THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF ELECTRICAL ENGINEERING

A PARALLELIZED REAL-TIME IMAGE CONVOLUTION ALGORITHM WITH SYSTEMVERILOG

BRIAN YU
SPRING 2015

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree in Electrical Engineering
with honors in Electrical Engineering

Reviewed and approved* by the following:

John Sustersic Jr.
Research Associate
Thesis Supervisor

Jeffrey Mayer
Associate Professor of Electrical Engineering
Honors Adviser

* Signatures are on file in the Schreyer Honors College.

ABSTRACT

Visual salience is a perceptual quality by which an item stands out from its neighbors, grabbing the attention of a viewer. A computer model that can detect salient objects quickly has many applications in the field of computer vision. However, a major limitation of implementing salience detection in computers is the computational power required to process the large amount of data at real-time speeds. In this paper, the implementation of a parallelized Difference of Gaussian pyramid convolution algorithm to be used on a cluster of high-performance FPGA is designed and characterized. Written in SystemVerilog, emphasis is made on parallelizability, design modularity, and parameterization. Under this design, each level of the Difference of Gaussians pyramid can be calculated as quickly as pixels come in, minus some initialization overhead. Performance is affected primarily by the number of pixels to be processed, and the number of levels in the pyramid has little impact. In this highly parallelized design, an FPGA implementation would be effective for real-time processing on high-resolution image inputs.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my supervisor, John Sustersic, for lending his expertise and support throughout the course of this project. Conducting research under him has been a rewarding endeavor, furthering both my education and work experience in ways not possible in the classroom. I would also like to thank the Applied Research Laboratory at Penn State for providing the opportunity and the resources to complete this thesis. I feel very lucky as an undergraduate student working for such an organization. In addition, I offer thanks to my fellow undergraduate co-workers at the Applied Research Lab, especially Matt Miller and Dan Bonness, for their help and support for my work.

Many thanks go out to Penn State University and the College of Engineering, as well as the Schreyer Honors College. This thesis serves as a culmination of an undergraduate career marked with substantial academic, professional, and personal growth. I thank all the professors, advisors, and faculty at this institution for having assisted me in my studies and in reaching my academic and professional goals.

Finally, I would like to extend special thanks to my family and friends, whose support has played an integral role in my success not only as a student, but also as a person. I couldn't have done this without them.

Chapter 1

Introduction

This chapter provides insight into the motivation behind the project. A brief description of visual salience and the role it plays in human vision is provided. In addition, the distinction between bottom-up and top-down visual attention is made. Finally, prior efforts to computationally model visual salience are briefly discussed.

Visual Salience

Visual salience is a perceptual quality by which an item stands out from its neighbors, grabbing the attention of a viewer. In living organisms, detecting and focusing on salient objects is an essential mechanism that allows the viewer to devote more perceptual and cognitive resources on important areas in a given field of view (Itti). Across species, a tendency to direct attention toward visually salient objects is evolutionarily advantageous, allowing rapid detection and reaction to potentially life-threatening dangers (Fine).

In humans, the ability to survey a scene visually and recognize areas of interest involves numerous complex cognitive functions working together in a high-level visual system. To determine what part of a scene is most interesting, a mechanism known as selective attention is employed in the brain (Frintrip et al.). By prioritizing sensory input in this way, the brain devotes processing in detail only to the most relevant portions of data—the region in the field of view

that is being attended. It is therefore essential to consider attention when approaching a model to human vision.

Factors of Visual Attention

The two major drivers of attention are split into bottom-up or top-down factors (Frintrip et al.). While bottom-up factors depend only on a response to the saliency of the visual scene itself, top-down factors are driven by cognitive features of the viewer, such as feature recognition, goals, and expectations. Both bottom-up and top-down factors influence the viewer's overall attention in specific ways.

Bottom-up factors trigger involuntary responses to salient features of an image; a target with stand-out color or shape will naturally draw more attention. Conversely, top-down factors exert a more conscious influence on attention based on context, prior knowledge, or object recognition. For example, a musician reading sheet music is more likely to differentiate between printed note lengths and intervals. Both the bottom-up and top-down mechanisms work together to influence attention during human perception (Frintrip et al.). This paper is concerned with the former type. Quantifying visual salience of a given image is the overall goal—no prior knowledge or contextual learning is considered.

Computer Modeling of Salience Detection

Much effort has been made to model the mechanism of human salience detection for computer vision. A computer model that can detect salient objects quickly has many applications

in the field of computer vision, including autonomous vehicles and robots, visual surveillance systems, and computer-human interface devices. To date, most models combine aspects of feature integration theory and a guided search model, computing these features in parallel to produce a saliency map (Judd, Durand, and Torralba).

A major challenge in computer vision is the computational power required to process a large amount of data at high speeds (Delp and Siegel). Even with the rapid advancement of hardware, processing all data indiscriminately is unfeasible. For this reason, the implementation of selective attention in computer hardware is extremely desirable: It provides a way to reduce the computational requirements of saliency and at the same time emulate natural human vision. To implement this selectiveness in software, Gaussian pyramids can be used to progressively blur and down-sample images with the intent of preserving full resolution in the images' region of interest (the fovea in human vision) and decreasing the amount of data required to represent the less interesting portions of the image (peripheries in human vision). Traditional Gaussian pyramid algorithms perform this in several layers, cropping out the discarded pixels for each layer. Combining this cropping with Difference of Gaussian computations, the total data needed to be processed is significantly decreased.

Coupled with selective attention, recent advancements in field-programmable gate array (FPGA) technology are now providing the computer architecture hardware required to meet ever-demanding processing requirements. The highly parallel structure of FPGAs lends itself to performing operations on images in hardware, and parallelized tasks running on FPGAs can be run at much higher bandwidths than their software-based counterparts.

Chapter 2

Background

This chapter provides background information related to the various components used in the design of the project. A brief history and description of FPGAs is given to provide context for the parallel nature of the algorithm. In addition, an overview of the SystemVerilog language is given. Finally, the image saliency model on which this implementation is based on is described.

FPGAs

History

In 1947, John Bardeen and Walter Brattain at AT&T's Bell Laboratories invented the transistor. Serving as the fundamental building block of practically all modern-day electronic devices, it revolutionized the electronics industry and was the driving force behind the rapid growth of computer technology in the 20th century ("This Month"). As transistor sizes shrunk at an incredible rate over the next few decades, integrated circuits (IC) became faster, smaller, and cheaper to mass produce. However, as the need to create more and more complex devices grew, so did the desire to have programmable hardware.

The first major step in the evolution of programmable hardware is the invention of Programmable Logic Devices (PLDs). Consisting of fixed AND and OR planes, they had the capability of programming simple logic functions at a factory or in the field (Lu). However, even though their logic gates were programmable, the interconnections between these gates were still

hard-wired (“History”). It wasn’t until 1985, when Xilinx produced the first commercially viable FPGA device, the XC2064, that fully reprogrammable hardware was realized. The XC2064 consisted of both programmable gates and interconnections, holding 64 configurable logic blocks and two lookup tables (Maxfield).

