

Architecture for Real-Time, Low-SWaP Embedded Vision Using FPGAs

A Thesis Presented for the
Master of Science
Degree

The University of Tennessee, Knoxville

Steven Andrew Clukey

December 2016

Copyright © 2016 by Steven Clukey

All rights reserved.

Abstract

In this thesis we designed, prototyped, and constructed a printed circuit board for real-time, low size, weight, and power (SWaP) HDMI video processing and developed a general purpose library of image processing functions for FPGAs.

The printed circuit board is a baseboard for a Xilinx Zynq based system-on-module (SoM). The board provides power, HDMI input, and HDMI output to the SoM and enables low-SWaP, high-resolution, real-time video processing.

The image processing library for FPGAs is designed for high performance and high reusability. These objectives are achieved by utilizing the Chisel hardware construction language to create parameterized modules that construct low-level FPGA implementations of image processing functions. Each module in the library can be used independently or combined with other modules to develop more complex processing functions. In addition, these modules can be used together with other existing firmware resources.

The circuit board and image processing library were then used together to develop, synthesize, and implement a real-time low-light video enhancement pipeline. It was also used to benchmark and estimate the performance of accelerating convolution for use in convolutional neural networks. In these applications our system was able to perform up to 20 times faster than the CPU at the same power consumption.

The final circuit board measures only 1.5x3x0.6 inches, weighs 1.00 ounce, and consumes less than 5 Watts when fully operational. It has been tested with HDMI video stream at 60 frames per second with input resolutions up to 1440x900 pixels and output resolutions up to 1920x1080 pixels. In addition, all of the modules in the library are optimized to be able to operate at no less than 60 frames per second on full high-definition (1920x1080 pixel) video.

Table of Contents

1	Introduction.....	1
1.1	Motivation	1
1.2	Typical Applications	3
1.2.1	Unmanned Aerial Vehicles.....	3
1.2.2	Small, Low Power, Real-time Multi-camera Systems	4
1.2.3	Autonomous vehicles	5
1.3	Synopsis	6
2	Literature Review	8
2.1	FPGA-based Image Processing	8
2.1.1	Edge and Corner Detection	8
2.1.2	Morphological Operations.....	10
2.1.3	Sliding Window Applications	11
2.2	Using FPGAs and CPUs together	11
2.2.1	General Examples.....	12
2.2.2	General Purpose Image Processing Accelerators	13
2.2.3	Dynamic Reconfigurability	14
2.3	Convolutional Neural Networks.....	16
2.3.1	In Image Processing	16
2.3.2	Using FPGAs.....	17
3	Embedded Systems for Image Processing	19
3.1	FPGAs vs. CPUs vs. GPUs	19
3.1.1	Design Architecture.....	21
3.1.2	Performance.....	22
3.1.3	Recommendations	24
3.2	Zynq SoC.....	24
3.3	Embedded Hardware Platforms	25
3.4	Custom Baseboard for the Mars ZX3	27
3.4.1	Custom Baseboard Version 1	29
3.4.2	Custom Baseboard Version 2	32

4	Image Processing Library for FPGAs	40
4.1	FPGA Programming.....	40
4.2	FPGA Languages	42
4.2.1	Traditional HDLs.....	42
4.2.2	High-Level Synthesis	42
4.2.3	OpenCL	43
4.2.4	Chisel.....	44
4.2.5	Recommendations	47
4.3	Chisel Video Processing Library.....	48
4.3.1	Utilities	50
4.3.2	Chisel Image Processing Primitives (ChIPPs)	59
4.3.3	IP Cores	68
5	Building and Benchmarking Image Processing Systems	77
5.1	Creating an IP Core for use in Xilinx Vivado block designs	77
5.1.1	Chisel Outputs	78
5.1.2	Vivado HLS.....	79
5.2	Creating a video processing design in Xilinx Vivado.....	79
5.2.1	HDMI Receiver and Transmitter.....	82
5.2.2	Frame Buffers and Clock Domains	83
5.2.3	Processing Functions	84
5.3	Results for several image processing operations.....	85
5.3.1	Lookup Table.....	86
5.3.2	Color Conversions	88
5.3.3	Histogram Equalization	92
5.3.4	2D Filters	92
6	Application Examples.....	96
6.1	Implementation: Enhancing Low-Light Real-Time Video	96
6.2	Benchmarking: Accelerating Convolutional Neural Networks	100
6.2.1	Overview of Convolutional Networks.....	100
6.2.2	Hardware and Software Architecture	103
6.2.3	Performance Results.....	107
6.2.4	Evaluation.....	115

7	Conclusions.....	116
7.1	Summary	116
7.2	Future Work	118
	List of References.....	121
	Vita	128

List of Tables

Table 2.1: Performance results for the Canny edge detector from [4].	9
Table 2.2: Performance results for the Harris corner detector from [4].	9
Table 2.3: Energy usage for the Canny edge detector from [4].	9
Table 2.4: Energy usage for the Harris corner detector from [4].	9
Table 2.5: Performance results from the top-hat transform implementation in [8].	10
Table 4.1: Chisel is able to maintain a low-level quality of performance while still providing high-level abstractions and hardware construction interfaces. Table recreated from [49].	45
Table 5.1: Utilization and maximum frequency for the Lookup Table module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.	87
Table 5.2: Utilization and maximum frequency for Xilinx's Gamma Transformation IP Core configured to be functionally equivalent to our Chisel-based LookupTable module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.	88
Table 5.3: Utilization and maximum frequency for the RGB to Grayscale conversion module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.	89
Table 5.4: Utilization and maximum frequency for the RGB to HSV conversion module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.	90
Table 5.5: Utilization and maximum frequency for the HSV to RGB conversion module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.	90

Table 5.6: Utilization and maximum frequency for the RGB to YCbCr conversion module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.....	91
Table 5.7: Utilization and maximum frequency for the YCbCr to RGB conversion module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.....	91
Table 5.8: Utilization and maximum frequency for the Histogram Equalization module written in Chisel. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.	93
Table 5.9: Utilization and maximum frequency of the Histogram Equalization module available in Vivado HLS. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.	93
Table 5.10: Utilization and maximum frequency for the passthrough function using the window module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.	94
Table 5.11: Utilization and maximum frequency for the median filter module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.....	94
Table 5.12: Utilization and maximum frequency for the single channel convolution module written in Chisel. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.	95

List of Figures

Figure 1.1: FPGAs can be used to perform a large amount of processing onboard the UAV platform and limit the bandwidth required between the ground station and the UAV.	4
Figure 1.2: Multiple sensors can be grouped and aligned to capture very large images at very high speed. Each small piece can be processing individually, or together with other sensors.....	5
Figure 1.3: Pipeline of contributions. The printed circuit board for low SWaP real-time video processing is developed, along with a library of FPGA implementations of image processing functions for quickly developing high-performance designs.....	7
Figure 2.1: Flow diagram for the road sign recognition algorithm in [14] (figure reproduced therefrom).	12
Figure 3.1: FPGAs are at the extreme end of the performance vs quantity spectrum: they have an extremely large number of very low performance “cores” as compared to CPUs or GPUs.	21
Figure 3.2: Enclustra Mars ZX3 System-on-Module. This module includes the Zynq xc7z020clg484-2, 1GB DDR3 SDRAM, a Gigabit Ethernet PHY, 512MB NAND Flash and power regulators in a 200 pin SO-DIMM form factor.	26
Figure 3.3: Some of the available development boards that use the Xilinx Zynq SoC. The Enclustra Mars ZX3 (far left) was chosen because it is the smallest one available. ..	27
Figure 3.4: Custom baseboard version 1, a breakout board for the Mars ZX3 and 2 HDMI connectors. This board was simply used for testing and experimentation with the pinout of the ZX3. This testing was used to develop the final pinout for the second board.	31
Figure 3.5: Version 1 baseboard fully connected with HDMI input and output. The power and ground pins are mostly at the bottom in the wider ribbons. The top left helper	

board is a UART to USB converter. The HDMI works, but only at low resolutions due to poor signal quality due to the wiring.	33
Figure 3.6: Version 2 baseboard. This board contains the UART to USB converter onboard, as well as logic level translators for the HDMI I2C interface.	34
Figure 3.7: Version 2 baseboard fully connected with HDMI input and HDMI output...	35
Figure 3.8: HDMI input circuit on the version 2 baseboard.	37
Figure 3.9: HDMI output circuit on the version 2 baseboard.	37
Figure 3.10: UART to USB circuit on the version 2 board.	38
Figure 4.1: Chisel outputs C++ for use in simulations and Verilog for implementation in either FPGAs or ASICs.....	45
Figure 4.2: The process of writing FPGA firmware in Chisel. The thicker arrows indicate a quicker path. The Chisel simulator provides a very wide, easy path to quickly develop a very good module before proceeding to the more time consuming parts of the development process.	46
Figure 4.3: The firmware library uses three separate projects: IP Cores, ChIPPs, and Utilities. The Utilities are used in both ChIPPs and IP Cores. The ChIPP modules are image processing primitives that are combined to build IP Cores to perform a specific processing operation.	50
Figure 4.4: The adder tree created by the Adder module adding 9 addends, 2 at a time. The latency of the input to the answer is 4 cycles.....	51
Figure 4.5: Maximum operating frequency of the Adder module with varying number of combinational additions.	52
Figure 4.6: Flowchart of the Division module's implementation of binary long division.	54
Figure 4.7: AXIVideoStream input showing buffering and backup registers that are used when the stream is restarted.....	56

Figure 4.8: Chisel's built-in ShiftRegister module has the enable signal only for the input data.....	58
Figure 4.9: This library's ShiftRegisterGood module uses the enable signal for all the stages.....	58
Figure 4.10: Example results of using the scalar math modules. They accept arbitrary constant and apply their operation to the image. The addition and subtraction are both accomplished by the AddS module because it accepts signed integers. The multiplication and division outputs were created by the MulS and DivS modules, respectively.	61
Figure 4.11: Examples of using the lookup table module.	65
Figure 4.12: Data path for the Window module. The pixels are streamed in and registered. Line buffers are used to save each line of pixels until there are enough to fill the entire window. The pixels are streamed in sequentially, with the first pixel corresponding to the top left pixel, therefore the last pixel in memory is the top left pixel in the window.	67
Figure 4.13: Flowchart of the operation of the Histogram Equalization module.	69
Figure 4.14: Example of the histogram equalization method of contrast stretching. The Histogram Equalization module calculates the histogram of the input image, then transforms it such that the colors are spread over the entire range of intensities. Images from [56] and [57].....	69
Figure 4.15: Color Histogram Equalization using two methods. On the left: Convert to HSV and equalize the V channel. On the right: Equalize the red, green, and blue channels individually.	72
Figure 4.16: Convolution process. The kernel is placed over the image at each pixel, then each kernel value is multiplied with the image pixel value under it. The result is summed and divided by a normalization factor and the result is placed in the output image.....	73

Figure 4.17: Convolution examples. The top image is the input, and the output of the convolution module is shown for several different 3x3 kernels.	75
Figure 4.18: Implementation of a 3x3 convolution module using blocks from the ChIPPs and Utilities projects.	76
Figure 4.19: Example inputs and output of the average and median filters modules.	76
Figure 5.1: Architecture of a general real-time video processing system using the Zynq FPGA and CPU.	77
Figure 5.2: Vivado HLS's "Export RTL" dialog to create Vivado IP cores from an HLS project.	80
Figure 5.3: Empty image processing function showing the directives needed to create an AXI-4 Stream interface input and output, with two variables added to an AXI-Lite interface.	80
Figure 5.4: Example image processing pipeline. The frame buffers are used to enable processing are used to enable the use of processing cores that block the signal path and for clock domain crossing between the input and output signals.	81
Figure 5.5: Example of a full block diagram for an image processing function. The highlighting has been added for clarity.	81
Figure 5.6: Example of a full block diagram for an image processing function showing only the interface connections. Here the flow of the video can be seen at the bottom. From the HDMI receiver on the bottom left, through the frame buffers and processing, to the video output on the bottom right.	81
Figure 5.7: HDMI clock and data timings. The HDMI data is transmitted at 10 times the HDMI clock. To deserialize this, the Zynq uses a clock 5 times the HDMI clock and double data rate (DDR) sampling.	83
Figure 5.8: Image processing operation benchmarking system. The Asynchronous FIFO blocks allow the DUT to be clocked at a higher speed than the rest of the design. This	

allows the DUT to run as fast as possible without being limited by the performance of the other blocks.	85
Figure 5.9: Device utilization of the LookupTable performance testing project on the Zynq xc7z020clg484-2 part.....	87
Figure 6.1: Architecture of the low-light video enhancement system implemented on the FPGA.	98
Figure 6.2: Architecture of the low-light video enhancement system implemented on the CPU.....	98
Figure 6.3: Example inputs and outputs from the low-light enhancement system. The inputs are on the left and the outputs are on the right. From top to bottom they show the least input light to the most.	99
Figure 6.4: Example architecture of a Convolutional Neural Network. Recreated from [60]. The input is a single channel image, it first goes through a series of convolution and pooling layers, then the final classification is performed by a two-layer fully connected neural network.	101
Figure 6.5: Architecture of the hybrid CPU/FPGA convolutional neural network. Each layer reads input from the RAM and outputs to another location in the RAM. The AXI-Lite interface block is used to communicate with the FPGA-based layers for synchronization and configuration.....	103
Figure 6.6: Volume convolution can be performed a single channel at a time by summing the results over the various input and kernel combinations.....	105
Figure 6.7: Operation of LeNet-5. The input is the image of the number 2 on the left. The ten outputs on the right correspond to the ten numbers. The output with the largest activation is number predicted to be in the image.	109
Figure 6.8: Performance of the forward pass of the LeNet network used for handwritten character recognition on the MNIST database implemented in Caffe and executed on the Zynq's ARM CPUs and approximated using hardened accelerators on the FPGA.	110

Figure 6.9: Overview of the facial landmark localization process. First the image is preprocessed, this includes locating the face and cropping the image, then normalizing the face based on trained average and standard face images. Then the image is sent to the convolutional neural network to locate the landmark. Finally the landmarks are backtracked to the original image.	112
Figure 6.10: Layers on the facial landmark localization network. There are 3 sets of convolution, activation, and pooling. These are followed by a final convolution and activation, then a 2 layer dense network.	113
Figure 6.11: Performance of the Vanilla CNN network used for facial landmark detection and localization implemented in Caffe and executed on the Zynq's ARM CPUs and approximated using hardened accelerators on the FPGA.	114

1 Introduction

This research develops an architecture for implementing real-time image processing systems on size, weight, and power (SWaP) constrained hardware. We accomplished this by constructing a real-time video processing hardware system that includes the Xilinx Zynq System-on-Chip (SoC) and by developing a library of image processing functions for field-programmable gate arrays (FPGAs) that accelerates the development of high-performance video processing firmware.

1.1 Motivation

The increasing complexity of modern computer vision and image processing algorithms coupled with the continued miniaturization of technology creates a demand for faster and more efficient computing resources. Modern intelligence systems often utilize large amounts of either 2D or 3D data and require large amounts of time to process and extract meaningful information. The use of distributed and heterogeneous processing systems, including multiple multicore CPUs, multiple GPUs, and one or more FPGAs or ASICs, is essential to improve the performance of these systems. Currently, many systems, and especially embedded systems, only utilize CPUs, with some high end systems adding GPUs. This thesis analyzes the impact of including FPGAs in real-time acquisition and processing systems that utilize large datasets, specifically, high-resolution, real-time video.

FPGAs are very well suited for high performance, real-time applications. First, they are programmable devices, so fabrication and maintenance costs can be kept low just like with CPUs and GPUs. Second, they are massively parallel devices that give access to levels of optimizations not possible with other devices. Third, they support configuring the input and output interfaces at the electrical level, which removes the need for dedicated hardware support for a wide variety of input and output interfaces. Fourth, the emergence of cheap FPGAs that include hardened, multicore CPUs in the same package with the FPGA creates an avenue for developing very high performance heterogeneous processing systems. This architecture allows the operations best suited for the CPU to be kept there, and the operations that can benefit from FPGA acceleration to be easily and quickly offloaded to the FPGA. This was possible in the past by using multiple discrete components, but

integrating them into a single component reduces the hardware complexity, physical size, power consumption, and cost of the final system.

This work also examines how to take advantage of an FPGA in conjunction with a CPU to increase the performance of real-time image processing tasks. This includes deciding how much of a given task to assign to the CPU, how much to assign to the FPGA, and how to handle the communication in to and out of the system so that the processing tasks can be completed in a timely manner. For real-time video, this means the processing must complete fast enough for a system to be considered “real-time”. In this thesis, “real-time” video is minimally defined as having at least 20 frames per second and no more than 1 second of latency. The aim of this work, however, is to perform at a minimum of 60 frames per second with the latency as close to 0 as possible. Such a high frame rate and the increasing use of large resolution imaging (full HD up to 4k and beyond) gives these systems a very large bandwidth requirement. To meet this requirement, an architecture was developed that will allow both serial and parallel processing through combinations of processing elements either in the CPU, the FPGA, or both simultaneously. If a process is too complex to complete on the CPU quickly, then it is broken into smaller pieces and sent to the FPGA to reduce the processing time and latency.

At the hardware level, the objective is to process data from a visible camera, infrared camera, LIDAR, or a combination of these (describing 2D or 3D information) through a single input, single output hardware platform. Additionally, if the input format is made compatible with the output format then the design can be cascaded through multiple stages to increase the processing capabilities. All the processing will be done on the fly with the synchronization being handled automatically by the input and output format. For a particular input image, typical operations that could be performed include simple enhancements like brightening or darkening the image, extracting outlines or silhouettes for measuring object sizes and positions, or more complex operations like object recognition or tracking.

In order to complete these operations quickly, some intelligent planning is required to determine how to use resources available in the FPGA and CPU. In general, the approach is to implement as much as possible in the CPU and offload operations to the FPGA when the CPU is not able to achieve the required framerate and latency.

1.2 Typical Applications

Nearly any application of real-time image processing can be considered a possible application of this work. Mainly, however, this thesis is focused on applications that will benefit not just from the FPGAs computational performance, but also from their massively parallel architecture, low power consumption, and high I/O bandwidth. In general, if there exists a sufficient budget of money, physical space, and power, then a large, expensive, high-performance CPU and GPU system can perform just as well as an FPGA-based system. However, if a system needs to scale down to very small embedded systems, or up to very large, warehouse or supercomputer sized systems, then computational performance is not the only (and sometimes not the most important) consideration.

1.2.1 *Unmanned Aerial Vehicles*

Consider a small unmanned aerial vehicle, perhaps of the type gaining widespread popularity for their low cost and unique capabilities. They can perform many useful imaging tasks including aerial photography, terrain mapping, inspection of tall or otherwise inaccessible structures, and possibly even package delivery. Very often, high resolution cameras are added to these devices that could be used for a multitude of machine vision and image processing tasks. Because they are powered with batteries, small UAVs have very strict size, weight, and power (SWaP) restrictions, so taking full advantage of high-resolution camera without significantly reducing the flight time is a challenge. There are two possible options for processing image data from the UAV. The first option is to stream all the data to a ground station and perform processing there. This requires very high-powered transmitters on the device to be able to handle the massive bandwidth of high-resolution and high-framerate data, but it allows the processing to be performed on large computers with no power restrictions. The second option is to process the data on the platform itself and to send only the results of processing and extracted information down to the ground station. Processing the data onboard decreases the response time of the platform to input images by reducing the latency between input and processed output, but it also requires a high-performance processing system onboard that can consume a large amount of power. These two options present a tradeoff between the antenna's power consumption and the processing element's power consumption. FPGAs can be used to

increase the onboard computational performance and decrease the power consumption of the processing elements to make the second option a more advantageous choice.

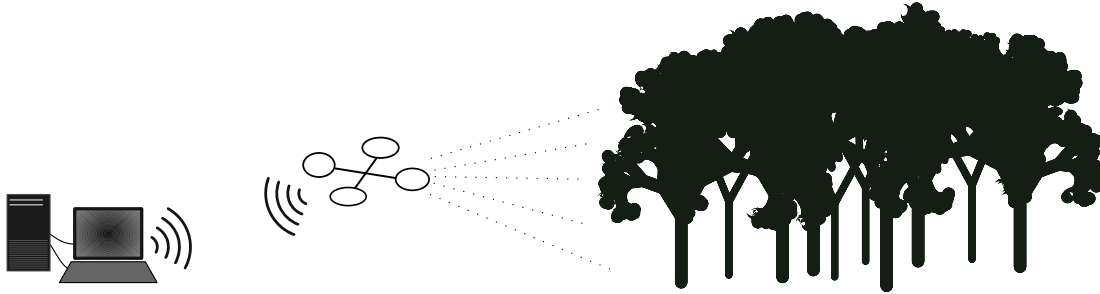


Figure 1.1: FPGAs can be used to perform a large amount of processing onboard the UAV platform and limit the bandwidth required between the ground station and the UAV.

1.2.2 Small, Low Power, Real-time Multi-camera Systems

A promising area of research in imaging is the use of arrays of small, relatively inexpensive cameras. Used together, these arrays of cameras can create very powerful and flexible imaging systems. Some systems that are built from arrays of sensors include light field cameras, gigapixel cameras, and single exposure multispectral imagers.

Light field cameras like the Lytro ILLUM use large arrays of micro-lenses to capture both the color and direction of light. Using the direction information, a “light field” can be generated and used to calculate depth information and focus the picture on any point in the scene. The Lytro ILLUM uses a Snapdragon processor that includes a CPU, GPU, DSP, and “other specialized cores” to process the light field images on the camera in real-time [1]. The nature of these specialized cores is not specified, but they are present because, according to the Lytro CTO, they “needed more than a processor with a CPU and GPU”. The nature of the specialized cores is not specified, but FPGAs have been shown to be capable of implementing very low power (5W), real-time (25 fps) light field cameras [2].

Cameras that achieve gigapixel (1 billion pixel) resolution can be made by using an array of lower resolution, commercially available sensors. If the sensors are placed in a known arrangement then their outputs can be stitched together to create a much larger image.

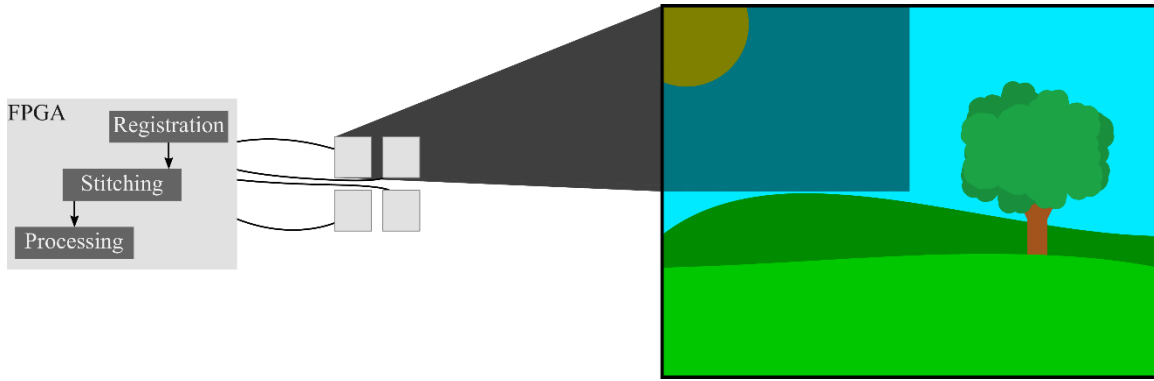


Figure 1.2: Multiple sensors can be grouped and aligned to capture very large images at very high speed. Each small piece can be processing individually, or together with other sensors.

These camera arrays require extremely high bandwidth to handle reading and stitching together the multitudinous stream of images coming from the sensors. This is an extremely large task, but it can be easily broken down into a tree of smaller tasks. The images can be grouped into manageable collections, each of which is stitches. The results are then grouped into manageable sized collections and stitched together. This process can continue until there is only a single image remaining. Because of the high bandwidth requirements and the highly constrained and highly parallel nature of the process, FPGAs are a natural fit for this application. An architecture like the one proposed in this thesis would allow a small, low-power, easily maintainable system to be developed.

Another use for camera arrays is to create low-cost custom multispectral imaging systems. Often, multispectral images must be built up by using a tunable filter on a single sensor and sequentially acquiring images at various wavelengths. In contrast, an array of cameras each with their own spectral response and exposure settings can be used to capture the entire multispectral image in a single shot. If the goal is to make the system small and low-power, then the high data bandwidth and synchronization requirements for such a system preclude the use of traditional CPUs and GPUs and make FPGAs a very attractive choice.

1.2.3 Autonomous vehicles

Fully autonomous vehicles require very sophisticated arrays of real-time processing systems. To work properly and ensure the safety of the passengers and pedestrians many

factors must be tracked, including: other vehicles, debris, lane markers, pedestrians, road signs, weather conditions, construction zones, traffic lights, and emergency vehicles, among others. Because of the completely unconstrained environment, achieving high accuracy in monitoring these factors requires extremely sophisticated sensors and processing. Imaging sensors are a vital part of the systems, and real-time performance is an absolute requirement. For accuracy these video streams are both high-resolution and high-speed, adding a high bandwidth requirement to an already difficult real-time video processing problem. FPGAs can help meet the demands for large bandwidth and processing speeds and also offer exceptional power performance. In vehicles, the power draw of the vehicle translates directly to vehicle efficiency and range. With the shift toward electric vehicles, that already have a limited range, reducing the power consumption of the vehicle's own systems is an important task. High-performance CPUs and GPUs are capable of performing very high speed processing, but they often require a lot of power to do so. Fully custom application specific integrated circuits (ASICs) provide the optimal solution to power and speed, but they are not reprogrammable. Given the high pace of innovation and improvement in the development of these systems, upgrades are required often so ASIC-based systems would incur huge costs. FPGAs offer a very good tradeoff between cost, upgradability, power consumption and performance in this application.

1.3 Synopsis

In Chapter 3 a small, low-power video processing baseboard for a Xilinx Zynq-based system-on-module (SoM) is designed, prototyped, and constructed. In Chapter 4 a high-performance, highly reusable, vendor-independent library of image processing functions for FPGAs is developed. Chapter 5 benchmarks the library and describes its use together with a CPU and other FPGA functions. Chapter 6 shows an example of using the hardware and FPGA library together to implement an image processing operation and analyzes its use for a much more general set of operations. Chapter 7 presents conclusions and possibilities for future work.

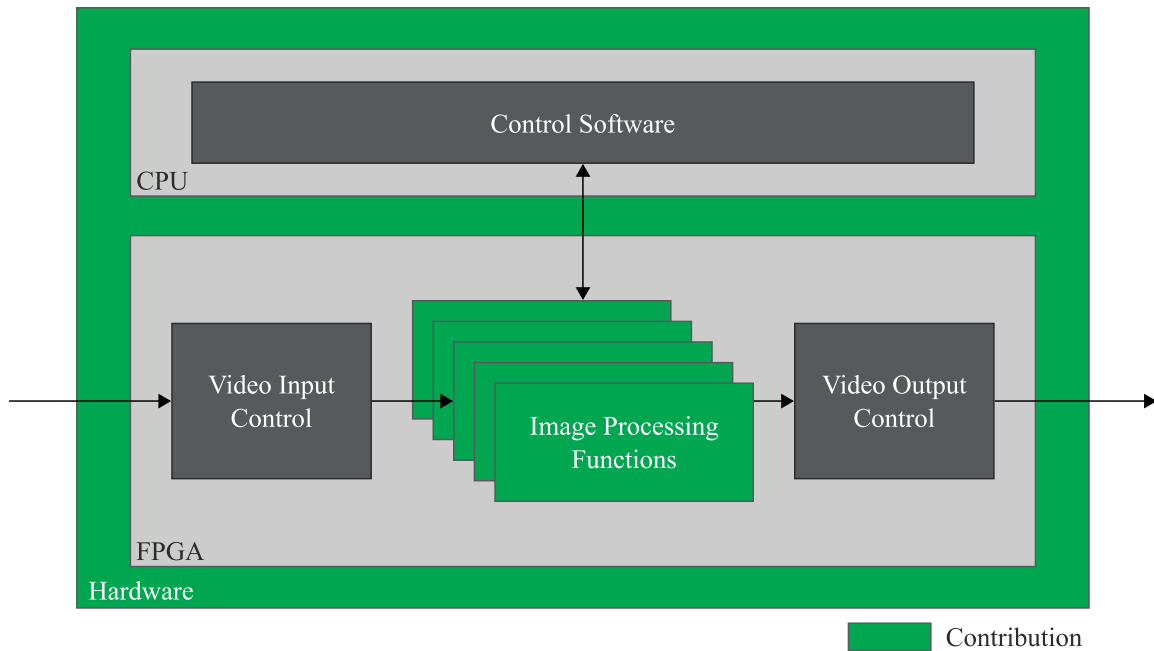


Figure 1.3: Pipeline of contributions. The printed circuit board for low SWaP real-time video processing is developed, along with a library of FPGA implementations of image processing functions for quickly developing high-performance designs.

2 Literature Review

2.1 FPGA-based Image Processing

FPGAs are very well suited for image processing because of their ability to exploit parallelism. They are able to achieve real-time performance in many applications that serial processors simply cannot without sacrificing resolution [3]. Often, image processing algorithms on FPGAs are implemented as one-off designs built to accelerate a particular task, however, there have been several attempts to develop more general purpose systems for image processing.

2.1.1 *Edge and Corner Detection*

Two very popular operations to perform on FPGAs are edge detection and corner detection. There have been many implementations of edge and corner detection on FPGAs, but very few offer comparisons with other architectures. One of the few that does this comparison is by Possa, et al. in [4]. They compared implementations of Canny edge detection and Harris corner detection on a CPU, GPU, and FPGA. The results of the work is shown in Table 2.1 through Table 2.4. This work shows an order of magnitude increase in the framerate from the CPU to the FPGA, and three orders of magnitude reduction in energy consumption for the FPGA below the CPU or GPU.

The Harris corner detection algorithm was also implemented on an FPGA in [5] achieving 144 frames per second on a 640x480 image, and in [6] achieving 640x480 at approximately 677 frames per second, which is comparable to the results in [4].

In [7], the authors report a speedup of 544x for an FPGA implementation of the Canny edge detection algorithm, and also a speedup of 9.72x and 1.57x for two image registration algorithms. This is an exemplar study of how FPGAs can dramatically increase performance, but only in certain situations. In this case, the authors attribute the low speedup of the registration algorithms, as compared to the Canny edge detection, to the ratio of computational complexity to memory usage of the algorithms. FPGAs are very powerful, but they have a limited number of built-in memory resources. If more memory is required than the FPGA can store then the memory must be stored outside the FPGA in

Table 2.1: Performance results for the Canny edge detector from [4].

Resolution (W x H)	CPU (ms)	GPU (ms)	FPGA (ms)
512x512	30	2.11	1.1
1024x1024	101	6.08	4.37
1476x1680	267	13.9	10.31
3936x3936	1497	59.94	64.16

Table 2.2: Performance results for the Harris corner detector from [4].

Resolution (W x H)	CPU (ms)	GPU (ms)	FPGA (ms)
512x512	20	2.32	1.15
1024x1024	60	4.49	4.56
1476x1680	171	13.16	10.75
3936x3936	1402	64.41	66.93

Table 2.3: Energy usage for the Canny edge detector from [4].

Image resolution (W x H)	CPU		GPU		FPGA	
	(W)	(mJ)	(W)	(mJ)	(W)	(mJ)
512x512	141	4200	231	500	1.5	1.6
1024x1024	147	14800	244	1500	1.5	6.4
1476x1680	149	39800	248	3400	1.5	15
3936x3936	153	229000	251	15000	1.5	93.6

Table 2.4: Energy usage for the Harris corner detector from [4].

Image resolution (W x H)	CPU		GPU		FPGA	
	(W)	(mJ)	(W)	(mJ)	(W)	(mJ)
512x512	141	2800	231	500	1.5	1.7
1024x1024	147	8800	240	1100	1.5	6.7
1476x1680	147	25100	242	3200	1.5	15.8
3936x3936	15	213100	249	16000	1.5	98.2

slower memory resources. Accessing this slow memory can often be a bottleneck in memory intensive processing operations. Therefore, optimizing the amount of memory requires and the memory access patterns is very important in FPGA designs.

2.1.2 Morphological Operations

In [8], the top-hat transform was implemented on an FPGA, CPU, and GPU to compare the performance and energy efficiency. The top-hat transformation, as they define it, is the difference of the input image and its opening, where the opening is defined as the dilation of the erosion of the input image. This process, therefore, requires three operations, a dilation, erosion, and difference. They implemented the operation using 5 different methods: using OpenCV on a CPU, using OpenCV on a GPU, hand coded on a GPU, using Vivado HLS's video processing library on an FPGA, and hand coded on an FPGA. Their results are shown in Table 2.5.

Table 2.5: Performance results from the top-hat transform implementation in [8].

Platform	Style	Frames per Second			mJ per Frame		
		3x3	7x7	15x15	3x3	7x7	15x15
CPU:							
Intel Atom	OpenCV	927	805	570	21.5	24.7	34.9
Intel Core i7	OpenCV	3826	2635	1940	17.4	25.2	34.3
GPU:							
Tesla K20X	OpenCV	454	156	55	-	-	-
Tegra K1	hand-coded	697	532	357	15.5	20.3	30.3
FPGA:							
Zynq 7020	Vivado HLS	1901	1317	171	2.3	3.7	19.2
Zynq 7020	hand-coded	1901	1901	1655	2.1	2.5	3.5

These results show that while the Intel Core i7 processor performs at approximately double the frame rate of the FPGA at a maximum, it also uses 8-10 times the power of the FPGA. If the choices are limited to those usable in low-power embedded systems (i.e. remove the Intel Core i7 and the Tesla K20X), then the FPGA has the highest performance both in terms of frame rate and energy consumption across the board.

2.1.3 Sliding Window Applications

In [9], the authors compare the performance and energy consumption of FPGAs, GPUs and CPUs in three different sliding window applications, convolution with a kernel, sum of absolute differences, and correntropy on high-definition video (1920x1080). In all cases the FPGA outperformed both the CPU and GPU for large kernels. For the correntropy function, they found that “the FPGA was the only device capable of real-time usage.” They also found that “the FPGA used orders of magnitude less energy than other devices in many situations, providing the only realistic embedded system implementation for high-definition video.”

Another study implemented the DMPDS algorithm for motion estimation on Stratix 4 and Virtex 5 FPGAs [10]. This algorithm is a high-performance motion estimation algorithm designed for use in video coding standards like H.264 where the motion estimation step accounts for roughly 80% of the computation burden of video encoding [11]. Their implementation on the Virtex 5 was able to achieve 53.3 fps for Quad Full High Definition (QFHD, 3840x2160 pixels) video and over 200 fps for Full HD (1920x1080 pixels).

Some of the previous algorithms are used together to create an omnidirectional motion detection and object localization system in [12]. This work used a catadioptric system for capturing an omnidirectional polar image 800x480 pixels in resolution. They then used the FPGA to locate motion and calculate the distance to the moving objects. The FPGA fabric contained modules for RGB to grayscale conversion, mean filtering, background subtraction, segmentation, erosion, and center of mass calculation, as well as including a Nios II soft-core processor. The entire system was able to run at real-time speeds of 26.6 fps, outperforming their previous CPU based implementation by nearly 14x.

2.2 Using FPGAs and CPUs together

An alternative method to using FPGAs is to use them as accelerators in tandem with a CPU. Using an FPGA together with a CPU is a useful way to decrease the time cost of developing FPGA firmware while still getting some of the benefits. In general, new algorithms and methods are developed first for CPUs, so it is relatively easy to find an existing CPU implementation of a method. FPGA implementations of image processing algorithms, on the other hand, are very difficult and time consuming to develop. To

alleviate the challenge of building an entire FPGA image processing pipeline, a CPU and FPGA can be used together. This allows the system to be built and deployed quickly, while time is spent to develop FPGA modules only for the specific pieces of the algorithm that are most in need of acceleration.

