

# DESIGN OF A HETEROGENEOUS ENVIRONMENT PROGRAMMING ENVIRONMENT

JARED BOLD

DEPARTMENT OF ELECTRICAL ENGINEERING

GRADUATE RESEARCH PAPER

ROCHESTER INSTITUTE OF TECHNOLOGY

# Contents

<b>1</b>	<b>Abstract</b>	<b>5</b>
<b>2</b>	<b>Introduction</b>	<b>6</b>
<b>3</b>	<b>Background</b>	<b>7</b>
3.1	Hardware - Xilinx ZC702 . . . . .	7
3.2	Operating System - Linux 3.14 . . . . .	9
3.3	Advanced Microcontroller Bus Architecture . . . . .	11
3.4	Image Filtering . . . . .	13
3.5	Bilinear Interpolation . . . . .	15
<b>4</b>	<b>Design</b>	<b>17</b>
4.1	Common Design Components . . . . .	17
4.1.1	Tagged Image File Format . . . . .	17
4.1.2	User Interface . . . . .	18
4.2	Processor Based Design . . . . .	20
4.2.1	Image Filter . . . . .	20
4.2.2	Bilinear Interpolation . . . . .	22
4.3	Programmable Logic Based Design . . . . .	22

4.3.1	Image Filter . . . . .	22
4.4	Homogeneous Design . . . . .	22
4.4.1	Bilinear Interpolation . . . . .	22
<b>5</b>	<b>Results</b>	<b>23</b>
5.1	3x3 Gaussian Filter Results . . . . .	23
5.2	9x9 LoG Filter Results . . . . .	23
5.3	Bilinear Interpolation Results . . . . .	23
<b>6</b>	<b>Conclusion</b>	<b>24</b>
	<b>Appendices</b>	<b>26</b>
<b>A</b>	<b>Useful Resources</b>	<b>27</b>
A.1	Programmable Logic . . . . .	27
A.2	Operating System . . . . .	27
A.3	Vivado . . . . .	27
<b>B</b>	<b>Verilog Sources</b>	<b>28</b>
<b>C</b>	<b>C Source</b>	<b>29</b>
<b>D</b>	<b>Reproduction of Work</b>	<b>30</b>

## List of Figures

3.1	Top-Level Diagram of Xilinx ZC702 Evaluation Board . . . . .	8
3.2	Top-Level Diagram of Xilinx Zynq device . . . . .	9
3.3	View of Linux kernel . . . . .	10
3.4	Example AXI processing pipeline . . . . .	12
3.5	Image filtering example . . . . .	13
3.6	Symmetric Filter . . . . .	14
3.7	Bilinear Interpolation of Grayscale Image . . . . .	15
4.1	TIFF File Structure[3] . . . . .	18
4.2	User Interface . . . . .	19
4.3	Processor based filter framework . . . . .	20
4.4	3x3 Gaussian blur filter kernel . . . . .	21
4.5	9x9 LoG filter kernel . . . . .	21

## List of Tables

3.1	Operations required for different filter kernels . . . . .	15
-----	--	----

## 1. Abstract

## **2. Introduction**

## 3. Background

### 3.1 Hardware - Xilinx ZC702

The hardware platform selected is the Xilinx ZC702 Evaluation Board, based around the Xilinx Zynq XC7Z020-1CLG484C. The board provides many standard peripherals such as 1 GB DDR3 RAM, 128 Mb QSPI flash memory, USB 2.0 transceiver, SD Card interface, USB JTAG interface, HDMI, Ethernet PHY with RJ-45 connector, and additional peripherals and user I/O as shown in Figure 3.1 [5]. The purpose of the evaluation board is primarily to allow for a system design to be tested prior to producing a custom board layout, and makes it an ideal development platform for this work.