Since the XC2064, FPGA capabilities have grown tremendously, with modern high-end devices containing hundreds of thousands of logic units and thousands of memory blocks (“Stratix”). They are used in a variety of applications, including telecommunications, networking, and consumer applications. Straddling the advantages of both hardware and software design paradigms, their flexibility and reusability make it an ideal solution to many computing problems. Today, the global FPGA market is valued at around 6 billion USD and is forecasted to reach almost 10 billion USD by 2020 (“FPGA”).

Structure and Functionality

The general structure of FPGAs consists of configurable logic blocks (CLBs) arranged in a grid connected via programmable interconnects. Surrounding this grid are I/O blocks that can be programmed by the user. Figure 1 shows the typical structure in an FPGA device.

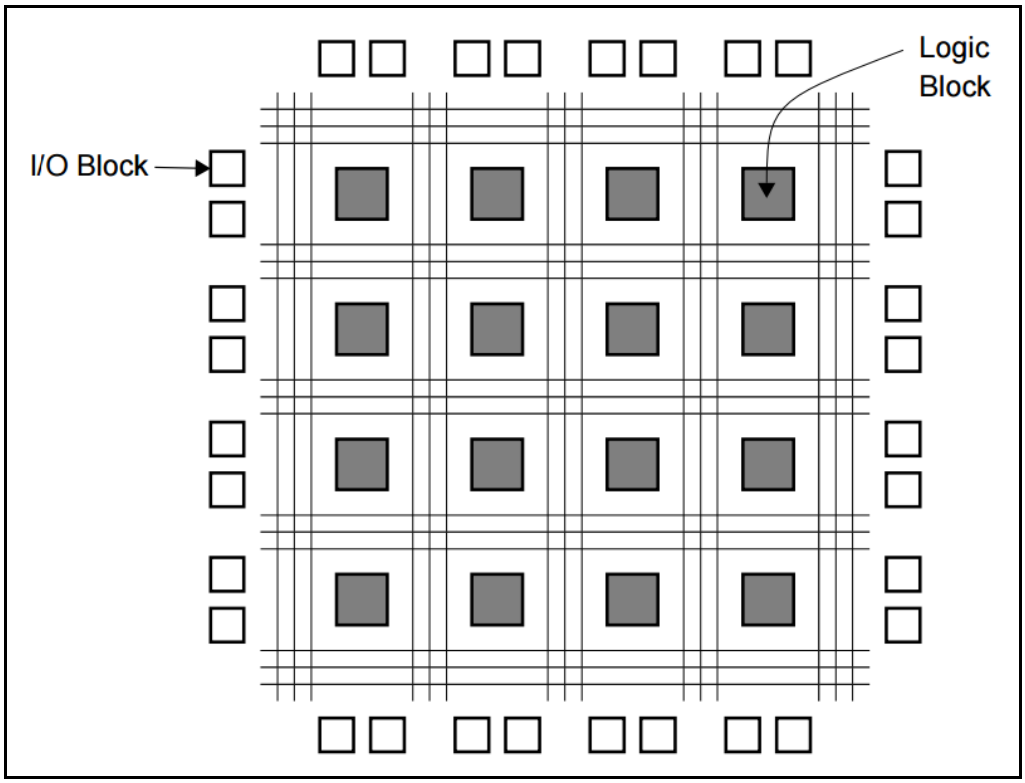

Figure 1: Typical FPGA Structure

Each CLB in an FPGA may consist of NAND gates, multiplexers, lookup tables, or flip-flops arranged in a way that is highly flexible for configuration. Different vendors implement their CLBs with different elements, but the main goal is for the block to be as flexible as possible for configuration of desired circuit logic. Various other elements may also be present in an FPGA device, such as digital signal processing blocks, microprocessors, or other hardwired cores.

Compared to typical CPUs, FPGAs perform operations at a lower clock rate. Typically, FPGAs run at a clock rate on the order of hundreds of megahertz. However, its inherently parallel structure allows it to execute computing tasks that can take advantage of the structure,

such as filtering or convolution. Additionally, because FPGAs operate at a lower level in hardware, it is more suited to computations at a bit level than are CPUs.

Verilog and SystemVerilog

Verilog is a hardware description language commonly used to design and verify digital circuits. As opposed to traditional software programming languages, hardware description languages like Verilog or VHDL describe circuitry and their behavior over time. Syntactically, it is similar to C; part of the motivation for Verilog was the desire for a C-like programming language that could describe hardware.

A Verilog design consists of modules and interconnections between these modules. By instantiating modules inside of other modules and using declared inputs, outputs, and bidirectional ports, a design hierarchy can be specified which represents the overall structure of a hardware system. Inside each module are three possible types of structures: continuous assignments, always blocks, and initial blocks. Continuous assignments serve as a connection to a wire, continuously assigning the left-hand side of the statement to the right. Initial blocks execute statements when the module is instantiated. Finally, always blocks execute repeatedly, being driven by wires or registers specified by the user.

SystemVerilog is an extension to Verilog that adds many capabilities primarily for design verification and modeling, namely object-oriented programming techniques. However, it also includes extensions that adds features and improvements to hardware design.

Image Convolution

Image convolution is a form of mathematical convolution that operates on the pixels of an image to implement filters such as blurring, sharpening, or edge detection. Convolution uses a kernel, a matrix consisting of coefficients that contains an anchor point, to produce a new value for each pixel of the image. To convolve an image with a kernel, the anchor point, typically located at the center of the kernel, is centered over each pixel of the image, with the rest of the kernel overlaying the corresponding pixels in the image. Each corresponding pixel (including the center pixel) is multiplied by the coefficients in the kernel. The sum of each result is then applied to the central pixel as the new value. Figure 2 illustrates this process for one pixel.

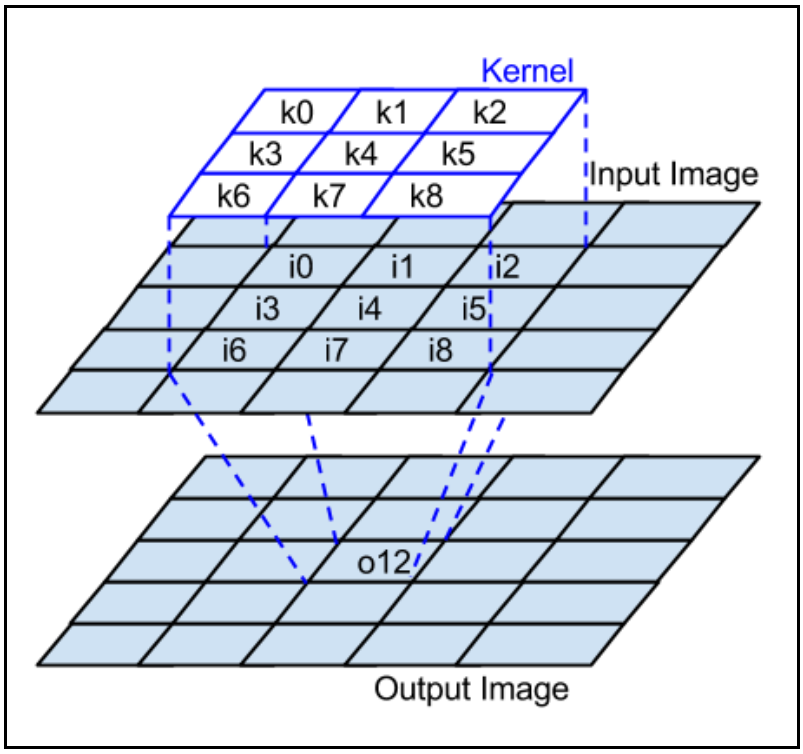

Figure 2: Convolution of a 3x3 kernel on a 5x5 image. o12 is the sum of the products of the corresponding kN and iN values.