2.2.1 General Examples

A good example of a CPU-FPGA system is the Visual-inertial sensor built by Nikolic, et al. [13] Their objective was to make a low power, real-time pose estimation and mapping system for flying Micro Aerial Vehicles (MAVs). Their original system implemented the entire process on a CPU, but it was not able to achieve real-time performance (defined as 20 FPS, in this case). They discovered that one particular step of their implementation was taking a majority of the processing time: detecting points of interest. To achieve real-time performance, the interest point detector (either a Harris corner detector or a FAST corner detector) was moved to the FPGA fabric. With this optimization alone the system was able to reach 20 FPS, so the remaining steps were left on the CPU.

An additional example is the road sign detection system by Russell and Fischhaber [14]. This system uses the Xilinx Zynq to detect road signs in full HD (1920x1080) images and performs at roughly .2 frames per second. Figure 2.1 shows the processing pipeline for their work.

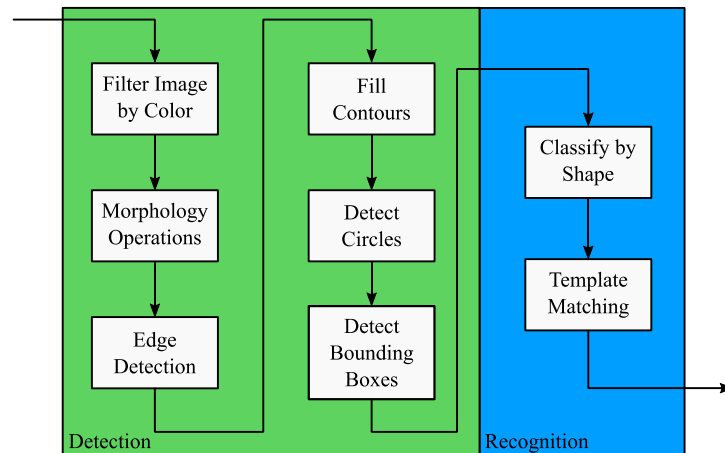


Figure 2.1: Flow diagram for the road sign recognition algorithm in [14] (figure reproduced therefrom).

To improve the performance of their system, the color filter block is implemented in the FPGA fabric. Unfortunately, with this alone they indicate that the FPGA fabric is 80% full. If they use a larger FPGA, though, they could implement further processing on the FPGA and accelerate their work even more. The morphological operations and edge detection account for 56% of the total processing time, and, as has already been shown, these operations can both be performed very well on FPGAs.

2.2.2 General Purpose Image Processing Accelerators

There have been several attempts at creating general purpose image processing systems on FPGAs. These generally follow one of two approaches, either they provide a library of image processing functions that can be synthesized into the FPGA, or they build a set of basic functional units that are synthesized into the hardware and can be configured to perform a given image processing task. Examples of image processing libraries would be Vivado HLS's Video library [15] and the Chisel-based library developed in this work, while an example of later is IPPro [16].

Xilinx maintains a high-level synthesis (HLS) product called Vivado HLS. They also provide a library of HLS functions that implement a slightly modified subset of OpenCV. Called the "HLS Video Library", this library includes functions for morphological operations, Harris edge detection, Sobel operators, and several others as well as data structures for line buffers and windows that can be used to create custom functions. The stated objective of this library is that it be used as a companion to OpenCV on the CPU. An algorithm written using the OpenCV library on the CPU can be updated to offload certain function calls from the CPU to the FPGA fabric with minimal changes and minimal effort. At this very specific task the library excels, but adding additional functions is a challenge. This library and high-level synthesis in general is discussed later in this thesis.

There has also been another library, written to be compatible with Vivado HLS and other C-based high-level synthesis systems, that is dedicated to image processing [17]. As described in the literature, however, this library seems to be in a very early stage of development. At this point it seems to provide nothing except perhaps an optimized version of kernel convolution.

The alternative approach is to develop a small processor core that can be instantiated many times within the FPGA and configured to work together to perform an image processing task. This is not unlike a highly customized GPU implementation, in that, rather than using the FPGAs hundreds of thousands of tiny logic elements separately, they cluster some of them together to have dozens or possibly hundreds of special purpose cores that can perform specific tasks very well. One such system is the Image Processing Processor (IPPro) core [16]. This is a small, signed fixed-point scalar processing core with an ISA optimized for efficiently performing image processing operations. These cores can be grouped together to form a single instruction, multiple data (SIMD) architecture just like a GPU. The size and shape of the so-called SIMD-IPPro can be optimized for the operation being performed and the number of FPGA resources available.

Using the SIMD-IPPro configuration, they have documented implementations of color filtering and morphology used to achieve road sign detection performance of 2.38 frames per second at 600x400 pixel resolution, and two tasks in the histogram of oriented gradients (HOG) algorithm to boost the performance to 209 frames per second for 1920x1080 images [18]. Currently, the programming and allocation of these cores is done manually; there is no working compiler to optimize the architecture or distribute the work. Therefore it is debatably not any easier to work with than traditional HDL. Additionally, even though this system does provide high performance, it also uses very many resources. As a comparison in [18] they hand-coded the same algorithms implemented in IPPro and achieved 79 frames per second on 1920x1080 pixel images. The IPPro system performed at 2.6 times the frame rate of the hand-coded implementation. In term of resource utilization, however, the IPPro implementation not only used 5.7 times the number of lookup tables (LUTs) as the hand-coded implementation, it also used 90 digital signal processing blocks (DSPs) and 24 block random access memories (BRAMs), compared to none in the hand coded implementation. Perhaps at a comparable frame rate the resource utilization would come down, but without an effective compiler the benefits are not apparent.

2.2.3 Dynamic Reconfigurability

Another emerging area of interest in FPGAs is dynamic reconfigurability. This is the ability of the FPGA to reconfigure its operation online and without stopping other activities. This opens up possibilities of not only parallel operation in space, but interleaving operations in

time. Intelligent systems could swap in and out hardware accelerators based on the situation at hand, or even within the same operation multiple accelerators could be used sequentially.

In [19], the authors implement an IPsec system using partial reconfiguration. The algorithms for Advanced Encryption Standard (AES), Secure Hash Algorithm (SHA256), and modular exponentiation (MODEXP) are implemented in hardware, but only one of them is available at a time. Using IPsec requires these several different algorithms, but generally only one at a time. The parameters of the connection are negotiated first, requiring the MODEXP module, then the session is started using either the Authentication Header (AH) protocol requiring the SHA256 module, or the Encapsulating Security Payload (ESP) protocol requiring the AES module. Implementing these three modules using partial reconfiguration showed a FPGA resource savings of more than 34% over implementing them all in parallel.

Evolvable hardware (EHW) takes the use of dynamic reconfiguration to the extreme. In EHW an evolutionary algorithm is used to develop hardware circuits on the fly. As the system is running it constantly modifies its own behavior and implementation to improve performance. This evolution could be used to adapt to changing environmental conditions, to update for new program objectives, or even to automatically recover from hardware faults. A test of this idea was performed by Dobai and Sekanina [20], wherein they evolved an image 3x3 filter. Their implementation is in contrast to the traditional method of implementing EHW. Before partial reconfiguration was fast and well supported, EHW circuits implemented all possible options inside the FPGA and used a series of multiplexers to switch between the options until the proper set of switches had been located. These are known as virtual reconfigurable circuits (VRCs). These virtual circuits are able to operate very quickly because they only need to set the value of switches to change their operation, whereas using partial reconfiguration requires reloading the configuration in every hardware resource within the reconfigured region. The VRCs do have their limitations, however. First, they cannot implement any arbitrary function, they are limited to only what was built in at the start. Second, their ability to perform a large number of complex functions is limited by the size of the FPGA. Partial reconfiguration, on the other hand, can get new functions at any time from interfaces like Ethernet, USB, or Wi-Fi, and it can store a huge number of complex functions by trading plentiful disk space for limited FPGA resources.

2.3 Convolutional Neural Networks

Convolutional neural networks are a modification of traditional neural networks that allow the neurons to react to regions of the input, rather than a single point. They do this by using convolution, with the convolution kernel defining the response regions of the system. These ConvNets, or CNNs, as they are sometimes called, have found success in many fields of study, including natural language processing [21, 22], speech recognition [23], musical chord recognition [24], and even electroencephalographic (EEG) understanding [25]. By far, however, their most popular application is image processing.

2.3.1 *In Image Processing*

Two of the most popular application has most assuredly been object detection and recognition. One of the first uses of CNNs was in handwritten character recognition. Back in 1998, LeCun et al. [26], were able to achieve the then state-of-the-art error rate of 0.8% on the MNIST image database of handwritten digits using a convolutional neural network. Currently, the best recorded performance on this database has reached a near-human performance, 0.23%, using a committee of 35 networks [27].

Face recognition is another use that has received a lot of attention. The first paper, to my knowledge, on using convolutional neural networks for face recognition is [28]. This used small images and a relatively simple convolutional network in conjunction with more tradition methods, like self-organizing maps and some basic local sampling. More recent methods, such as [29], use the ConvNet as an end-to-end solution and focus more on increasing the size of the network (number of layers, activation functions, etc.) and improving the training data (by size jittering, translation, rotations, etc.).

In the more general task of image classification, ConvNets have consistently been advancing the state-of-the-art. In the ImageNet competitions, the majority of entries in the past several years have used ConvNets, and have been the winners since 2012. This is a natural use of the networks and is a relatively common starting point when new architectures or patters are proposed.

Some other applications that have been recently explored using ConvNets in some interesting ways are super-resolution [30], deconvolution [31], facial landmark location

[32], denoising and blur removal [33, 34], and no-reference image quality assessment [35]. These applications show that ConvNets generalize well and are able to be used effectively in a wide variety of computer vision applications, even applications that are not usually associated with machine learning.

2.3.2 Using FPGAs

There have been several discussions of using custom hardware to implement a generic convolutional neural network accelerator on an FPGA. Work by Farabet et al. [36] indicated that FPGAs are able to provide around two orders of magnitude faster computation and one order of magnitude less power consumption than CPUs on the same network. Another important discovery they made is that there was no difference in detection accuracy of the network when using floating point numbers versus fixed-point numbers. This held true even when the fixed-point numbers were quantized down to 8 bits. The same author completed a more concrete example of this in [37] by implementing a face detection network and achieving 10 frames per second on 512×384 pixel grayscale images and a peak performance of 9.8 billion connections per second.

The implementation of this work used a special purpose processor core that contains the operations that may be needed for the network. It has a compiler that translates a high-level description of a network to a low-level implementation that uses these cores to perform the computations. Another example of this approach is in [38]. In that work, however, the structure of the FPGA fabric is already set so that the FPGA is filled as much as possible with resources. The host computer then compiles a network into a structure that maps to these fixed resources. The results were slightly better, achieving 6 frames per second in face recognition on 640×480 pixel color images. In this case they also compared the performance to the host computer, which achieved only 1 frame per second on the same data. To run either of these networks a CPU is used as a host and controller, and the data is passed from the CPU to the coprocessors in the FPGA fabric to perform the computation.

Next, there is the “nn-X” architecture [39]. This architecture, like the others, is also implemented by using a collection of special purpose processing cores. The memory performance and scalability has been improved, however, which led this implementation to achieve 42 frames per second on 500×350 pixel full color video when performing face

detection, and over 200 frames per second performing road scene parsing on a 500×288 pixel full-color video.

Lastly, there is the implementation of the Convolutional Face Finder (CFF) algorithm [40] on an FPGA [41]. This implementation was built specifically for the CFF algorithm, and still uses sequential processing elements as the building blocks of the network implementation. Depending on the size of the FPGA this implementation can reach 35 frames per second on 640×480 pixel images or up to 127 frames per second on 320×240 pixel images.

The majority of the computation time in implementing ConvNets is in the convolution layers. To address this [42] explores several methods of implementing the convolution as matrix multiplication and taking from the advancements that have been there to minimize the amount of calculation that needs to be performed. Using proper techniques there can be up to a 47% reduction in the workload of the convolution, further improving their performance.

3 Embedded Systems for Image Processing

In this chapter an FPGA-based low-SWaP printed circuit board for real-time image processing is developed. First, FPGAs are described and compared to CPUs and GPUs for image processing tasks. Second, the Xilinx Zynq All-Programmable System-on-Chip (SoC) is described. Third, several existing hardware modules that utilize the Zynq are examined for their ability to meet our objectives, and one is chosen. Fourth, a breakout board is developed for the chosen module. Fifth, the final baseboard that can stream and processing real-time, high-resolution video is designed and constructed.

3.1 FPGAs vs. CPUs vs. GPUs

Here we give a brief description of how FPGAs work, then compare their design architecture and performance to CPUs and GPUs in the context of image processing.

Field-programmable gate arrays (FPGAs) are very low-level, massively parallel devices. In essence, they contain an array of flip-flops and logic gates that can be reprogrammed to perform any operation. In fact, an FPGA is a massively parallel cluster of extremely simple processing elements called configurable logic blocks, logic cells, or one of many other names. FPGAs are often used to prototype application specific integrated circuits (ASICs) before they are fabricated. For this reason, even though the physical device does not ever change, FPGAs are often considered programmable hardware. Each processing element takes its input from several single bit signals, then performs an operation and outputs a new set of signals to other processing elements. The processing elements are connected to each other by a complex set of interconnects that run throughout the entire device. The interconnect forms a mesh of the processing elements. This mesh is often called the FPGAs “fabric”. The processing elements themselves each perform only very simple operations. They often only contain lookup tables and flip-flops, but they sometimes include comparators, or full adders, or various other circuits [43]. In addition, there is usually a small set of elements dedicated to memory and DSP operations. The FPGA is configured by setting the operation of each processing block and configuring the interconnect to route signals between and among the processing blocks. Usually this is done only once for the entire device when it boots, but in certain cases the entire device can be reprogrammed

while it is running. More recently vendors have also been providing tools for performing partial reconfiguration. That is, specific parts of the FPGA can be changed without affecting the rest of the device. The configuration of the device, either the entire device or only a portion, is stored in a “bitfile”. In almost all cases the configuration file is created by a set of tools distributed by the FPGA’s vendor. Among these tools are the synthesizer and place-and-route tool. The synthesizer takes human readable code written in a register transfer language (RTL) and builds a list of all the operations and connections that are needed to implement the code (called a netlist). Then, the place-and-route tool maps the requirements to a specific FPGA’s available hardware resources. Both of these operations are very time intensive. It can take anywhere from several minutes to several hours or even days depending on the size and complexity of the design, the optimizations requested, and the size of the target device.

When building an image processing system, FPGAs are often overlooked because they are considered too difficult to use, or too niche, or only used to develop ASICs. CPUs and GPUs are generally considered much more approachable and the better choice for general use. The register transfer languages that FPGAs use can be quite difficult to learn, let alone master, and they require a drastically different approach to programming than the approach used for CPUs or GPUs. FPGAs do not “run software” in the traditional sense. The code written for an FPGA describes how to program a set of processing elements that all operate concurrently. GPUs also have many concurrently operating “cores”, but their number is on the order of hundreds or thousands, and code within the cores is still executed sequentially. FPGAs contain hundreds of thousands to millions of processing elements that all operate concurrently and only perform a single function. This completely removes any shred of sequential behavior that is not explicitly built back into the system. There are attempts to use sequential languages like C++ to program FPGAs, but they are fraught with issues and have so far not performed nearly well enough to be widely accepted or replace the traditional languages.

Parallelism is at the heart of FPGA performance. For functions that can be effectively parallelized, there are major benefits to using a massively parallel architecture. In general it is true that an FPGA will perform well with operations that parallelize well, and because of their low clock speed they will perform poorly with operations that do not. Image

processing generally performs very well on FPGAs because many image processing algorithms have a large amount of parallelism that can be exploited [44].

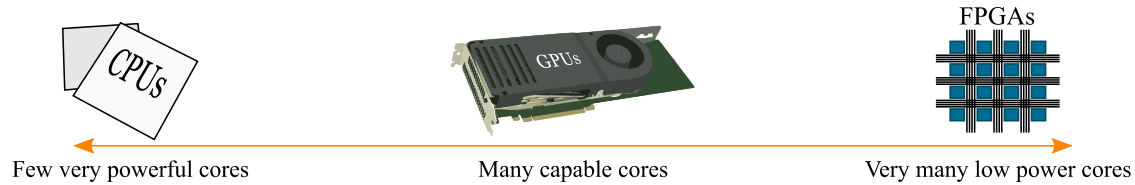


Figure 3.1: FPGAs are at the extreme end of the performance vs quantity spectrum: they have an extremely large number of very low performance “cores” as compared to CPUs or GPUs.

3.1.1 Design Architecture

When deciding between FPGAs, CPUs, and GPUs, much of the decision is determined by which architecture is best suited for the application being targeted. Image processing is a specific case of 2D signal processing. Because of the size of the images, many operations become memory bandwidth limited, so very high memory throughput is required. Further, images are most commonly transmitted as 3-channels of 8-bit integers, which means there is often no need for large bit-depth types or floating point arithmetic.

Compared to CPUs

CPUs are the oxen in Seymour Cray’s famous question: “If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?” In this case, the decision is between ~1-10 extremely complex CPU cores and ~100,000-1,000,000 extremely simple FPGA processing blocks. Often the few oxen are preferred because they are relatively easy to control and are quite well behaved, but, given the sheer size of modern FPGAs, most image processing applications can see performance increases by exploiting parallelism with an FPGA.

Compared to GPUs

GPUs are highly parallel devices, similar to FPGAs. Unlike FPGAs, however, they can be programmed efficiently in the same language as CPUs because the cores all execute the same sequential program. GPUs are generally use single instruction, multiple data (SIMD)

parallelism, and this simplifies the code that needs to be written. The GPUs processing cores are more complex than the FPGAs programmable blocks, to the point that they are basically miniature CPUs. This means they incur the overhead of instruction set processing and control logic that enables general purpose operation. As well, they have fixed 32-bit or 64-bit math functions that are useful in general, but are often not needed for image processing because most images use only 8 bits per channel. FPGA can be optimized for 8-bit numbers and perform 3 or 4 8-bit operations in nearly the same space and time that the GPU performs a 32-bit operation on a single 8-bit channel.

3.1.2 Performance

Performance in a given task can be measured in many ways. For real-time video processing the most important factors are speed, measured in framerate, and latency. Other considerations programing difficulty, or how long it will take to develop the code for an application; energy efficiency; and physical size.

Speed

As shown by Asano, et al. [44], whether FPGAs, CPUs, or GPUs will be the fastest is very application dependent. The major tradeoff is between the FPGAs very fine-grained parallelism but low clock speed, the GPUs high clock speed and moderately large parallelism but slow and difficult memory management, and the CPUs high clock rate, easy memory management but very limited parallelism. In image processing applications there is very often a large amount of inherent parallelism that can be extracted. Therefore, FPGAs and GPUs will generally outperform CPUs. Between FPGAs and GPUs the performance difference is not obvious. For example, FPGAs have been shown to perform better in stereo vision and K-Means clustering tasks [44] and matched filter computations [45], while GPUs performed better for two-dimensional filters [44] and normalized cross-correlation [46]. Some conflicting results further complicate the issue. For example, in [8] it is shown that FPGAs outperform GPUs for the top-hat transformation, with a high-powered CPU actually outperforming both. However, we saw in [44] that two-dimensional filters perform better in GPUs, and the top-hat transformation is simply a sequence of two-dimensional filters. Given the complexity of these devices, it is very difficult to state categorically that any device will perform the best for image processing in general or even for a particular operation specifically.

Programming Challenges

FPGAs are notoriously difficult to program. They provide a very limited number of built-in features and there are very few libraries available upon which to build up a design. Additionally, there are fewer learning opportunities for FPGAs as compared to CPUs and GPUs. CPUs and GPUs both have a plethora of very good libraries built for almost any task and there are thousands of resources for learning how to use them. For now, however, the situation is that FPGAs almost definitely require much more time and effort to use than either CPUs or GPUs.

Energy Efficiency

FPGAs excel in the area of energy efficiency. In [9], the authors implement a sliding window operation, sum of absolute differences, and correntropy, and determine that, “the FPGA used orders of magnitude less energy than other devices in many situations”. In [8] the same order of magnitude energy reduction is shown. In addition, since the release of the Xilinx Zynq System-on-Chip (SoC), which has an ARM-based CPU and an FPGA in the same package, neither a discrete CPU nor a soft-core CPU is needed in FPGA-based hardware systems. This increases their energy efficiency of the entire system as a whole, in addition the energy efficiency of the FPGA itself. There have been improvements on the GPU side as well. The NVIDIA Tegra K1 is similar to the Zynq, in that it includes an ARM-based CPU in the same package as the GPU. Here also the energy efficiency of the system is significantly improved. On the whole, however, out of FPGAs, CPU, and GPUs of comparable performance, FPGAs are almost always the most energy efficient.

Size

Size is measured in two ways: the size of the device itself, and the size of the required supporting devices. In this arena CPUs are the winner if not for the simple reason that most FPGA and GPU systems also include a CPU (in the same package or otherwise). In the case of GPUs, using a CPU as a host is required, while for FPGAs it is optional but very common. As an alternative FPGAs can simply embed a small CPU like MicroBlaze or NIOS II within their fabric (so called “soft-core” processors), but this reduces their capacity and the CPU is not very performant. In any case there is very little variance in this factor because the most recent Zynq and Tegra SoCs are very similar in size to CPUs used in embedded systems. In fact, for many devices the size is dominated by the number of

external pins they have. The external package dimensions are generally always much larger than the actual silicon device.

3.1.3 Recommendations

FPGAs are very good devices for implementing small, low power embedded real-time image processing systems. Because of the ease of use of CPUs and GPUs, however, these will likely continue to be the first choice for new designs. If and when the performance of these devices is unable to achieve a system's objectives, or achieving the objectives consumes too much energy to be usable, FPGAs are often able to improve the system's performance. The firmware library developed in this thesis aims to improve FPGAs ease of use for image processing tasks and allow them to become a first choice option.

3.2 Zynq SoC

The Xilinx Zynq All-Programmable SoC is an FPGA that also includes a dual-core CPU. CPUs are very common in FPGA designs because they provide a place for orchestrating the hardware blocks and easily handling less time-sensitive operations like Ethernet, USB, and persistent storage. If they are sufficiently capable, they can also be used to assist the FPGA in actual processing tasks as well. When deciding to add a CPU to a FPGA design, there are three options: first, add a soft-core CPU to the FPGA design; second, add a separate physical CPU to the design; third, use the Zynq.

Soft-core CPUs provide all the ease of use of CPUs with almost none of the computational performance. In addition, including the CPU inside the FPGA's fabric reduces the number of resources that are available for the operations that necessitated the FPGA in the first place. CPUs with advanced features like USB, Ethernet, or MMC can quickly become very large and dominate the resource and power usage of the FPGA design. Second, FPGAs run quite slow compared to hardened CPUs, so even with all the resources dedicated to the CPU it would run much slower than a hardened CPU. Third, the most popular soft-core CPUs like Microblaze or Nios II have their own instruction set architectures and require their own toolchains for compiling or cross-compiling software. This means none of the mainstream software repositories are directly usable.

A two-chip solution solves the issues with soft-core CPUs, but it introduces some of its own. Most importantly, a two-chip solution enlarges the physical hardware design and slows down communication. On a single package solution (soft-core or SoC), the RAM resources can be shared between the CPU and FPGA fabric. This is not the case for two-chip solutions, and this alone will greatly degrade system performance. In GPU-based systems, that always use separate RAM for the CPU and GPU, proper memory management is a dominant factor in the performance of the program and quite a challenge to do well. Using a two-chip solution for and FPGA and CPU would create the same challenge.

The Zynq provides a single chip that houses both the FPGA and hardened CPU. This reduces the complexity of the physical hardware, and it still allows the entire FPGA to be used for operations. In addition, the RAM is connected through the CPU, so the same memory space is shared between the CPU and FPGA. There are also facilities to integrate into lower levels of the CPU and add instruction level accelerators and enforce cache coherence. This solution provides the fastest, smallest, and most power efficient solution of the three options.

3.3 Embedded Hardware Platforms

With the choice made to use the Xilinx Zynq SoC, the next step is to choose a development board to use. The objective is to have a small, low power device for image processing. This requires video input and output interfaces and as little else as possible. Unfortunately, such a specialized, minimal system does not already exist, so a board is chosen that allows the necessary components to be added and is as small as possible. The best board available for this purpose is the Enclustra Mars ZX3. In the following paragraphs we will describe the ZX3, and compare it to several of the other available options.

The main draw of the Mars ZX3 is its size. It is a 67.6 x 30 mm board that has no discrete connectors. This offers very little functionality initially, but it provides a very good platform for developing a small custom system. The ZX3 module is shown in Figure 3.2. It uses the XC7Z020-2 device, which is at the high end of the Artix class Zynq devices, though the Artix class is the lowest class of Zynq devices. It contains all the components necessary for a fully functioning FPGA system, including: external clock source, power

regulators, 1 GB DDR3 RAM, and 512 MB NAND Flash. The only input required is a single 3.3 volt power supply and the module handles the rest.

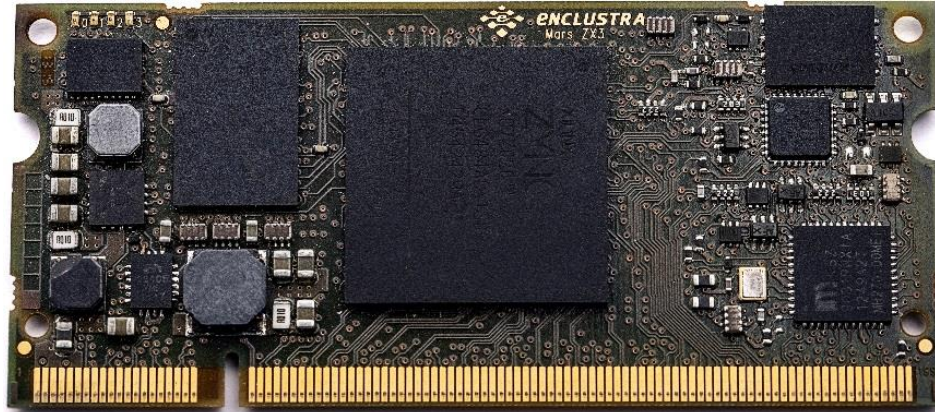


Figure 3.2: Enclustra Mars ZX3 System-on-Module. This module includes the Zynq xc7z020clg484-2, 1GB DDR3 SDRAM, a Gigabit Ethernet PHY, 512MB NAND Flash and power regulators in a 200 pin SO-DIMM form factor.

There are many other boards that use the Zynq SoC, most were available when this work was started, though several were not. Notable boards include the Xilinx's own ZC702 and ZC706 boards, the *Zed* family of boards, namely the Zedboard, MicroZed, and PicoZed. The Xilinx boards are rather large and expensive, and they only have a single HDMI port that is connected to an Analog Devices ADV7511 HDMI transmitter and no HDMI input. The Zedboard was one of the first, and is likely the most popular development board for the Zynq. It uses the XC7Z020-1 device, and has many peripherals, including an HDMI output and VGA output port. However, it also does not have a video input port readily available. The MicroZed and PicoZed are small, minimal SoMs comparable to the Mars ZX3, but they are not as small as the ZX3. In addition, the MicroZed's inclusion of a gratuitous and humongous combination Ethernet and USB connector on an otherwise small board is confusing at best. The Mars ZX3 is a truly minimal board that is extremely small and provides a solid platform for developing a custom hardware solution. Figure 3.3 shows a size comparison of these other boards with the Mars ZX3.

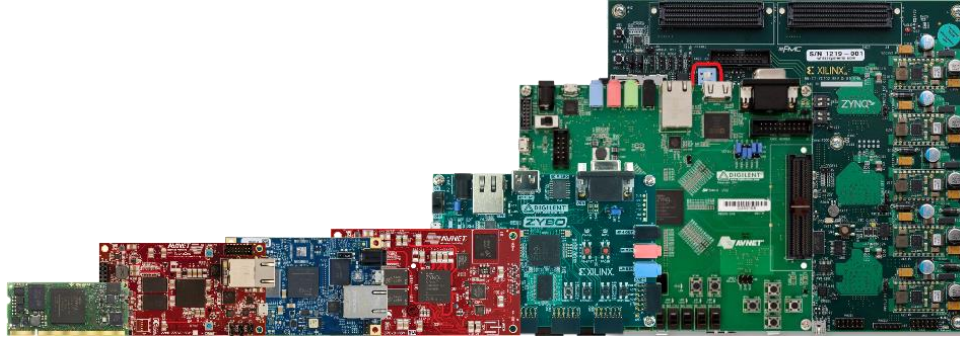


Figure 3.3: Some of the available development boards that use the Xilinx Zynq SoC. The Enclustra Mars ZX3 (far left) was chosen because it is the smallest one available.

3.4 Custom Baseboard for the Mars ZX3

To achieve the lowest possible size and power consumption for the final system, a custom baseboard for the Enclustra Mars ZX3 was developed. Off-the-shelf evaluation boards include many features that are not required for a video processing system. These extra features make them larger and use more power than a board built for a single purpose. This section describes the design, prototyping, and construction of a special purpose printed circuit board for video processing. It starts by discussing the video I/O interface that will be used. Then the development of the breakout board used to prototype the final design is described. Finally, the final circuit board is designed and constructed.

Before developing the system, a video I/O interface had to be chosen. There are quite a few options for video interfaces, and the choice is mostly application specific. For this application the best choice will be high speed, compatible with FPGAs, support uncompressed RGB video, and popular enough to allow testing many different inputs and outputs. Interfaces built specifically for video data include: VGA, Component video, Composite video, DVI, HDMI, DisplayPort, MIPI CSI and DSI, CameraLink, SDI, general LVDS, and basic parallel video. More general purpose interfaces such as USB, FireWire, Ethernet, PCI, Wi-Fi, or Thunderbolt could be used as well. The main considerations for the video I/O interface are quality, data bandwidth, complexity, and to some degree, popularity.

Image transmission quality is extremely important in image processing systems. Images degraded by transmission losses can result in poor processing results regardless of the

sophistication of the processing being applied. For video interfaces, VGA, component, and composite video pose the most obvious concerns for quality because they are analog interfaces. Maintaining high quality video over these interfaces would require very high quality analog-to-digital converters (ADCs), digital-to-analog converters (DACs), and short, high-quality wires to minimize signal degradation. In the digital domain, signal quality is not an issue when the interface is used properly, so the issues arise not in losing what is transmitted, but in what gets transmitted in the first place. The major quality concerns in digital video are compression and bit depth. All of the digital interfaces are capable of transmitting uncompressed video, assuming they have enough bandwidth. The video specific interfaces, however, do impose some restrictions on the bit depth, but it is high enough to support most all applications. The general purpose interfaces can, of course, support any bit depth. In the end the issue of quality is limited more by the interface's bandwidth than by what the interface itself supports.

The main consideration for a video interface is its data bandwidth. The most modern interfaces such as HDMI, DisplayPort, MIPI CSI and DSI, USB 3, and Thunderbolt 2 have plenty of bandwidth to handle most modern high resolution video, but it comes at the cost of extra hardware for transmission, encoding, or decoding. For example, the Xilinx FPGAs do not currently natively support the MIPI D-PHY specification, so external hardware is required at the physical layer to support CSI and DSI. USB 3, and Thunderbolt 2 require entire external encoder and decoder chips to handle the speed and formatting of the data. In general digital interfaces serialize the data transmitted over them, so at minimum high-speed clocks deserializers are required.

Interface complexity can be a huge impediment to building a small, lower power system. Ideally there would be no wasted time or effort in simply sending and receiving data. This is the case for extremely simple interfaces like parallel video or VGA, but the majority of the interfaces have at least some overhead. A simple interface would be DVI, which uses 8b/10b encoding and serializes the data. An example of a complex interface is Ethernet, which comes with an entire protocol stack to implement and handle. The general purpose interfaces like USB, Ethernet, and Thunderbolt are much more complex than the video specific interfaces. This complexity is wasteful in two ways: consuming resources in the FPGA, or requiring physical hardware at the cost of power consumption, cost, and physical size.

While not a major concern, the interface's popularity is worth considering. Choosing a popular interface means there will be many more devices available to use with our system, and much more support in developing it. For example, almost all monitor and displays support VGA and DVI, and many support HDMI or DisplayPort. Most televisions generally support HDMI, component video, composite video, and VGA. Using any interface other than these will require uncommon hardware to monitor the inputs and outputs of the video processing system. Cameras, on the other hand, usually output their data as USB, MIPI CSI, HDMI, LVDS, GigE, component video, or parallel video.

As a goal, we chose to aim for supporting video data at full HD 1920x1080 resolution, 60 FPS, 3 channel, uncompressed, 24-bit video. With a bit rate of around 2.78 gigabits per second, the bandwidth requirement immediately rules out USB 2, Fast and Gigabit Ethernet, FireWire, Wi-Fi, and component video. Quality concerns rule out the analog interfaces altogether. Excessive complexity rules out Thunderbolt, USB 3, and higher bitrate Ethernets. Even with these restrictions there is still DVI, HDMI, CameraLink, DisplayPort, LVDS, and SDI. In this list a popularity contest would show DVI and HDMI as the winners, and fortunately the DVI specification is a proper subset of the HDMI specification. The choice, then, is to use the DVI interface with HDMI connectors.

3.4.1 Custom Baseboard Version 1

The first baseboard that is developed for the Mars ZX3 module is a breakout board. This board was used to prototype the electrical connections of the ZX3, HDMI connectors, and control console.

The Mars ZX3 is in a standard 200 pin 1.8 volt SO-DIMM form factor, so connectors are readily available. On this breakout board the pins of the connector are all broken out to standard 0.1 in pin headers. There are 57 GPIO differential pairs on the Mars ZX3 module, and their positive and negative pins have been connected to adjacent locations on the pin headers. The trace length of the positive and negative signals in differential pair is matched to within 2.5 mils. The differential pairs have not been length matched among each other. The HDMI connectors were connected to 2-row, 20-pin 0.1 inch headers. The HDMI connectors only have 19 pins, so the 20th location on the pin header is unused. The differential pairs of HDMI connectors are also length matched between each pair, but not among all the pairs. Length matching among the differential pairs is important for data

integrity, and forgoing this step limited the usable resolution of this board. Unfortunately, this is something that was learned the hard way.

The 3D model of the board and the physical result is shown in Figure 3.4. The board is 5x2.3 inches, and breaks out all 238 pins from the Mars ZX3 and two HDMI connectors. Building a working systems with this board required manually making many connections to the pin headers. At minimum, the power and ground signals need to be connected. The power and ground connections account for the vast majority of the signals connected to the board. The Mars ZX3 has 16 power pins (17 including battery power input) and 31 ground pins that need to be connected to power the system. Part of the reason for the multitude of power pins is that the module can be run in several different configurations. There are only 2 power pins that are required to be 3.3 volts. These pins are used to power the Ethernet PHY, Flash, Oscillator, RTC, EEPROM, and LEDs. The remainder of the power pins are used to power voltage regulators or as voltage references for the I/O banks of the FPGA. The inputs to the power regulators support voltages of 3.15 to 5.5 volts. The FPGA has 3 banks of I/O that can each individually be run at different voltages. Banks 34 and 35 support voltages from 1.8 to 3.3 volts and bank 33 supports 2.5 to 3.3 volts. The common voltage supported by all of the power pins is 3.3 volts, and the module does support being powered with a single 3.3 volt supply.