The Xilinx Zynq XC7Z020-1CLG484C device combines a Processing System (PS) and Programmable Logic (PL) on the same chip, forming a System on a Chip (SoC). The PS is comprised of two ARM Cortex-A9 MPCore application processors (APUs), each containing a NEON co-processor, a 32 KB L1 instruction cache, and a 32 KB L1 data cache. The APUs share a common 512 KB L2 cache. The PL is designed using the Xilinx Artix-7 fabric, their middle-grade fabric, with 85K Logic Cells (LCs), 53,200 Look-Up Tables (LUTs), 106,400 Flip-Flops (FFs), 560 KB Block RAM, and 220



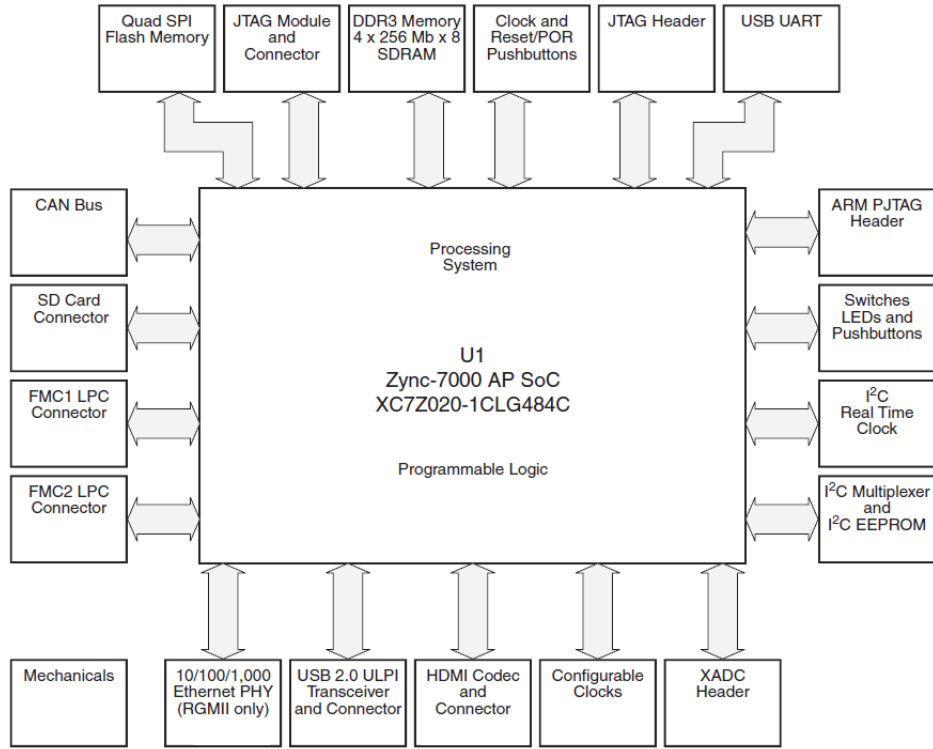


Figure 3.1: Top-Level Diagram of Xilinx ZC702 Evaluation Board

DSP Blocks [6]. Figure 3.2 shows the top level view of the composition of the Zynq device. Communication between the two portions of the chip occurs via a subset of the communication buses specified by the Advanced Microcontroller Bus Architecture (AMBA), known as the Advanced eXtensible Interface (AXI). The PL can be user programmed to provide access to additional peripherals or to offload processing from the APUs.

The reason the ZC702 was selected is two-fold. The first being the use of the Xilinx Zynq device, which provides an ideal environment for building and testing homogeneous processing environments without limitations of using separate CPU and FPGA chips. Secondly, the Zynq device and the ZC702 board have a large amount of support from the open source community. The