There are a number of ways to handle the cases when the corresponding pixel over a kernel entry lies outside the image. One way is to extend the nearest border pixel to provide the pixel value. Other methods include taking the pixel from the opposite edge of the image or cropping the outer edges. In the design discussed in this paper, a value of zero is used for out-of-bounds pixels. This has the effect of darkening the edges of the convolved image.

Chapter 3

Design

This chapter is concerned with the implementation of the parallelized Difference of Gaussians convolution algorithm. The target hardware setup is described, enumerating the hardware components and the interfaces used to communicate between them. Diving into the algorithm itself, each SystemVerilog module is described in detail and the overall dataflow process is given.

Hardware

The convolution implementation discussed in this paper is written for the BittWare S5PE-DS custom-configured PCI Express board. On this board resides two high-density Altera Stratix V GS FPGA devices, each connected to four external SODIMMS, three of which contain QDRII+ SRAM modules with the other containing a DDR3 SDRAM module. Each FPGA device is programmed with Altera intellectual property (IP) cores for executing direct memory accesses over PCI Express. Working in tandem are Altera QDRII+ SRAM controllers and the convolution engine. In hardware, these modules are connected using the Altera Qsys system integration tool.

When the host computer is powered on, it detects the board on the PCI Express bus. The device is mapped using the BittWorks II libraries and drivers provided by BittWare. To facilitate data transfers between the host computer and the external memory on the PCIe board, a program

is executed. This program utilizes the BittWorks Host Interface Library (BwHIL) to initialize the memory and perform data transfers. Once the image data is transferred, the hardware system can begin the convolution.

A PCI Express hardware IP provided by BittWare is adapted to work with the custom-made boards to provide the high-speed interface between system memory and on-board memory. A master software application handles the transfer of image data between the two. Finally, RTL code implements the image convolution algorithms (currently written with C++ & OpenCV) on the FPGA. Figure 3 shows a high-level block diagram of the overall operation.

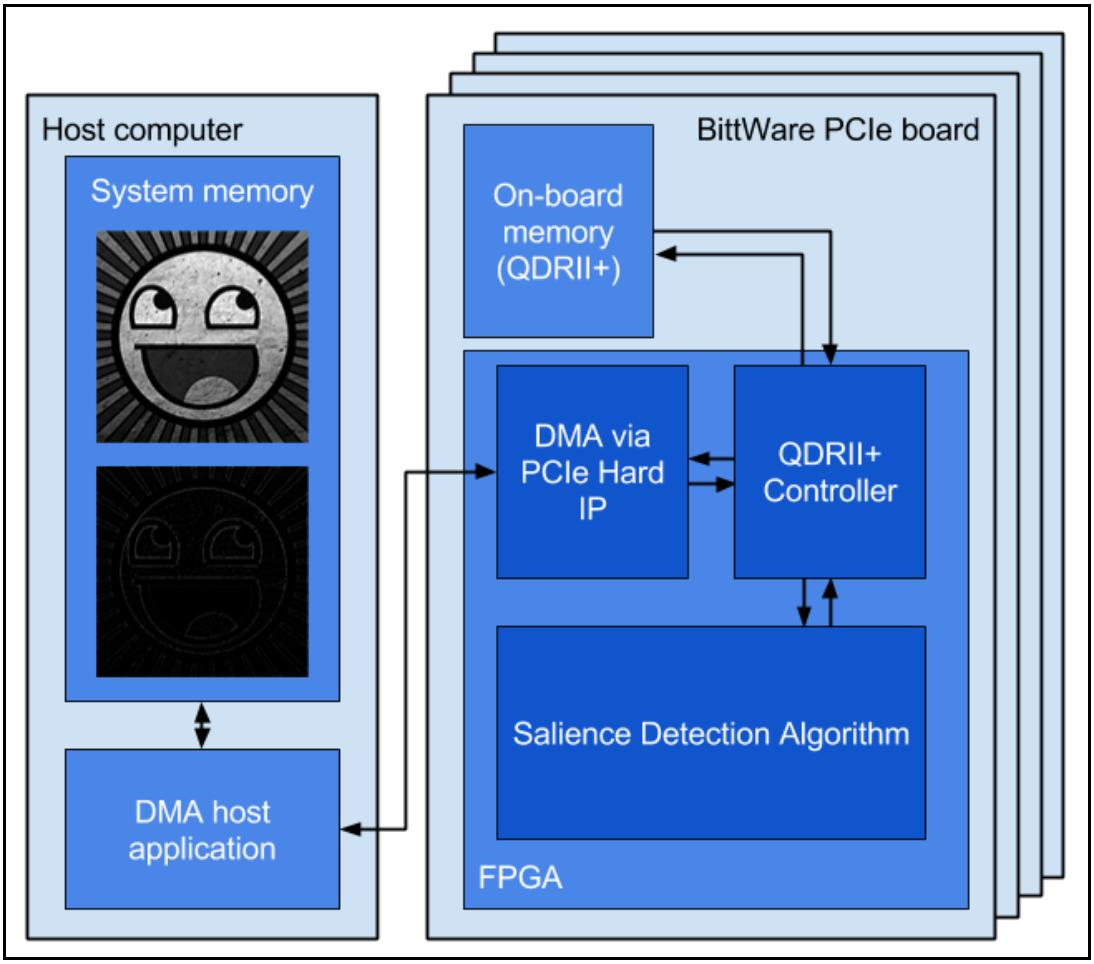

Figure 3: High-level block diagram of target hardware

PCI Express DMA Design

To facilitate data transfers between the host computer's memory and the memory on the BittWare board, a PCI Express reference design provided on the BittWare developer website is adapted. The design consists of the PCI Express Hard IP Core for Stratix V FPGA devices generated in Qsys, an Altera Avalon Memory-Mapped (Avalon-MM) DMA subsystem, also in Qsys, and the software application that runs on the host system as the master for the design.