HDMI uses 4 differential pairs: 3 for data channels and one for the clock. Both the data and clock pins can be directly connected to the FPGA's I/O pins. The HDMI clocks, though, should be connected to the dedicated clock-capable I/O pins on the FPGA to ensure that the FPGA can meet timing on the clock signals. Bank 34 and Bank 35 each have clocking resources that can route clock signals within their bank. These resources only support one clock, so one of the HDMI connectors should use signals in Bank 34 and the other should use signals in Bank 35. For the HDMI's DDC protocol, the power and I²C interface of the HDMI input are connected directly to the power and I²C of the HDMI output. This makes the module invisible to the input source and ensures that any input to the breakout board is compatible with the display connected to the board's output. It also avoids any requirements to handle the 5 volt I²C signals or implement the EDID protocol.

For control, a UART to USB converter is connected to the UART pins on the module. This provides access to the console used by the bootloader and by Linux.

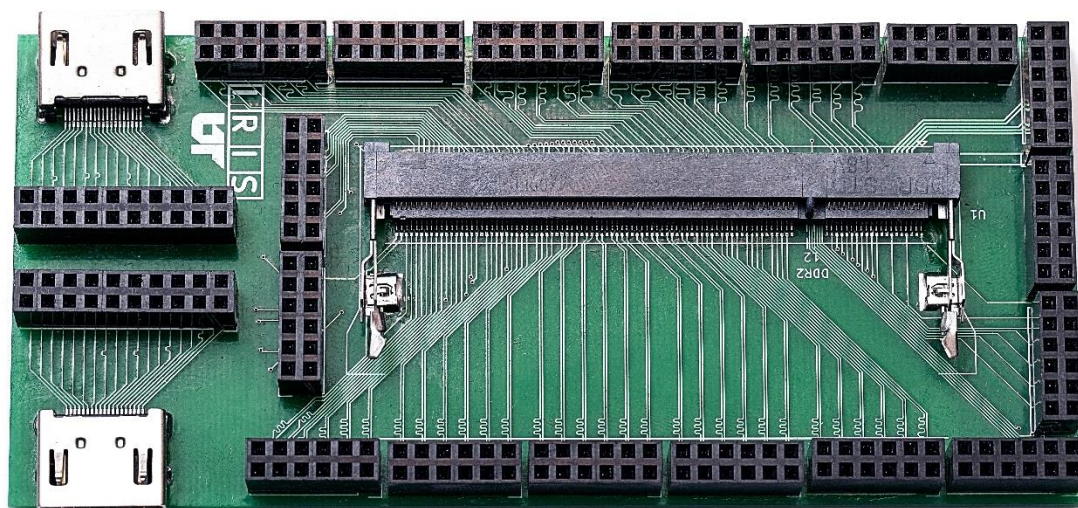
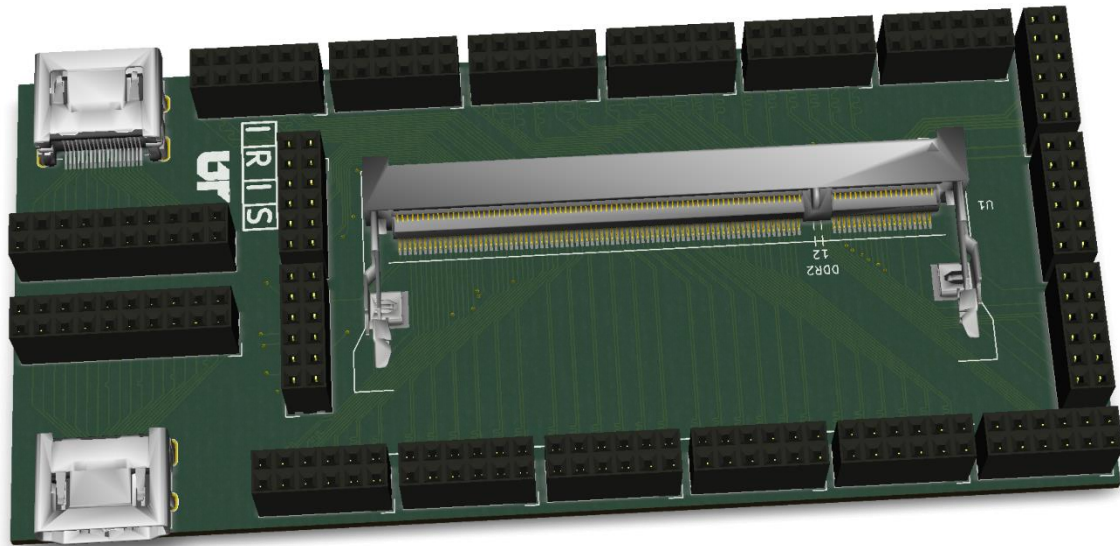


Figure 3.4: Custom baseboard version 1, a breakout board for the Mars ZX3 and 2 HDMI connectors. This board was simply used for testing and experimentation with the pinout of the ZX3. This testing was used to develop the final pinout for the second board.

Results

With the system shown in Figure 3.5 the device can be booted, receive HDMI input, transmit HDMI output, and use the UART interface to communicate with a host computer. Unfortunately, it is only able to reliably receive and transmit HDMI signals up to 640x480 at 60Hz. This is because of poor signal quality on the HDMI data and clock lines. The poor signal quality is caused by mismatched delay (from not length matching among the differential pair), poor impedance control, and poor signal isolation allowing significant crosstalk. These issues will be addressed by developing a printed circuit board that contains all the wires internally.

3.4.2 Custom Baseboard Version 2

After using the breakout board to prototype and develop a video processing system, a new, optimized board was developed. This second board routes all of the necessary connections within a single small board. The new design is a 4-layer printed circuit board that includes all the connections and components necessary for the HDMI input and output interfaces, power input, and control console. Additionally, this board includes logic level translators for the I²C bus of the HDMI's DDC interface so that the FPGA can interact with the source and sink devices. The power supplied from the HDMI input connection is used as the output power of the logic level translators and is also sent along to the HDMI output connector, so that the board itself still does not generate any 5 volt power itself, but rather uses the 5 volt power available through the HDMI protocol. Finally, the HDMI connections on this board are all length matched between and among the differential pairs and impedance controlled to 100 Ω differential impedance and 50 Ω single ended impedance. The 3D model and physical realization of this new design is shown in Figure 3.6. The module is shown connected to an input source, output device, and power in Figure 3.7.

The goal of this second baseboard is size. The HDMI, power, and UART connectors are placed on one side of the board, while the Mars ZX3 is on the other to minimize the surface area of the board. Also, the height of the connector for the Mars ZX3 creates 2 mm of clearance between the ZX3 module and the board. This space allows small components to be placed underneath the module. The final board 3x1.5x0.56 inches, weighs 1.00 ounce, and consumes less than 5 Watts.

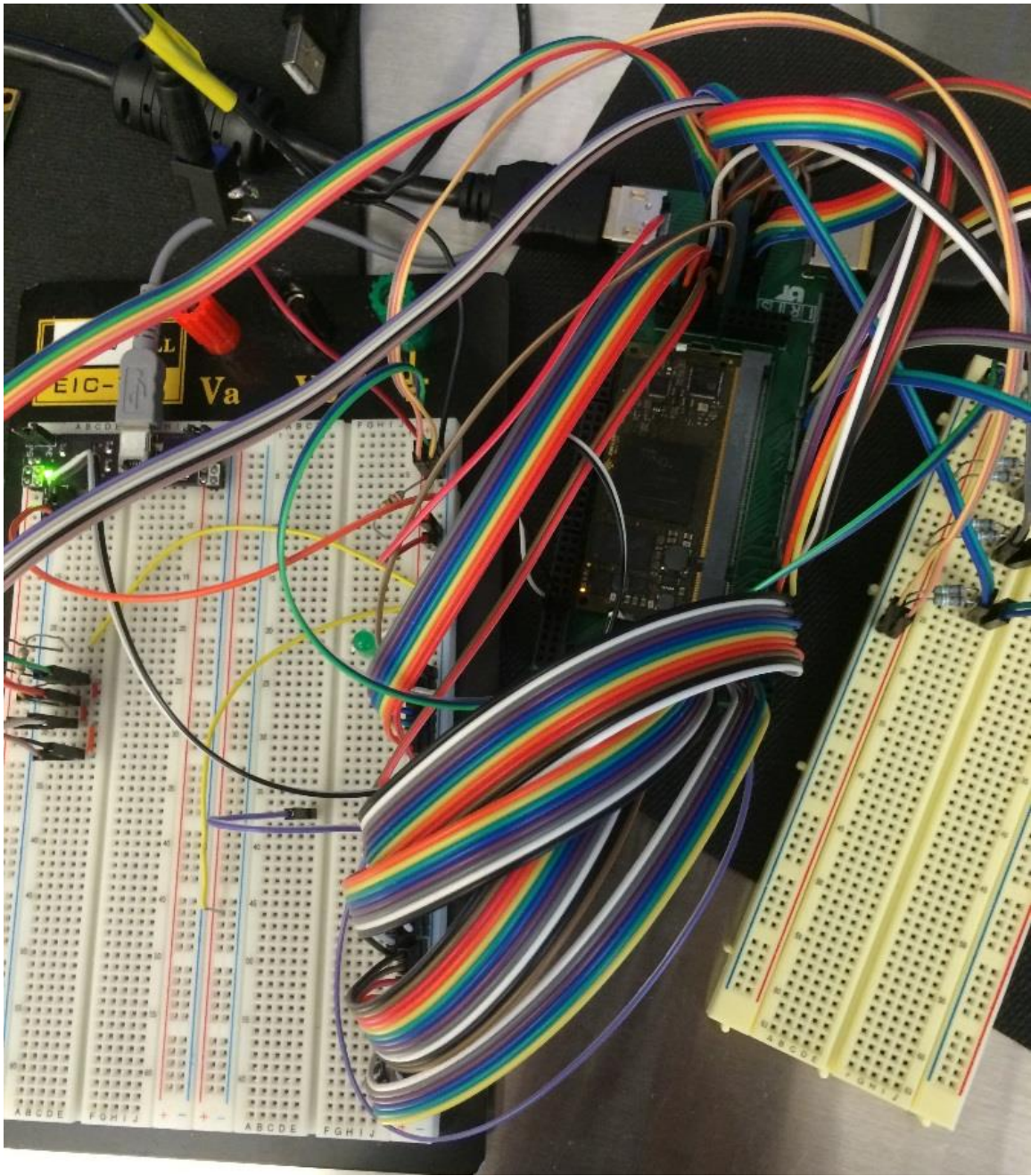


Figure 3.5: Version 1 baseboard fully connected with HDMI input and output. The power and ground pins are mostly at the bottom in the wider ribbons. The top left helper board is a UART to USB converter. The HDMI works, but only at low resolutions due to poor signal quality due to the wiring.

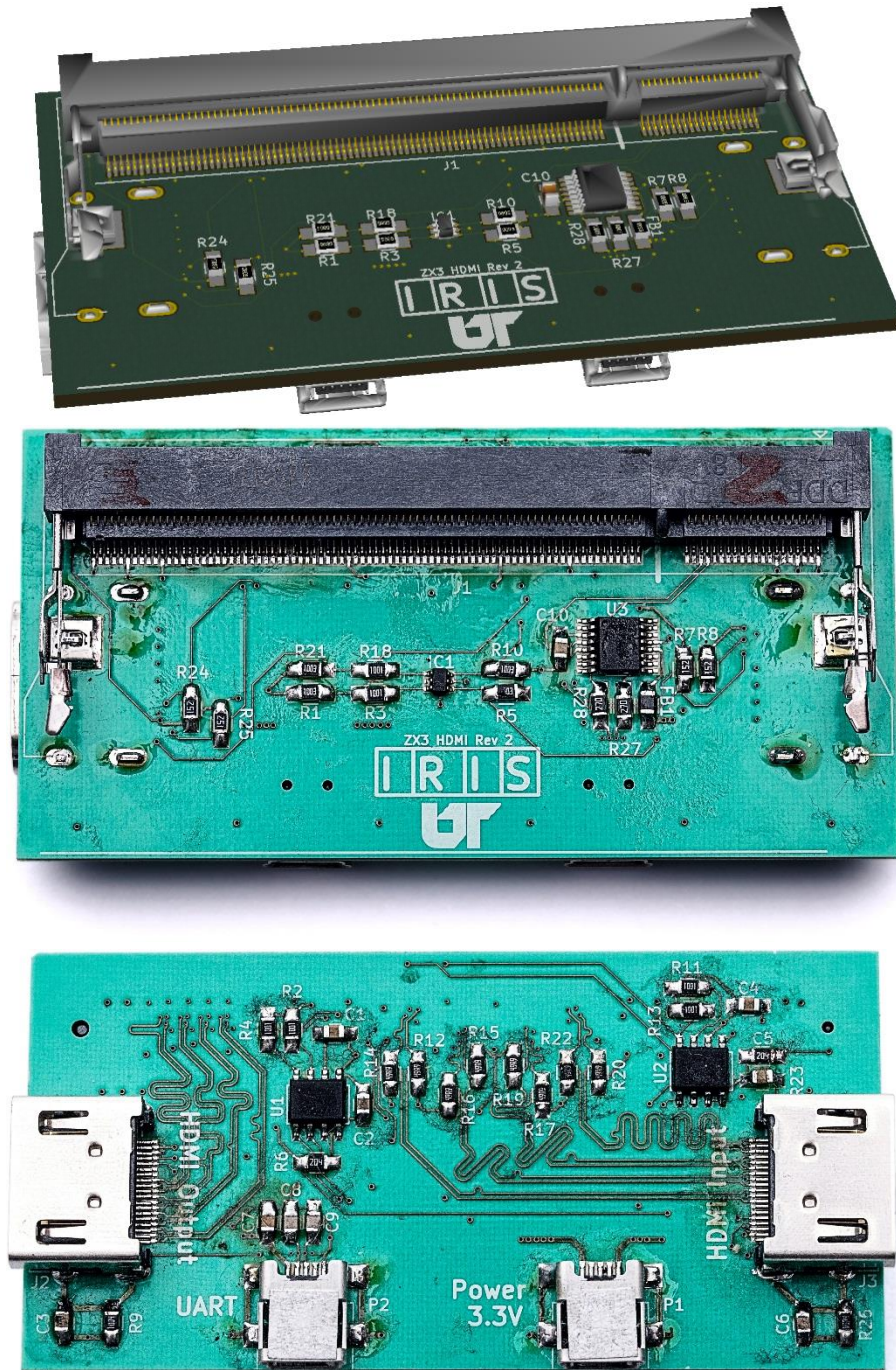


Figure 3.6: Version 2 baseboard. This board contains the UART to USB converter onboard, as well as logic level translators for the HDMI I2C interface.

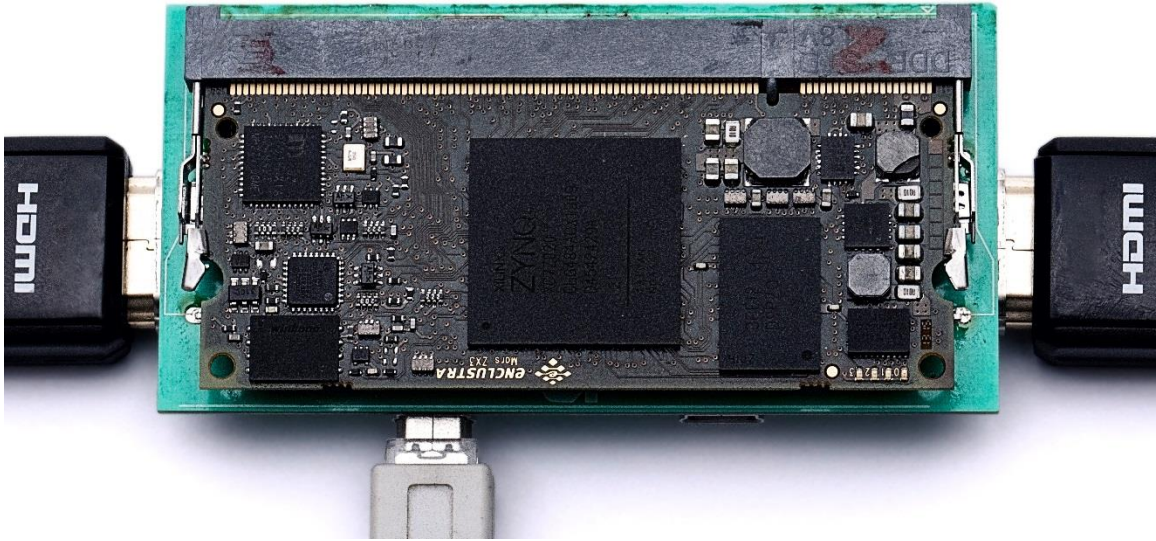


Figure 3.7: Version 2 baseboard fully connected with HDMI input and HDMI output.

Design

The design of the board was intentionally kept as minimal as possible. For the purpose of symmetry with the control port connection, a mini USB connector is used for the power supply input. The supply voltage for the input power is 3.3 volts, even though USB power supplies are always 5 V, so this does require a custom power supply. The power input is not 5 volt tolerant. Because the Mars ZX3 works with only a single 3.3 volt supply, the power and ground pins can be connected directly to the ZX3 module. No other power regulation is required. Other than the power, there are three subsystems in this design: the HDMI input, HDMI output, and the UART to USB converter.

The HDMI input circuit is shown in Figure 3.8. The differential inputs have $49.9\ \Omega$ pull-up resistors to 3.3 volts as required by the TMDS standard. No input termination resistors are required [47]. The DDC signals are translated from 5 volts down to 3.3 volts to communicate with the FPGA, and the hot plug detect signal is buffered to create the required 1 k Ω output resistance for the output to the transmitter.

The HDMI output circuit is shown in Figure 3.9. It is nearly the same as the HDMI input circuit, but there is no need for the pull-up resistors. The logic level translator is still present on the DDC signals. The hot plug detect signal is still buffered, though in this case the buffering acts as a logic level translator as well, to ensure the FPGA inputs do not see 5 volts.

The UART to USB circuit is shown in Figure 3.10. The FT230X part from FTDI is used as a single chip interface between the UART signals from the FPGA and USB port. This chip uses the standard FTDI USB part that has drivers readily available for Windows, Linux, and OS X, so it is very easy to use. It is configured as a bus powered USB to RS232 converter. Being bus powered means that the chip is powered from the USB bus and will turn on regardless of whether or not the rest of the board is powered. This is extremely helpful when debugging the software and firmware on the board because the UART console can be active as soon as the board is turned on. If the USB were powered by the board, then when the board is powered off the computer could not connect to the USB device. Only once the board was powered on would the USB device enumerate on the host computer and accept connections. This is a problem because the FPGA also begins booting

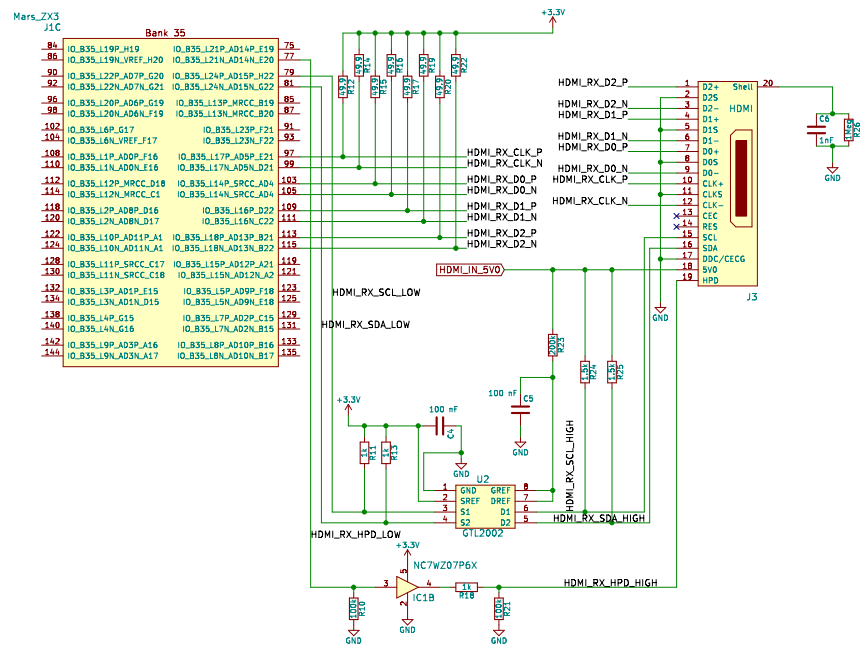


Figure 3.8: HDMI input circuit on the version 2 baseboard.

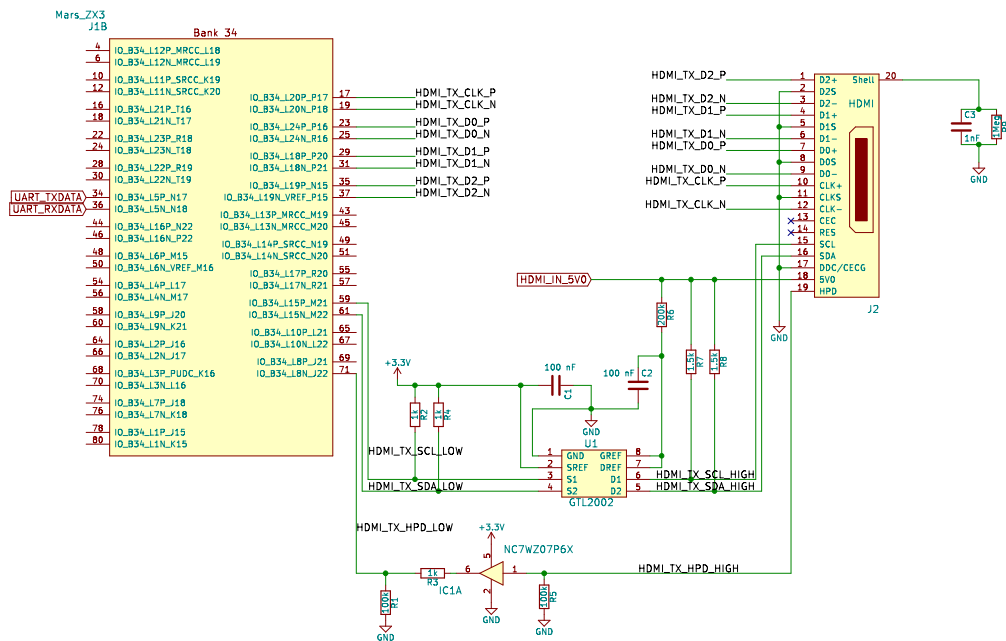


Figure 3.9: HDMI output circuit on the version 2 baseboard.

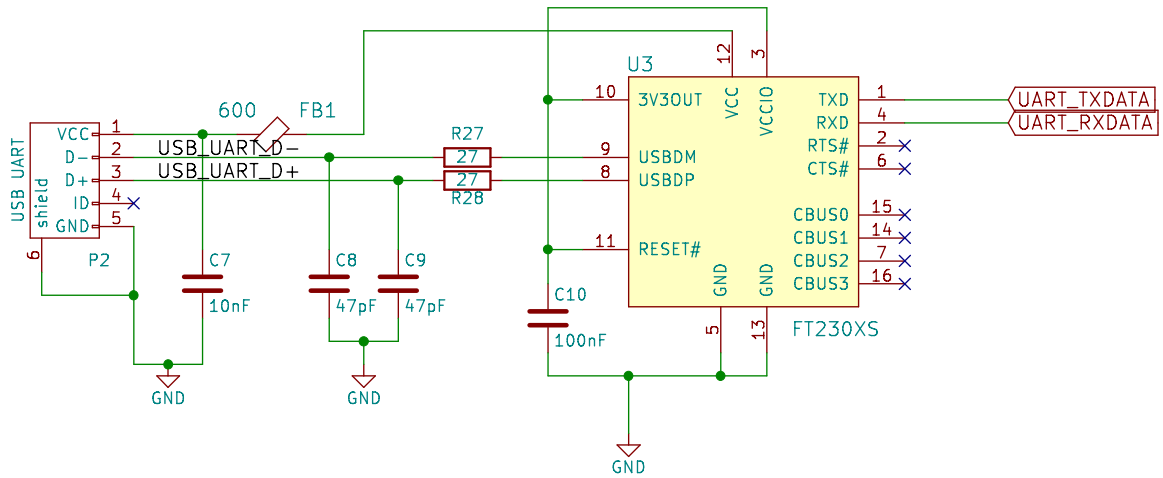


Figure 3.10: UART to USB circuit on the version 2 board.

as soon as power is applied, and it does not wait for the USB connection. Therefore, if the FPGA were to send any output to the console before connection was made on the host computer it would be completely lost. This would make it extremely difficult to debug errors that occur at or soon after booting. The first stage bootloader (FSBL), the standard bootloader (generally Das U-Boot), the early stages of the Linux kernel, or a bare metal application that doesn't use an OS are all examples of software that could fail before the USB would have enumerated and well before the host computer would be connected. Since the USB is bus powered, however, the USB connection remains active even when the board is not powered. Then, when the board is powered and begins to boot the USB connection is able to capture all of the output.

Results

This board is able to transmit and receive much larger resolutions than the breakout board. It can achieve full HD 1920x1080 at 60 Hz output, and up to 1440x900 at 60 Hz input with under 5 Watts of power consumption. Once built, it was discovered that the HDMI input clock is not connected to a clock-capable I/O pin on the Zynq FPGA. This makes it much more difficult for the FPGA to properly route the clock to the clocking resources, and therefore, it is more difficult to lock on to the input clock. This results in a lower achievable resolution. If a proper clock pin were used, however, full HD 1920x1080 at 60 Hz HDMI input is achievable.

4 Image Processing Library for FPGAs

This chapter describes the development of a library of modules that implement image processing algorithms on FPGAs. Reusing code by building libraries is ubiquitous in CPU programming, but is much more challenging in FPGAs. We will start by describing the process of implementing a function on an FPGA. Then we will briefly cover several languages that are used to develop these implementations. From these discussions we will motivate the choice to use the Chisel hardware construction language for developing our image processing library. Finally, the organization, implementation, and contents of the actual library are discussed.

4.1 FPGA Programming

Programming for FPGAs involves building a complete configuration for the FPGAs hardware resources. Rather than a CPU program, which creates a series of instructions that are sent to the hardware sequentially, an FPGA “program” is a file of configuration information that defines how the FPGA’s hardware is configured and routed. Once this configuration is done, the FPGA can run itself without any further intervention. The configuration is described by a bitstream file. This bitstream is created by first taking a human readable hardware description, synthesizing the description to a netlist, mapping the netlist into the FPGAs hardware resources, and capturing this mapping in the bitstream file.

The exact format of this file is proprietary information, so the only way to generate a bitstream file is to use the FPGA vendor’s tools to create it. In general, this file describes how to configure all of the resources in the FPGA, whether they are used or not. Therefore, the size of this file is fixed, regardless of the design it contains. With some newer FPGAs it is also possible to create partial bitstreams. That is, bitstreams can be created that only configure some of the resources, rather than all of them. These allow configuring, or reconfiguring, only part of the FPGA without affecting the rest. When an FPGA first powers on there is no configuration information on the device, so a complete bitstream file must be available for the FPGA every time it boots. This is usually done by providing a small external static memory device that the FPGA can read at boot-time to configure the

FPGA. Additionally, the FPGA can be reconfigured in its entirety or partially while it is running.

The bitstream file is created by using the outputs of the placement and routing (PAR) tools. These tools map a design to the physical resources of the FPGA. The most basic PAR tools would simply assign each piece of the design to its own hardware anywhere in the device. Real PAR tools, however, generate a mapping that optimizes many factors of the design including resource utilization, clock speed, power usage, and many other. The main factor that determines whether or not a design can be mapped into a physical FPGA is the number of resources the FPGA contains and the number of resources the design requires. For a given design, the FPGA must have enough resources to be able to fully contain the design, otherwise the implementation will fail. Another important factor for placement and routing is timing delays. Much of the work of the PAR tools is assigning resources such that the signals propagate between themselves fast enough to ensure proper functioning of the design. This generally means that resources associated with the same operation will be located close to each other, resources associated with high-speed I/O will be located near the clocking resources and external pins, and so on. There are many other optimizations that can be performed during implementation, but these two are the most important for determining the proper functioning of the design.

The inputs to the PAR tools are netlists generated by the synthesizer. Synthesis is the process of taking a description of a hardware circuit in a human-readable language and converting it to a netlist. A netlist is a description of the operations and connectivity of nodes in a design. Current synthesis tools provided by Xilinx and Altera only support synthesis from the Verilog, VHDL, and SystemVerilog languages. These are low level languages, so generally only low level optimizations to these descriptions are possible. This is the point in the process that the hardware descriptions are mapped to specific hardware resources, a process called inferring. For example, arrays of vectors by default are implemented using lookup tables (LUTs), but in certain situations they can be inferred as either block RAM or distributed RAM. Registers can be implemented as flip-flops, but in certain circumstances they are inferred as latches. These translations can be critical to a designs performance, but they are usually not explicit in the code.

4.2 FPGA Languages

Verilog, VHDL, and SystemVerilog are the only hardware description languages (HDLs) supported by the Xilinx and Altera synthesis tools. Many more languages have been created, but they must always be converted into one of these three to be implemented on a physical FPGA. This section describes these traditional HDLs, as well as high-level synthesis languages, the OpenCL architecture for FPGAs, and the Chisel hardware construction language.

4.2.1 *Traditional HDLs*

By far, the two most popular languages for FPGA designs are Verilog and VHDL. Verilog was developed in the early 1980s as a simulation language for hardware designs. Synthesis was not introduced until quite a bit later. VHDL was developed as a documentation language, with use for simulation even being an afterthought. Because of this history, many of the features in both Verilog and VHDL are not synthesizable and remain only for simulation or documentation purposes. These remain the dominant language in the industry because they are time tested, well supported, and there are no sufficiently useable alternatives to unseat them. Verilog and VHDL are both considered Register-Transfer Level languages. This classification indicates that they are used to model hardware designs as a sequence of clocked registers with combinational logic between them. This paradigm is in contrast to more abstract behavioral descriptions and lower gate-level descriptions.

4.2.2 *High-Level Synthesis*

High-level synthesis (HLS) is the process of taking a high-level description of a hardware design and automatically determining the proper low-level design. This process abstracts the architecture of the design in favor of the behavior. It leaves the architecture as a detail for the compiler to sort out. This arrangement often does not achieve the best possible performance because the compiler will not always make the best (or sometimes even correct) choice, but it does make the programming easier. The architecture of hardware designs is extremely important to efficient implementation, but designing proper architecture is not easy, especially for people unfamiliar with hardware design.

High-level synthesis also involves at least some amount of automatic parallelization – a fundamentally ill-posed problem. One of the most popular HLS tools is Xilinx’s Vivado HLS. Vivado HLS converts C and C++ into Verilog or VHDL automatically. To make this process even reasonably effective, compiler directives must be added to the code to guide the compiler to the proper implementation. Things like loop unrolling or pipelining, dependency management, input and output busses, and many other details can be specified as compiler directives (inline as pragmas, or in a separate file). In general, using Vivado HLS effectively requires a detailed understanding of the low-level hardware [48] to a point that the ease of programming can become unnecessary and limiting. While it can make it reasonably easy to make something work, it takes a lot of effort to make it efficient and high-performance. Even so, it will likely never be as efficient as if it were written directly in the low-level language. Additionally the C level simulator in Vivado HLS is not guaranteed to be cycle accurate, and in practice, it is not uncommon for the simulation results to not match the hardware results. Dealing with these behavior discrepancies requires additional knowledge of the high-level C and C++ compiler and adds an additional level of debugging that should not exist.

4.2.3 OpenCL

Other high-level environments for programming FPGAs have been, and are being, developed. OpenCL is an environment originally built to take advantage of GPUs. There is a very strict host-accelerator relationship between the CPU and GPU (or FPGA, as is becoming the case) where the CPU controls the data and the program flow, and offloads some of its workload to the accelerator. The programming architecture is very similar to NVIDIA’s CUDA environment, in that, they both are used by writing small “kernels”, or small programs intended to be run many times in parallel on the accelerator. The kernel’s are written in a variant of C and are themselves sequential programs. This architecture was developed for GPUs, wherein there are effectively hundreds or more small CPUs, all of which run the same code. To implement complex functions when there will be thousands of instances of a single kernel, there ends up being a large amount of control logic needed. This extra control code is used to locate the correct memory for the current instance to operate on, detect edge conditions and behave appropriately, and other application specific control code.

To map the OpenCL programming architecture to FPGAs (and to ensure code compatibility with GPUs), OpenCL uses the FPGAs fabric to implement what are effectively small, application optimized processor cores (Xilinx calls them “compute units”). The kernel code then executes on these cores and the system will operate effectively the same as it does on a GPU. This implementation basically amounts to implementing a very special purpose GPU inside the FPGA. This approach is something in between the ideas of Vivado HLS and lower-level FPGA languages, in that, the programmer is required to explicitly define the parallelism (no automatic parallelization like in HLS). To ease the programmer’s burden, however, the FPGA’s massive parallelism is reduced to the same level as GPUs. This underutilization of the FPGA may make it nicer for CPU programmers to move in the FPGA space, but the performance cost is large. In [49], the OpenCL environment is compared against 4 FPGA programming alternatives: LegUp, a C-based high-level synthesis language; Verilog, a traditional FPGA programming language; Bluespec Verilog, an extension of SystemVerilog that adds abstraction layers but stays low-level; and Chisel, a high-level language that constructs low-level FPGA systems. They implemented 4 different algorithms used in Database Management Systems and recorded the performance of each architecture and language relative to a theoretical solution. Their table of results is shown in Table 4.1. LegUp performed the worst, achieving at most only 6.22% the theoretical throughput. OpenCL performed generally bad as well. In one case it only reached 0.21% of the theoretical throughput. In the best cases it reached 50.7% and 79.7%. The three low-level languages, Verilog, Bluespec Verilog, and Chisel, all achieved at least 98.6% of the theoretical throughput in every test performed. At the end of the day, OpenCL and other high-level approaches take an architecture that is easy for CPU programmers, but fundamentally slow on FPGAs, and attempt to increase its performance through compiler optimizations. The alternative approach, taken by this thesis, is to use a low-level architecture that is fundamentally fast and efficient on FPGAs and attempt to make it easier to learn and use.

4.2.4 Chisel

Chisel [50] is a hardware construction language intended for developing FPGA and ASIC modules. It is a domain specific language written within the Scala programming language. The outputs of a Chisel software program are shown in Figure 4.1. Chisel produces entirely synthesizable, vendor independent Verilog outputs, so it can be used with existing

Table 4.1: Chisel is able to maintain a low-level quality of performance while still providing high-level abstractions and hardware construction interfaces. Table recreated from [49].