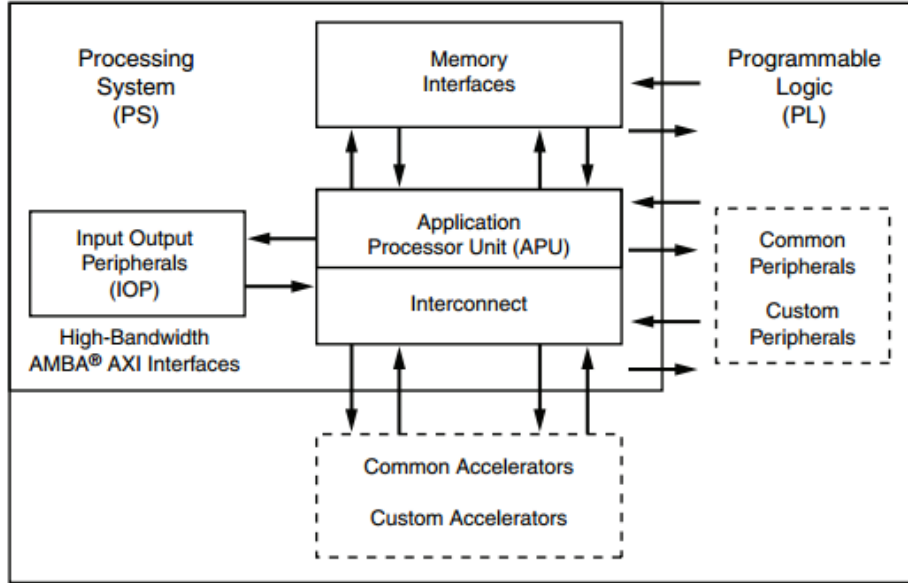


Figure 3.2: Top-Level Diagram of Xilinx Zynq device

wealth of information and example code provide an excellent starting place.

## 3.2 Operating System - Linux 3.14

For purposes of this work, the Linux operating system has been selected to run on the ZC702 evaluation board. This Linux kernel is compiled directly from source maintained by Xilinx and is based off of the mainstream Linux kernel 3.14. The Linux kernel separates the operating system into two distinct spaces, the User Space and the Kernel Space. User applications run in the User Space and use calls to the system call interface and the GNU C Library to communicate and interact with the Kernel Space, as shown in 3.3. In the Kernel Space, system resources such as memory and peripherals are accessible and can be interacted with. The memory seen within the User Space is known as a virtual memory space which maps to a physical memory

in the Kernel Space. The Linux kernel provides several features including process management, memory management, a virtual file system, a network stack, and device drivers [2]. These features allow for easier development and interaction between User Space applications and the hardware running underneath.

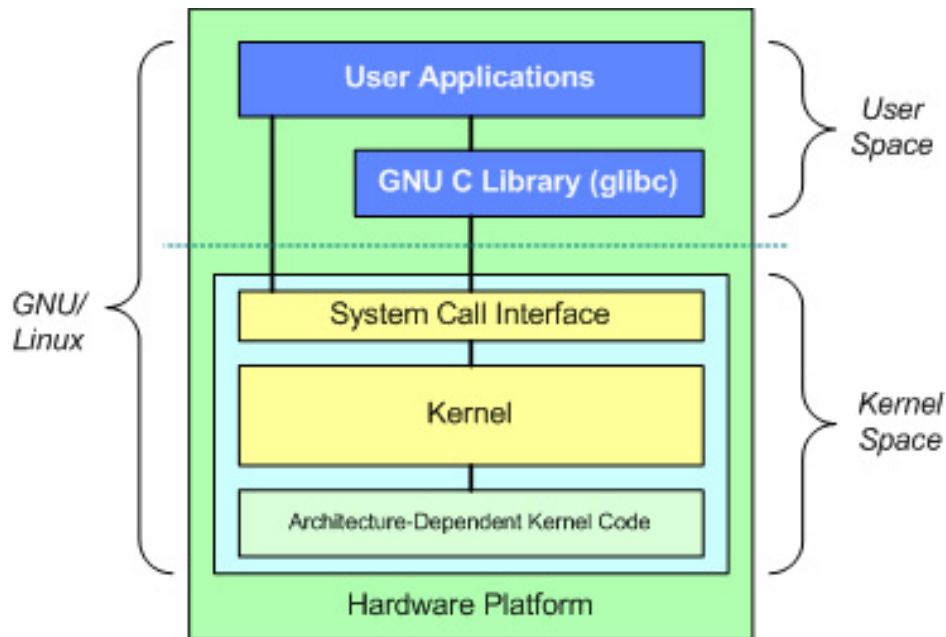


Figure 3.3: View of Linux kernel

Using Linux has several advantages over writing a bare-metal application. One of the greatest benefits comes from the portability and ease of access to both development tools and open source applications. An application that had previously been compiled for a Linux operating system can be recompiled to run in the embedded environment. These applications can range from image processing libraries (libtiff) to communications (ssh) to file transfer (ftp) to any number of other things. The tools needed to do this, such as gcc, are readily available and open source, making them free to use. In