PCI Express IP Core

The PCI Express Hard IP for Stratix V FPGA devices is a complete implementation of the PCI Express protocol, which includes the transaction, data link, and physical layers. The transaction layer communicates with the application layer, which in this design is the software program executing on the host computer. The transaction layer also manages the receive and transmit channels and executes flow control. The data link layer maintains packet and data integrity between the physical and transaction layers. Its job is to generate error-checking codes and manage acknowledgements and retries. Finally, the physical layer manages the speed type in addition to the lane number and width for each packet received from the above layers. Figure 4 shows a high-level block diagram of the PCI Express Core.

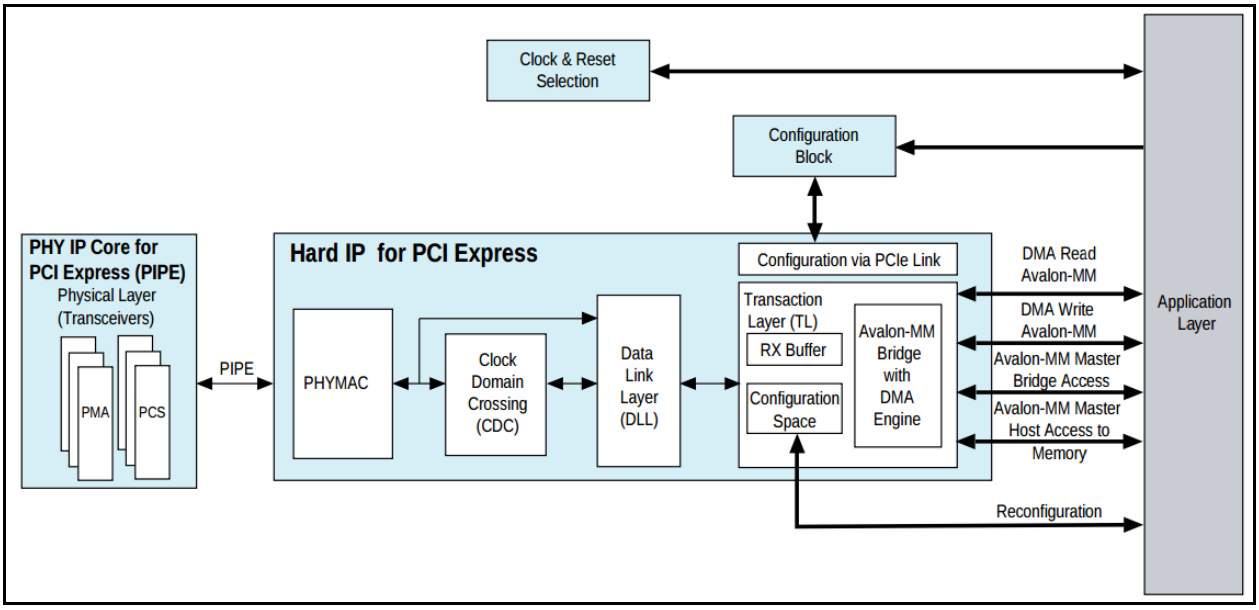

Figure 4: Block diagram of PCI Express Core (from Altera)

Avalon-MM DMA Qsys Subsystem

The second Qsys component in this design is the Avalon-MM DMA subsystem. This connects the application layer and the transaction layer in the PCI Express protocol stack. In Direct Memory Access (DMA), descriptor tables are used to control the flow of data. The DMA descriptor table consists of 128 descriptors, each keeping track of the source address, destination address, and the size of the data that is to be transferred. The DMA controller loads the descriptors from the descriptor table in order to perform the DMA, and when it does, writes a flag back to the table signifying its completion. Figure 5 shows an overview of this data flow.

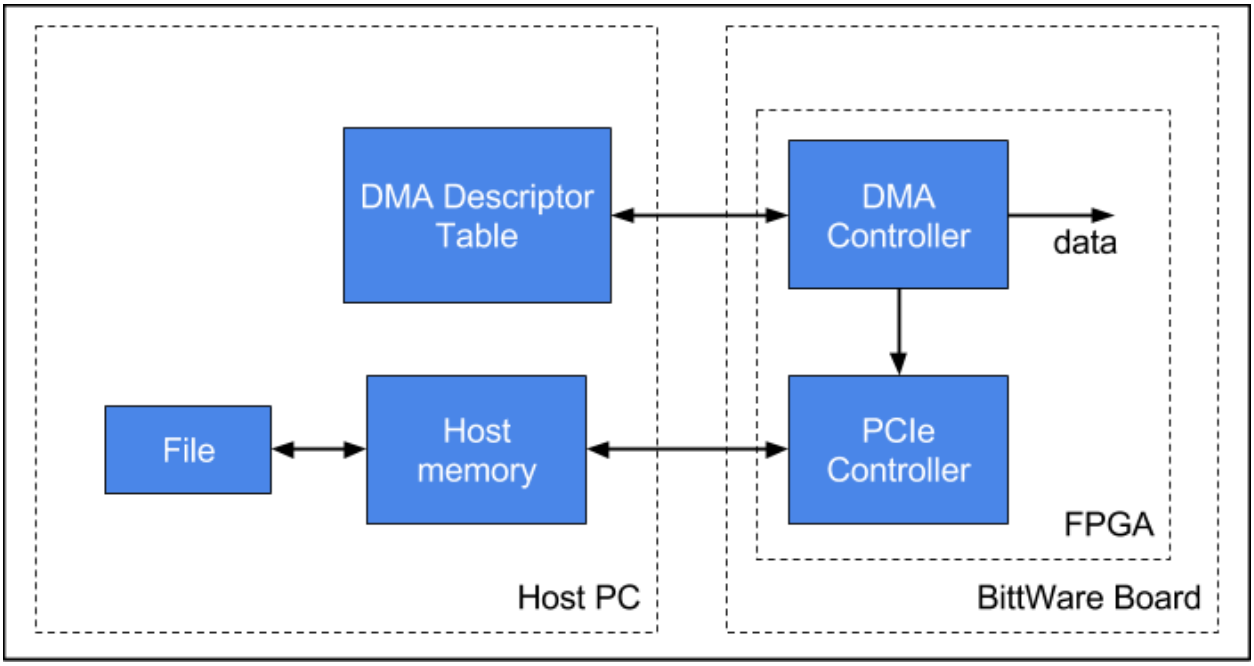

Figure 5: Block diagram of DMA via PCIe

Software Application

The final component of the PCI Express DMA design is the software application, which serves as the application layer on the PCI Express protocol stack. It is not a Qsys subsystem, executing instead on the host machine in software. The program uses the BittWorks II libraries to initialize the BittWare Host Interface Library (BwHIL). Using the BwHIL, the resources of the mapped PCI Express board can be accessed. Once a device handle is obtained and the system memory is allocated for the 128 descriptors in the descriptor table, the DMA begins. The DMA controller's registers are initialized with the location of the descriptor table in the host's memory. Next, the descriptor table is initialized with a source address, destination address, payload size, and identifier for each descriptor. Once the table is filled, the program polls the completion flag of its last entry. When this is updated, the DMA transfer is complete. Figure 6 shows the data sequence for each memory transfer.

