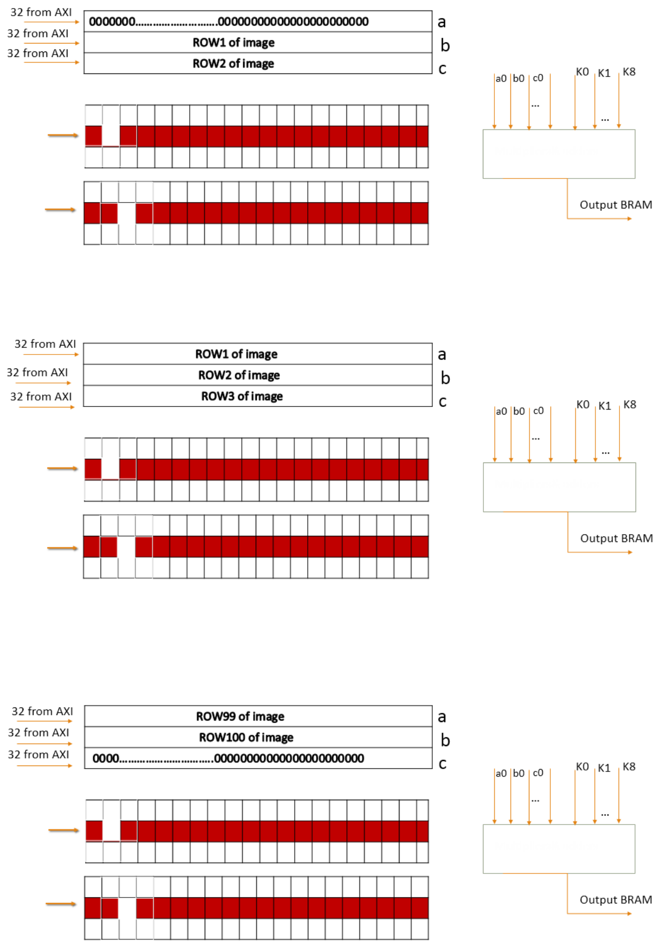

*Figure I-III: Iteration over the image.*

## 1) Implementation

The Gaussian filter was designed in VHDL first,as shown in Figure I-IV. We used the following input signals; Imaginput, addr_input, addrb. Where Imaginput is 32 bit in length that contains 4 of 8bits of input gray image and addr_input that is the address of input dual port RAM is 12 bits and also addrb which is read address of output dual port RAM is 14 bits. The outputs consisted of the following signals; image_out and done. Where image_out is 16bits in length and done is a 16bits length.

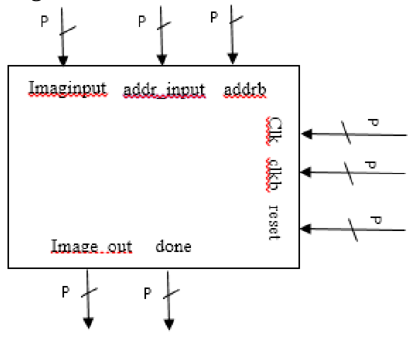

*Figure I-IV: block diagram of design*

### a) Hardware Implementation

To design the filter, different steps are considered.

In the first step, 32 bits data input is read and save in a Dual port RAM which its address is sending by processor at the same time with input pixels.

In the second step, three rows of the image, which is stored in a BRAM in the first step, are read and save in registers a,b,c.

In the last step, convolution are performed on the pixels. Then output pixels are saved in the output dual port RAM. In this step, convolution is performed on one row of the input image then we should go back to step 2 and read another row and also swap row b and c with a and b. we should switch between step 2 and 3 until complete convolution for the whole rows.

Convolution as mentioned before is multiply 9 pixels of image and kernel and then add them together. The multiplier is just * and adder is + in hardware design. The result of convolution is a pixel in 16 bit length.

Two Bram are used in this design for input image and convolution results. The input one saves 32 bit * 2550=10200 byte which is the image pixels with two extra zero row for zero-padding of the first line and the last line. The output BRAM saves 16*10000 =20000 byte which is convolution results.

The steps are implemented in hardware with a finite state machine in four steps (S1,S2,S3,done).

## B. *Grayscale Filter*

A grayscale filter is a filter in which each value of each pixel is a single sample. This means that each pixel only carries intensity information. The grayscale filter gets its name from the colors it represents. The grayscale is composed exclusively of shades of gray that vary from black to white. Where black is the weakest in intensity and white is the strongest in intensity.

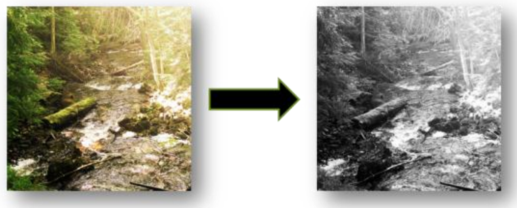

*Figure II-3: Converted Image to Grayscale*

Grayscale is used for when color is not a necessity. Using a gray color range allows for better and faster image processing compared to its color component. Grayscale filtering is also one of the first steps finding an image's edge due to how it simplifies the image.