addition to applications, Linux provides device drivers for much of the hardware in use today, and if a driver does not exist, one can be created based on an existing driver allowing for easy access and interaction with hardware peripherals. This is of great benefit when compared with having to write to device registers by hand in a bare-metal application.

In addition to device drivers, the Linux kernel provides a method of directly mapping physical memory in the Kernel Space to virtual memory in a User Space application using the `mmap` system call. This proves very useful in SoC applications where the peripherals are memory mapped and may change based on the configuration of the device. Using this mapping, the registers of an external device can be directly written to from the User Space, bypassing the need to develop a custom device driver and allowing for faster development and testing.

### **3.3 Advanced Microcontroller Bus Architecture**

Central to the design of heterogeneous systems is the necessity to move data between the different processing units of the SoC. In order to facilitate this, ARM has developed a set of specifications known as the Advanced Microcontroller Bus Architecture (AMBA) which consists of a number of protocols and interfaces to allow for the efficient movement of such data based on a Master/Slave relationship using a Ready/Valid handshake. The Zynq device supports a number of memory-mapped interfaces for communication between

the PS and PL including two 32-bit Advanced eXtensible Interface (AXI) master interfaces, two 32-bit AXI slave interfaces, four 32/64-bit high performance AXI ports for direct access to DDR, and one 64-bit AXI slave interface known as the Accelerator Coherency Port (ACP). While the memory-mapped interfaces allow for data to be easily transferred into and out of the PL, a method of easily streaming data through the PL for processing is also provided by the AXI streaming interface. Using IP developed around the AXI stream protocol, different processing blocks are easily linked together to form a processing pipeline. Figure 3.4 shows an example processing pipeline utilizing both the memory-mapped and streaming AXI interfaces. Xilinx provides a number of IP cores to be used in hardware development to interface with both memory-mapped and streaming AXI interfaces.