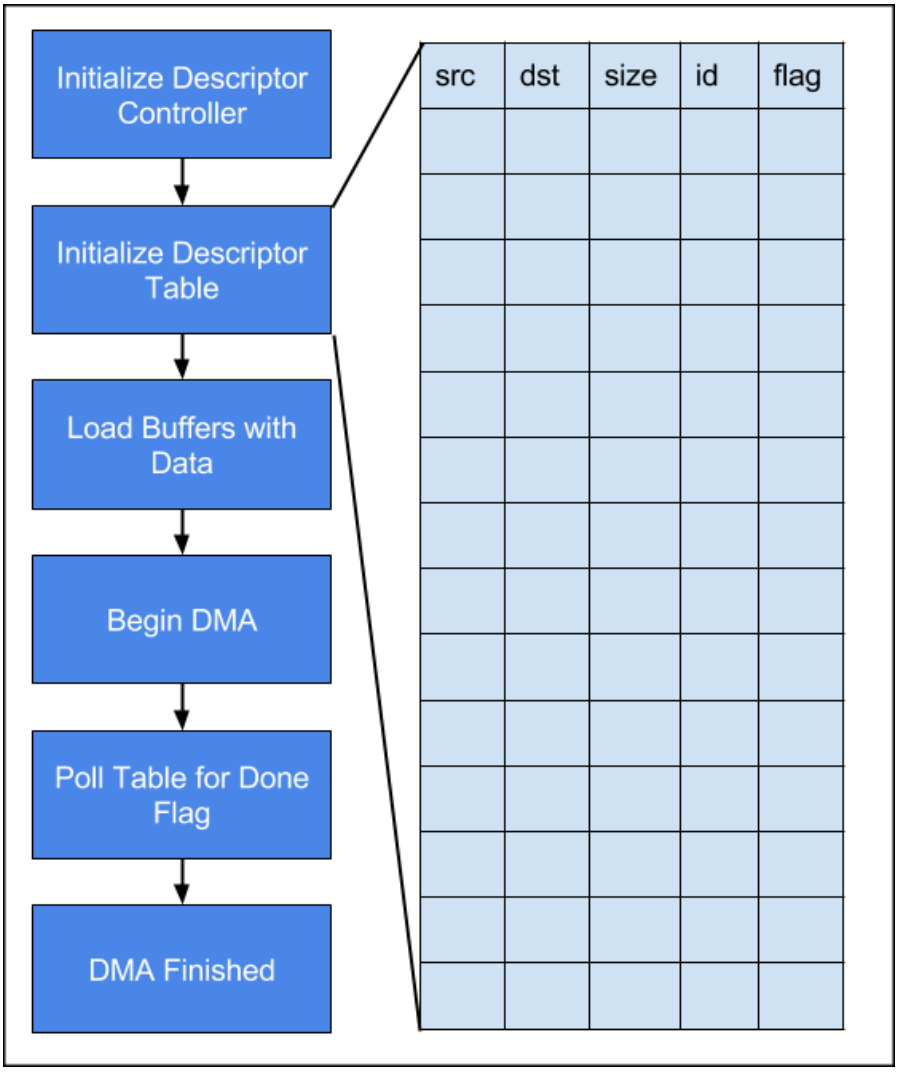

Figure 6: Data flow diagram of DMA software application

QDRII+ Controller Hard IP

The second Qsys component in the design is the QDR II+ SRAM controller. Six QDR II+ SODIMMS reside on each of the PCI Express boards, three for each FPGA device. Though QDR transfers data on both edges of the clock like DDR SRAM, it differs from DDR in that uses unidirectional data buses, making read and write operations independent of each other. QDR has

two separate clocks and separate buses for reading and writing data. This architecture makes QDR operate with extreme efficiency when both read and write operations occur at high clock frequencies.

In Qsys, the QDR controllers translate memory reads and writes on the Avalon Memory-Mapped interface to the physical layer of the external QDR memory device. In the design, it is used whenever data is read from or written to the host computer's memory.

Difference of Gaussians Pyramid Algorithm

The parallelized Difference of Gaussians pyramid algorithm implemented on the FPGA is the third and final Qsys component in the design. This component consists of SystemVerilog source files, each representing a separate module. Some of the modules created are for simulation purposes only, namely the bitmap reading and writing modules. This algorithm computes center and surround 2D Gaussian convolutions for each of the eight levels. Half-wave rectification is performed on the two convolved images to produce a Difference of Gaussians for each level. Each level has a growing kernel stride, increasingly down-sampling the image as the level increases. The final output is eight Difference of Gaussian-filtered images comprising the Difference of Gaussian pyramid. Figure 7 shows a high-level block diagram of the overall algorithm data flow.