There are many different ways on how to achieve grayscale. The first way is called the averaging method, aka "quick and dirty method". This method uses the formula, $Gray = \frac{(red+green+blue)}{3}$. Even though this method is easy to implement and generates a nice grayscale equivalent, it does a poor job of representing shades of gray relative to the way we as humans perceive brightness. The way that our grayscale filter was created allows for the changing of brightness and you are able to control the brightness for each color. This method is what we call the percentage method. The percentage method uses the formula,

$$Gray = R * R\% + G * G\% + B * B\%;$$
$$where\ R\% + B\% + G\% = 100\%$$

By controlling the percentages, we are able to represent a better way that we as humans perceive brightness. One of the other reasons of doing this is because not all humans see color the same. Depending multiple factors main ones being race and gender, the weight of each color will vary. This is one of the main reasons why the average method is an inaccurate method.

Instead, one of the common formulas that is used through all image processors is called luminosity or Luma for short.

$$Luma = R * 0.2126 + G * 0.7152 + B * 0.0722$$

This formula results in a more dynamic grayscale image. Even though this formula is an ITU-R recommendation BT.709, sometimes also called ITU656, there are other recommendations that go by different coefficients. Due to this fact, the grayscale filter was designed by us allows for integer numbers, 0-100, to be used as inputs to allow for a wide arrange of grayscale conversions.

### 1) *Implementation*

The filter was first designed in VHDL and used the iterative method. The design used can be seen in Figure B-1. We used the following input signals; RGBin, R%, G%, B%, start. Where RGBin, R%, G%, B% are 8bits in length and start is a single bit. The outputs consisted of the following signals; RGBout and done. Where RGBout is 8bits in length and done is a single bit. With the inputs and outputs already decided we were able to then design how the filter would work.

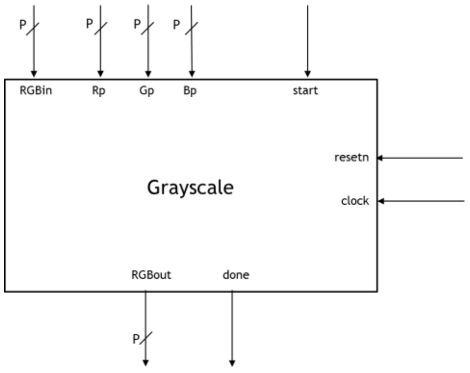

*Figure II-4: Grayscale Hardware Overview*

#### a) *Hardware Data Path*

The filter, seeming complex, actually utilizes simple components. The design can be seen in Figure B-2. The first step in the filtering process is to separate the red, green and blue bits from each other and create their own signals. To do this, in software it would take multiple clock cycles but in FPGA it takes no time.

From the signal RGBin, three separate signals were created; R, G and B. Where the three colors were separated according to their bit length and set to the most significant bit(MSB) and concatenated with 0's at the LSB until the length of 8bits was achieved for each signal.

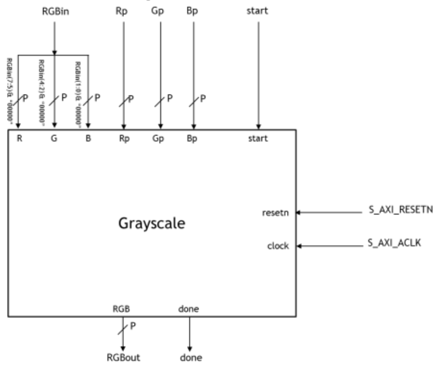

*Figure II-5: Grayscale Hardware RGB Decoder*

The circuit for the following can be seen in Figure B-3 below. From the splitting of the colors, we had two different multiplexors to receive the 8bit color signals (RGBColorMux) and the RGB percentages (RGBPercentMux). A 2bit select line was used to control which color was selected. When a color was selected based on the state machine, which will be described later in the paper. From the multiplexors, it was sent to a serial multiplier circuit.

The multiplier circuit was provided to us on Prof Daniel Llamocca website [1]. This multiplier circuit takes two numbers in size N in length and outputs a number that is 2*N in length. In our case the two numbers are the color percentage and the color. The signal then travels to the adder circuit.

The adder circuit consists of two components, the adder and an adder register (addreg) both have an input and output of size 2*N. The adder is iterative since it only takes in one number at a time and is also controlled by the state machine. Since we have three colors the adder circuit must add all three colors together that were multiplied by the multiplier circuit. In the first iteration the color red would go through and would be added by a constant 0 for the first iteration only. From here it would be loaded into the adder register to be added with the next color. After doing this until the blue color was added in it would be then sent to the divider circuit.