Implementation	Registers		Logic		ALM*	Memory (Kbits)	F _{max} (MHz)	Throughput		LoC
	FF	% incr.	LUT	% incr.				MB/s	%	
bitonic analytical	10 240	0.00%	2 560	0.00%						
bitonic Verilog	10 250	0.10%	2 640	3.13%	7 210.8	0.00	311.43	38 016.36	100.00%	134
bitonic BSV	10 250	0.10%	2 640	3.13%	6 997.5	0.00	313.97	38 326.42	100.82%	57
bitonic Chisel	10 272	0.31%	2 649	3.48%	5 571.0	0.00	314.96	38 447.27	101.13%	114
bitonic LegUp	4 210		5 180		3 973.4	0.00	211.86	1 034.47	2.72%	101
bitonic OpenCL	38 455		5 221		15 842.6	361.38	307.12	1 317.21	3.46%	140
STL sort C++ (host CPU)							2 300.00	570.42	1.5%	3
spatial analytical	2 048	0.00%	640	0.00%						
spatial Verilog	2 081	1.61%	641	0.16%	1 359.5	0.00	341.30	1 301.96	100.00%	98
spatial BSV	2 112	3.13%	1 701	165.75%	1 081.0	0.00	343.52	1 310.42	100.65%	181
spatial Chisel	2 112	3.13%	720	12.50%	1 053.0	0.00	345.30	1 317.21	101.17%	87
spatial LegUp	1 115		823		612.5	0.00	309.12	3.13	0.24%	28
spatial OpenCL	26 059		14 667		15 072.3	877.84	236.85	660.53	50.73%	66
median analytical	4 544	0.00%	2 240	0.00%						
median Verilog	4 555	0.24%	2 352	5.00%	4 009.5	0.00	302.76	1 154.94	100.00%	159
median BSV	4 554	0.22%	6 168	175.36%	3 359.5	0.00	334.67	1 276.66	110.54%	70
median Chisel	4 577	0.73%	2 351	4.96%	3 321.5	0.00	338.98	1 293.11	111.96%	132
median LegUp	10 449		5 262		3 781.4	0.00	174.98	34.25	2.97%	97
median OpenCL	19 366		7 590		9 309.5	190.06	312.10	920.60	79.71%	84
median C++ (host CPU)							2 300.00	836.10	72.39%	6
hash probe Verilog	995		174		327.5	3 136.00	174.06	995.98	100.00%	66
hash probe BSV	1 150		166		365.5	3 136.00	181.46	1 038.32	104.25%	124
hash probe Chisel	1 020		179		333.5	4 096.00	171.59	981.85	98.58%	83
hash probe LegUp	345		397		262.5	4 096.00	302.85	61.90	6.22%	50
hash probe OpenCL	35 536		21 854		19 175.6	3 876.08	270.19	2.14	0.21%	59
hash probe C++ (host CPU)							2 300.00	433.32	43.51%	18

*ALM (Adaptive Logic Module): Altera's basic cell blocks, with an 8-input fracturable LUT and four 1-bit registers.

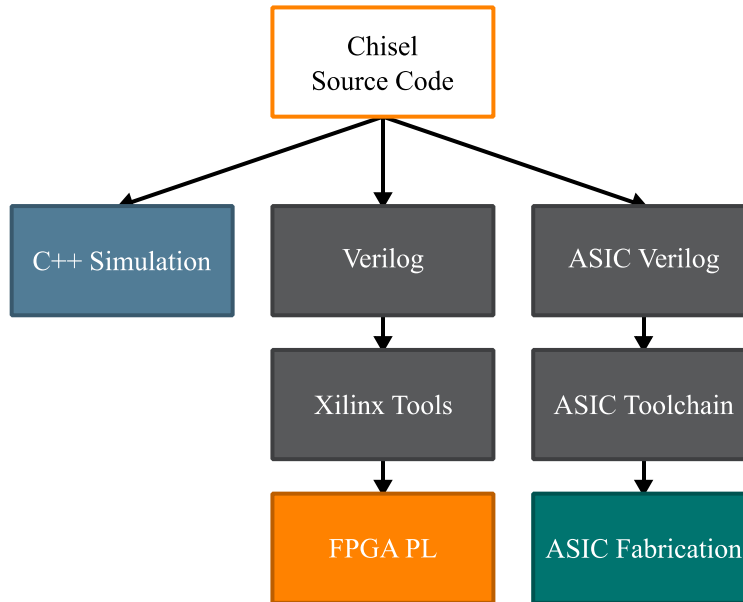


Figure 4.1: Chisel outputs C++ for use in simulations and Verilog for implementation in either FPGAs or ASICs.

synthesizers. Because Chisel is embedded within a high-level language, the high-level features can be used to construct the low-level hardware designs in very powerful ways. Effectively, Chisel is a functional programming language, and programs written in Chisel generate hardware. Therefore, abstractions can be made in the Scala language, and since the Chisel code itself is low-level, the same low-level capabilities of Verilog or VHDL can be achieved. Because Chisel code is a step before Verilog outputs, parameterization is very easy and powerful. The features of Scala can be leveraged to build highly customizable descriptions of hardware that result in optimized Verilog that is easy for the synthesizer to parse and optimize. Another major benefit of Chisel is simulation speed. Chisel guarantees cycle-accurate simulations that run significantly faster than standard simulations. This reduces the development time of Chisel modules significantly. Even better, the tests are written in Scala so they can be very advanced, high-level tests that verify correctness at a behavioral level as well as a waveform level. This can be very difficult to accomplish with the traditional HDLs.

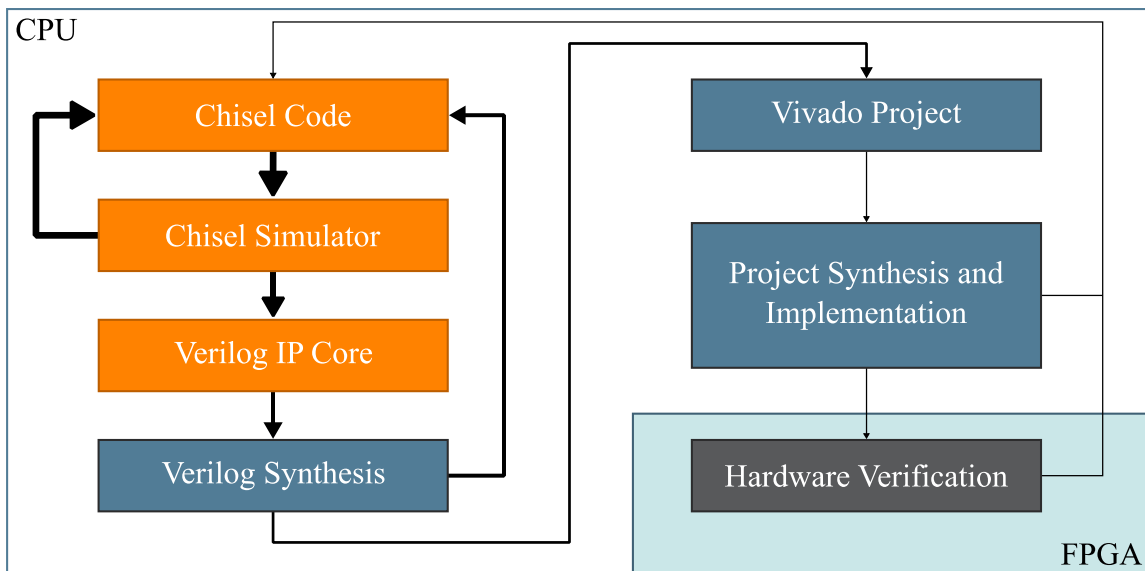


Figure 4.2: The process of writing FPGA firmware in Chisel. The thicker arrows indicate a quicker path. The Chisel simulator provides a very wide, easy path to quickly develop a very good module before proceeding to the more time consuming parts of the development process.

4.2.5 Recommendations

There have been a plethora of alternative languages over the years that have seen various levels of support and adoption. Wikipedia alone mentions over 30. However, the current state is that Verilog and VHDL, the oldest of these languages, are still the major players and have survived the test of time after a multitude of competitors. In my view, their continued dominance is mostly due to the existing codebase and software tools, continued vendor support, and minimal abstraction. The sustained effort for many years to develop a new programming language for FPGAs shows a dissatisfaction with the currently available languages.

Since they are the oldest, Verilog and VHDL have by far the majority of FPGA code developed. Given this history and the experienced hardware designers that use these languages, it is not easy to suddenly change technology. Without a clearly superior alternative there has been no sufficient motivation to change. In addition, the software tools for development and verification in Verilog and VHDL are very mature, trusted, and actively supported. Most alternative languages that have been developed have either not had these tools at all, or have faded to obscurity before the tools reached stability. Finally, much like trying to push Python or another very high-level language on a seasoned assembly language programmer, many languages simply abstract too much and obscure the benefits of an FPGA in the first place. Verilog and VHDL provide access to the lowest level of abstraction, and with it, the most freedom and possibility for performance. In my opinion, a language that can gain traction in the world of Verilog and VHDL will still need to be low-level (or at least provide access to this level), it will need to interoperate with Verilog and VHDL software tools, and it will need to provide a significant advantage over Verilog or VHDL. Chisel has put itself in a good position to meet these needs, and with sufficient development it could gain serious traction. Chisel provides access to the lowest levels of the design, but provides high-level features for constructing and testing these designs. This increases code reuse, behavioral understanding, and programmer efficiency. This is why Chisel was chosen as the development language for the image processing library.

4.3 Chisel Video Processing Library

We have developed a library of image processing functions for FPGAs written in the Chisel hardware construction language. The purpose of this library is to enable quick development of image and video processing systems for FPGAs. FPGAs can often offer large performance benefits over CPUs or GPUs for image processing, but they require so much time and effort to program that they are not used as often as they could be. The purpose of this library is to provide a starting point for using FPGAs in image processing and reduce the entry barrier to these devices. In many cases this library could completely remove the development time of an FPGA-based system, if the functions are already included. Otherwise, the library provides facilities for creating new functions as easily as possible and integrating them into the FPGA. In the remainder of the chapter we first, identify 6 factors that will influence the library’s reusability; second, present the organization of the library’s contents; and third, describe the functions that are currently available in the library.

When developing source code libraries for FPGAs, there are many factors to consider to ensure that the code can be reused. Specifically, [51] has identified 6 key factors: complete documentation, problem generality, design built using good development practices, comprehensive testing, compatibility with existing software, and usability with standard interfaces. The first factor, comprehensive documentation, is always a good requirement regardless of the programming language or environment. Chisel, or any other language, cannot do much to help with this. The second factor is the generality of the problem being solved. If the code is too specific then it will not necessarily fit the next situation. This is an issue mostly handled when the module is designed, but because Chisel adds an additional layer between the original code and the synthesizable hardware representation, it adds the ability to include specific optimizations for niche applications while maintaining its ability to generalize. Because Chisel constructs hardware descriptions from a high-level language, a generalized Chisel module can be built to construct specific hardware. The third factor, building designs with good development practices, is again not language specific, so Chisel can handle this as well as any other language. The fourth factor, testing, is improved by Chisel’s fast and cycle-accurate simulator written in Scala. Adding a high-level testing interface before the low-level implementation allows many test procedures to be easily developed at the behavioral level to catch errors before formal, low-level

verification. It also enable very quick iteration when a design is being developed. This leads to faster convergence to a correct design. The fifth factor, compatibility with existing software, comes for free with Chisel because the output of Chisel is fully synthesizable Verilog. All the existing software written to simulate and synthesize Verilog code is still fully usable. The sixth and final factor, usability with standard interfaces, is also improved with Chisel. Because Chisel programs represent the construction of hardware rather than describing a specific hardware module, it is easy to integrate with any bus or interface. The internal module can be developed minimally, and a parameter in the Chisel code can select an arbitrary bus to be wrapped around the module when outputting the hardware representation. Overall, Chisel has many features that make it possible to develop highly reusable FPGA designs.

The library is organized into three separate projects. The first project, “Utilities”, contains general functions that are not imaging specific. These should be available in a separate library already, but since they are not they are included here. Utilities include modules for pipelined multiplication, addition, subtraction, division, and multiplication; AXI Lite and AXI Stream bus interfaces; and others. The second project, “Chisel Image Processing Primitives” or “ChIPPs”, is the main library that implements the image processing functions. This is the most important part of the library, and provides a base for developing finished image processing systems. The third project, “IP Cores”, provides reference implementations of several functions built specifically for the AMBA bus architecture. These show the power of the Chisel implementation, in that, they are easily able to combine modules from the ChIPPs and Utilities projects to develop larger functions that utilize a standard bus architecture.

Figure 4.3 shows how these three projects interact with each other. The Utilities are the most basic modules; they never contain either ChIPPs or IP Cores. The ChIPP modules often use Utilities internally, but never contain IP Cores. The modules in the IP Cores project are the top-level modules. Each IP Core contains one or more modules from the ChIPPs or Utilities projects to develop a complete implementation of a processing function. The IP Cores are also designed to interoperate with the AMBA bus architecture used by the Xilinx Zynq and the modules that are available in the Xilinx Vivado software. The IP Cores are the modules that will be used to generate the final Verilog code that is used in Vivado to develop video processing systems.

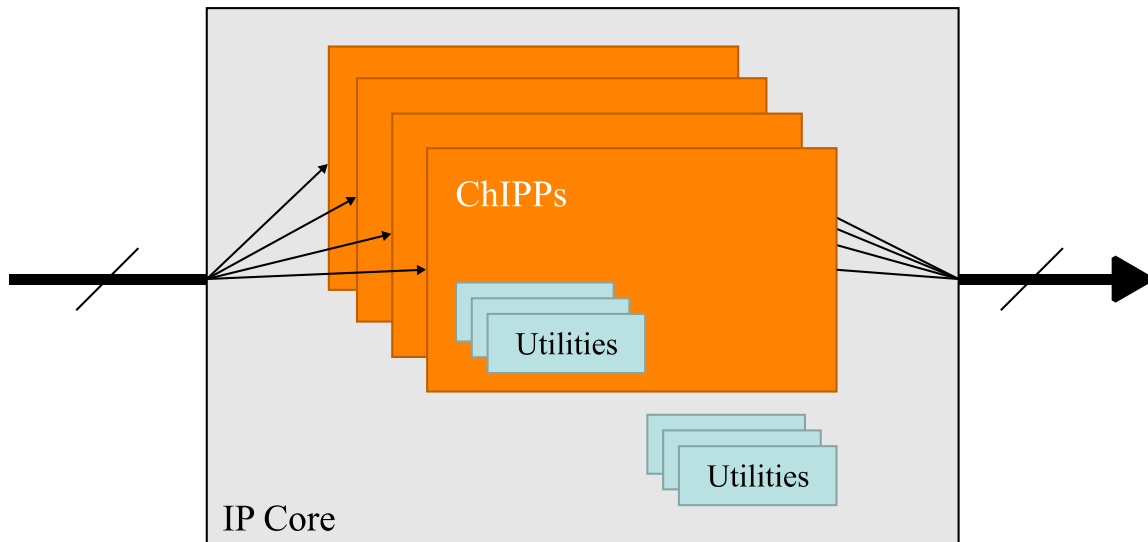


Figure 4.3: The firmware library uses three separate projects: IP Cores, ChIPPs, and Utilities. The Utilities are used in both ChIPPs and IP Cores. The ChIPP modules are image processing primitives that are combined to build IP Cores to perform a specific processing operation.

4.3.1 Utilities

These utilities implement general functions that are used in the image processing cores. Currently, there are 8 FPGA modules:

- Adder
- Divider
- Multiplier
- Saturate
- Wrap
- AXILite
- AXIVideoStream
- ShiftRegisterGood

and 2 Scala classes:

- FixedPoint
- ImageTester

Adder

The adder module implements pipelined addition of an arbitrary number of addends. Due to the useful properties of 2s-compliment, it can also perform subtraction.

The module is configured with only two parameters: the number of addends and the maximum number of combinational additions. These parameters are used to build a pipeline that resembles an upside down tree. An example of the pipeline is shown in Figure 4.4. In this example there are 9 addends and a maximum of 2 combinational additions. The gray circles are registers, therefore, this particular arrangement has a latency of 4 cycles. In general, the latency is $\lceil \log_c(n) \rceil$ where n is the number of addends and c is the maximum number of combinational additions.

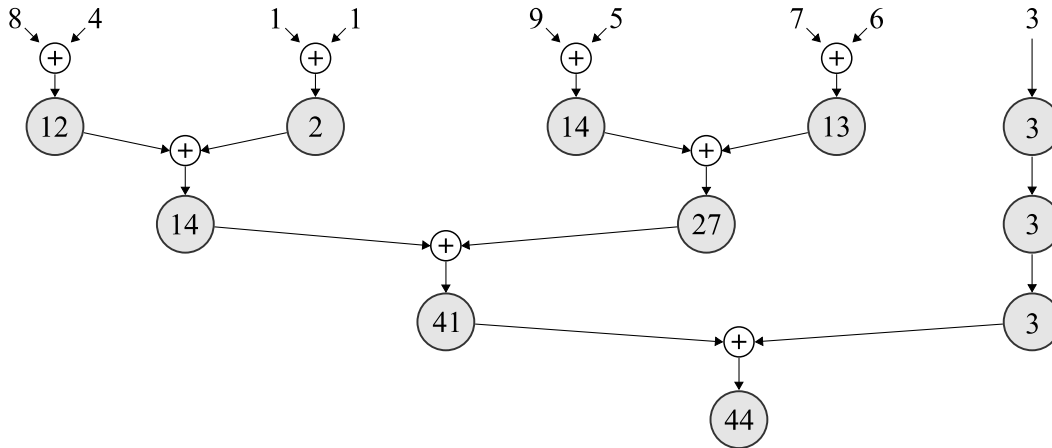


Figure 4.4: The adder tree created by the Adder module adding 9 addends, 2 at a time. The latency of the input to the answer is 4 cycles.

This module is necessary when adding several numbers together because many additions can quickly become a bottleneck in an image processing system. The FPGA synthesis tools can automatically build combinational adders when they see the “+” symbol, but adding more than two numbers combinational will reduce the maximum operating frequency of the circuit. To increase the frequency, then, the additions must either be treated as a multi-cycle path, which can be complicated to deal with, or the additions can be pipelined. Pipelining is a tradeoff between latency and throughput, as adding pipeline stages will

increase the number of clock cycles that are required to complete the addition, but it will also increase the maximum operating frequency. Because video processing systems require a relatively high clock frequency to maintain a high frame rates and resolution, pipelining is necessary in most cases. Figure 4.5 shows the maximum operating frequency of the Adder core with 9, 32-bit while varying the maximum number of combinational additions. The results were obtained by implementing the Adder module by itself on the Zynq 7020 part, -2 speed grade. The input clock speed was increased until the implementation failed to meet timing constraints. The highest clock speed that results in a successful implementation is considered the maximum clock frequency of the module. Clearly, even with the entire fabric available for operation, the fully pipelined Adder can operate with ~2.5 times the clock frequency of the Adder with even 5 combinational additions. With an almost full FPGA in a complex design the combinational additions would perform even worse, and make it difficult for the design to meet all timing constraints.

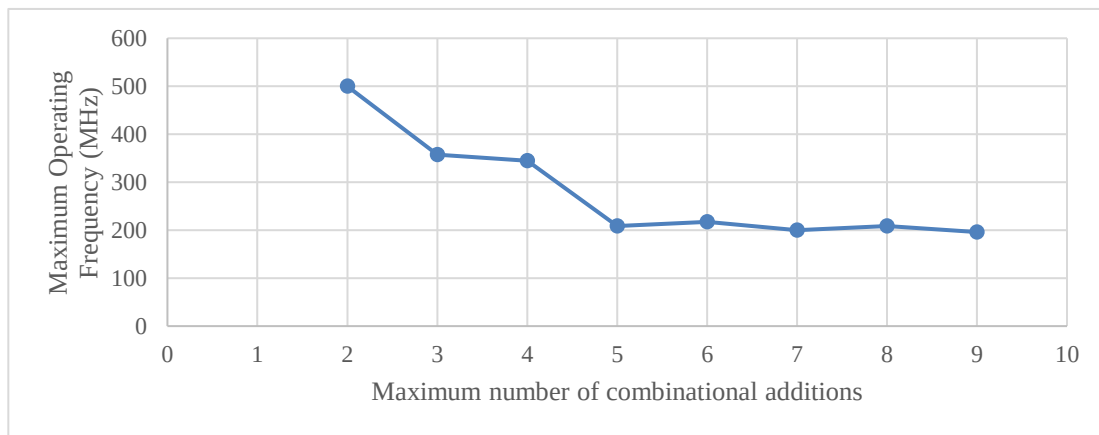


Figure 4.5: Maximum operating frequency of the Adder module with varying number of combinational additions.

Divider

Division is the slowest and most area inefficient of the four basic mathematical operations. In general, it is best to be avoided, but sometimes that is not possible. This divider module implements pipelined binary long division signed integers and is parameterized with the bit depth of the input integers. The inputs to the module are the dividend and divisor, and

the outputs are the quotient and remainder. The latency of this core is the bit width of the inputs plus 1 clock cycle. The algorithm used is simple binary long division. This algorithm is rather simple, but it is easily pipelined and has a fixed latency from input to answer. Pipelining is necessary to achieve real-time frame rates, so the latency is unavoidable. Fixed latency simplifies all the logic that goes around this divider. If the latency was variable, then matching timing with other modules, or with auxiliary data within the same module becomes much more difficult. Figure 4.6 shows the flowchart of the operation of the division module.

Future work for this module is to implement other different division algorithms. Each algorithm has its advantages and disadvantages, and the choice and configurability to make modules suit any particular need is what makes a Chisel-based library so appealing.

Multiplier

The multiplier module rounds out the basic math operations. It operates on two 16 bit signed integers and produces a 32-bit signed product. The algorithm is pipelined binary partial product multiplication. The latency of the module is 8 cycles.

Future work for this modules is to implement additional algorithms and use the vendor specific hardware multipliers. Chisel currently does not have very good facilities for using vendor specific hardware or external Verilog models in simulations, so this feature will need to wait until Chisel adds these features.

Saturate and Wrap

These are simple modules that scale numbers so that they fit within a specified bit width by either saturating or wrapping. Both modules assume that the inputs are signed. The Saturate module is configured with the input bit width and scale (used for fixed point numbers) and the output bit width. If the input number is negative (the most significant bit is 1), then the output of the module is 0, if the input is larger than the output bit width can hold, then the output is the largest positive number the output can hold (a single 0 followed by all 1's). The Wrap module is configured with the input bit width and scale, the output bit width, and also the maximum output number. In this case, if the input is negative, then the output is the input plus the maximum number. Note that this only happens once, so if the input is less than twice the maximum number then the output will still be negative. Conversely if the input is larger than the maximum, the output is the input minus the

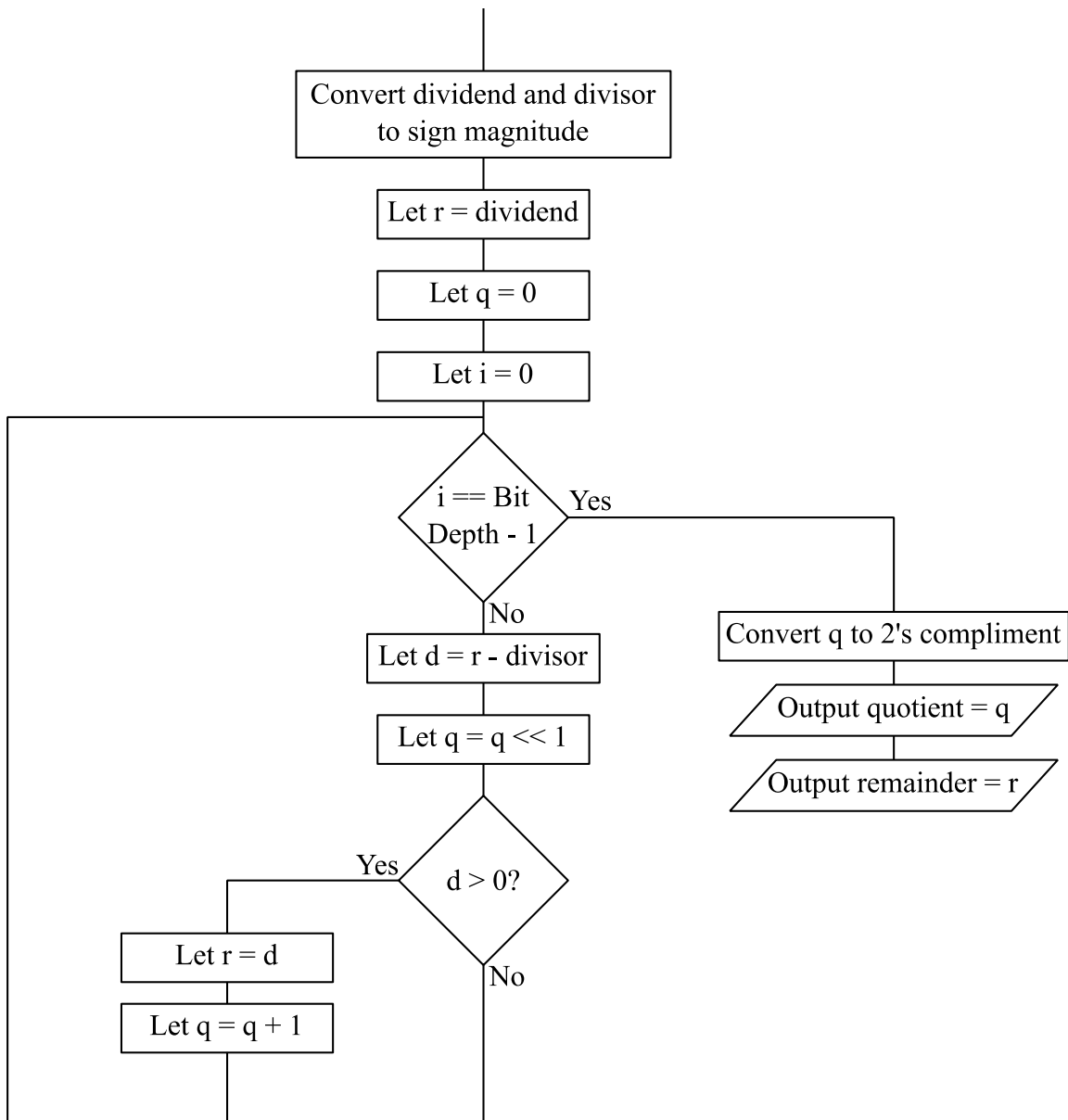


Figure 4.6: Flowchart of the Division module's implementation of binary long division.

maximum. In this case also there is only one subtraction, so if the input is larger than twice the maximum then the output will still be larger than the max. If it is possible that the input will be outside the range $[-\text{max}, 2*\text{max}]$, then multiple wrap modules can be chained together till the output is stable.

AXILite and AXIVideoStream

These modules are used for interacting with the AXI4 bus interface used in the Xilinx FPGAs.

The AXILite module implements the AXI4-Lite protocol and is used for presenting a memory mapped register interface to the processor. These registers can then be used to configure parameters of the image processing functions on the fly.

The AXIVideoStream module implements the AXI4 Stream protocol in the manner specified by Xilinx to be used for video streams in [52]. The video-based changes to the protocol define how to pack the pixel on the data bus, how to use AXI Stream protocol's 'tlast' signal as an end-of-line marker for a video frame, and how to use the 'tuser' signal as a start-of-frame marker. Additionally, Xilinx makes several suggestions for developing video stream modules and those suggestions are followed as much as possible. For example, Xilinx suggests using registers on all the input and output signals [53]. This induces some special considerations because registering valid and ready signals causes a delay between the signal changing and the signal being received. To account for this, the AXIVideoStream module includes pixel buffers that automatically save and resend an output pixel when the downstream module becomes not ready, or automatically store valid input pixels sent when the current module becomes not ready. The datapath of the input version of the AXIVideoStream is diagramed in Figure 4.7. The "First Backup" and the "Second Backup" registers are used when the video stream is started or stopped. When the sink's ready signal is deasserted, the source will not see the new signal until the next clock cycle, so it may send another valid input. This valid input cannot be sent to the sink because the sink is not ready, so it is stored in these backup registers until the sink asserts its ready signal. Once the ready signal is asserted, the backed up data is sent to the sink in the single cycle lag time before the source sees the newly asserted ready signal.

ShiftRegisterEnable

This module is a simple shift register with an enable flag. The tag “Enable” is appended because Chisel actually already provides a shift register with an enable signal, but the enable signal is actually a valid signal (sometimes called “shift in” or “input enable” or “strobe”) rather than an entire module enable. A true enable flag will disable the entire module, both input and output. In Chisel’s built-in ShiftRegister module, however, deasserting the enable flag only disables the input and the module will still shift out its values. There is then no way to stop the output. The layout of Chisel’s built-in ShiftRegister module is shown in Figure 4.8. In the ShiftRegisterEnable module, the enable signal disables both input and shifting, so that the data inside the shift register is maintained until the enable flag is asserted. This layout is shown in Figure 4.9.

FixedPoint

The FixedPoint class helps in using fixed point numbers in both hardware circuits and simulation by providing methods for creating, modifying, parsing, and viewing fixed point numbers. The constructors for the class convert numbers in other types to fixed point notation with an explicit bit width and scale. The numbers can then be displayed, modified, or represented as integers and passed to the hardware modules. This enables the use of fixed point numbers. The class also implements the standard math and logical operators that would be expected from a numerical type. The mathematical operations return a new fixed point number that has the scale adjusted properly. Therefore, the scale of the output may not be the scale of the input. Addition and subtraction return a number with the width and scale being the maximum width and maximum scale of the input numbers. Multiplication returns a number that has the same width, but the scale is the sum of the input scales. Division returns a number that has the same width, and the scale is the scale of the dividend less the scale of the divisor.

ImageTester

The image tester class is a class used for testing image processing algorithms that use the AXI bus interface. The class provides functions for reading and writing registers on an AXI-Lite interface, setting the input and output images, and packing and extracting the channels from a pixel bus. Most importantly, however, is the function used to actually send an image to the module being tested and generate an image from the output of the module.

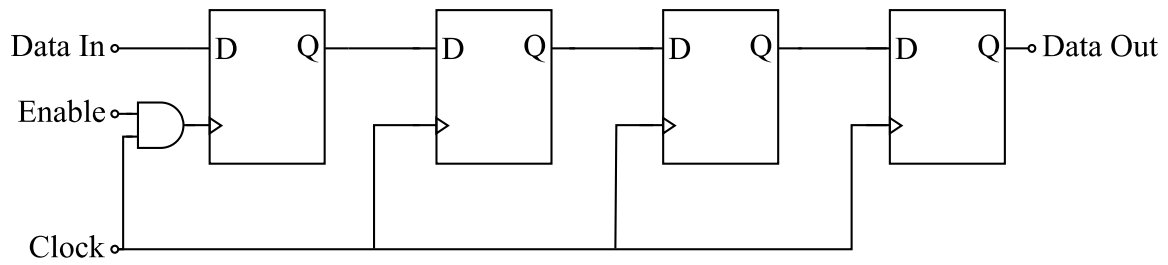


Figure 4.8: Chisel's built-in *ShiftRegister* module has the enable signal only for the input data.

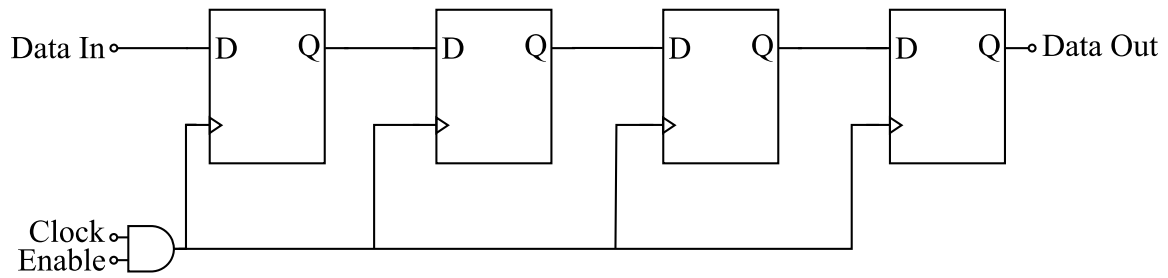


Figure 4.9: This library's *ShiftRegisterGood* module uses the enable signal for all the stages.

When the test is run, the tester streams the input image to the module being tested over the module's AXI4-Stream input interface. The valid, end of line, and start of frame signals are all generated, and the module's ready signal is respected. Simultaneously, the AXI4-Stream output of the module is monitored for valid output data and, upon receiving output data, it is formatted back into an image. Additionally, the input valid signal and the output ready signal are toggled randomly, to simulate unknown conditions surrounding the module as may be seen in a real system. Often, this is the part of the test that causes the most trouble, because properly stopping and starting in the middle of an image can often cause issues with the pipeline stages.

4.3.2 Chisel Image Processing Primitives (ChIPPs)

The ChIPPs project is the main portion of this work; it contains the modules that implement the image processing primitives. There are currently 12 modules in this project:

- AddS
- MulS
- DivS
- RGB2YCbCr
- YCbCr2RGB
- RGB2Gray
- Gray2RGB
- RGB2HSV
- HSV2RGB
- LUT
- Window
- HistogramEqualization

Scalar Math Operations

The first three modules – AddS, MulS, and DivS – implement the basic 4 math operations on input images (AddS also performs subtraction). Each module has inputs for 3 scalars and applies the operation to each channel of the input image. The scalars in the AddS module are signed fixed point numbers that have 12 integer bits and 12 fractional bits (1.12.12 format). The scalars in the MulS module are in the 1.5.10 signed fixed point

format. The scalars in the DivS are only signed integers. In each case, the output of the mathematical operations are saturated to 8 bits. Figure 4.10 shows an example input and output of these functions.

RGB and YCbCr Color Conversions

A very common color space in the transmission of video signals is YCbCr (or YPbPr). The RGB2YCbCr and YCbCr2RGB modules implement the forward and backward conversions of RGB to YCbCr using the ITU-R BT.709 standard adjusted to use the entire 8-bits. The inputs are all 8 bit and the output are 8 bit, but in the future this will be extended to arbitrary bit widths.

As far as color conversions go, the conversions to and from RGB and YCbCr are relatively simple. In each case, each channel is a linear combination of the components from each space. From the ITU's BT.709-6 recommendation [54], the forward conversions are:

$$\begin{aligned}
 Y &= 0.2126 \cdot R + 0.7152 \cdot G + 0.0722 \cdot B \\
 Cb &= \text{round} \left(\left(-\frac{0.2126}{1.8556} \cdot R - \frac{0.7152}{1.8556} \cdot G + \frac{0.9278}{1.8556} \cdot B \right) \cdot \frac{224}{219} + 128 \right) \\
 Cr &= \text{round} \left(\left(\frac{0.7874}{1.5748} \cdot R - \frac{0.7152}{1.5748} \cdot G - \frac{0.0722}{1.5748} \cdot B \right) \cdot \frac{224}{219} + 128 \right)
 \end{aligned}$$

In this case the R, G, and B values have been quantized between 16 and 235, therefore they do not use the entire range of 8 bits. This conversion module uses the following formulae:

$$\begin{aligned}
 Y &= \frac{54.4256}{256} \cdot R + \frac{183.0912}{256} \cdot G + \frac{18.4832}{256} \cdot B \\
 Cb &= -\frac{29.3305}{256} \cdot R - \frac{98.6695}{256} \cdot G + \frac{128}{256} \cdot B + 128 \\
 Cr &= \frac{128}{256} \cdot R - \frac{116.2631}{256} \cdot G - \frac{11.7369}{256} \cdot B + 128
 \end{aligned}$$

In this case the fractional multipliers are approximately the same, but the conversion factor of 224/219 has been removed because the R, G, B, Y, Cb, and Cr values are all in the range [0, 255].