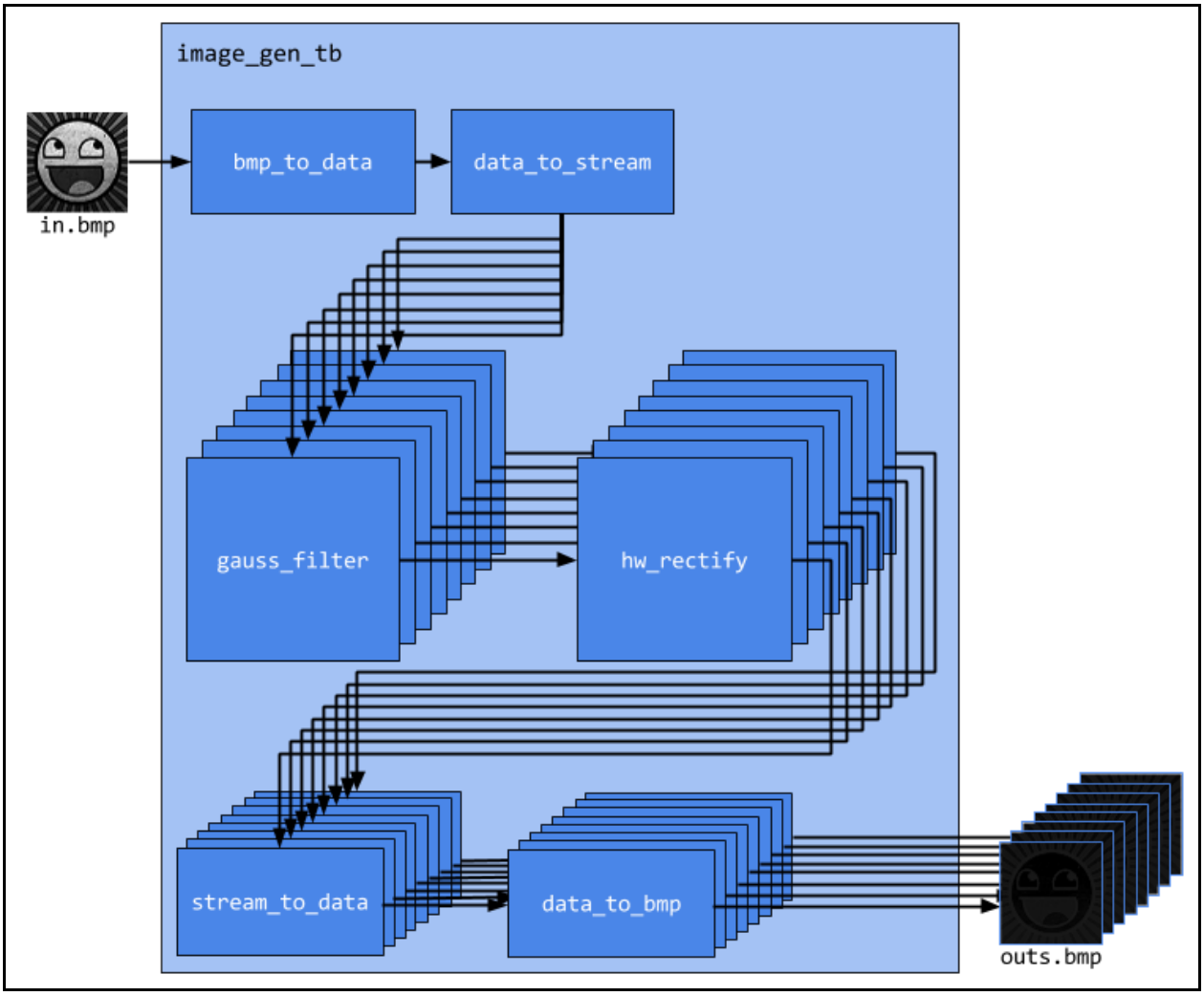

Figure 7: Data flow between modules in the parallelized convolution algorithm

Modules

The algorithm is split into individual parameterized modules, each running in parallel. Some modules, including the Gaussian filter and half-wave rectification modules, have multiple instantiations for each level or convolution. Because the modules are all running in parallel, pixels can be processed as quickly as they come in after an initialization period. All modules have a standard clock, reset, and enable input port. The clock and reset ports are common

between all modules, and the enable port is utilized differently depending on the module's functionality. The following sections detail each of the modules, describing their functions, parameters, and any ports they have in addition to the clock, reset, and enable ports.

Top-level module (image_gen_tb) (simulation)

The test bench serves as the top-level module. It instantiates all the modules and makes connections between their ports. For simulation purposes, it produces simulated clock, reset, enable signals to drive the simulation. It then initiates a simulated run, toggling the clock and asserting or de-asserting various signals at specified times. This simulation sequence is not included in the hardware implementation. More details of the simulation setup and process can be found in Chapter 4.

bmp_to_data.sv (simulation)

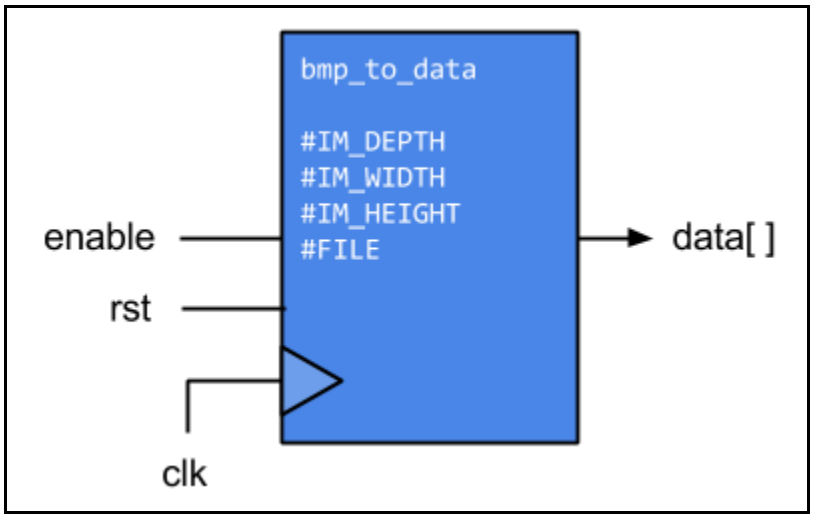

Figure 8: Ports and parameters of bmp_to_data.sv

This module provides sample image data from a bitmap file specified in the test bench. Purely for simulation purposes, it simplifies verification of the algorithm by giving realistic data on which to perform the image processing algorithm. It has a single output consisting of a 2D array containing the pixel data retrieved from the bitmap.

The filename of the bitmap is given as a parameter, in addition to the image's bit depth, width in pixels, and height in pixels. The file is retrieved using the SystemVerilog system function **`$fopen()`** in read-binary mode and its contents are placed into a memory register within the module using the system task **`$fread()`**. Once the data is retrieved, the binary information is parsed for pixel data. This module is written to parse 24-bit bitmaps without a color table and only saves the eight bits of the green channel. Table 1 shows the bit format that is used when parsing the binary file.

One consideration made when retrieving the pixel data is that the pixels are stored in a vertically-flipped order. That is, the first pixel entry in the binary file is the first pixel of the last row in the image. This module accounts for this storage pattern, and the first pixel stored is the pixel at the first column of the first row of the image. Subsequent entries are taken in a left-to-right, top-to-bottom order. The result is an array consisting of a total of width * height 8-bit pixels. This output array is connected to the input of *data_to_stream.sv*.

Offset (hexadecimal)	Size	Description
BMP Header		
0	2	ID field = “BM” = 0x424D
2	4	Size of file (unreliable)
6	2	Unused
8	2	Unused
A	4	Offset of pixel data = 0x00000036
Windows DIB Header		
E	4	Number of bytes in DIB header = 0x00000028
12	4	Width of image (in pixels)
16	4	Height of image (in pixels)
1A	2	Color planes used = 0x0001
1C	2	Bits per pixel
1E	4	Compression used = 0x00000000
22	4	Size of bitmap data including padding
26	4	Horizontal print resolution in pixels/meter
2A	4	Vertical print resolution in pixels/meter
2E	4	Number of colors in palette = 0x00000000
32	4	Number of important colors
Beginning of Pixel Data		
36+	3	Pixel data in BGR, starting from the last row and first column and moving horizontally