The divider circuit is the most complex component of the circuit. This component was also provided to us by Prof Daniel Llamocca [1]. There were multiple versions of the divider. The version used in this project is the iterative divider. The divider circuit contains multiple signals that are being used to drive it. The main signals are E and done. These signals allow us to start the divider and know when the divider is done. The next two signals that are used are the dividend and divisor. The circuit takes the output from the adder register (the dividend) and divides it by a constant 100 (divisor). The reason for doing this is because during our multiplication circuit we took a number that ranged [0,100] and multiplied by another number that ranged [0,255]. This gave us a range of numbers of [0,25500]. This meant that our output would be 2*N bits long. This is not what we want, what we want is a number that is in the range of [0,255]. By diving by a constant 100 we were able to achieve this range but with how the divider circuit is made we had to output a bit length of 2*N. Since the division had left the range of our numbers under 256, we knew that the top N bits of our 2*N bit number were all 0 we could just take the lower N bits and that would be our RGB grayscale output.

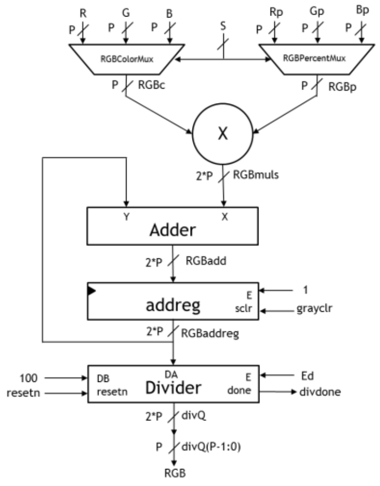

*Figure II-6: Grayscale Data Path*

### b) Hardware Finite State Machine

The state machine for the grayscale filter consists of three states (S1, S2 and S3). The state machine starts only when resetn is ‘1’. When resetn is ‘1’, we start in S1. In S1 we set our counter variable C to 0 and our grayclr signal to ‘1’. The variable C controls how many iterations we go through for our adder circuit and grayclr controls the clearing of the registers. We stay in S1 until a ‘1’ is received on the start signal. In S2 we increase our counter variable C until C = 3. When C = 3, we set C back to 0 and enable our divider circuit by sending a ‘1’ to the signal Ed. In S3, we send a ‘1’ to grayclr to clear our registers and wait until our divider is done. When the divider is done it sends a ‘1’ back to our state machine on divdone signal. When a ‘1’ is received we move back to S1 and wait for another start signal.

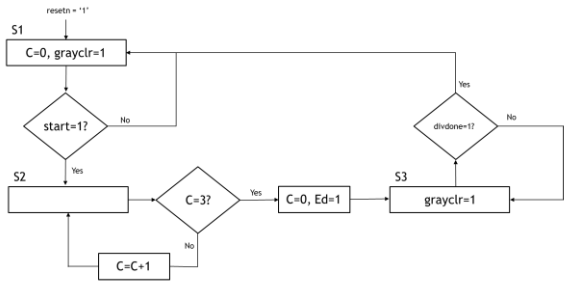

*Figure II-7: Grayscale Finite State Machine*

## C. Processor

### 1) Gaussian Filter

Three different files were created; main.c, Gaussian.c and image.h. In the file, Gaussian.c, we included our Gaussian filter header so we were able to read and write to our filter using the AXI lite bus that was created. A function called Gaussian Filter(), was also created in this file. This function took two different parameters; photo height, photo length.

The photo height and photo length were used to control the increasing of address input which is the address of input BRAM and also determine how many pixels we want to send by AXI. This file takes outputs by AXI and prints it to the terminal window.

2) *Grayscale Filter*

Three different files were created; main.c, Grayscale.c and image.h. In the file, Grayscale.c, we included our grayscale filter header so we were able to read and write to our filter using the AXI lite bus that was created. A function called grayscaleFilter(), was also created in this file. This function took five different parameters; photo height, photo length, red percent, green percent and blue percent. The photo height and photo length were used to control the look of the output. Also they determined the size of the array that was being called. The red, green and blue percentage parameters all referred to how much intensity you wanted to give to each of the three colors. This file being basic, printed everything to the terminal window.

The image.h file, contained the image that was to be converted. It also allowed so that more filter could be implemented in the future and could use the same image.

The main.c file contained our main function. This files only function was to call the filter functions and give the defined parameters.

## III. Experimental Setup

The project was written using Xilinx Vivado IDE and Xilinx SDK tools. The hardware portion of the project was written in Xilinx Vivado IDE. Testing was done for each of the filters using test benches. After verifying that the filters were working, each filter was package in its own IP. Using Vivado's block diagram software, all of the inter connections were made. The system was then placed into its own HDL wrapper and a bit file was created. Once the bit file was created the program was exported to Xilinx SDK for the programming of the microprocessor.