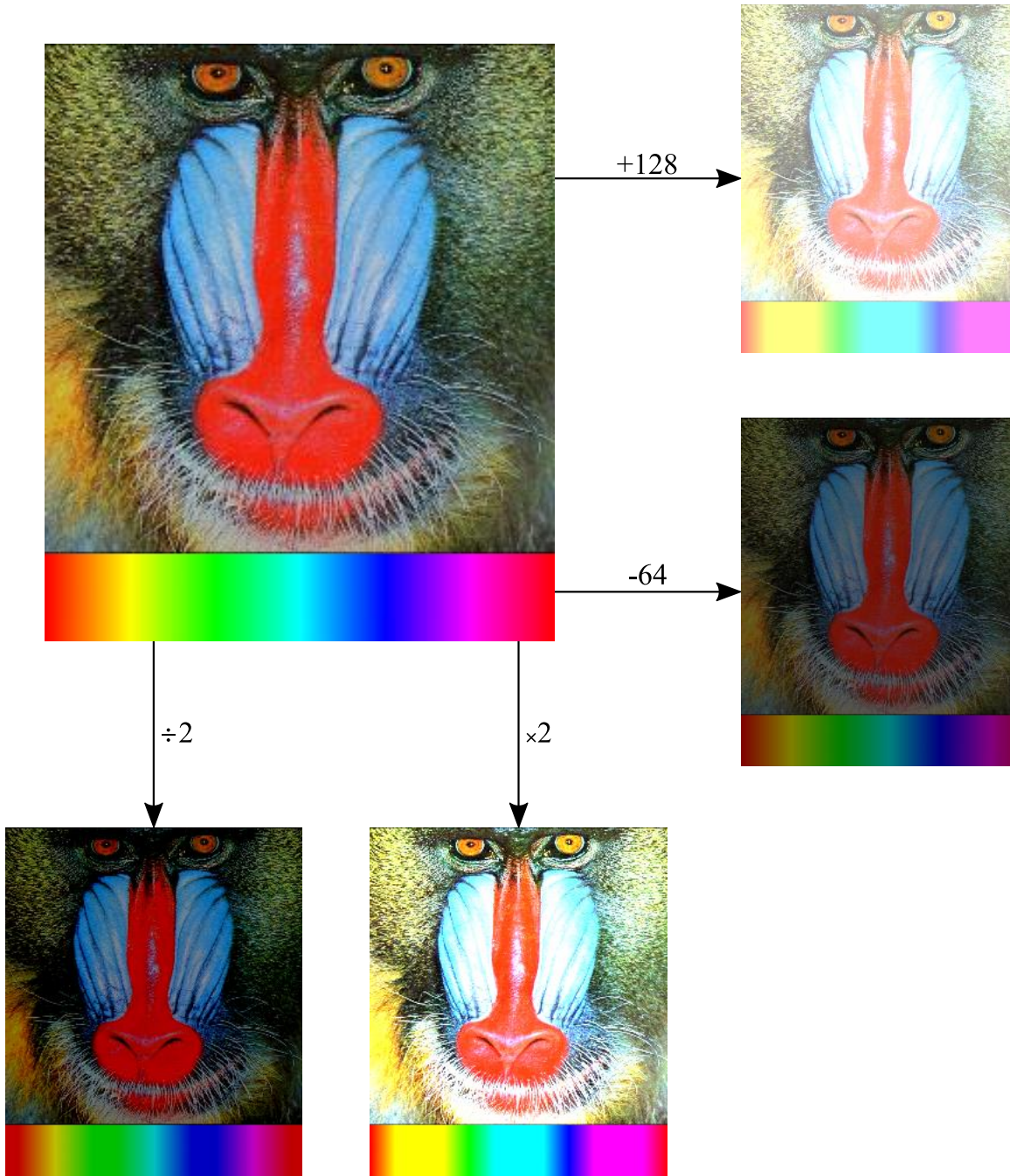


Figure 4.10: Example results of using the scalar math modules. They accept arbitrary constant and apply their operation to the image. The addition and subtraction are both accomplished by the AddS module because it accepts signed integers. The multiplication and division outputs were created by the MulS and DivS modules, respectively.

The standards document does not explicitly state the backward transformation functions, but they can be derived from the forward transformations, and this module implements them with the following equations:

$$\begin{aligned} R &= Y + \frac{403.1488}{256} \cdot Cr - 201.5744 \\ G &= Y - \frac{47.9550}{256} \cdot Cb - \frac{119.8398}{256} \cdot Cr + 83.8974 \\ B &= Y + \frac{475.0336}{256} \cdot Cb - 237.5168 \end{aligned}$$

These equations also assume that Y, Cb, and Cr are in the range [0, 255], and the output R, G, and B are also in the range [0, 255].

Future work for this module is to support compressed range values, arbitrary bit widths, and the multiple standards that exist defining various constants for the conversion factors.

RGB and Grayscale Color Conversions

The RGB to grayscale conversion uses the ITU-R Recommendation BT.601 standard's recommendation for the luminance channel [55]. The conversion is therefore:

$$I = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

Similar to the YCbCr conversions, future work for this module is to support arbitrary bit widths and multiple conversion standards.

RGB and HSV Color Conversions

The forward and backward transformations between RGB and HSV are much more complex than the other conversions. Additionally, there are several different definitions of very similar spaces. HSV, HSB, HSI, and HSL are all very similarly defined. Though there are two names, HSV and HSB are, in fact, the same, and that is the space that is implemented in the RGB2HSV and HSV2RGB modules.

For the forward conversion from RGB to HSV the following formulae are used:

$$\begin{aligned} M &= \max(R, G, B) \\ m &= \min(R, G, B) \end{aligned}$$

Then, to calculate the hue channel, two intermediate values need to be found

$$h_{difference} = \begin{cases} G - B & , & M = R \\ B - R & , & M = G \\ R - G & , & M = B \end{cases}$$

$$h_{offset} = \begin{cases} 360 & , & M = R \text{ \& } B > G \\ 0 & , & M = R \text{ \& } G \geq B \\ 120 & , & M = G \\ 240 & , & M = B \end{cases}$$

Finally, the components can be defined as:

$$H = \frac{60 \cdot h_{difference}}{M - m} + h_{offset}$$

$$S = \frac{M - m}{M} \cdot 255$$

$$V = M$$

The outputs of the module use 8 bits for the saturation and value channels, and 9 bits for the hue. This is because the saturation and value channels are in the range [0,255], while the hue channel is in [0,360], and 8 bits would require halving or rescaling the hue value and losing information so that it would fit.

For the backward conversion, from HSV back to RGB, the operation is quite different. First, define several intermediate values as

$$C = S \cdot V$$

$$X = C \cdot (1 - |(H/60) \bmod 2 - 1|)$$

$$m = V - C$$

Then, the output R, G, and B depend on what range the hue is in

$$(R', G', B') = \begin{cases} (C, X, 0) & , & 0 \leq H < 60 \\ (X, C, 0) & , & 60 \leq H < 120 \\ (0, C, X) & , & 120 \leq H < 180 \\ (0, X, C) & , & 180 \leq H < 240 \\ (X, 0, C) & , & 240 \leq H < 300 \\ (C, 0, X) & , & 300 \leq H \leq 360 \end{cases}$$

Then the final output values become

$$(R, G, B) = (R', G', B') + m$$

Lookup Table

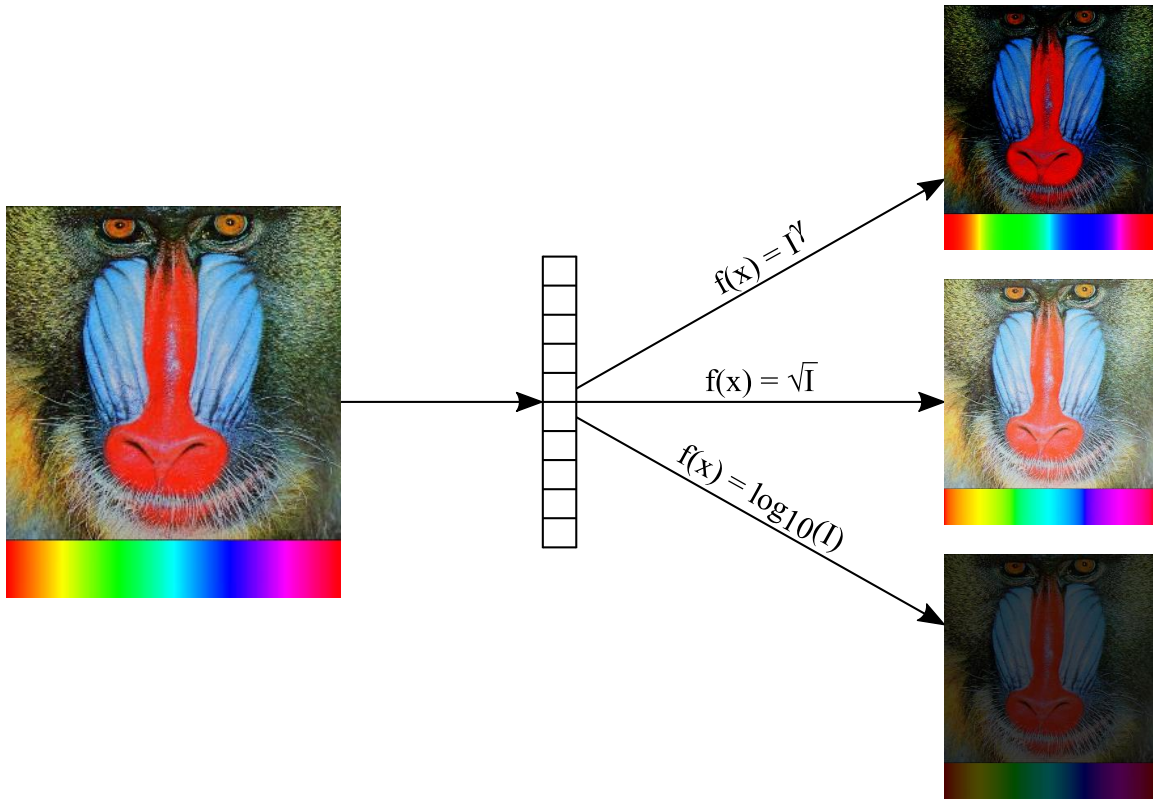
The lookup table module is extremely useful because it can be used to implement any pixel-based, per-channel operation, and it can be reprogrammed on the fly. The module uses 2 sets of 3 lookup tables. The lookup tables contain 256, 8 bit values. When an input pixel arrives, it is used to index into the lookup table, and the output is the value returned. Figure 4.11 shows example uses of lookup tables.

There are 2 sets so that, if the function needs to be changed, the lookup tables can be reprogrammed without affecting the currently running process, then switched to instantly change the lookup table. Without 2 sets, changing the lookup table would have to occur on the tables that are actively being used. This would result in partially updated tables being used at some point. As well, if there are only two functions to perform, they can both be loaded in the lookup tables at the same time and switched in a single clock cycle.

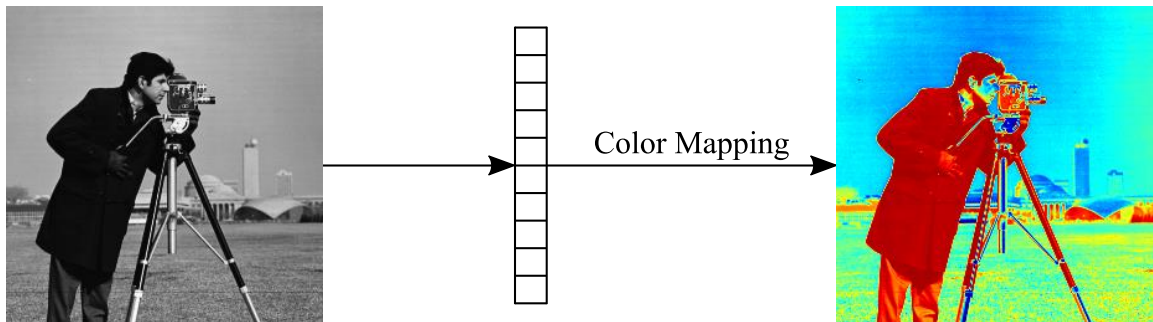
Using a lookup table trades memory usage for computational complexity. They are used to implement complex functions that would require a large number of resources such as n th roots, exponentials, logarithms, etc. Lookup tables work especially well when the domain of the operation is small, as is the case for image with 8-bit channels. The lookup tables are generated in the CPU by calculating the answer for a given operation or set of operations. The answers are then transferred to the lookup tables in the hardware using memory mapped registers. This way, the hardware does not need to waste space implementing the complex operation, and instead simply stores that answer for every possible input.

Window

The Window module inputs an image and outputs an $n \times n$ region of pixels. This is used to implement region based operations such as kernel convolutions, average or median filters, morphological operations like dilation and erosion, and many others. The module is configured with the size of the window to output, and the maximum expected width of a single line of the input image. The maximum line width is used to size the FIFO line buffers used in the core to hold values while it builds the region. The outputs of the module are the



(a): Lookup tables can be used to perform complex mathematical operations by pre-calculating the answer for each input and saving the result in a table. These operations require significant effort to compute in an FPGA, but when using a lookup table set by the CPU the operation is simple.



(b): Color mapping is performed by using the single input channel as three separate channels with a different lookup table on each channel to create the color.

Figure 4.11: Examples of using the lookup table module.

$n * m$ pixels in the window, and a signal called 'all_valid' that indicates whether all of the output pixels are actually from the image (i.e., this is not a boundary). The 'all_valid' signal is used to implement various type of border handling operation like zero padding, pass through, or edge pixel duplication.

Because the image is streamed into the module one pixel at a time, the module must save all of the pixels that it needs to build the window. This means storing $n - 1$ rows of the image in memory inside the FPGA fabric. For large images this can be relatively large, and will require Block RAM resources. To build the window, the input pixels are first streamed into an n pixel wide shift register. This shift register holds the bottom line of the window. The output of the shift register is put into a line buffer. Once an entire line of the image has been input, the line buffer starts sending data into another n pixel shift register. The pixels in this shift register are in the second to bottom line of the window. This process continues until the end of the image. This data path is shown in Figure 4.12. Once all of the image's pixels are streamed in, the shift registers and line buffers continue to move data through the window even though there is no input. This causes the module to deassert its input ready signal until all of the image's pixels have made their way through the module (i.e. all the line buffers and shift registers are empty). Once the image has finished, the input ready signal is asserted and the next frame of the image can be streamed in.

Histogram Equalization

The histogram equalization module inputs a single channel image, calculates the equalization lookup table, and applies it to the input. The operations of histogram equalization requires reading the entire image twice. Therefore, frame buffering in the middle of the operation is required. This module simply uses the transformation generated by the previous input on the frame being currently input as it calculates the new transformation from the current frame. With this setup, to equalize a single image it must be sent to the core twice. In a video stream, every frame must be sent twice, otherwise the histogram used for the transformation will lag by one frame. If the video frame rate is 30 or 60 frames per second, this lag is generally acceptable because the frame does not change much in only $1/60^{\text{th}}$ of a second. In video applications then, each frame is usually only sent once and transformed by the histogram of the previous frame. If it is not acceptable, then this module simply forces external frame buffering to be performed.

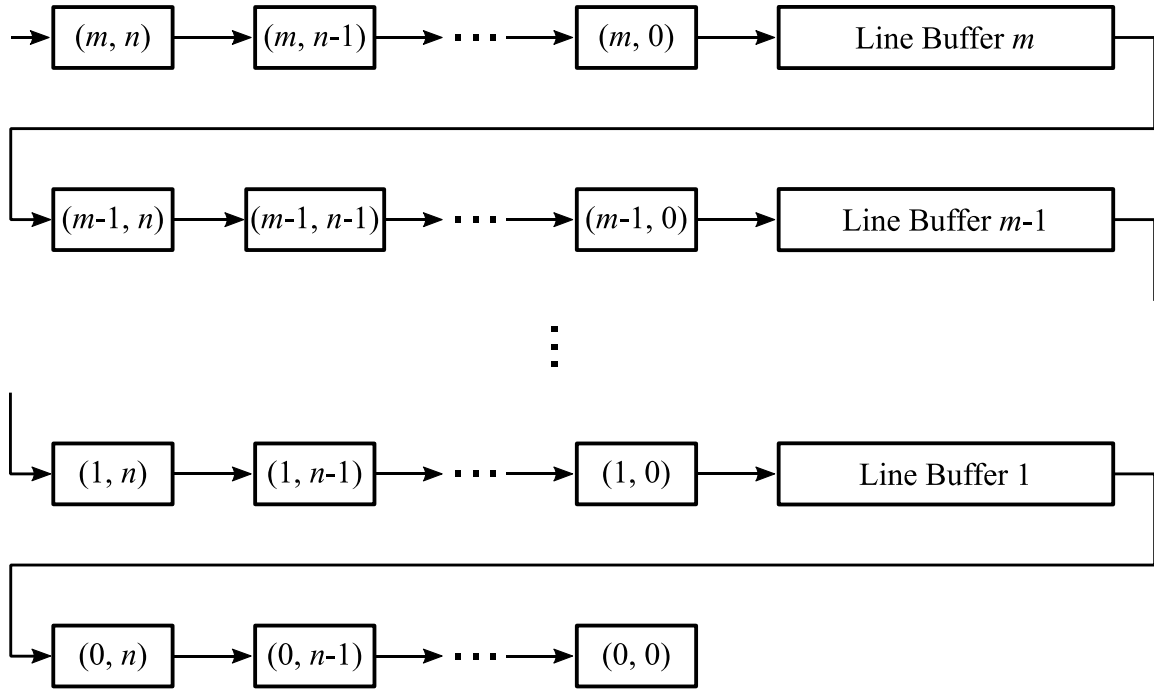


Figure 4.12: Data path for the Window module. The pixels are streamed in and registered. Line buffers are used to save each line of pixels until there are enough to fill the entire window. The pixels are streamed in sequentially, with the first pixel corresponding to the top left pixel, therefore the last pixel in memory is the top left pixel in the window.

The flowchart of the operation of the histogram equalization module is shown in Figure 4.13, and example inputs and outputs are shown in Figure 4.14. There are two states of operation, either it is running and processing inputs, or it is not. When running, the input ready signal is asserted, the input value is transformed using a lookup table, and the histogram is generated. Once the entire image has been streamed through the module, the module enters the second mode of operations where the input is not ready so that the new lookup table can be calculated. To calculate the new transformation lookup table, the histogram is accumulated and normalized by the size of the input image. The normalized values are used to update the lookup table, and once all 255 values have been updated the module returns to the running mode of operation.

Histogram equalization is a single-channel operation. It can also be extended to color images but the problem is ill-posed and has several possible solutions. The modules in ChIPPs are intended to be basic processing primitives, so this module is only a single channel implementation. For color images, two IP Cores have been developed that use this primitive to build a color-based system. These modules are discussed in the next section.

4.3.3 IP Cores

The IP Cores project contains implementations of image processing functions built to work in the Xilinx Zynq. Most specifically, they are built to use the AXI Video Stream interface for the video input and output, the AXI4-Lite interface for memory mapped registers in the processor, and interoperate with Xilinx's other video processing IP cores made available within their tools. These modules are the final outputs of the image processing library. These are the modules that are converted to Verilog and used to build an IP core using Xilinx Vivado that is included within a block design to build up the entire video processing system. There are currently 17 IP cores included in the library:

- Passthrough
- ScalarAddition
- ScalarAdditionYCbCr
- ScalarAdditionHSV
- ScalarMultiplication
- ScalarDivision

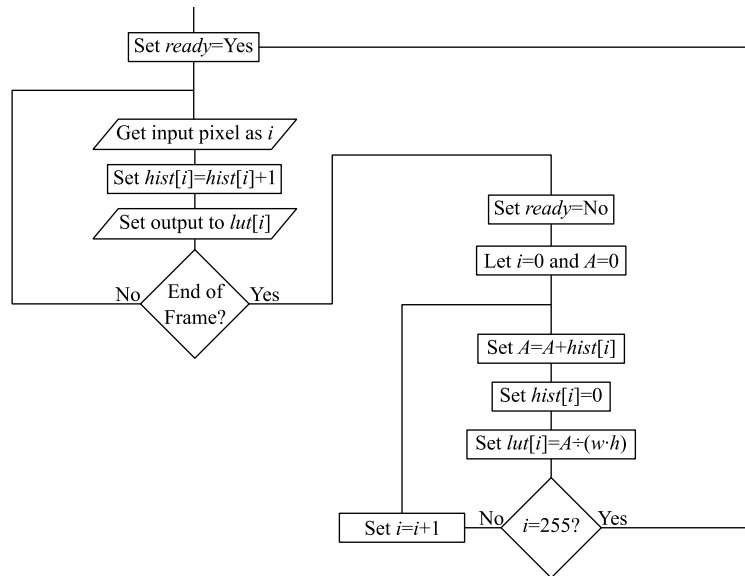


Figure 4.13: Flowchart of the operation of the Histogram Equalization module.

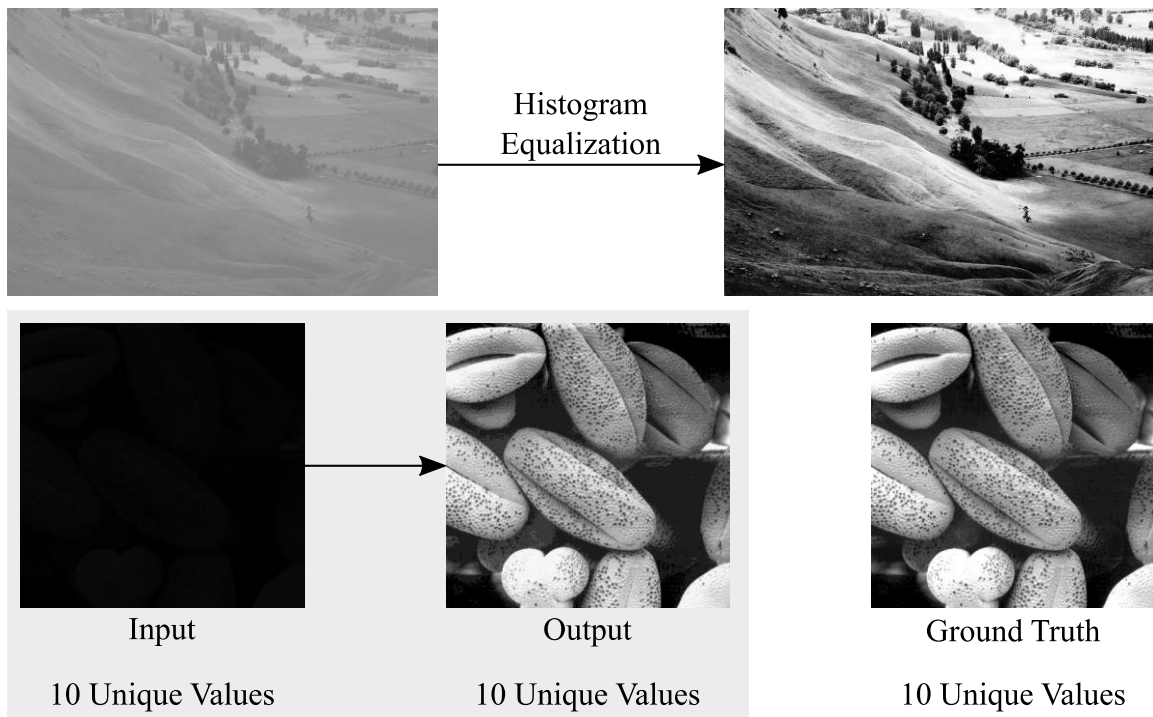


Figure 4.14: Example of the histogram equalization method of contrast stretching. The Histogram Equalization module calculates the histogram of the input image, then transforms it such that the colors are spread over the entire range of intensities. Images from [56] and [57].

- Square
- Invert
- ConvertColorRGB2HSV
- ConvertColorHSV2RGB
- ConvertColorRGB2Gray
- HistEqRGB
- HistEqHSV
- Convolution
- AverageFilter
- MedianFilter
- LookupTable

Passthrough

The Passthrough module is the minimum module which simply passes the input to the output. This was used for testing, but it is also the starting point IP cores as it provides the boilerplate that needs to be set up to develop a new module. So indeed, these are used as the starting points for new modules.

There are, in fact, three Passthrough modules: the plain Passthrough, PassthroughAXI, and PassthroughWindow. The plain Passthrough module simply has an AXI Video Stream input and an AXI Video Stream output and passes the input to the output. The PassthroughAXI module additionally includes the AXI4-Lite register interface. The PassthroughWindow module includes the previous and also the basic setup of the Window module. These modules are used as starting points for building new designs, and also used to benchmark the performance of the AXI bus interfaces.

Scalar Math Operations

Several of the modules, ScalarAddition, ScalarAdditionYCbCr, ScalarAdditionHSV, ScalarMultiplication, ScalarDivision are just bus wrappers around ChIPP modules for scalar math manipulations of an image. The modules whose names end in YCbCr and HSV also include conversions to and from YCbCr and HSV color spaces, respectively. These are used for simple manipulations of images, for example, changing the brightness, contrast, tint (hue), or saturation, are all simple additions to various components of various

color spaces. Addition in the RGB space is also commonly used. It is used for color calibration of monitors and displays that have unequal power output for each color.

Color Conversions

There are IP cores available that implement the color conversion functions. If the goal is to build a processing pipeline that uses a different color space, then it is not a good idea to convert back and forth between color spaces between each stage of the pipeline. These modules can be used to convert to a desired color space once at the beginning of the pipeline. Then the pipeline of operations can be performed with other modules or with external functions all in the new color space. Then at the end of the pipeline the image can be converted back to the original space.

Color Histogram Equalization

Histogram equalization is inherently a single channel operation. There are, however, ways to apply the operation to color images. If the input image is RGB, there are two common methods. The first is to perform histogram equalization on each channel of the image. The second is to convert to a color space that has a quasi-intensity channel, perform the equalization on this channel only, then converting back to RGB. Performing the equalization on the R, G, and B channels individually can give very nice looking results, but it can cause color distortion by scaling each channel differently. Converting to a different color space like HSV that contains a channel dedicated to the image intensity separate from the color information seems like a much better approach. In practice, however, it is very difficult to completely decouple the color information from the intensity, so the equalization in color spaces like HSV can still cause color distortion. Figure 4.15 shows an example of performing histogram equalization on a color image using these two methods. In the first method the module instantiates three independent histogram equalization modules and applies them to each channel. The second method uses the RGB2HSV module, then a HistogramEqualization module on the V channel, then the HSV2RGB module.

Square and Invert

These are modules that, in fact, do not use modules from the ChIPPs project at all. They are simple enough to use the modules in the Utils project, or perform simple math directly. The Square module uses three multipliers, one for each channel. The multipliers have both

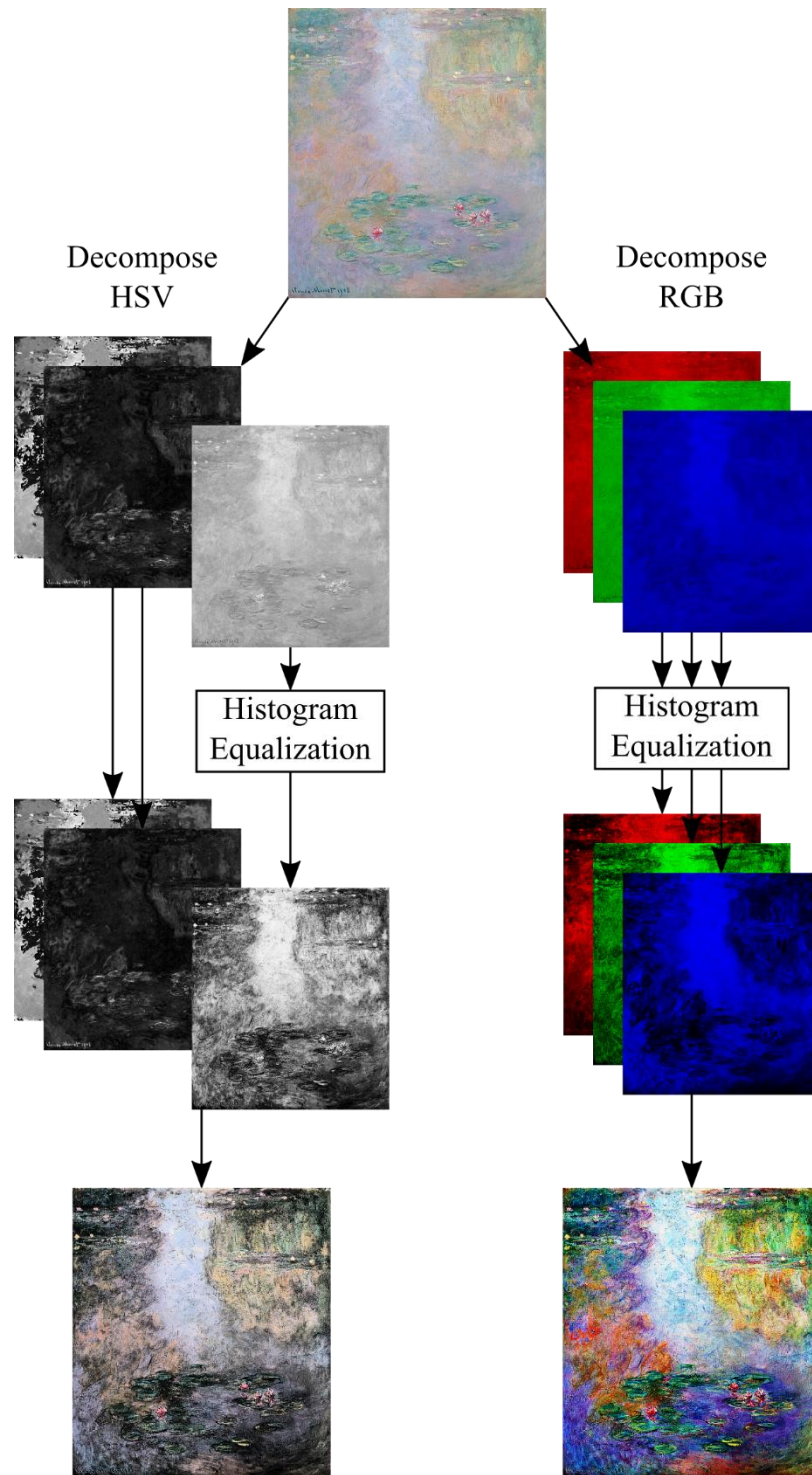


Figure 4.15: Color Histogram Equalization using two methods. On the left: Convert to HSV and equalize the V channel. On the right: Equalize the red, green, and blue channels individually.

their inputs tied to the corresponding channel's input pixel. The Invert module simply subtracts the input data from 255 directly, without the need for additional modules. These modules showcase how easy it is to develop simple IP cores in Chisel with this library. All the streaming and signaling standards are automatically handled, and the only code that needs to be written is the processing code itself.

Convolution

Convolution is a basic operation in many image processing algorithms. In effect, it replaces each pixel by a weighted average of its neighbors. It does this by taking a kernel, a $k \times k$ matrix, and passing it over the input image one pixel at a time. At each step the input image pixel values are multiplied by the value in the kernel, all the values are summed, and the result is divided by a normalizing factor. Figure 4.16 shows this for one position within the image.

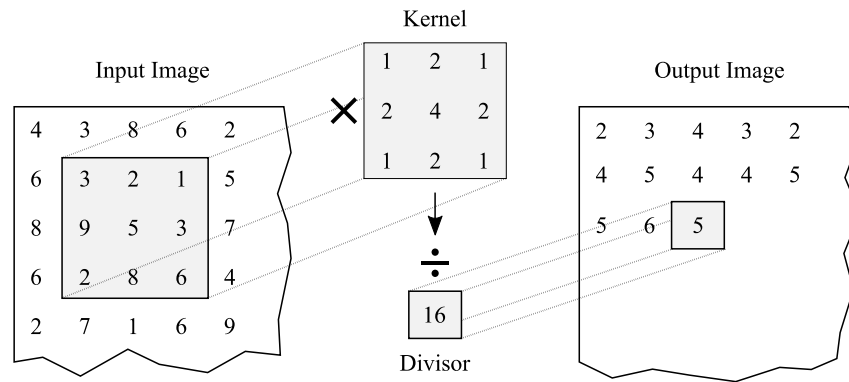


Figure 4.16: Convolution process. The kernel is placed over the image at each pixel, then each kernel value is multiplied with the image pixel value under it. The result is summed and divided by a normalization factor and the result is placed in the output image.

In the FPGA, for a $k \times k$ kernel the module first instantiates a $k \times k$ Window module. The Window module collects the inputs and creates a vector that contains the input data for the current step of the convolution. For the computation, this module instantiates k^2 Multiplier modules, an Adder module with k^2 addends, and a Divider module. The data from the window is first sent to the multipliers which all operate in parallel. Once the multiplications finish, the matrix is sent to the adder module which sums the addends using an addition

tree. Next, the sum is sent to the divider and the output of the divider is the output image pixel at that location. Depending on the values in the kernel, many operations can be performed with this simple computation. Figure 4.17 shows several examples of kernels and the results they have when applied to an image. Figure 4.18 shows the implementation of this 3x3 convolution operation (note that different window sizes would create an implementation with more multipliers).

Like histogram equalization, convolution is an inherently single channel operation. It can be applied to color images, but the procedure is very application dependent. This module provides a single channel convolution so that any color process that needs to be built can be built from these single channel modules.

Other 2D Filters

We have also implemented IP Cores for implementing average filtering and median filtering. These are two of the most common blurring filters. Both implementations use a 3x3 window, with average filtering returning the average value of the input window, and median filtering returning the median.

The average filter operation could be implemented as a convolution filter with the kernel all ones and a divisor of 9, but an optimized implementation allows for completely removing the multipliers. Nine unnecessary multiplications represents a lot of wasted hardware time and resources, so it is worth the separate module. The median filter cannot be implemented as a convolution because locating the median is a sorting problem. An example of the input and output is shown in Figure 4.19.

Lookup Table

The lookup table IP Core is simply a wrapper around the LookupTable module in the ChIPPs project. This IP Core module makes it easy to use the lookup table from the CPU by providing a memory mapped, register interface for accessing the values. This module is used to perform gamma transformation, power and log transformations, and any other operation that are applied to the pixels individually.

Input Image



Gaussian Blur

1	2	1
2	4	2
1	2	1



Sharpen

0	-1	0
-1	5	-1
0	-1	0



Horizontal Sobel Operator

-1	0	1
-2	0	2
-1	0	1



Vertical Sobel Operator

-1	-2	-1
0	0	0
1	2	1



Outline

-1	-1	-1
-1	8	-1
-1	-1	-1



Emboss

-2	-1	0
-1	1	1
0	1	2



Figure 4.17: Convolution examples. The top image is the input, and the output of the convolution module is shown for several different 3x3 kernels.

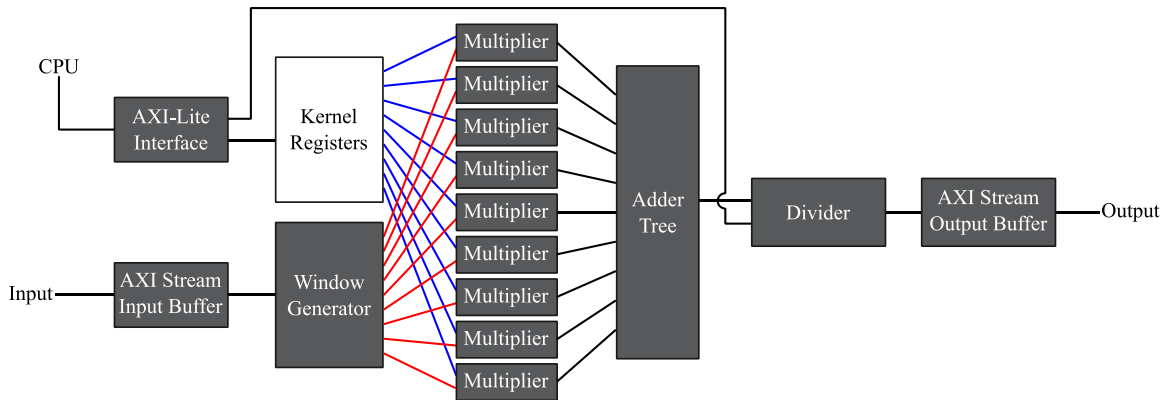


Figure 4.18: Implementation of a 3x3 convolution module using blocks from the ChIPPs and Utilities projects.