Table 1: Bit format for bitmaps parsed in simulation

data_to_stream.sv

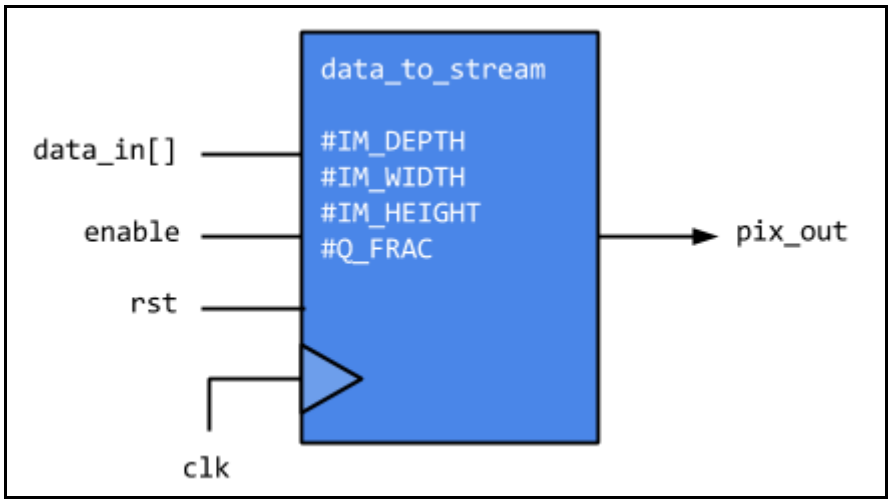

Figure 9: Ports and parameters for data_to_stream.sv

This module converts the 2D array containing the green-channel pixel values from **`bmp_to_data.sv`** to a stream of pixel data converted to fixed point. Essentially a parallel-in-serial-out shift register, it consists of an input for the 2D data array, an output for the pixel data, and a done signal for when the last entry in the input array is sent to the output. The module parameters include the image bit depth, width, and height, and fraction bits for the output pixel.

Pixels are converted to unsigned QM.N fixed point numbers before being streamed, where M and N are determined from the image bit depth and fraction bit module parameters, respectively. Because the input pixels are unsigned integers, the fractional bits will always be zero. Thus, the conversion is implemented simply by bit-shifting the pixel to the left by the specified N fractional bits.

gauss_filter.sv

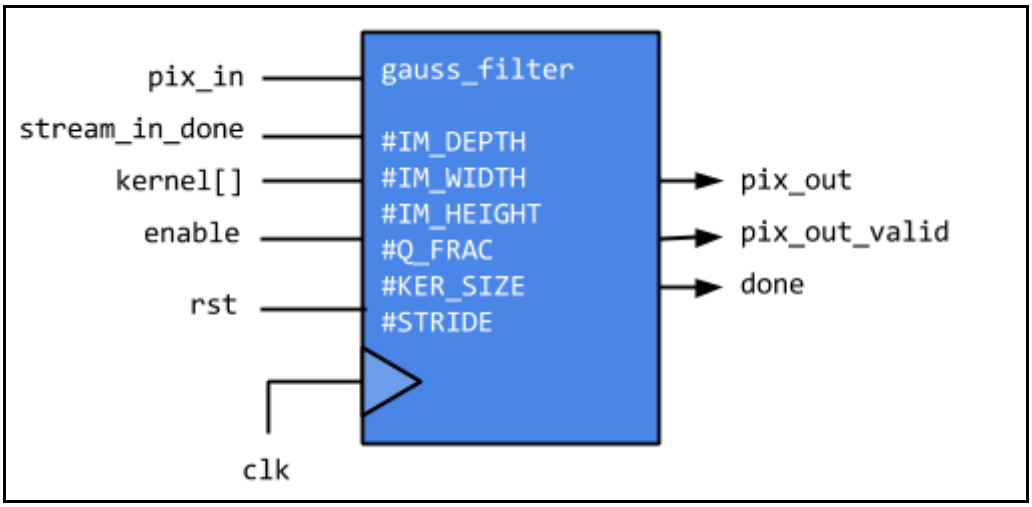

Figure 10: Ports and parameters for gauss_filter.sv

This module implements Gaussian filtering. To construct the Difference of Gaussians pyramid, two of these modules are instantiated for the center and surround kernels in each of the eight levels. The pixel stream from **`data_to_stream.sv`** is connected to the input of each module. The other inputs consist of the convolution kernel for the filter as well as the done signal from the **`data_to_stream.sv`** module. Its outputs are a stream of pixels convolved with the kernel, a done signal, and a pixel valid signal used for when the kernel stride is greater than one.

The module consists of three main blocks running concurrently: a pixel accumulator block, a kernel multiplication block, and a summing block. The pixel accumulator block handles the storage of pixels coming in from the pixel stream. As pixels come in, they are accumulated in a 1D array called **`pixel_store[]`**, whose capacity is initialized to be as small as possible. In this algorithm, the maximum number of pixels needed to be kept in storage at one time is equal to the image width multiplied by the kernel size. Figure 11 illustrates this concept.

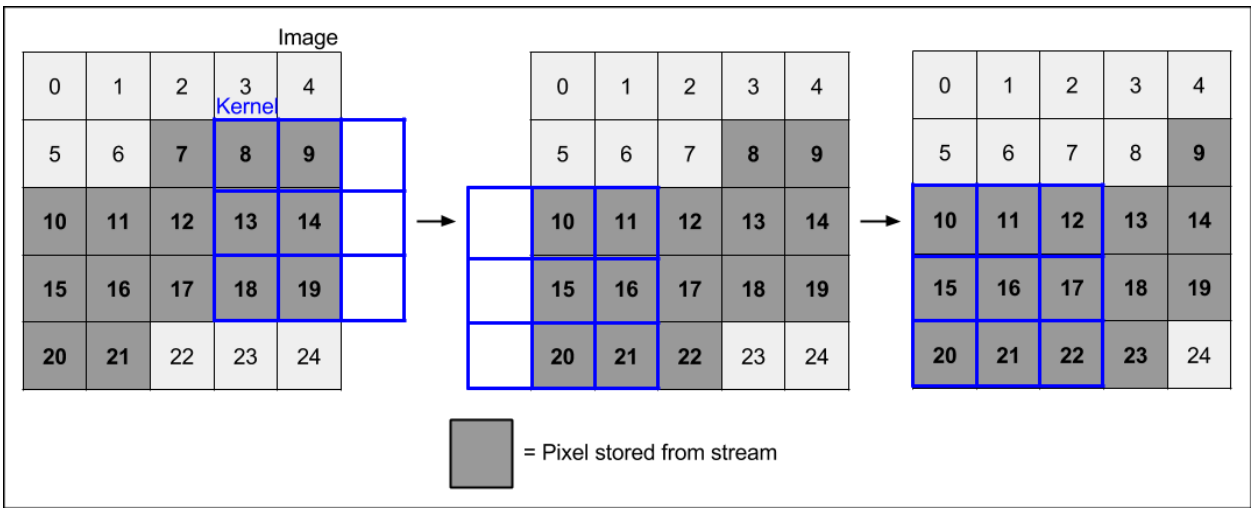

Figure 11: An example sequence of convolving a 5x5 image with a 3x3 kernel. With this streaming pixel scheme, only 15 pixels need to be stored at once.