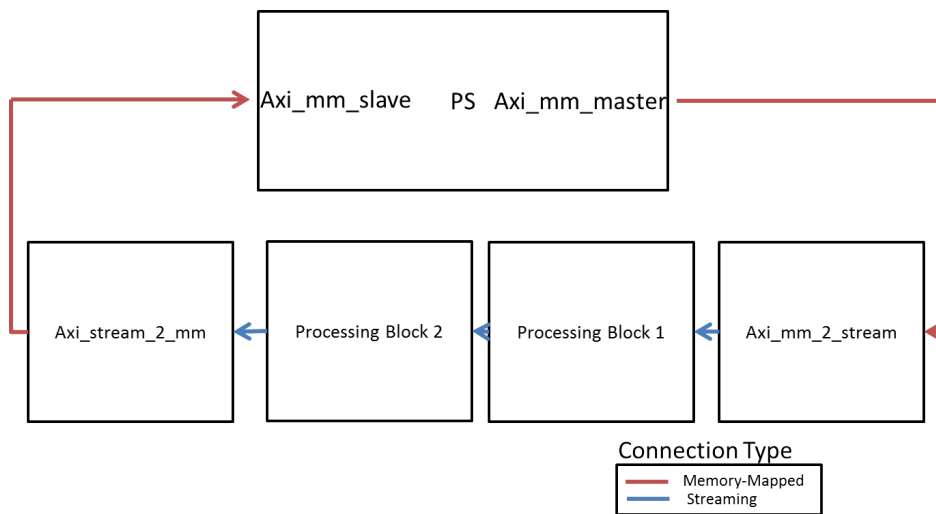


Figure 3.4: Example AXI processing pipeline

### 3.4 Image Filtering

Image filtering is a core operation in the field of image processing. It is the process of subjecting an input image to a linear, sapce invariant (LSI) system resulting in an output image that is the convolution of the input and the LSI system, in this case the filter kernel [1]. The filter kenrl is a matrix of coefficients that when convolved with the input image results in a weighted sum of the current pixel and neighboring pixels, which is then stored in the output image. Figure 3.5 shows an example of how a filter kernel would be applied. Due to the nature of convolution, the kernel must be flipped about the horizontal and vertical axis prior to being applied to the input image and calculating the output.

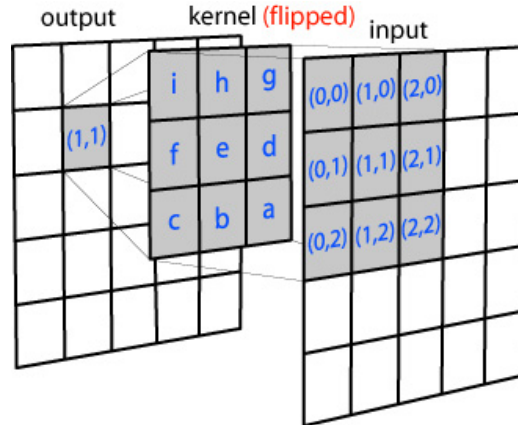


Figure 3.5: Image filtering example

The process of filtering an image can become increasingly resource intensive as the size of the kernel increases. For the example shown in 3.5, a

single output pixel would be calculated using equation (3.1).

$$\begin{aligned}
O(1,1) = & I \times I(0,0) + H \times I(1,0) + G \times I(2,0) + \\
& F \times I(0,1) + E \times I(1,1) + D \times I(2,1) + \\
& C \times I(0,2) + B \times I(1,2) + A \times I(2,2)
\end{aligned} \tag{3.1}$$

In order to reduce the computational load, symmetric filters are typically used of the format shown in 3.6. A filter of this form has an output pixel equation given by (3.2).

$$\begin{aligned}
O(1,1) = & A \times I(1,1) + \\
& B \times (I(0,1) + I(1,0) + I(2,1) + I(1,2)) + \\
& C \times (I(0,0) + I(2,0) + I(0,2) + I(2,2))
\end{aligned} \tag{3.2}$$

A comparison of the number of computations between the different sized filters and their symmetric counterparts, shown in table 3.1, shows the benefit of using symmetric filters.

	0	1	2
0	C	B	C
1	B	A	B
2	C	B	C

Figure 3.6: Symmetric Filter

Filter Type	Additions	Multiplications	Computational Load
3x3 Filter	8	9	100%
9x9 Filter	80	81	100%
3x3 Symmetric Filter	8	3	64.7%
9x9 Symmetric Filter	80	15	59.0%

Table 3.1: Operations required for different filter kernels

### 3.5 Bilinear Interpolation

Bilinear interpolation (BI) is the process by which an unknown value is interpolated based on a 2 dimensional interpolation process using 4 known values. In its application in image processing, BI is used to scale images up and down. Unlike nearest-neighbor interpolation, which relies on only one known value resulting in blocky images, BI has creates smoother images when scaling due to its estimation of intermediate values.

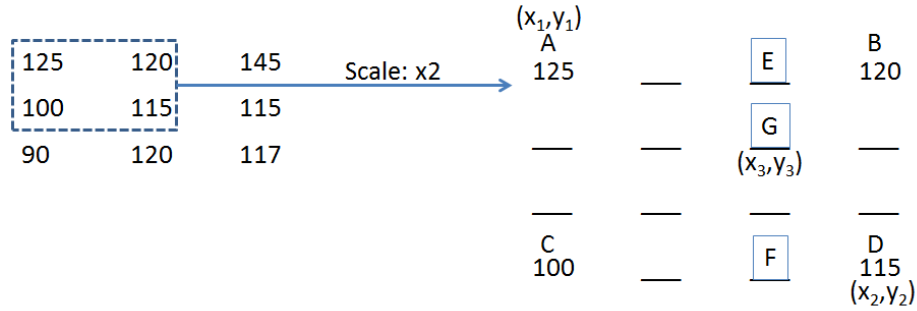


Figure 3.7: Bilinear Interpolation of Grayscale Image

The process of bilinear interpolation begins by expanding the known pixels to their scaled locations. This expansion process generates a window of unknown pixels, bounded by four corners of known pixels, as shown in 3.7. In order to calculate the the unknown pixel values, three equations must solved.