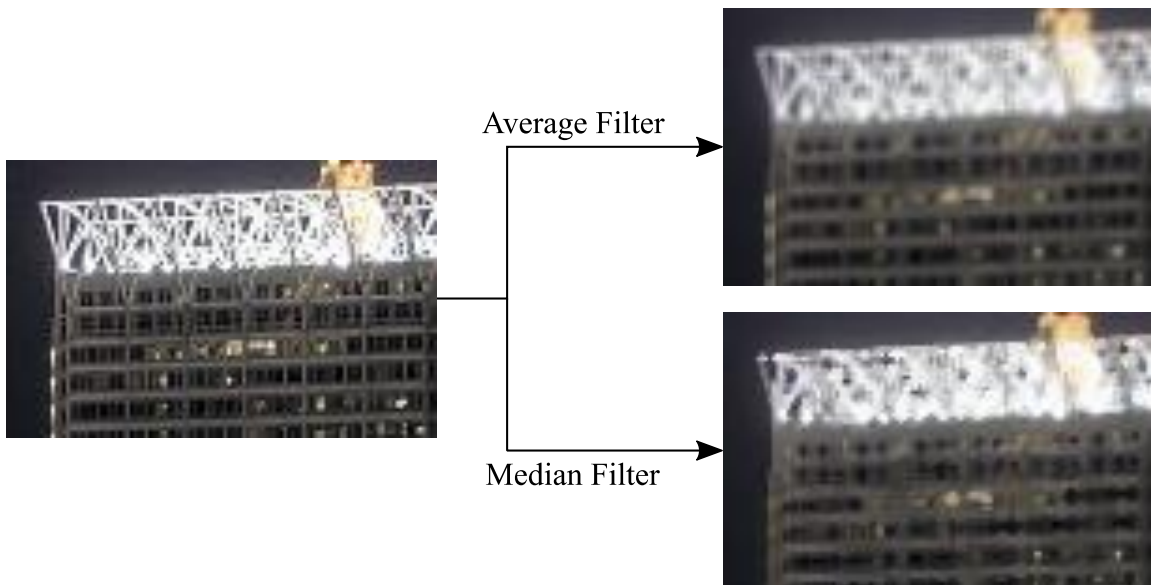


Figure 4.19: Example inputs and output of the average and median filters modules.

5 Building and Benchmarking Image Processing Systems

The process of developing full image systems changes drastically depending on the technology being used. Each FPGA vendor uses their own proprietary tools for implementing designs on the part. Further, even within a single vendor there are large variation depending on the specific part. This chapter describes how to use the FPGA firmware library to build an image processing design on a Xilinx Zynq using the Vivado software. After that, the performance of several image processing functions is evaluated on the actual hardware.

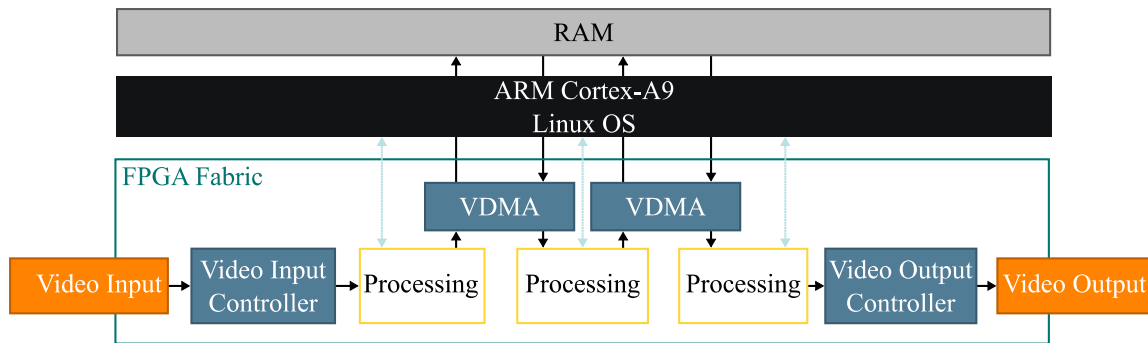


Figure 5.1: Architecture of a general real-time video processing system using the Zynq FPGA and CPU.

5.1 Creating an IP Core for use in Xilinx Vivado block designs

The most common way to build complex system designs in Xilinx Vivado is to use block diagrams. These block diagrams contain modules called “IP Cores” that present inputs and outputs that can be connected to other blocks in the diagram. Therefore, the first step to utilizing image processing functions built with the Chisel Video Library in Vivado is to convert them into Vivado compatible IP Cores. Because of the multitudinous options and specific variations in the process of creating IP cores from process to process and version

to version, the process of creating an IP core is not described here in too much depth. Rather, only the major factors and some important considerations are mentioned.

As a quick note, the existing IP Cores in Vivado use the AXI Stream interface, which is why the IP Cores project in the firmware library uses AXI Stream interfaces for the inputs and output. This allows them to easily connect the existing Xilinx IP along with any other IP compatible with the AMBA bus architecture.

5.1.1 Chisel Outputs

Chisel creates a single Verilog file as its output. This single file contains the top level module definition and the module definitions for all the various functions that are needed by the top level module. This file can be converted to an IP core using Vivado, and the newly created IP can be imported into a block design. During the conversion, the interfaces need to be mapped to the signals they contain and the clock and reset signals need to be configured.

When creating the IP core with Vivado, all of the signals that can be, should be grouped into interfaces. Generally, every processing core will have at least an AXI-4 stream interface (named “axis” in Vivado) for the input and output. Additionally, there is often also an AXI-Lite interface for communication with the CPU cores. These interfaces must be created and properly connected in order to interact with the rest of the IP cores in the Vivado software and the Zynq CPU.

Once the bus interfaces are made, the clock and reset signals need to be configured. Specifically, the polarity of the reset signal and the bus associations of the clock need to be configured. Currently, Chisel only ever uses active high resets. This is in contrast to Vivado’s default of using active low resets, so the polarity of the reset signal should be explicitly defined as active high in the Vivado software. The clock signals can then be associated with the bus interface by using the “ASSOCIATED_BUSIF” property of the clock interface. The value of this property is a colon separated list of the names of the bus interfaces present in the core.

5.1.2 Vivado HLS

Vivado HLS provides facilities for creating Vivado IP cores directly from HLS projects. Once the core has been synthesized, the option to “Export RTL” will be enabled. Figure 5.2 shows the options presented by this function. This function creates a packaged IP core that already contains the proper bus interfaces, resets, clocks, and associations to be immediately used within a Vivado block design. Using this, however, requires the HLS code to include directives that indicate what type of bus interfaces should be present and which variables should be made available on them. This boilerplate code is shown in Figure 5.3.

The Vivado HLS functions also include an “AP_CTRL” interface by default. This interface contains the “ap_start” signal that indicates to the function whether or not it should run at all. For the vast majority of projects, this signal should be tied to a 1 so that the core is always running. Resetting the core can be done using the reset signal, rather than using this generally extraneous signal.

5.2 Creating a video processing design in Xilinx Vivado

Once every processing function has been converted to an IP core, a block design can be created that implements an entire image processing system. A usual video processing block design includes cores for the HDMI input and output, the Zynq’s CPU cores, the Video DMA engines, and all the supporting blocks. We will start by discussing the entire block diagram in general, but quickly move on to more detailed information about the HDMI receiver and transmitter, the frame buffers, and the processing functions.

Figure 5.4 shows an example of an image processing pipeline that performs color histogram equalization as well as including a lookup table that can adjust the final output’s hue, saturation, or brightness. Figure 5.5 and Figure 5.6 show the block diagram in Vivado that implements this processing system. Figure 5.5 shows the entire block diagram including all the clocks, resets, and auxiliary signals. Figure 5.6 shows the same diagram with only showing the bus interface connections visible.

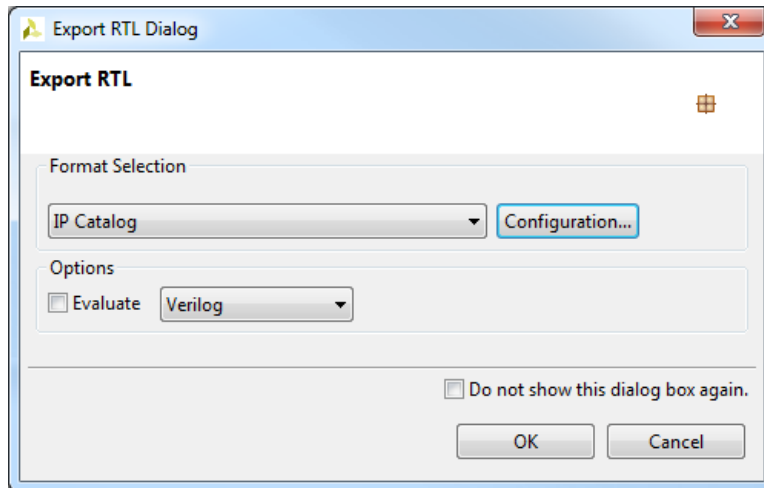


Figure 5.2: Vivado HLS's "Export RTL" dialog to create Vivado IP cores from an HLS project.

```
#include <hls_video.h>

typedef hls::stream< ap_axiu<32,1,1,1> > AXI_STREAM;

void image_processing_function(AXI_STREAM& input, AXI_STREAM& output, int rows, int cols) {

    // Indicate that the input and output should be AXI4-Stream interfaces
    #pragma HLS INTERFACE axis port=inputStream
    #pragma HLS INTERFACE axis port=outputStream

    // Add the rows and columns variables to an AXI-Lite interface
    #pragma HLS RESOURCE variable=rows core=AXI_SLAVE metadata="-bus_bundle CONTROL_BUS"
    #pragma HLS RESOURCE variable=cols core=AXI_SLAVE metadata="-bus_bundle CONTROL_BUS"

    #pragma HLS INTERFACE ap_stable port=rows
    #pragma HLS INTERFACE ap_stable port=cols

    // ...
}
```

Figure 5.3: Empty image processing function showing the directives needed to create an AXI-4 Stream interface input and output, with two variables added to an AXI-Lite interface.

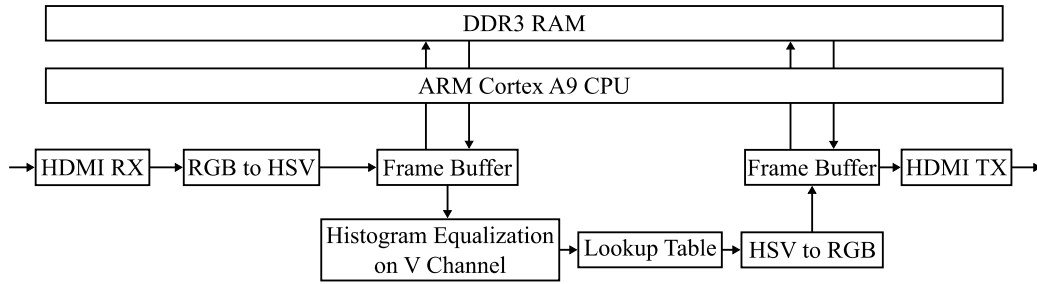


Figure 5.4: Example image processing pipeline. The frame buffers are used to enable processing are used to enable the use of processing cores that block the signal path and for clock domain crossing between the input and output signals.

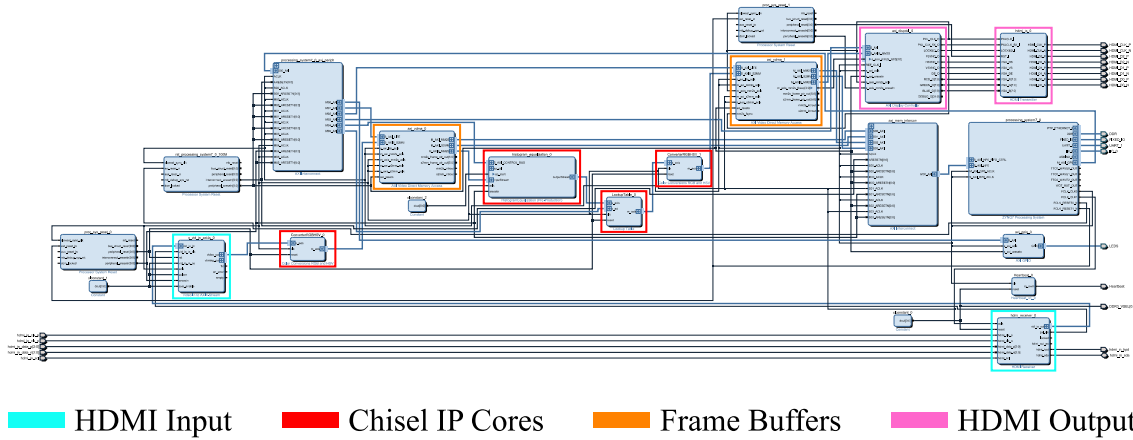


Figure 5.5: Example of a full block diagram for an image processing function. The highlighting has been added for clarity.

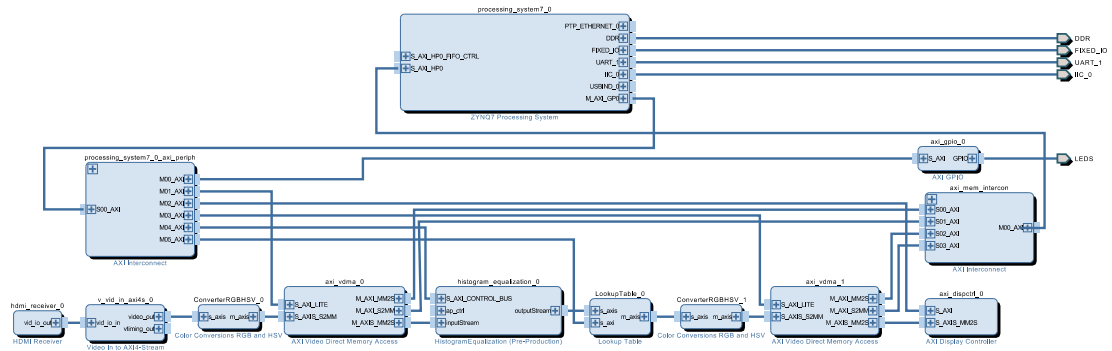


Figure 5.6: Example of a full block diagram for an image processing function showing only the interface connections. Here the flow of the video can be seen at the bottom. From the HDMI receiver on the bottom left, through the frame buffers and processing, to the video output on the bottom right.

While the most important blocks are those directly related to the video processing, there are many supporting, though required, IP in the design. These include three Processor Reset Systems that synchronize reset signals to a clock, two AXI Interconnects, one for the low speed AXI Lite register interfaces and one for the high performance full AXI interface, a Constant block to tie the Histogram Equalization block's "ap_start" signal generated by Vivado HLS to 1, and lastly a ZYNQ7 Processing System block that represents the Zynq's CPU cores and other configurable built in features. In general, these supporting blocks are needed for any processing system. They are usually generated automatically by the software though, so the most important blocks are the ones directly along the path of the video data.

5.2.1 HDMI Receiver and Transmitter

The blocks that interface directly with the external video input and output signals are the HDMI receiver and transmitter. These blocks decode and encode the high speed differential signals transmitted by the HDMI protocol and convert them to or from a parallel video interface. The parallel video interface is then converted into an AXI-4 Stream by additional IP.

The HDMI specification uses the DVI protocol for the video portion of its own protocol. For the sake of minimizing the resources utilized for the I/O, the more advanced features of HDMI, such as audio, HDCP, CEC, Ethernet, etc., are not implemented. Therefore, the HDMI receiver and transmitter are, more accurately, a DVI receiver and transmitter. However, because the physical connection is made using an HDMI connector, and because DVI is a proper subset of the HDMI specification, the term HDMI is used.

The HDMI receiver has two main jobs, first is to decode the incoming video, the second is to provide an Extended Display Identification Data (EDID) interface for the source to access. The incoming video is transmitted as three differential data channels and one differential clock channel. The clock speed is the pixel clock, but the 8-bit data is serialized into 10 bits and transmitted 10 times faster than the clock. Figure 5.7 shows the timings of the HDMI's clock and data lines, and the clock used for deserializing the data. Deserialization of the data requires a very high clock rate for large resolutions, and takes advantage of dedicated hardware resources within the FPGA to achieve this. The Artix 7, and therefore the Zynq 7020, contains the ISERDESE2 primitive that deserializes the input

data. This primitive can operate at double data rate (DDR), which means that it samples data on both the rising and falling edges of the clock signal. This halves the speed of the serialization clock so that it only needs to be 5 times the pixel clock, which significantly increases the resolutions that can be decoded and encoded. The second job of the HDMI receiver is to provide an EDID to the source. The EDID indicates to the source what resolutions this particular device can accept. Without it, many sources will not even enable their output, so it is crucial that it function properly.

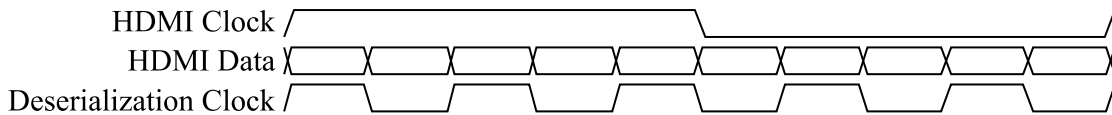


Figure 5.7: HDMI clock and data timings. The HDMI data is transmitted at 10 times the HDMI clock. To deserialize this, the Zynq uses a clock 5 times the HDMI clock and double data rate (DDR) sampling.

The HDMI transmitter has an easier task than the receiver. Its only job is to serialize the data and send it out. Of course, this does leave out many possible features, including hot plug detection, automatic resolution changes, and EDID decoding, but for the sake of conserving FPGA resources, these features are not included.

5.2.2 Frame Buffers and Clock Domains

Frame buffers play an important role in video processing systems. For a system that uses video from an external source, their most important function is enabling clock domain crossing. The HDMI receiver creates an output stream that is synchronized to and controlled by the input pixel clock. The HDMI transmitter, on the other hand, must run at its own pixel clock speed in order to allow different resolutions, or in order to operate without the input functioning. This requires using a frame buffer between the receiver and transmitter. The buffer allows the input to stream the frame to memory as fast as it needs to, and the output to stream the frame to the transmitter as fast as it needs to, each without interfering with the other's operation. With only these two clock domains, the processing function would need to be locked in to one of them, and this is sometimes not possible. Some processing functions need to start and stop during the processing, and if the clock is locked to an external interface this will create signaling error. To allow the processing cores

to function properly, a third clock domain is added. The first is exclusively for the receiver and possibly any processing that is guaranteed to not block. The second clock domain is controlled independently of the receiver and transmitter and can be used for any processing operation including those that require starting and stopping the flow of the stream. The third clock domain is exclusively for the transmitter and, again, any processing that is guaranteed to not block the stream.

To reduce frame tearing, each frame buffer actually uses 3 memory buffers. The input will lock one of the 3 buffers and write a frame to it, and the output will lock a different buffer and read from it. When the write completes, the input will unlock its current buffer and lock the next buffer. Once the read finishes the output does the same, it will unlock its current read buffer, lock the next buffer, and begin outputting its contents. This process ensures that the image is not changed by the input while it is being read by the output, resulting in an image that looks “torn”. Within the Xilinx Zynq ecosystem, frame buffers are implemented with the Video Direct Memory Access (VDMA) block. This block is a 2D optimized DMA that interacts with the RAM at a very high speed. The VDMA implements both writing a frame to the RAM and reading a frame from the RAM. It can be configured to automatically lock its frame buffers to implement the multiple buffer scheme described previously.

5.2.3 Processing Functions

Because the Zynq SoC uses the AXI bus protocols, the image processing blocks are all built to operate within this system. The input is one or more AXI-4 Streams, and the output is also one or more AXI-4 Streams. This streaming interface uses a ready-valid handshake protocol. This allows the stream source to pause by deasserting the valid signal, and it allows the stream sink to block by deasserting the ready signal.

In addition to the main data stream, many processing functions also include an AXI-Lite interface that connects to the CPU cores and presents a memory mapped register interface. This interface can be used for controlling run-time parameters of the functions as well as monitoring their operation or possibly receiving the outputs of functions whose result is not another image.

5.3 Results for several image processing operations

To benchmark the performance of individual image processing functions, the system outlined in Figure 5.8 was used. The design under test (DUT) block is replaced by the specific processing algorithm that is being tested. This system does not include the HDMI receiver or transmitter, or any additional processing as they would influence the placement and routing of the FPGA and would likely be the limiting factor in the design rather than the image processing function itself. The “Async FIFO” blocks are used to allow the DUT to be clocked with a dedicated, fast clock while the test pattern generator, and the frame buffer run at a low clock speed that they are known to handle. This allows the frequency of the DUT to be increased without influencing other blocks in the design so that a more accurate maximum is found. Indeed, it is not uncommon for the DUT to support higher frequencies than the test pattern generator or the frame buffer. Next we describe the metrics that will be measures, then the performance of several of the modules are benchmarked using these metrics.

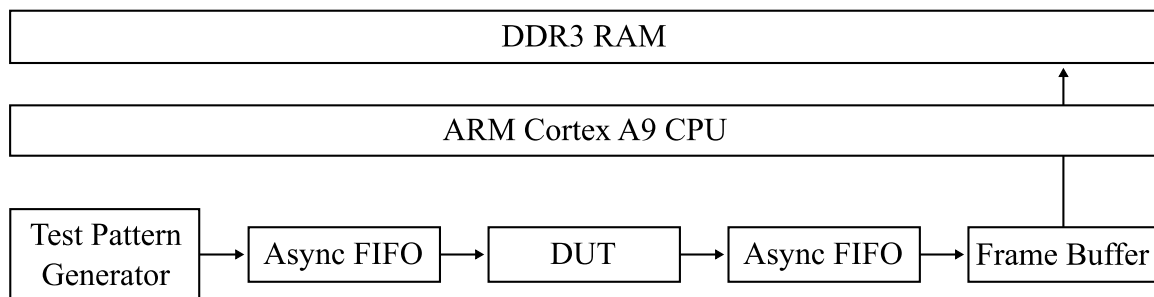


Figure 5.8: Image processing operation benchmarking system. The Asynchronous FIFO blocks allow the DUT to be clocked at a higher speed than the rest of the design. This allows the DUT to run as fast as possible without being limited by the performance of the other blocks.

The two factors we will measure to calculate the performance of this core are its resource utilization and maximum operating frequency. The resource usage indicates how much space this function takes in the fabric, and therefore how many other functions can be included in the system before the FPGA becomes full. Indirectly, the number of resources can also influence the maximum operating frequency. The maximum operating frequency is the limiting factor in the performance of the processing block. All of the processing

blocks that we have created have a bus that inputs 1 pixel per clock cycle, therefore, the maximum frame rate of the processing module for an $m \times n$ image is

$$r = \frac{f_{max}}{m * n + b}$$

where r is the maximum frame rate, f_{max} is the maximum operating frequency, and b is the number of cycles between frames needed to finish processing. For most functions there is no between frame processing, so $b = 0$. Clearly, a faster clock allows more pixels to be processed in the same amount of time, thus higher resolutions can be supported by the processing module for a given frame rate.

All the resource utilizations and maximum frequencies were calculated using the Xilinx Zynq 7020 with -2 speed grade in the CLG484 package and the Vivado 2014.4.1 toolchain with default options for both Synthesis and Implementation.

5.3.1 *Lookup Table*

The lookup table module reads an input pixel and uses the value to index an array. The data stored at that location in the array is the output pixel. Lookup tables can be used to implement any per-channel single pixel operation by first computing the result in the CPU, then loading the lookup table with the result. The module includes an AXI-Lite interface to present a memory mapped register interface to the CPU for configuring the lookup table's memory resources.

The utilization and timing results for the Chisel-based lookup table module are shown in Table 5.1. This is a relatively simple operation, but it has a low maximum frequency because of the use of distributed RAM. The memory for the lookup tables is distributed around the FPGA's fabric, so it can take a long time to read from to this memory. The distribution of the resources can be seen in Figure 5.9. This figure shows the physical locations of the resources used in the FPGA. The core only uses 1-2% of the available resources of the FPGA, but they are spread over much more than 1-2% of the physical space.

Table 5.1: Utilization and maximum frequency for the Lookup Table module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.

Chisel-based Lookup Table					
Max Frequency: 220 MHz					
Bit Depth	8		Resolution	Pixels per frame	Maximum Frame Rate
Channels	3				
LUT-FF Pairs	670	1.25%	1920x1080	2073600	106.1 FPS
LUTs as Logic	299	0.56%	1440x900	1296000	169.8 FPS
LUTs as Memory	288	1.65%	1024x1024	1048576	209.8 FPS
Slice Registers	296	0.28%	1280x720	921600	238.7 FPS
RAM 36/18	0	0.00%	1024x768	786432	279.7 FPS
DSP48	0	0.00%	640x480	307200	716.1 FPS
			512x512	262144	839.2 FPS

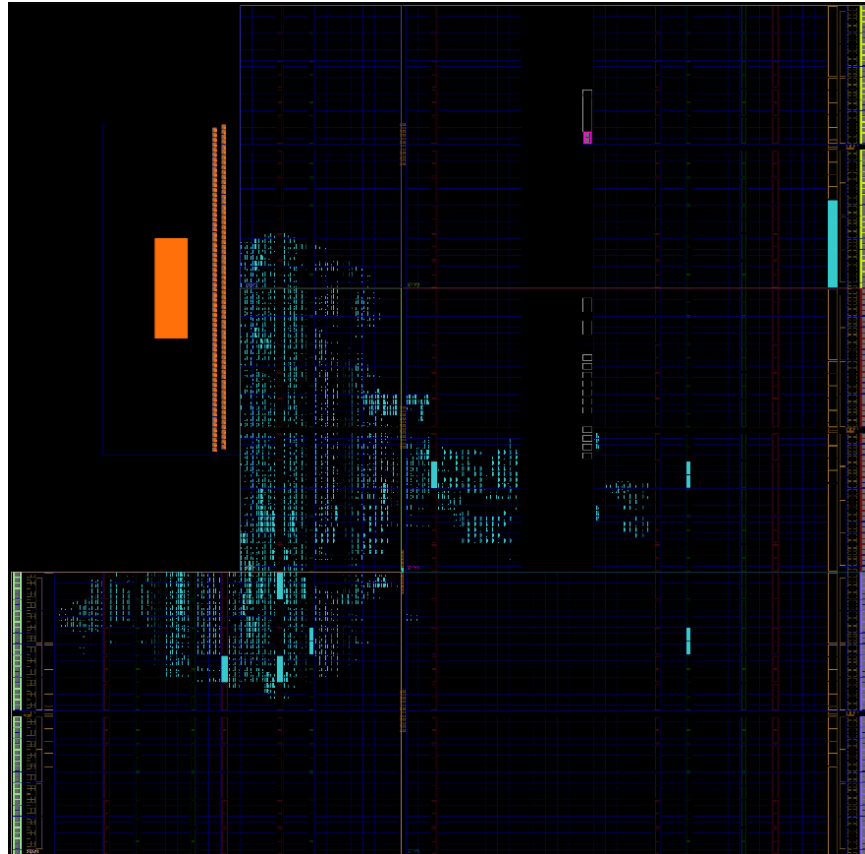


Figure 5.9: Device utilization of the LookupTable performance testing project on the Zynq xc7z020clg484-2 part.

We compared this module to Xilinx’s Gamma Transformation IP Core. Even though it is named for the gamma transformation, it is implemented using lookup tables. The core is configured to enable the AXI-Lite interface, to have 3 channels with 3 independent lookup tables (one for each channel), and to use double buffering. This configuration makes this core functionally equivalent to our Chisel-based lookup table module. The utilization and maximum frequency are shown in Table 5.2. The Chisel-based module has a higher maximum frequency while also using significantly fewer resources.

Table 5.2: Utilization and maximum frequency for Xilinx’s Gamma Transformation IP Core configured to be functionally equivalent to our Chisel-based LookupTable module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.

Xilinx Vivado Gamma Transformation IP Core				
		Max Frequency: 214 MHz		
Bit Depth	8		Resolution	Pixels per frame
Channels	3			Maximum Frame Rate
LUT-FF Pairs	986	1.85%	1920x1080	2073600
LUTs as Logic	620	1.16%	1440x900	1296000
LUTs as Memory	71	0.41%	1024x1024	1048576
Slice Registers	1033	0.97%	1280x720	921600
RAM 36/18	1.5	1.07%	1024x768	786432
DSP48	0	0.00%	640x480	307200
			512x512	262144
				816.3 FPS

5.3.2 Color Conversions

The Chisel Video Library has color conversions between RGB and Grayscale, YCbCr, and HSV. These modules use many more resources than the lookup table module, but they also have a higher maximum frequency because they use very little distributed RAM (the “LUTs as Memory” field in the utilization table).

The RGB to grayscale conversion uses the least resources of any of the color converters because it only needs to calculate the result for a single output data channel. Its utilization is shown in Table 5.3.

The RGB to HSV and HSV to RGB conversions are very asymmetric, so that the implementations of the two result in very different utilizations and maximum frequencies. The conversion from HSV to RGB is much simpler and results in a much higher maximum frequency as compared the RGB to HSV converter. The utilizations of these cores are shown in Table 5.4 and Table 5.5.

The RGB to YCbCr and YCbCr to RGB conversions are quite simple and support very high frequencies, but they require a large number of resources. This is because each output channel is a linear combination of the inputs, so the forward conversion requires 9 multipliers and the backward conversion requires 5. The difference in the number of multipliers accounts for why the backward conversion is able to operate a nearly 300 MHz, while the forward conversion can only support around 250 MHz. The utilizations are shown in Table 5.6 and Table 5.7.

Table 5.3: Utilization and maximum frequency for the RGB to Grayscale conversion module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.

RGB to Grayscale				
			Max Frequency: 250 MHz	
Bit Depth	8		Resolution	Pixels per frame
Channels	3			Maximum Frame Rate
LUT-FF Pairs	1587	2.98%	1920x1080	120.6 FPS
LUTs as Logic	1407	2.64%	1440x900	1296000
LUTs as Memory	0	0.00%	1024x1024	1048576
Slice Registers	1300	1.22%	1280x720	921600
RAM 36/18	0	0.00%	1024x768	786432
DSP48	0	0.00%	640x480	307200
			512x512	262144
				953.7 FPS

Table 5.4: Utilization and maximum frequency for the RGB to HSV conversion module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.

RGB to HSV				
Max Frequency: 226 MHz				
Bit Depth	8		Resolution	Pixels per frame
Channels	3			Maximum Frame Rate
LUT-FF Pairs	2293	4.31%	1920x1080	2073600
LUTs as Logic	1902	3.57%	1440x900	1296000
LUTs as Memory	28	0.16%	1024x1024	1048576
Slice Registers	2179	2.05%	1280x720	921600
RAM 36/18	0	0.00%	1024x768	786432
DSP48	0	0.00%	640x480	307200
			512x512	262144

Table 5.5: Utilization and maximum frequency for the HSV to RGB conversion module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.

HSV to RGB				
Max Frequency: 252 MHz				
Bit Depth	8		Resolution	Pixels per frame
Channels	3			Maximum Frame Rate
LUT-FF Pairs	1784	3.35%	1920x1080	2073600
LUTs as Logic	1541	2.90%	1440x900	1296000
LUTs as Memory	27	0.15%	1024x1024	1048576
Slice Registers	1439	1.35%	1280x720	921600
RAM 36/18	0	0.00%	1024x768	786432
DSP48	0	0.00%	640x480	307200
			512x512	262144

Table 5.6: Utilization and maximum frequency for the RGB to YCbCr conversion module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.

RGB to YCbCr				
Max Frequency: 252 MHz				
Bit Depth	8		Resolution	Pixels per frame
Channels	3			Maximum Frame Rate
LUT-FF Pairs	4410	8.29%	1920x1080	2073600
LUTs as Logic	3864	7.26%	1440x900	1296000
LUTs as Memory	4	0.02%	1024x1024	1048576
Slice Registers	3602	3.38%	1280x720	921600
RAM 36/18	0	0.00%	1024x768	786432
DSP48	0	0.00%	640x480	307200
			512x512	262144
				961.3 FPS

Table 5.7: Utilization and maximum frequency for the YCbCr to RGB conversion module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.

YCbCr to RGB				
Max Frequency: 293 MHz				
Bit Depth	8		Resolution	Pixels per frame
Channels	3			Maximum Frame Rate
LUT-FF Pairs	2296	4.31%	1920x1080	2073600
LUTs as Logic	1938	3.64%	1440x900	1296000
LUTs as Memory	6	0.03%	1024x1024	1048576
Slice Registers	1869	1.76%	1280x720	921600
RAM 36/18	0	0.00%	1024x768	786432
DSP48	0	0.00%	640x480	307200
			512x512	262144
				1,117.7 FPS

5.3.3 Histogram Equalization

The resource utilization and maximum frequency of the Histogram Equalization module currently available in the Chisel library is shown in Table 5.8. As comparison to this, we also tested the histogram equalization core built using Vivado HLS’s video library. The utilization and maximum frequency of this core are shown in Table 5.9. The Vivado HLS core performs better in this case because it optimizes the memory usage of the core so that it can take advantage of the device’s available block RAM resources. This optimization likely accounts for both the additional resource usage of the Chisel-based module and the lower maximum frequency. In the Chisel-based module the memory is still created, but it is implemented as distributed memory across the LUT resources in the CLBs, increasing the usage of slice registers, LUTs as memory, and (by extension) LUT-FF pairs.

Another issue to be aware of is that, in order to have persistent memory across multiple function calls, the C-based HLS core uses function static variables for its memory storage. This means that it is not possible to equalize multiple input streams in the same HLS module without each one overwriting each other’s memory. In the end, both implementations could be improved.

5.3.4 2D Filters

Two-dimensional filters are used for many image processing operations. The base of all of their implementations is the Window module. This module builds a rectangular window of pixels that the processing can be performed on. Because of this, there are relatively strict limits on the performance and utilization of any processing module that utilizes a 2D window. To understand the impact of the Window module Table 5.10 shows the performance of the Window module alone on a 3 channel image.

Using the Window module, several operations can be performed. One very important operation is the Median filter. This is a blurring filter that replaces the pixel value at each point by the median value of itself and its neighbors. The resource usage and maximum frequency for this module are shown in Table 5.11. As expected for another 3-channel module, the maximum frequency is lower than that of the window module alone.

Another important window-based operation is convolution. Many operations, including edge detection, sharpening, blurring, and embossing, are built using various kernels in a

Table 5.8: Utilization and maximum frequency for the Histogram Equalization module written in Chisel.
Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.

Chisel Histogram Equalization				
Max Frequency: 164 MHz				
Bit Depth	8		Resolution	Pixels per frame
Channels	1			Maximum Frame Rate
LUT-FF Pairs	3766	7.08%	1920x1080	2073600
LUTs as Logic	2796	5.26%	1440x900	1296000
LUTs as Memory	289	1.66%	1024x1024	1048576
Slice Registers	3372	3.17%	1280x720	921600
RAM 36/18	0.5	0.36%	1024x768	786432
DSP48	0	0.00%	640x480	307200
			512x512	262144

Table 5.9: Utilization and maximum frequency of the Histogram Equalization module available in Vivado HLS. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.

Vivado HLS Histogram Equalization				
Max Frequency: 181 MHz				
Bit Depth	8		Resolution	Pixels per frame
Channels	1			Maximum Frame Rate
LUT-FF Pairs	2851	5.36%	1920x1080	2073600
LUTs as Logic	2335	4.39%	1440x900	1296000
LUTs as Memory	96	0.55%	1024x1024	1048576
Slice Registers	2218	2.08%	1280x720	921600
RAM 36/18	1.5	1.07%	1024x768	786432
DSP48	4	1.82%	640x480	307200
			512x512	262144

Table 5.10: Utilization and maximum frequency for the passthrough function using the window module.
Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.

Passthrough Window				
Max Frequency: 191 MHz				
Bit Depth	8		Resolution	Pixels per frame
Channels	3			Maximum Frame Rate
LUT-FF Pairs	1258	2.36%	1920x1080	2073600
LUTs as Logic	503	0.94%	1440x900	1296000
LUTs as Memory	544	3.13%	1024x1024	1048576
Slice Registers	549	0.52%	1280x720	921600
RAM 36/18	0	0.00%	1024x768	786432
DSP48	0	0.00%	640x480	307200
			512x512	262144
				747.7 FPS

Table 5.11: Utilization and maximum frequency for the median filter module. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.