In Xilinx SDK separate c files were created for each filter. A single header file was created that also contained the image to be filtered. For the grayscale filter, the filtered image was outputted to a terminal window. This output was then copied and pasted into a text file. The text file was then ran and converted back into an image file by using an MATLAB script.

## IV. Results

### A. *Gaussian Filter*

We test the output of Gaussian Blur with the Matlab output for Gaussian blur that was unfortunately different. But we test input of hardware with the samples inputs that is corrected. The output of designed filter plotted by Matlab is shown in Figure IV-I.

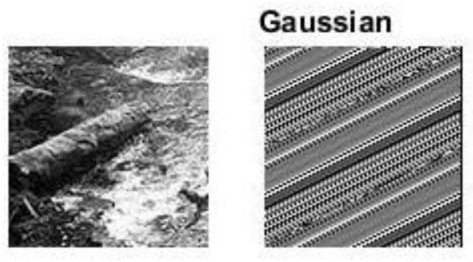

*Figure IV-1: input and output of the filter*

When I started implementing of Gaussian filter I thought it is simple because the concept seems simple. I didn't have enough knowledge about image processing and spend a lot of time to display the output with VGA but I couldn't display properly because I was in a wrong way. When I talked with Dr.Llamocca and found what I should do, it was late. And also my image output at this moment isn't a blur version of input.

### B. *Grayscale Filter*

After converting each converted image back into viewable image. We found that the filter did what it was supposed to. The images were converted using Octave, a free alternative compared to MATLAB. Using the function rgb2gray(), we were able to convert an image to grayscale using their standards and compare it to our filter. The rgb2gray filter uses the following formula:

$$rgb2gray = 0.2989 * R + 0.5870 * G + 0.1140 * B$$

Since our filter can only accept integer numbers we had to round our values to the nearest one's place. Even with this rounding our custom made filter still looked remarkably close to the one that was generated by Octave

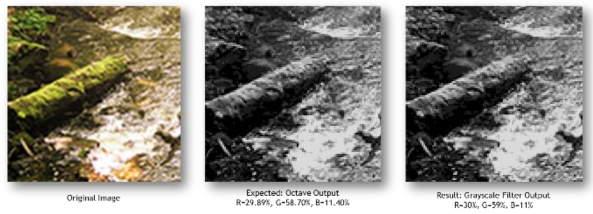

*Figure IV-2I: Comparing Images*

The differences in the images is due to how compression of saving them took place. The images were saved into a JPEG format. This compressed the images and caused slight change in some of the coloring. Before compression, if you looked at each image the pixels would match up almost pixel to pixel.

What I found while developing the grayscale filter was even though the concept may have seemed easy at first, implementing it was the hard part. Designing a system that works with both the FPGA and microprocessor seems like an easy task to do but when you start to design how you are going to implement it you realize there are many variables you have to take into account. At first, I tried to design my filter so a start signal would not have to be sent to the filter. A FSM would take care of it and when it saw something on the slave registers it would start. With how I had my filter designed this lead to a lot of headache on trying to figure out why my filter was not working.

## Conclusions

What we found was that filters, even though they may seem to be simple in code. They are actually complex when designed in hardware. We found that our hardware filters matched up to that of a computer algorithm filter. There are some slight differences due to rounding of the numbers and that a computer can do a lot more complex math then what our filters are capable doing at the moment.

One feature that was added but never able to implement was the ability to send and receive text files to the Zybo from a laptop using UARTlite.  This would have made our filter more generic, instead of having an image that was hardcoded into our code. We would have been able to send over any image that we wanted to. Receiving the image back would make the processing easier as well. We could have converted the image, sent the image, filtered the image, received the image and converted the image back in one step compared to the multiple steps we had to do now. This function could had been all done through a MATLAB script, with MATLAB's ability to use the COM port.

One other feature that we wanted to implement was the ability for the user to choose which filter they wanted and then input the parameters they wanted for the filter. This was to be done over the terminal or with the use of a GUI that would have been written in Java. This was not implanted due to not being able to get the UART completed.

Overall, we believe that we had a great result for our filters. Even though we were not able to complete everything we wanted to do for this project. We were able to successfully filter an image through the use of the Zybo-7000.

## References

[1] Llmocca, Daniel. *Reconfigurable Computing Research Laboratory*. Ed. Daniel Llmocca. N.p., n.d. Web. 2 Dec. 2015. <http://www.secs.oakland.edu/~llamocca/VHDLforFPGAs.html>.

[2] Helland, Tanner. Seven Grayscale Conversions. Ed. Tanner Helland. N.p., 1 Oct. 2011. Web. 3 Dec. 2015. <http://www.tannerhelland.com/3643/grayscale-image-algorithm-vb6/>.