To generate an final equation for the value, the variables shown in 3.7 will be used. The first equation is the horizontal interpolation of the value at  $E$ .

$$E = \frac{x_3 - x_1}{x_2 - x_1}B + \frac{x_2 - x_3}{x_2 - x_1}A \quad (3.3)$$

Next, the horizontal interpolation of the value at  $F$  is calculated.

$$F = \frac{x_3 - x_1}{x_2 - x_1}D + \frac{x_2 - x_3}{x_2 - x_1}C \quad (3.4)$$

Finally, the vertical interpolation of value at  $G$  is calculated.

$$G = \frac{y_3 - y_1}{y_2 - y_1}F + \frac{y_2 - y_3}{y_2 - y_1}E \quad (3.5)$$

$$\alpha = x_3 - x_1, \beta = x_2 - x_3, \gamma = y_3 - y_1, \omega = y_2 - y_3 \quad (3.6)$$

By substituting (3.3) and (3.4) into (3.5) and allowing variables to be re-named as in (3.6), the BI equation for a pixel is determined.

$$G = \frac{1}{(\omega + \gamma)(\beta + \alpha)}[\gamma(\alpha D + \beta C) + \omega(\alpha B + \beta A)] \quad (3.7)$$

By applying (3.7) to every unknown pixel in every window that is generated from the scaling process, an image is enlarged or shrunk with minimal loss of quality. This technique is easily extended to multiple color channel images such RGB by operating on each colorplane individually and ensuring that pixel offsets are correctly calculated.

## 4. Design

### 4.1 Common Design Components

#### 4.1.1 Tagged Image File Format

All images used throughout this research are Tagged Image File Format (TIFF) images. TIFF images consist of an image header that describes the byte order, TIFF version number, and offset to the image file directory[4]. The image file directory stores the offsets to the directory entries and varies in size based on the number of images contained within the TIFF image. The directory entry is then broken down into several sections including a Tag section which consists of a number of tags which give information about the image including the bits per pixel, compression scheme, image length and height, and many more. The directory entry is lastly completed with the offset to the image data. Figure 4.1 shows a more indepth representation of the TIFF structure for multiple image TIFFs.

For the purposes of this research, only uncompressed TIFF images are used in order to eliminate the need for resource intensive decompression and compression of input and output images. To support the reading and writing

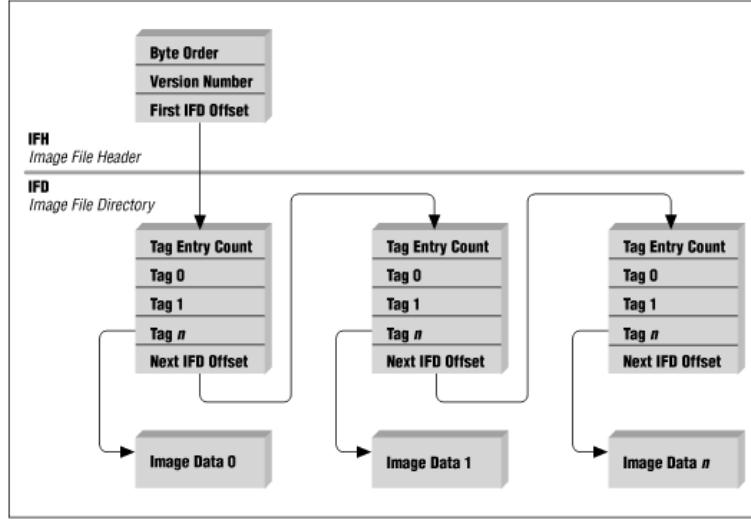


Figure 4.1: TIFF File Structure[3]

of these image files, the LibTiff library is cross-compiled for use on the ZC702's PetaLinux OS.