Median Filter				
Max Frequency: 181 MHz				
Bit Depth	8		Resolution	Pixels per frame
Channels	3			Maximum Frame Rate
LUT-FF Pairs	4744	8.92%	1920x1080	2073600
LUTs as Logic	2381	4.47%	1440x900	1296000
LUTs as Memory	1936	11.13%	1024x1024	1048576
Slice Registers	1924	1.81%	1280x720	921600
RAM 36/18	0	0.00	1024x768	786432
DSP48	0	0.00	640x480	307200
			512x512	262144
				747.7 FPS

convolution. Table 5.12 shows the performance of the convolution module with a 3×3 kernel. The performance is slightly higher than that of the Window module alone because this operation only uses a single channel and a total of 8 bits, rather than 3, 8-bit channels.

Table 5.12: Utilization and maximum frequency for the single channel convolution module written in Chisel. Utilization percentage is relative to the resources available in the Zynq xc7z020clg484-2 part.

Single Channel Convolution				
Max Frequency: 196 MHz				
Bit Depth	8		Resolution	Pixels per frame
Channels	1			Maximum Frame Rate
LUT-FF Pairs	7244	13.62%	1920x1080	2073600
LUTs as Logic	5885	11.06%	1440x900	1296000
LUTs as Memory	481	2.76%	1024x1024	1048576
Slice Registers	6010	5.65%	1280x720	921600
RAM 36/18	0	0.00%	1024x768	786432
DSP48	0	0.00%	640x480	307200
			512x512	262144

6 Application Examples

The objective of this chapter is to demonstrate the usefulness of the hardware system and the firmware library in developing low-power, real-time applications. This is done in two ways. First, we implement an image enhancement pipeline for real-time low-light video. Second, we benchmark and analyze the use of the firmware library for accelerating convolution for use in convolutional neural networks.

6.1 Implementation: Enhancing Low-Light Real-Time Video

Dark images can contain a large amount of information, even though it may be very difficult to see. The human eye is only able to discriminate one or two dozen different intensities at a given illumination, and this number decreases as the overall scene gets darker [57]. In an 8-bit image there are 256 possible intensity values, but an image that looks dark may only use a small fraction of that. This limits a human's ability to discriminate the intensities in the image. To give our eyes access to the information in a dark image, the range of values in the image can be stretched to fill the entire 8-bit space. This can be done in many ways, but here we will use histogram equalization as discussed and implemented in previous chapters. To handle colors, the histogram equalization is performed in the HSV color space: a color space that separates the color information from the intensity. Figure 4.15 shows that histogram equalization in the HSV color space can have a negative impact on the saturation of the image, so a gamma transformation is performed after the transformation to improve the color of the final result. The pipeline of this process is shown in Figure 6.1. First we discuss how the pipeline is implemented and tested. Second, several example images using this pipeline are shown. Third, the performance of the implementation is quantified and compared to the CPU implementation.

All of the processing blocks for this application are directly available in the firmware library. First, the RGB2HSV module performs the forward conversion to the HSV color space. Then the Histogram Equalization module is used to equalize the Value channel. The LookupTable module then performs the gamma transformation on the equalized result. Finally, the HSV2RGB performs the backward conversion back into the RGB color space

for viewing. Because these functions are readily available, developing this entire pipeline was very quick and easy. Developing this type of pipeline entirely from scratch in Verilog, VHDL, or even a high-level environment like HLS would require a major effort to implement efficiently.

For performance comparison to another embedded, low power alternative, the OpenCV library is used to perform the same processing on the Zynq's ARM CPU. The implemented architecture for the CPU and FPGA is shown in Figure 6.2.

Because the image data is buffered in memory before and after the processing, the processing can be applied to live video input from the external video source, or to videos or images stored in memory by simple copying them to this memory buffer.

Several example inputs and outputs of this system are shown in Figure 6.3. The top row represents an extreme example, and very little information can be recovered. This image was taken with a low quality webcam sensor, which contributes to the abundance of noise. The image on the second row was taken at night with an iPhone 5S on automatic exposure settings (but no flash) in an area lit with street lights. At this level of darkness the input image could not be properly white-balanced, so the output is yellowed. The third row shows a more reasonable example. The image is not completely dark to begin with, so the enhancement will output an image that is of reasonably good quality, but also noticeably brighter. In addition, this image was taken with a high quality sensor, so the input and output do not have much noise.

Performance

Using the FPGA, the entire pipeline runs at a full 60 FPS, well over the limit of real-time. When using the CPU, on the other hand, the process only runs at ~2.7 FPS. This speedup of over 22x can be attributed not only to the performance of the individual modules, but also to their collective use of pipelining and concurrent operation. Extensive and expansive pipelined execution is a major advantage of programmable hardware for achieving real-time performance.

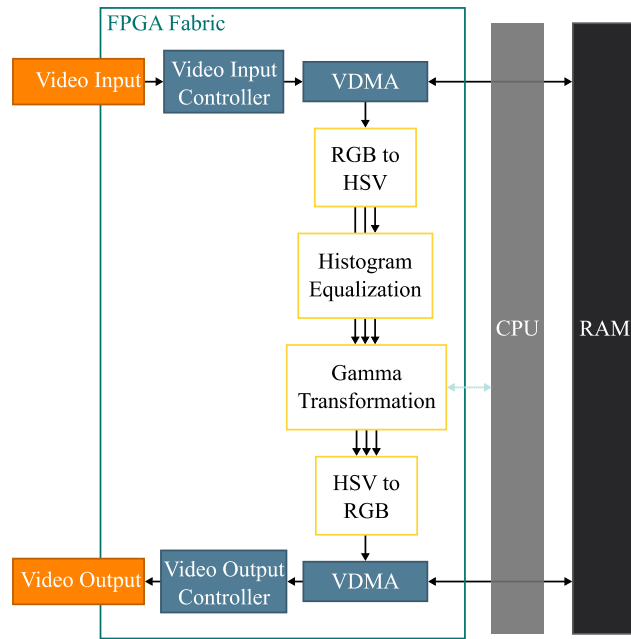


Figure 6.1: Architecture of the low-light video enhancement system implemented on the FPGA.

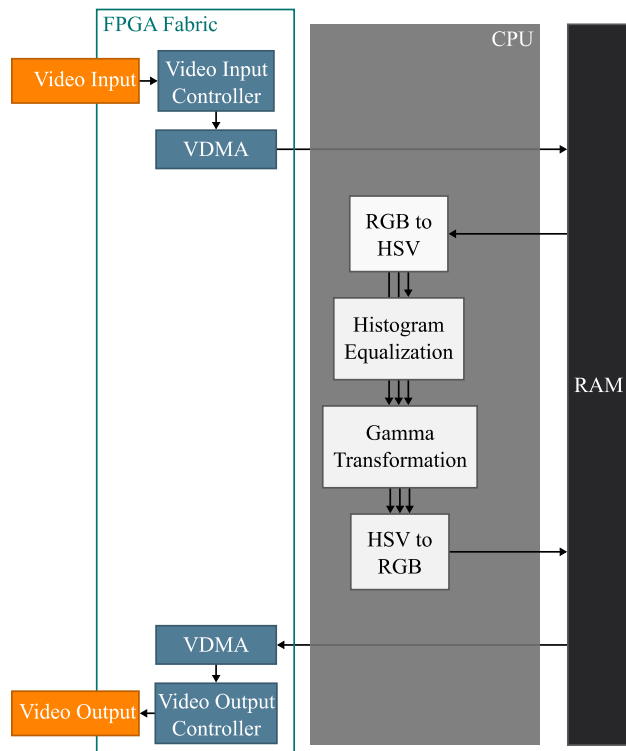


Figure 6.2: Architecture of the low-light video enhancement system implemented on the CPU.

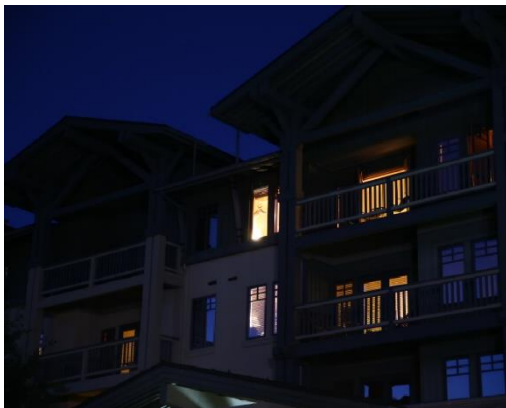
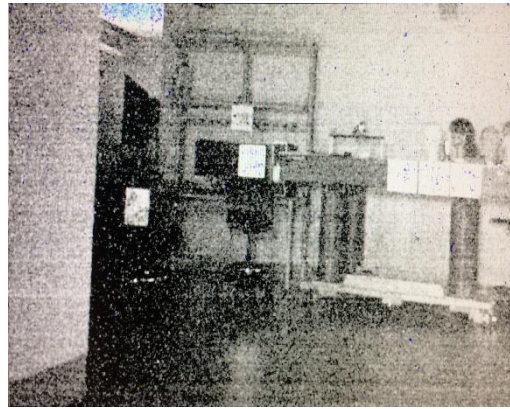
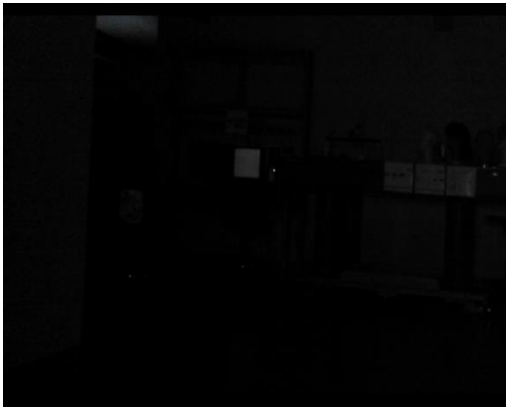


Figure 6.3: Example inputs and outputs from the low-light enhancement system. The inputs are on the left and the outputs are on the right. From top to bottom they show the least input light to the most.

6.2 Benchmarking: Accelerating Convolutional Neural Networks

Convolutional neural networks are very effective for a wide variety of image processing tasks. They are computationally intensive, though, so they can be difficult to use in real-time applications. Fortunately, many of the operations involved can be mapped to an FPGA very efficiently. Here we explore the architecture of these networks and how the firmware library can be used to accelerate their performance. First, an overview of convolutional neural networks is given. Second, we discuss how to use the library for volume convolutions, the expected performance of the FPGA of the operation, and the memory bandwidth required for the operations. Third, the FPGA convolution performance is benchmarked for two convolutional neural networks. Fourth, we conclude by examining the performance of the FPGA and the usefulness of the image processing library.

6.2.1 Overview of Convolutional Networks

A convolutional neural network (also called ConvNet or CNN) is a modification of a traditional neural network that includes a series of convolutions and pooling on the front end. Originally, the idea was inspired by research on the mammalian primary visual cortex [58] and how it responds to regions of an image. This made it a natural candidate for use in the field of computer vision. The networks include several convolution and subsampling stages that act as feature extractors for a subsequent dense, fully connected network. This significantly reduces the dimensionality of the data sent to the dense network and the number of parameters required by it. This reduction is especially important in image processing where there can be several million pixels per image. ConvNets have become quite popular in machine learning and computer vision tasks because they can achieve very good results. In fact, of the 36 teams in the 2014 ImageNet competition, all except 2 used some form of convolutional neural network [59].

Similar to traditional neural networks, convolutional neural networks have an input layer, several hidden layers and an output layer. The difference is in the specific architecture of the layers. An example of the architecture of one of these networks is shown in Figure 6.4. There are several distinct types of hidden layers in a ConvNet in addition to the fully connected layer used in traditional neural networks. The most common layers used before the fully connected network are convolution layers, element-wise activation layers, and pooling layers.

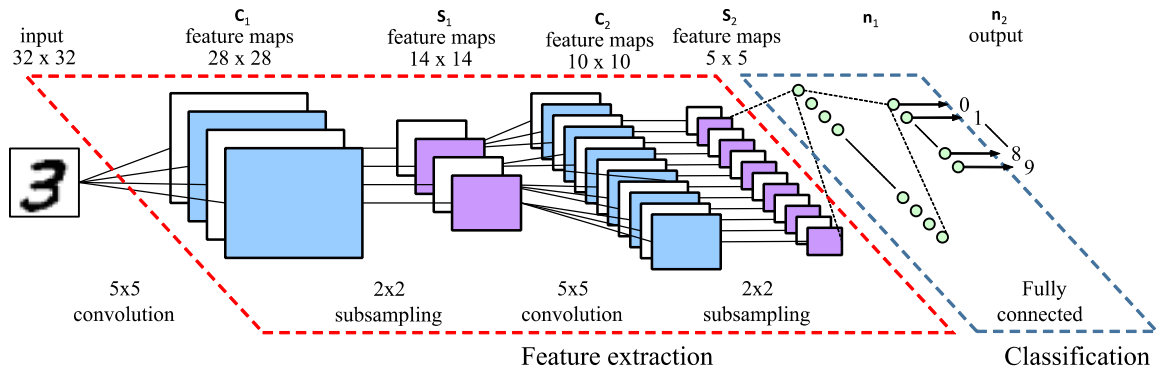


Figure 6.4: Example architecture of a Convolutional Neural Network. Recreated from [60]. The input is a single channel image, it first goes through a series of convolution and pooling layers, then the final classification is performed by a two-layer fully connected neural network.

The convolutional layers are what give the method its name. This is a layer that accepts multidimensional input data, convolves it with a given set of kernels, and outputs the results. The kernels connect the output to only a subset of the inputs, rather than all of the inputs as in a traditional neural network. The kernels are generally small compared to the input (often 3x3, 5x5, 7x7, etc.), so there are far fewer free parameters in these layers than in traditional dense layers. These layers act as feature extractors for the network.

Often, the outputs of the convolution layers are followed by an element-wise activation layer. These layers apply an activation function to each input value. The activation functions are sometimes seen as built into the convolutional layers, but they can be used separately from convolution operations so they are better seen as independent layers themselves. There are several activation functions that are often used including: The Rectified Linear Unit (ReLU) function $f(x) = \max(0, x)$, the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$, the hyperbolic tangent function $f(x) = \tanh(x)$, the absolute value function $f(x) = |x|$, the power function $f(x) = (a + b * x)^c$, and the binomial normal log likelihood function $f(x) = \log(1 + e^x)$. These layers have many uses, including accelerating the convergence of the gradient descent learning and saturating to avoid divergence.

The next type of layer is a pooling layer. This performs downsampling, or subsampling on the input along the spatial dimensions. One common method is so-called “max pooling”. This takes a small region of the input (2x2, 3x3, etc.) and outputs the maximum value of

this region. This provides some amount of rotation and translation invariance to the network. Pooling also greatly reduces the number of parameters in the following layers.

At the end of the network are the fully connected layers. These layers are equivalent to what exists in traditional neural networks and are used as the final output of the system to generate the final activation values. In a real way, the majority of the convolutional neural network can be seen as simply a feature extractor whose output is sent to a standard neural network. In fact, there has been research on replacing the fully connected neural network layers with other types of classifiers, for example, support vectors machines.

Training

Training a convolutional neural network is done via back propagation, and is very similar to training a standard neural network. The difference lies in the fact that many of the weights are shared, which is what creates the convolutional approach to the network. In practice, building an accurate classifier requires a very large dataset and an extremely long time. Using the ImageNet database of 1.2 million images takes 2-3 weeks to train a modern convolutional neural network, even when using multiple GPUs [61]. So, more often than not pretrained models are used. These models can be used directly or “fine-tuned” for a particular application. This fine-tuning takes a pretrained model and continues the training with a much smaller, but much more application specific dataset. This increases the speed of the convergence to the final trained solution because the starting point is not random. It also builds in some amount of instinct, so the network may respond to inputs that it was not explicitly tuned for because the training existed in the pretrained model.

Classifying

The classification output of a network is the activations of the final layer (the output layer). The output activations are created by completing a single forward pass in the network. The information that these output activations represent can vary depending on the network. They can represent the likelihood of a set of options, as in a classification problem that predicts a single variable based on the input. Alternatively, the activation value itself could have meaning. If the function of the network is to predict the location of an object within an image, perhaps the output activations correspond to pixel offsets of the object from the edge of the image. If the network is used for image enhancement, then another possibility is to set the number of output activations equal to the number of pixels in the image. Then

the activation values could be the pixel values of the enhanced image. There are very many possibilities depending on the intended operation of the network.

6.2.2 Hardware and Software Architecture

Normally, CNNs are implemented on a CPU, with certain parts being sent out to a GPU for acceleration. This is the same basic structure that will be used for the FPGA with one major difference. When using GPUs, the CPU and GPU have separate memory spaces (often on separate physical RAM devices), therefore anything the GPU needs to use must be copied from the CPU memory to the GPU memory before the operation is performed. When using the Xilinx Zynq, however, the CPU and FPGA share the same memory, so the FPGA can complete an operation by simply passing the physical address of the source data and the physical address of the destination without the need for copying the memory. The architecture is shown in Figure 6.5. Each layer is initially implemented in the CPU, and one-by-one they are moved to the FPGA until either there are no more layers left or the process has been sufficiently accelerated.

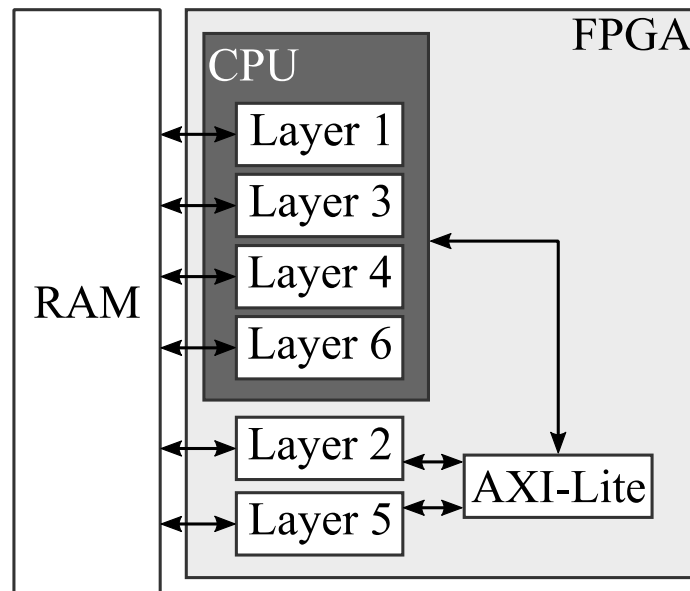


Figure 6.5: Architecture of the hybrid CPU/FPGA convolutional neural network. Each layer reads input from the RAM and outputs to another location in the RAM. The AXI-Lite interface block is used to communicate with the FPGA-based layers for synchronization and configuration.

Performance on the FPGA

The firmware library includes a convolution module that can be used to implement the convolutional layers in these networks. The module that is available, however, only computes convolution for single channel images (i.e., for volumes with a depth of 1). This section analyzes how to use single channel convolutions to implement the volume convolutions that are used in convolutional neural networks. It then estimates how much time is required to complete a volume convolution using the benchmark results of the convolution module in the image processing library for the FPGA.

These single channel convolutions can be summed together to form the full volume convolutions that are required. Consider a convolutional layer whose input volume, I , has dimensions $x_i \times y_i \times n$ and output volume, O , has dimensions $x_o \times y_o \times n$. Then there are m independent convolution kernels with depth n . The m output slices each correspond to performing n single channel convolutions to create n slices from the input volume then computing the dot product of these intermediate slices to produce a single output slice. This process is then repeated m times to create the m output slices. The operation is fundamentally a single channel convolution operations with a dot product used on certain sets of outputs. More precisely, each slice of the output volume is computed as

$$O_j = b_j + \sum_{i=0}^n I_i * K_{i,j}$$

Where O is the output volume, j indexes the slices of the output volume, I is the input volume, i indexes the slices in the input volume, K is the set of convolution kernels (indexed on i and j), and b is a vector of bias values for each output slice. Figure 6.6 shows how this operation can be implemented on an FPGA using only a single channel convolution module. In total, $n \cdot m$ convolutions required to build up the entire set of output images. Performing each of these convolutions one at a time is not the most efficient use of the parallel resources of the FPGA. However, it can be done with minimal resources and will represent a worst case assumption when estimating the performance of these operations.

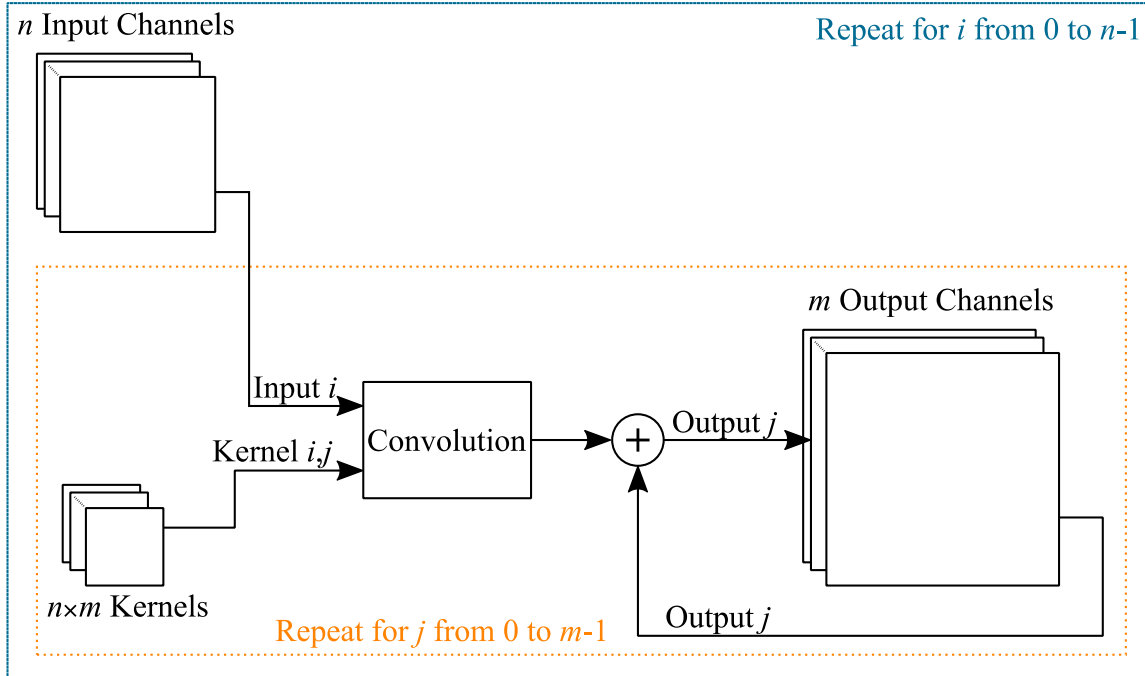


Figure 6.6: Volume convolution can be performed a single channel at a time by summing the results over the various input and kernel combinations.

The time required to complete a convolution layer can be computed by calculating the number of cycles the convolution module needs to complete a single channel convolution and then multiplying by the total number of convolutions that need to be performed to build up the volume. For these calculations consider the input volume from above, I , with dimensions $x_i \times y_i \times n$ and a kernel with dimensions $k \times k \times n$. In this case there are n single channel convolutions. The outputs of these convolutions are then summed with a bias to compute a single output slice. To begin each convolution operation the module needs to gather a $k \times k$ window of the input values. In general, the input window is only considered valid when the center pixel of the window is a valid pixel. Then, since the input values are streamed in sequentially, $k/2$ lines of input must be read before the window is valid. After that, the entire $x_i \times y_i$ image is streamed through the module to compute the output values. The last valid window occurs when the last value in the input is in the center of the window. At this point the line buffers will still contain partial windows, so the line buffers must be cleared out before the next convolution can begin. Clearing the final values from the window buffers requires a final $k/2$ lines of output to be sent. Finally, there is an

extra constant latency because of the pipelining. The constant latency incurred will be denoted as L , and it is the sum of 2 cycles for the input and output buffers, 8 cycles for the multiplications, 33 cycles for the division, 1 cycle to saturate the output value, and $\lceil \log_2(k^2) \rceil$ cycles for the adder tree. The values are read and written one per clock cycle, so streaming an entire image through the convolution module requires

$$\begin{aligned} S &= x_i \cdot \frac{k}{2} + x_i \cdot y_i + x_i \cdot \frac{k}{2} + L \\ &= x_i(y_i + k) + L \end{aligned} \quad \text{cycles.} \quad (1)$$

Building the entire output volume requires

$$C = S \cdot n \cdot m \text{ cycles.} \quad (2)$$

In Chapter 5 it was shown that the convolution module can run at 196 MHz. However, because it is not uncommon for Xilinx modules to not operate at this high frequency, and because the maximum clock rate will be lower in a fully populated design, and simply to create more of a worst case estimate, we will assume the module is clocked at 150 MHz. In this case the time required to complete the volume convolution is

$$T = \frac{C}{150 \cdot 10^6} \text{ seconds.} \quad (3)$$

Memory Bandwidth Considerations

Another factor to consider is the available memory bandwidth of the Zynq SoC. The input and output volumes are stored in an off-chip RAM. The data is streamed out of the RAM, through the CPU, into the Video DMA inside the FPGA fabric, and finally into the convolution module. This process occurs in reverse to store the memory back into the RAM. Each volume convolution will require two memory reads: one for the input slice, one for the previous output slide, and one memory write to store the result. The theoretical maximum bandwidth for the DDR3 RAM with the Zynq 7020 part is 4266.67 MB/s [62]. Therefore, each convolution operation can only require up to a third of this, or 1422.22 MB/s.

If the data requires more memory bandwidth than is available, the memory will not be abide. The transfers will occur only as fast as the RAM can provide the memory and this

will limit the performance of the computation. Note also that the RAM is shared with the CPU and between all of the other FPGA modules that may exist in the design, and they all may be performing memory transactions of their own. It is ideal, then, to not even approach this limit.

6.2.3 Performance Results

In this section the performance of FPGA-accelerated convolutional neural networks is benchmarked. We use the Caffe framework [63] for the CPU implementations of the convolutional neural networks. This library claims to be the “fastest convnet implementation available” [64] so it will provide a good baseline for the FPGA to be compared against. The networks will first be implemented with the Caffe framework on the CPU, then the performance of several of the layers implemented in the FPGA will be estimated using the procedure of the previous section and the benchmarking results of Chapter 5.

LeNet on MNIST

One of the most well-known convolutional neural network applications is on handwritten character recognition. The “LeNet” network architecture was introduced in [26] and represented a major improvement on the state-of-the-art recognition rate for the MNIST database. The MNIST database contains 70 000 images of handwritten digits, 60 000 for the training set and 10 000 for the test set. The images are all 28x28 pixels and contain a single centered digit to be recognized. Several LeNet networks architectures were created to test variations. As well, some networks were trained with additional training images generated by applying distortions to the standard training set.

The architecture of the network is shown in Figure 6.4. The first layer inputs the single channel 28x28 image and outputs 20, 24x24 images. The first pooling layer reduces the size of the images to 12x12. This is followed by another convolution layer that outputs 50, 8x8 images, followed by pooling to 4x4. Finally, this is sent to a neural network with a 500 node hidden layer and a 10 node output layer. Figure 6.7 shows the outputs of each layer for an example image.

The LeNet-5 architecture is implemented in Caffe and can be applied to the MNIST database. On the Zynq’s ARM CPU the network was trained using the training set for

10 000 iterations. The training process took 4 hours and 48 minutes to complete and in the end the network reached a final accuracy of 0.9921 (which matches the 0.8% advertised error rate of the network). After training, the performance of the network on the forward pass of the classification process was tested. The network was run through 100 classifications and the average time for the entire forward pass was 561 ms (only 1.8 frames per second). Figure 6.8 shows the time taken for each layer of the network in an average forward pass. Clearly, the vast majority of the time is spent in the convolution layers, with the second convolution layer being the most costly.

Using the processing functions available in the firmware library, both of the convolution layers can be accelerated as discussed in the previous section. For the first convolution layer the input volume is 28x28x1 (28 pixels wide, 28 pixels high, 1 channel deep), and the output volume is 24x24x20. The convolution uses a 5x5 kernel, so by Equation (1) it takes 973 cycles to complete a single convolution. The depth of the output volume is 20, so by Equation (2) the entire layer takes 19 460 cycles to complete. At a clock rate of 150 MHz this takes only 0.13 ms. On the CPU this layer takes 79.23 ms, giving the FPGA a speedup of ~611 times. In addition, this layer requires 346 MB/s of bandwidth to the RAM. This is well below the limit of 1422.22 MB/s.

The second convolution layer is the slowest layer in the network because of the large depth of both the input and output volumes. On the CPU this layer takes 338.68 ms. The input volume to this layer is 12x12x20, and the output volume is 8x8x50. Using the same calculations as for the first convolution layer, this layer requires 253 cycles per convolution and 253 000 cycles per volume. At a clock rate of 150 MHz this layer will execute in 1.69 ms and require 244 MB/s of RAM bandwidth. This is a speedup of ~200 over the CPU performance and well within the RAM bandwidth limit.

Unfortunately, due to Amdahl's Law, these two relatively large speedups only increase the performance of the entire forward pass by around 4 times to 145 ms per image (nearly 7 frames per second). While still not real time performance, it is much closer. With future work on the library, the fully connected layers and the pooling layers could be accelerated as well to further increase the performance of the network.

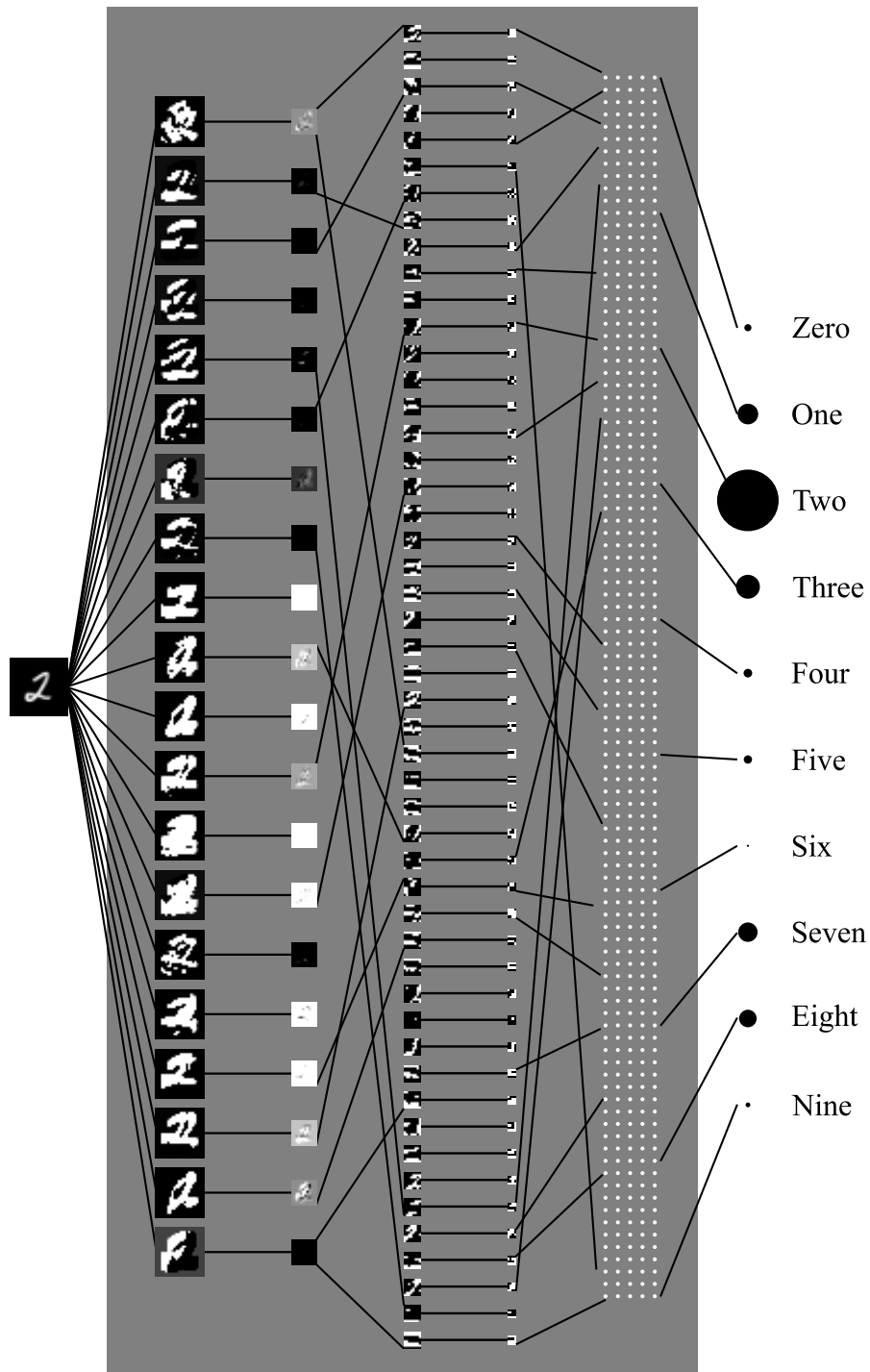


Figure 6.7: Operation of LeNet-5. The input is the image of the number 2 on the left. The ten outputs on the right correspond to the ten numbers. The output with the largest activation is number predicted to be in the image.

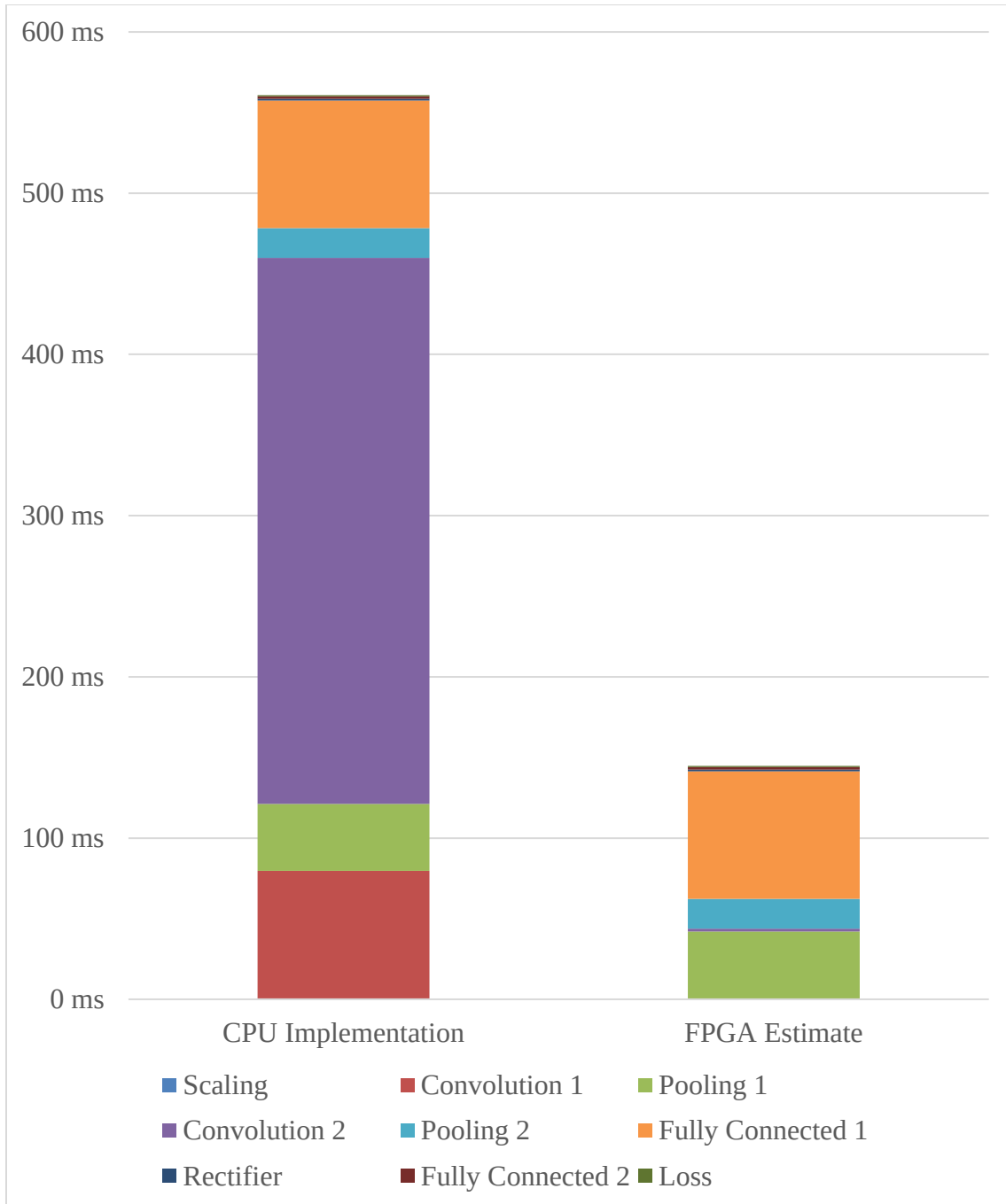


Figure 6.8: Performance of the forward pass of the LeNet network used for handwritten character recognition on the MNIST database implemented in Caffe and executed on the Zynq's ARM CPUs and approximated using hardened accelerators on the FPGA.

Facial Landmark Detection

Another application of convolutional neural networks is facial landmark localization. A recent study [32] developed two convolutional neural networks that can accurately perform landmark regression on face images. Facial landmarks are used for many tasks, including head pose estimation, emotion classification, face alignment in 2D and 3D, and face recognition. This convolutional neural network approach improves landmark localization performance in unconstrained images of faces with various poses, expressions, and occlusions.

Two similar networks are presented in the work. The first, “vanilla”, network is a traditional convolutional neural network layout, with a series of convolutional, activation, and pooling layers followed by a dense neural network. The second network is “tweaked” to increase the accuracy of the localizations. The single dense portion of the vanilla network is replaced by multiple dense networks each fine-tuned for the localization of a specific facial landmark. The networks are exactly the same up until the dense network, and because this chapter is mostly focused on accelerating the convolutional layers, the vanilla network is used here.

The model for this network to be used in Caffe is available in Caffe’s Model Zoo [65]. Using this network on the Zynq’s CPU, the entire network took 46.8 ms to complete a single forward pass. This is approximately 21 frames per second, so this process could already be considered real-time. In a useful application, however, landmark localization requires face detection and preprocessing to occur before this network is used to locate the landmarks. Additionally, given that an image may have multiple faces requiring this network to be run multiple times per frame, this implementation will quickly lose real-time performance in a real-world environment. The performance per layer for this network on the CPU is shown in Figure 6.11. This network has many more layers than the LeNet network, which increases the complexity of the FPGA system, but it also adds more opportunity for acceleration.

The process of landmark localization in a general image is shown in Figure 6.9, and the CNN architecture is shown in Figure 6.10. The process begins with a preprocessing stage that includes detecting the entire face and normalizing the face based on trained average face and standard face images. This preprocessed image is then sent through the convolutional neural network to locate the facial landmarks. These landmark positions are

then backtracked into the original full-size image. Figure 6.10 shows the inputs and output of the layers in this network.

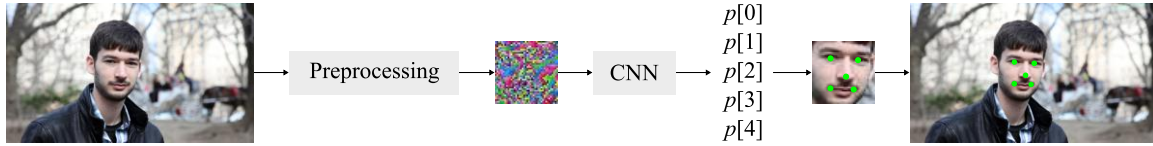


Figure 6.9: Overview of the facial landmark localization process. First the image is preprocessed, this includes locating the face and cropping the image, then normalizing the face based on trained average and standard face images. Then the image is sent to the convolutional neural network to locate the landmark. Finally the landmarks are backtracked to the original image.

The convolutional layers in this network are accelerated exactly as in the LeNet-5 network. The first convolution layer inputs a $40 \times 40 \times 3$ volume, uses a 5×5 kernel, and outputs a $40 \times 40 \times 16$ volume. Using Equations (1)-(3), each convolution takes 1849 cycles, and the entire volume takes 88 752 cycles, or 0.59 ms, to compute. The CPU takes an average of 10.4 ms to execute this layer, giving the FPGA a speedup of ~ 17 times. The execution time for the second, third, and fourth convolution layer on the FPGA can be calculated the same way to get 2.60 ms, 2.78 ms, and 1.66 ms, respectively. On the CPU these layers take an average of 10.03 ms, 6.26 ms, and 0.93 ms, giving the FPGA a speedup of 3.86x, 2.25x, and 0.56x, respectively. In addition, the maximum memory bandwidth required by any of these layers is 371 MB/s, well under the calculated limit.

In addition to the convolution layers, this network also has many other layers that can be very easily accelerated. There are 10 activation layers, each using either the hyperbolic tangent function or the absolute value. Since the input image is only 8-bits per channel, and since ConvNets are not very sensitive to reasonable quantization [66], these functions can be implemented using lookup tables, specifically the lookup table module available in the FPGA library developed herein. The lookup table module is shown in Section 5.3.1 to have a maximum frequency of 220 MHz and, because it does not use a window, this function does not incur the multi-line latency that the convolution module does. Again, in this case we will use 150 MHz for the clock speed rather than the actual maximum so that it matches with the convolution modules and to account for any other modules that might not be able

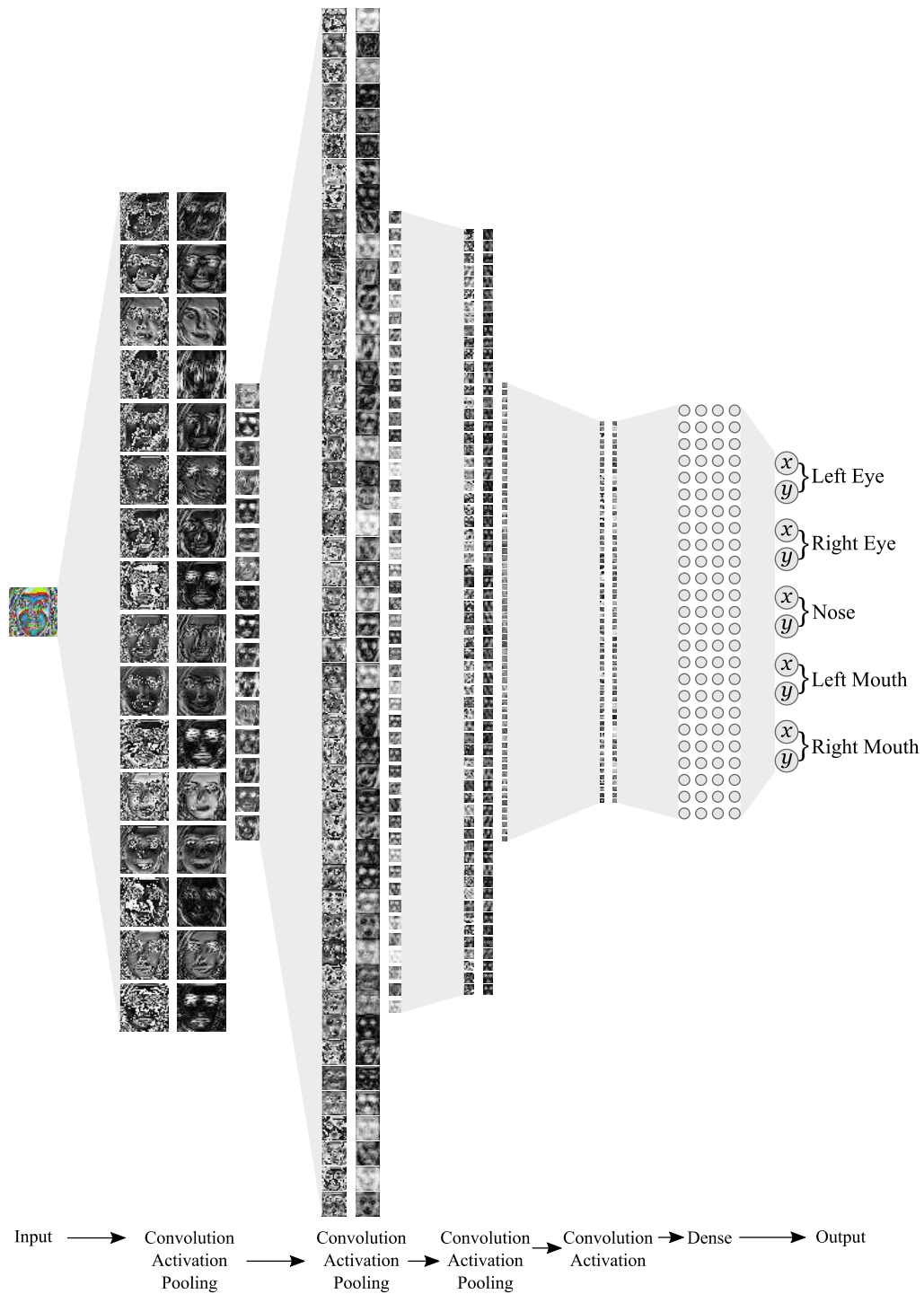


Figure 6.10: Layers on the facial landmark localization network. There are 3 sets of convolution, activation, and pooling. These are followed by a final convolution and activation, then a 2 layer dense network.

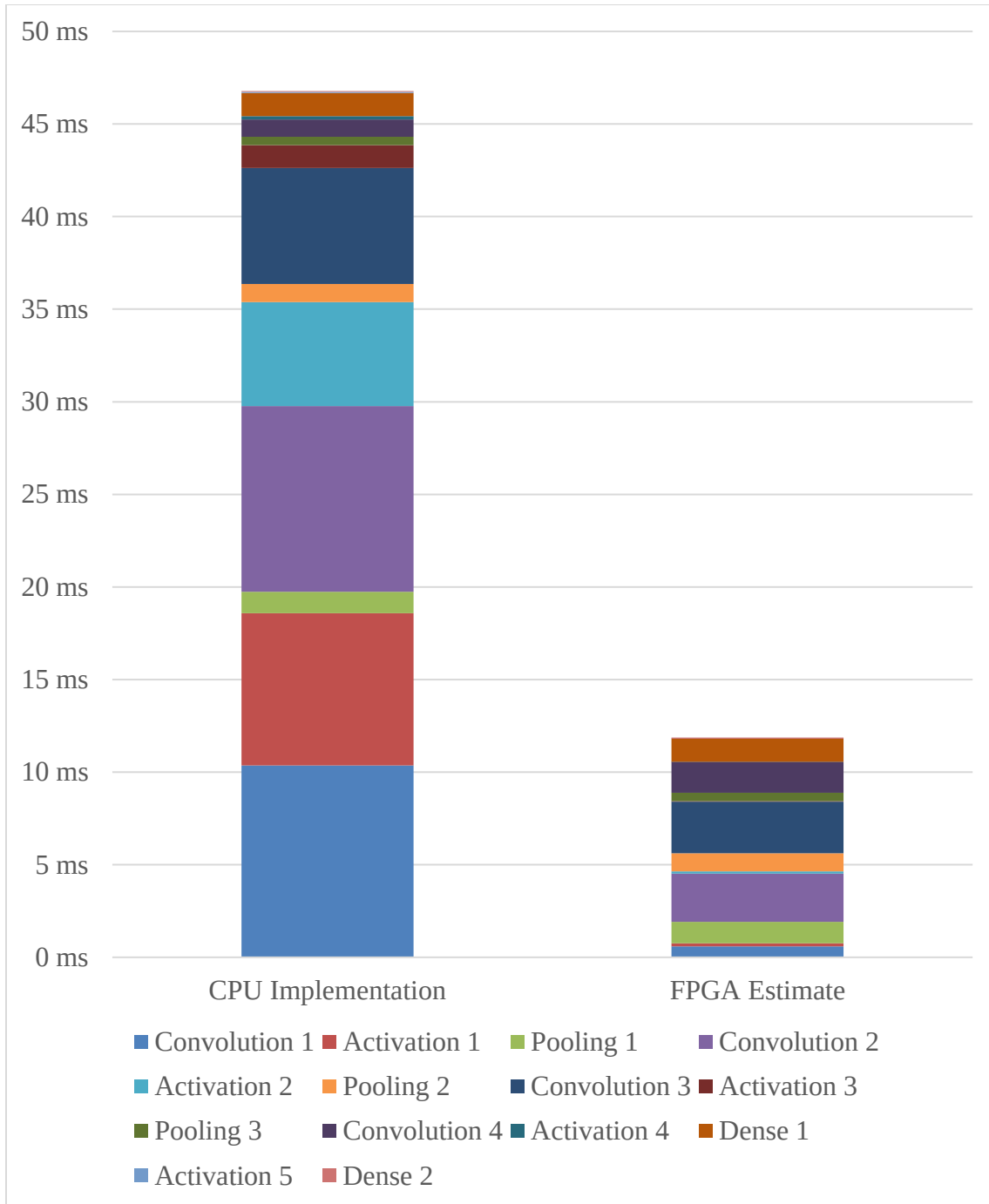


Figure 6.11: Performance of the Vanilla CNN network used for facial landmark detection and localization implemented in Caffe and executed on the Zynq's ARM CPUs and approximated using hardened accelerators on the FPGA.

to perform at such a high speed. The output is delayed only 1 cycles from the input, and an entire image can be processed in only as many cycles as there are pixels in the image. Even better, the hyperbolic tangent activation and absolute value activate are always performed back to back, so they can be collapsed into a single layer and use a single lookup table module in the FPGA instead of performing two separate operations.

There are 5, $|\tanh(x)|$ combined activation layers in the network which, on the CPU, take on average 8.2 ms, 5.6 ms, 1.2 ms, 0.2 ms, and 0.05 ms, respectively. On the FPGA the speed of the layers is determined by the number of pixels in the input image. Using the lowered clock rate of 150 MHz, the layer will complete in $\frac{w \cdot h + 1}{150000}$ seconds, where w is the width of the input image and h is its height. Using this, on the FPGA the 5 activation layers should complete in approximately 0.17 ms, 0.13 ms, 0.03 ms, 0.004 ms, and 0.0006 ms, respectively.

The only layers left inside the CPU in this network are the pooling layers and the dense layers. For the convolutional layers, the FPGA shows an average speedup of ~ 6 times, while for the activation layer the FPGA shows an average speedup of around ~ 46 times. These speedups take the performance of the forward pass of the entire network from around 46.8 ms on the CPU to approximately 11.9 ms when using the FPGA. This takes the framerate from 21.4 FPS to 84.3 FPS.

6.2.4 Evaluation

In conclusion, these calculations show that FPGAs are able to increase the performance of the convolution layers in convolutional neural networks significantly. More than that though, the firmware library can be used to quicken the development of the FPGA-based accelerators. In fact, because a single-channel convolution module is already implemented and available in the firmware library no extra FPGA development is needed. The use of the FPGA for acceleration has thus been converted from the very difficult problem of programming an FPGA into the relatively simple problem of programming the CPU. In general, the easy availability of FPGA-based image processing primitives creates a very easy path to use FPGAs as accelerators for CPU- or GPU-based image processing functions.

7 Conclusions

This work has shown the ability of FPGAs to be used very effectively in small, low-power, real-time image processing applications. In addition, the image processing library for FPGA that was developed makes it much easier to develop high-performance implementations of image processing tasks and quickens the development of entire processing pipeline.

7.1 Summary

As image processing research advances, the computational complexity of the algorithms being developed also increases. Additionally, the use of image processing in portable, real-time environments for hardware like phones and UAVs is increasing in popularity. This creates a significant need for high-performance image processing systems that are small and low power. The recent improvements in FPGA hardware, software, and firmware have made them a very attractive platform for implementing these systems, but the difficulty in programming and lack of existing resources makes their use a real challenge. In this thesis these challenges were addressed by developing a small, lightweight, low-power printed circuit board that can perform real-time high-resolution video processing and by creating a library of high-performance implementations of several image processing operations for FPGAs.

To ease implementation with existing systems and demonstrate the small size and low power abilities of FPGAs, a custom hardware platform was developed. This small board inputs live HDMI video at 60 frames per second, processes the video, and outputs the processed stream over HDMI. The entire system consumes less than 5 Watts of power, and occupies a volume of less than 2.25 cubic inches. This hardware is capable of processing HD video coming from any source including a camera or a computer and interacting seamlessly in an existing system by providing the same output format that is presented as input. This shows the capability of an FPGA based system to be used in low-power situations such as in battery operated devices. It also shows that the size of the system can be reduced enough to be used in lightweight, compact environments.

From there a firmware library was developed that aids in the development of advanced image processing algorithms on the FPGA. This library includes many of the basic building blocks of image processing systems and allows them to easily be composed and modified to implement the requirements of a given system. The library is implemented using low-level descriptions of the hardware, but these low-level descriptions are embedded in a high-level programming environment. This high-level environment makes the library more easily accessible to novice FPGA programmers who simply want to take advantage of the available functions without getting overwhelmed. It also allows advanced FPGA designers to delve deep into the architecture and extend the functionality or create very highly optimized functions that use advanced features of the FPGA.

The firmware library was then integrated into the standard vendor-specific toolset for creating real FPGA designs. A real-time video processing pipeline was developed using a combination of blocks from Xilinx and Digilent to implement the video inputs and outputs, while the processing was performed by the functions available in the new firmware library. Using this environment, the firmware library was then tested and benchmarked to show its performance and quantify its abilities. The available modules were implemented into a real FPGA design to determine their maximum performance and calculate their resource utilization. These numbers demonstrate the performance capability of the system and its ability to handle high resolution video at very high framerates. They are also used to discover places for further improvement and optimization.

Finally, several examples were given of how the firmware library can be used to implement and accelerate real-time video processing systems. First, a real-time low-light video enhancement engine was developed. This system improves the visibility of images that have limited contrast due to insufficient lighting. In this example the FPGA was able to outperform the CPU by over 20 times. Next, the performance of the firmware library for accelerating convolutional neural networks was benchmarked, and the usefulness of the library to achieve this acceleration was analyzed. This deep learning approach was chosen because of its performance and popularity in image processing applications, and because of the wide variety of applications that they are useful for. The ability to accelerate convolutional neural networks unlocks the potential to accelerate a host of specific applications including face detection and recognition; general object detection for things like pedestrians, street signs, curbs and lane markers, trees and rocks and other obstacles;

super resolution; deblurring; denoising; image quality analysis; gesture recognition; optical character recognition; and many others. The two applications specifically shown were handwritten digit recognition and facial landmark detection and localization. Handwritten character recognition is useful in machine understanding of visual imagery that contains text, as it allows the computer to understand information that is intentionally conveyed therein. Recently, there have been augmented reality systems that analyze video in real-time to display context aware information to the user. These systems would be greatly benefited by faster text reading speed and accuracy to deliver a much more accurate and intelligent experience. Using the firmware library to accelerate the CPU implementation of this system shows a performance improvement of nearly 4 times, bringing one of the most accurate algorithms currently available much closer to real-time operation. Facial landmark detection and localization can be used to implement facial recognition systems, among other things. These systems are used in security surveillance systems to validate authorized users upon entry to an area, detect unauthorized access to restricted locations, or automatically locate persons of interest. Using an FPGA in this application improved the performance of the CPU only system over 5.5 times, making it very usable in real-time video systems.

7.2 Future Work

In developing this work many ideas were generated for future work. Some of these ideas are presented here.

For the hardware system, even though DVI was chosen as the input specification, many others are available and useful. Implementing a full HDMI interface would allow the system to handle auxiliary data like audio or Ethernet. It could also allow the system to handle higher bandwidth data like 4K video. Many other video interfaces such as SDI, MIPI CSI and DSI, parallel camera interface, LVDS, GigE Vision, and Camera Link are popular interfaces that could be supported as well. In addition, other Zynq SoCs could be used. Currently the Zynq 7020 is used in the hardware board, but the 7035, 7045, and 7100 offer increased performance by using the Kintex-7 FPGA architecture. These devices include many more programmable logic cells, much faster DSP, and faster transceivers. Xilinx has also announced a new line of Ultrascale+ Zynq devices that will have 4 ARM cores (2 Cortex-A53 application processors and 2 Cortex-R5 real-time processing units),

Ultrascale Kintex and Virtex FPGA fabric architectures, and built in graphics processing units. These devices are much more powerful devices, but their cost and power consumption may be prohibitive in certain applications.

Other hardware improvements could be made to the baseboard as well. Firstly, the addition of a programming interface for the board would make it much easier to use for all stages of development and testing. Secondly, other data connections and interfaces could be added to extend the capabilities of the system. For example, adding a micro SD card would enable storing hundreds or thousands of functions on the board directly so they can be swapped in and out on the fly. Adding GPIO connectors or analog inputs would enable the use of external physical controls for modifying the behavior of the system as it is running. Adding a network connectivity interface like Wi-Fi would enable remote access to configure and monitor the system without physically accessing it.

The system could also be extended to support partial reconfiguration. Partial reconfiguration is the process of updating the operation of part of the FPGA while it is running without restarting the entire chip. This sort of behavior can be used to accelerate very large or complex operations that would otherwise not fit into the FPGA. The processing can be interleaved in time, allowing almost an infinite amount of processing to be performed with even the smallest and cheapest FPGA. It can also be used to develop adaptive image processing system. The module could automatically analyze the scene and chose the best type of processing to perform in the given situation. If there are many options, it is not feasible to implement them all in the FPGA, so partial reconfiguration can be used to automatically load and enable the different processing algorithms from a persistent storage device.

For the firmware library, there is almost no end to future work. First of all, the current functions should be improved to increase their parameterization and allow them to be used in more applications. The more basic elementary function modules such as multiplication and division should be extended to include multiple implementation algorithms including taking advantage of the vendor DSP functions. This will allow them to be better suited for a particular task and allow further optimization of the functions that use them. Additionally, the library should make other processor bus architectures available. Currently, only the AMBA bus is supported because it is the bus used by the Xilinx Zynq devices, but there are several others that are in common use.

Finally, there are a host of additional processing operations that can be added to the library. As a starting goal the library could be extended to include all the functions that are available in OpenCV and other CPU-based processing libraries, which would require a very large amount of development on its own. Along with the algorithm implementation should come an equally significant goal of fully testing these implementations. Verification is extremely important in hardware level design, and each implementation that has been or will be developed needs to be fully tested and verified for accuracy. This process will make the library suitable for use even in industry as well as academia.

List of References

- [1] P. Jacobowitz, "LYTRO ILLUM: a light field camera with the power of a Snapdragon processor | Qualcomm," 2014. [Online]. Available: <https://www.qualcomm.com/news/snapdragon/2014/08/11/lytro-illum-light-field-camera-power-snapdragon-processor>.
- [2] H. Afshari, A. Akin, V. Popovic, A. Schmid, and Y. Leblebici, "Real-Time FPGA Implementation of Linear Blending Vision Reconstruction Algorithm Using a Spherical Light Field Camera," in *2012 IEEE Workshop on Signal Processing Systems*, 2012, pp. 49–54.
- [3] C. T. Johnston and K. T. Gribbon, "Implementing image processing algorithms on FPGAs," *Proc. Elev. Electron. New Zeal. Conf.*, no. August, pp. 118–123, 2004.
- [4] P. R. Possa, S. A. Mahmoudi, N. Harb, C. Valderrama, and P. Manneback, "A Multi-Resolution FPGA-Based Architecture for Real-Time Edge and Corner Detection," *IEEE Trans. Comput.*, vol. 63, no. 10, pp. 2376–2388, Oct. 2014.
- [5] T. L. Chao and K. H. Wong, "An efficient FPGA implementation of the Harris corner feature detector," in *2015 14th IAPR International Conference on Machine Vision Applications (MVA)*, 2015, pp. 89–93.
- [6] H. Orabi, N. Shaikh-Husin, and U. U. Sheikh, "Low cost pipelined FPGA architecture of Harris Corner Detector for real-time applications," in *2015 Tenth International Conference on Digital Information Management (ICDIM)*, 2015, no. Icdim, pp. 164–168.
- [7] M. Huang, O. Serres, T. El-Ghazawi, and G. Newby, "Parameterized Hardware Design on Reconfigurable Computers: An Image Processing Case Study," *Int. J. Reconfigurable Comput.*, vol. 2010, pp. 1–11, 2010.
- [8] C. Brugger, L. Dal'Aqua, J. A. Varela, C. De Schryver, M. Sadri, N. Wehn, M. Klein, and M. Siegrist, "A quantitative cross-architecture study of morphological image processing on CPUs, GPUs, and FPGAs," in *2015 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*, 2015, pp. 201–206.
- [9] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays - FPGA '12*, 2012, p. 47.
- [10] G. Sanchez, F. Sampaio, M. Porto, S. Bampi, and L. Agostini, "DMPDS: A fast motion estimation algorithm targeting high resolution videos and its FPGA implementation," *Int. J. Reconfigurable Comput.*, vol. 2012, 2012.

- [11] Y.-S. Cheng, Z.-Y. Chen, and P.-C. Chang, "An H.264 spatio-temporal hierarchical fast motion estimation algorithm for high-definition video," in *2009 IEEE International Symposium on Circuits and Systems*, 2009, pp. 880–883.
- [12] J. Y. Mori, J. Arias-Garcia, C. Sánchez-Ferreira, D. M. Muñoz, C. H. Llanos, and J. M. S. T. Motta, "An FPGA-Based Omnidirectional Vision Sensor for Motion Detection on Mobile Robots," *Int. J. Reconfigurable Comput.*, pp. 1–16, 2012.
- [13] J. Nikolic, J. Rehder, M. Burri, P. Gohl, S. Leutenegger, P. T. Furgale, and R. Siegwart, "A synchronized visual-inertial sensor system with FPGA pre-processing for accurate real-time SLAM," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 431–437.
- [14] M. Russell and S. Fischhaber, "OpenCV based road sign recognition on Zynq," in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, 2013, pp. 596–601.
- [15] S. Neuendorffer, T. Li, and D. Wang, "Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Librarie," *Xilinx Wiki*, vol. 1167, p. 1, 2013.
- [16] F. M. Siddiqui, M. Russell, B. Bardak, R. Woods, and K. Rafferty, "IPPro: FPGA based image processing processor," in *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, 2014, pp. 1–6.
- [17] M. Schmid, N. Apelt, F. Hannig, and J. Teich, "An image processing library for C-based high-level synthesis," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–4.
- [18] C. Kelly, F. M. Siddiqui, B. Bardak, and R. Woods, "Histogram of oriented gradients front end processing: An FPGA based processor approach," in *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, 2014, pp. 1–6.
- [19] A. Salman, M. Rogawski, and J. P. Kaps, "Efficient hardware accelerator for IPSec based on partial reconfiguration on Xilinx FPGAs," in *Proceedings - 2011 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2011*, 2011, pp. 242–248.
- [20] R. Dobai and L. Sekanina, "Image filter evolution on the Xilinx Zynq Platform," in *2013 NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2013)*, 2013, pp. 164–171.
- [21] X. Zhang, J. Zhao, and Y. LeCun, "Character-level Convolutional Networks for Text Classification," Sep. 2015.

- [22] Y. Kim, “Convolutional Neural Networks for Sentence Classification,” Aug. 2014.
- [23] T. N. Sainath, A. Mohamed, B. Kingsbury, and B. Ramabhadran, “Deep convolutional neural networks for LVCSR,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 8614–8618.
- [24] E. J. Humphrey and J. P. Bello, “Rethinking Automatic Chord Recognition with Convolutional Neural Networks,” in *2012 11th International Conference on Machine Learning and Applications*, 2012, vol. 2, pp. 357–362.
- [25] H. Cecotti and A. Graeser, “Convolutional Neural Network with embedded Fourier Transform for EEG classification,” in *2008 19th International Conference on Pattern Recognition*, 2008, pp. 1–4.
- [26] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [27] D. Ciresan, U. Meier, and J. Schmidhuber, “Multi-column deep neural networks for image classification,” in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3642–3649.
- [28] S. Lawrence, C. L. Giles, Ah Chung Tsoi, and A. D. Back, “Face recognition: a convolutional neural-network approach,” *IEEE Trans. Neural Networks*, vol. 8, no. 1, pp. 98–113, 1997.
- [29] N. P. Ramaiah, E. P. Ijjina, and C. K. Mohan, “Illumination invariant face recognition using convolutional neural networks,” in *2015 IEEE International Conference on Signal Processing, Informatics, Communication and Energy Systems (SPICES)*, 2015, pp. 1–4.
- [30] C. Dong, C. C. Loy, K. He, and X. Tang, “Image Super-Resolution Using Deep Convolutional Networks,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 38, no. 2, pp. 295–307, Feb. 2016.
- [31] L. Xu, J. S. Ren, C. Liu, and J. Jia, “Deep Convolutional Neural Network for Image Deconvolution,” in *Advances in Neural Information Processing Systems*, 2014, no. 413113, pp. 1790–1798.
- [32] Y. Wu, T. Hassner, K. Kim, G. Medioni, and P. Natarajan, “Facial Landmark Detection with Tweaked Convolutional Neural Networks,” Nov. 2015.
- [33] J. Sun, Wenfei Cao, Zongben Xu, and J. Ponce, “Learning a convolutional neural network for non-uniform motion blur removal,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 769–777.

- [34] V. Jain and S. Seung, “Natural Image Denoising with Convolutional Networks,” in *Advances in Neural Information Processing Systems*, 2009, vol. 21, pp. 769–776.
- [35] L. Kang, P. Ye, Y. Li, and D. Doermann, “Convolutional Neural Networks for No-Reference Image Quality Assessment,” in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1733–1740.
- [36] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, B. Martini, P. Akselrod, and S. Talay, “Large-Scale FPGA-based Convolutional Networks,” in *Machine Learning on Very Large Data Sets*, R. Bekkerman, M. Bilenko, and J. Langford, Eds. Cambridge University Press, 2011.
- [37] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, “CNP: An FPGA-based processor for Convolutional Networks,” in *2009 International Conference on Field Programmable Logic and Applications*, 2009, vol. 1, no. 1, pp. 32–37.
- [38] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, “A Massively Parallel Coprocessor for Convolutional Neural Networks,” in *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2009, pp. 53–60.
- [39] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, “A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks,” in *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 696–701.
- [40] C. Garcia and M. Delakis, “Convolutional face finder: a neural architecture for fast and robust face detection,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 11, pp. 1408–1423, Nov. 2004.
- [41] N. Farrugia, F. Mamalet, S. Roux, Fan Yang, and M. Paindavoine, “Fast and Robust Face Detection on a Parallel Optimized Architecture Implemented on FPGA,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 19, no. 4, pp. 597–602, Apr. 2009.
- [42] J. Cong and B. Xiao, “Minimizing Computation in Convolutional Neural Networks,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8681 LNCS, 2014, pp. 281–290.
- [43] “7 Series FPGAs Configurable Logic Block.” Xilinx Inc., 2014.
- [44] S. Asano, T. Maruyama, and Y. Yamaguchi, “Performance comparison of FPGA, GPU and CPU in image processing,” in *2009 International Conference on Field Programmable Logic and Applications*, 2009, pp. 126–131.

- [45] Z. K. Baker, M. B. Gokhale, and J. L. Tripp, "Matched Filter Computation on FPGA, Cell and GPU," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, 2007, no. Figure 1, pp. 207–218.
- [46] E. Fykse, "Performance Comparison of GPU , DSP and FPGA implementations of image processing and computer vision algorithms in embedded systems Egil Fykse," Norwegian University of Science and Technology, 2013.
- [47] "7 Series FPGAs SelectIO Resources." Xilinx Inc., 2015.
- [48] D. Koch, F. Hannig, and D. Ziener, Eds., *FPGAs for Software Programmers*. Cham: Springer International Publishing, 2016.
- [49] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, Wei Song, J. Mawer, A. Cristal, and M. Lujan, "An empirical evaluation of High-Level Synthesis languages and tools for database acceleration," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–8.
- [50] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel," in *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*, 2012, p. 1216.
- [51] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-a-Chip Designs*, 3rd ed. Norwell, MA: Kluwer Academic Publishers, 2002.
- [52] Xilinx, "AXI Reference Guide," *UG1037*, vol. v3.0, pp. 1–175, 2015.
- [53] "AXI4-Stream Video IP and System Design Guide." Xilinx Inc., 2016.
- [54] "Recommendation ITU-R BT.709-6," 2015.
- [55] "Recommendation ITU-R BT.601-7," 2011.
- [56] P. Capper, "Glory be to God for dappled things." CC-BY-2.0. Modified by User:Konstable. <https://www.flickr.com/photos/42033648@N00/74545131/>
- [57] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 3rd ed. Upper Saddle River: Prentice-Hall, 2008.
- [58] C.-B. Ng, Y.-H. Tay, and B.-M. Goi, "Comparing Image Representations for Training a Convolutional Neural Network to Classify Gender," in *2013 1st International Conference on Artificial Intelligence, Modelling and Simulation*, 2013, pp. 29–33.

- [59] “ImageNet Large Scale Visual Recognition Competition 2014 (ILSVRC2014),” 2014. [Online]. Available: <http://image-net.org/challenges/LSVRC/2014/results>. [Accessed: 02-Feb-2016].
- [60] J. Ward, S. Andreev, F. Heredia, B. Lazar, and Z. Manevska, “PARsE | Education | GPU Cluster | Efficient mapping of the training of Convolutional Neural Networks to a CUDA-based cluster.” [Online]. Available: <http://parse.ele.tue.nl/education/cluster2>. [Accessed: 10-Aug-2016].
- [61] “CS231n Convolutional Neural Networks for Visual Recognition.” [Online]. Available: <http://cs231n.github.io/transfer-learning/>. [Accessed: 09-Aug-2016].
- [62] J. Lucero and B. Slous, “Designing High-Performance Video Systems with the Zynq-7000 All Programmable SoC,” pp. 1–15, 2014.
- [63] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional Architecture for Fast Feature Embedding,” *arXiv Prepr. arXiv1408.5093*, Jun. 2014.
- [64] “Caffe | Deep Learning Framework.” [Online]. Available: <http://caffe.berkeleyvision.org>. [Accessed: 09-Aug-2016].
- [65] “Model Zoo · BVLC/caffe Wiki.” [Online]. Available: <https://github.com/BVLC/caffe/wiki/Model-Zoo#facial-landmark-detection-with-tweaked-convolutional-neural-networks>. [Accessed: 31-Aug-2016].
- [66] S. Anwar, K. Hwang, and W. Sung, “Fixed point optimization of deep convolutional neural networks for object recognition,” in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, pp. 1131–1135.

Vita

Steven Clukey was born September 16, 1989 on Whiteman Air Force Base in Missouri, U.S.A. From there he moved to Prattville, Alabama; Ramstein AFB, Germany; Hampton, Virginia; and finally Cleveland, Tennessee where he graduated from A Beka Academy in 2008. From there he went to Cleveland State Community College and attained an Associate of Science degree in 2010 before transferring to The University of Tennessee. He completed his Bachelor of Science in Computer Engineering in 2012. He then began working for the Imaging, Robotics, and Intelligence Systems Laboratory in 2013 pursuing a Master of Science degree in Computer Engineering which was completed in 2016.