#### 4.1.2 User Interface

A common user interface (UI) was developed for each algorithm in order to maintain continuity as well as ease development. The UI shown in figure 4.2 is used to fully test, validate, and exercise the C and hardware implementations of the 3x3 Gaussian blur filter. The UIs for the 9x9 Laplacian of Gaussian filter and bilinear interpolation algorithms are identical with the exception of the title shown above the selection options. From the UI, the user is prompted with an option selection; which based on the input performs different operations. Selection option 1 runs the C version of the algorithm only, displays the runtime of a single run of the algorithm, and also outputs the resulting output image to a TIFF image file. Selecting option 2 runs the

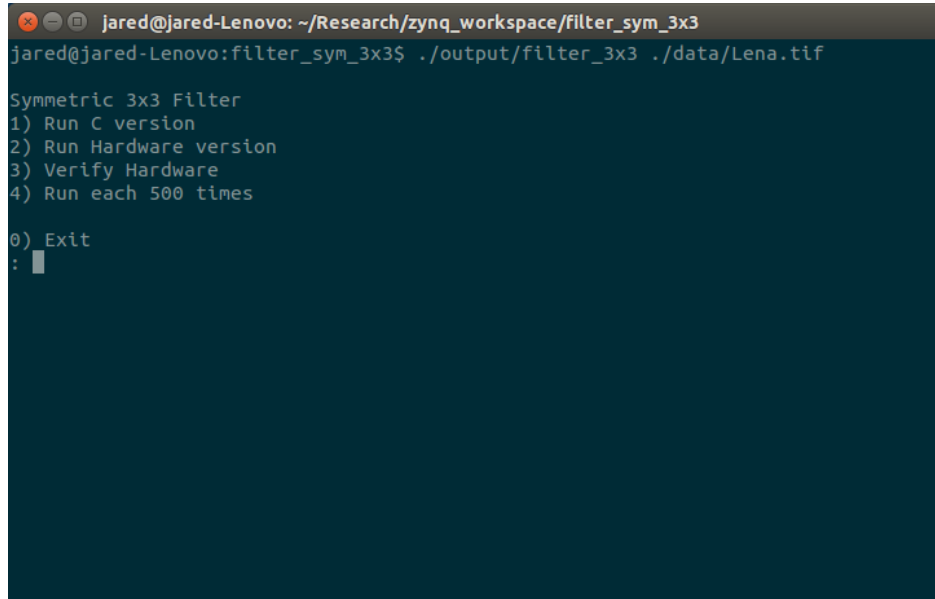
A terminal window with a dark blue background and white text. The window title is 'jared@jared-Lenovo: ~/Research/zynq\_workspace/filter\_sym\_3x3'. The prompt is 'jared@jared-Lenovo:filter\_sym\_3x3\$' followed by the command './output/filter\_3x3 ./data/Lena.tif'. The program output is 'Symmetric 3x3 Filter' followed by a numbered menu: '1) Run C version', '2) Run Hardware version', '3) Verify Hardware', '4) Run each 500 times', and '0) Exit'. A cursor is positioned after the colon following '0) Exit'.

Figure 4.2: User Interface

hardware version of the algorithm, displays the runtime of a single run of the algorithm in hardware, and also outputs the resulting output image to a TIFF image file. Selecting option 3 runs both the C and hardware versions of the algorithm and compares the outputs pixel by pixel. This is used to verify that the hardware implementation yields an identical output as the software algorithm. Depending on the algorithm, border pixels may be ignored due to the use of uninitialized values resulting in inconsistencies not only between the hardware and software implementations, but also between runs in these regions. Selecting option 4 performs the C and hardware implementations 500 times and displays the average runtimes for each.

## 4.2 Processor Based Design

### 4.2.1 Image Filter

For this experiment, two filter kernels are used. The first is a 3x3 Gaussian blur filter and the second is a 9x9 Laplacian of Gaussian (LoG) filter. The framework in the processor based design is the same for both of these kernels as shown in Figure 4.3. The processing occurs in the following steps:

1. Read in the TIFF input image from the file system
2. Allocate the output image in memory
3. Perform the filtering by applying the filter kernel pixel by pixel, iterating over the entire image
4. Convert the output image to the TIFF specification
5. Write the TIFF output image to the file system

Figure 4.3: Processor based filter framework

Since the Linux operating system is used, the Zynq device provides a static file system that is used to store the input and output images. This file system eliminates the need to transmit the processed image back to a host machine for storage.

#### 3x3 Gaussian Blur Filter

The 3x3 Gaussian blur filter used in this experiment is shown in Figure 4.4. The filter acts as smoothing filter, removing some of the higher spatial

1/16 x	1	2	1
	2	4	2
	1	2	1

Figure 4.4: 3x3 Gaussian blur filter kernel

frequencies present in the image and has a blurring effect. The scale factor in front of the filter normalizes the filter so that there is no overall gain in the output image.

### 9x9 Laplacian of Gaussian Filter

0	1	1	2	2	2	1	1	0
1	2	4	5	5	5	4	2	1
1	4	5	3	0	3	5	4	1
2	5	3	-12	-24	-12	3	5	2
2	5	0	-24	-40	-24	0	5	2
2	5	3	-12	-24	-12	3	5	2
1	4	5	3	0	3	5	4	1
1	2	4	5	5	5	4	2	1
0	1	1	2	2	2	1	1	0

Figure 4.5: 9x9 LoG filter kernel

The 9x9 LoG filter used in this experiment is shown in 4.5. The filter is the equivalent of applying a Gaussian smoothing filter followed by a Laplacian high pass filter to an input image. The reason that this is typically employed is to reduce the noise in an image using the smoothing filter and then to extract the edges within the image using the high pass filter. Since the output image is high pass filtered, the mean value (DC component) is removed and the resulting output can be either a positive or negative number that would be unable to be displayed in a standard image format. To resolve this issue, each

output pixel is offset by 128 in order to remove any negative numbers. Unlike the Gaussian blur filter, there is no need for a scale factor when applying the LoG filter because all of the coefficients sum to 0.

#### **4.2.2 Bilinear Interpolation**

### **4.3 Programmable Logic Based Design**

#### **4.3.1 Image Filter**

**3x3 Gaussian Blur Filter**

**9x9 Laplacian of Gaussian Filter**

### **4.4 Homogeneous Design**

#### **4.4.1 Bilinear Interpolation**

## **5. Results**

### **5.1 3x3 Gaussian Filter Results**

### **5.2 9x9 LoG Filter Results**

### **5.3 Bilinear Interpolation Results**



## 6. Conclusion

# Bibliography

- [1] Ramesh Jain, Rangachar Kasturi, and Brian G Schunck. *Machine vision*, volume 5. McGraw-Hill New York, 1995.
- [2] M. Tim Jones. Anatomy of the linux kernel: History and architectural decomposition. *IBM developerWorks*, 2007.
- [3] James D. Murray and William Van Ryper. *Encyclopedia of Graphics File Formats, 2nd Edition*. O'Reilly Media, 1996.
- [4] Richard H. Wiggins, H. Christian Davidson, H. Ric Harnsberger, Jason R Lauman, and Patricia A. Goede. Image file formats: Past, present, and future. *RadioGraphics*, 21:789–798, 2000.
- [5] Xilinx. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC*, 6 2014. v1.3.
- [6] Xilinx. *Zynq-7000 All Programmable SoCs*, 2014.

# Appendices

## **A. Useful Resources**

### **A.1 Programmable Logic**

### **A.2 Operating System**

### **A.3 Vivado**

## B. Verilog Sources

## C. C Source

## D. Reproduction of Work