When enough pixels are accumulated so that the kernel can operate on the first pixel, the block asserts a signal to enable the multiplication block. The number of pixels required to enable convolution on the first pixel is, like **`pixel_store[]`**'s capacity, a function of the image's width and the kernel's size. It is equal to **`IM_WIDTH*floor(KER_SIZE/2)+ceil(KER_SIZE/2)`**. When **`pixel_store[]`** is full, the oldest pixel (which is no longer needed by the kernel) is thrown out by shifting the entire contents. At the same time, a new pixel comes in from the pixel stream. This shifting occurs until the last pixel is streamed. Figure 12 shows the required contents of **`pixel_store[]`** for a 5x5 image convolved with a 3x3 kernel, and Figure 13 shows the accumulation of a 5x5 image for a 5x5 kernel convolution.

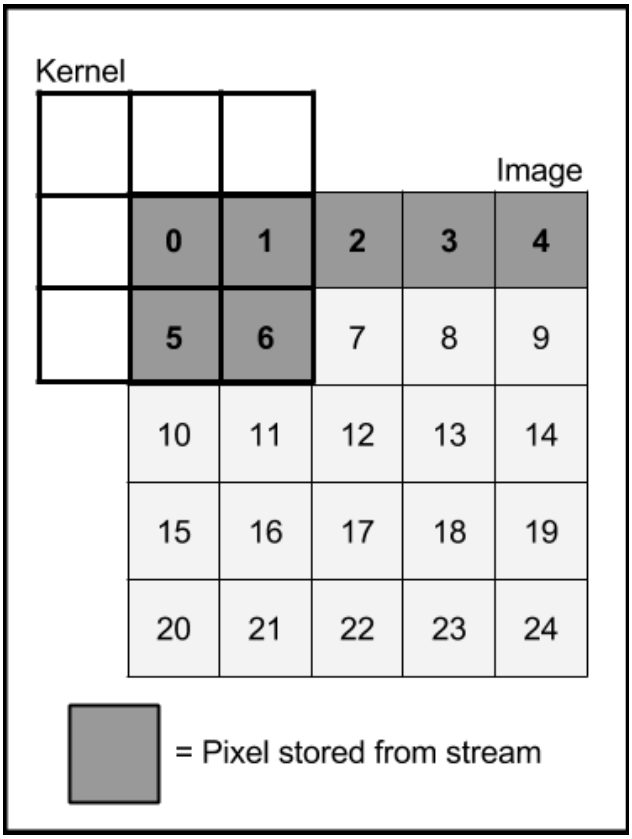

Figure 12: The first seven pixels are required to operate on the first pixel of a 5x5 image with a 3x3 kernel

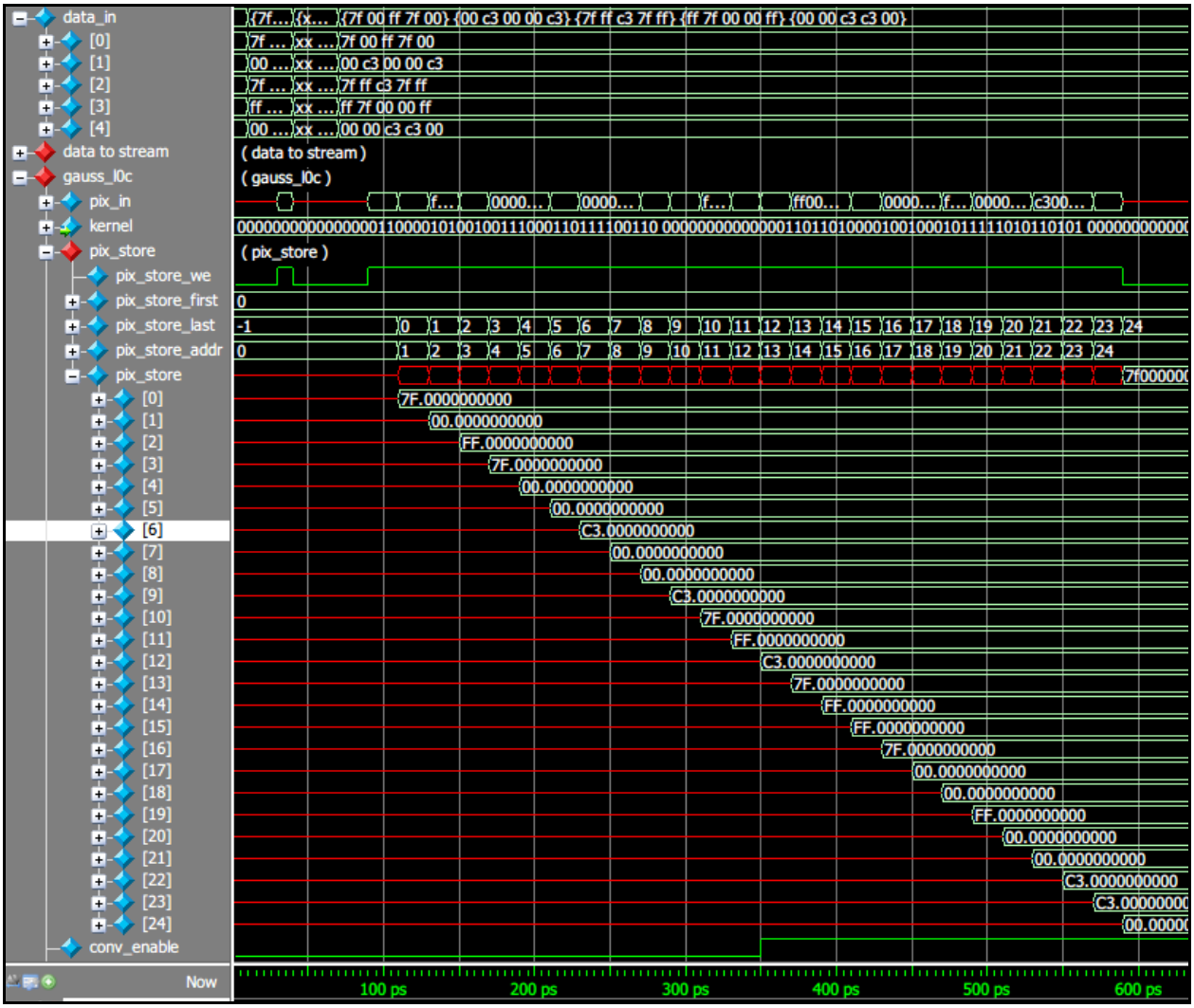

Figure 13: Accumulation of a 5x5 image for a 5x5 kernel convolution in ModelSim. Note that the `conv_enable` signal is asserted after the 12th bit is stored (as shown in Figure 15).

The multiplication block handles the convolution of each pixel with the kernel. The center of the kernel is initialized to the first pixel of the image and increments for each pixel in the image. For each entry in the kernel, the corresponding pixel on the image is determined and stored in a table called **`products[]`**. If the kernel entry corresponds to a location outside of the image, zero is used for the pixel (this has the effect of darkening the outer edge of the image). Running concurrently is the summation block, which sums all entries in **`products[]`** to give the

final pixel value. The final output is an unsigned fixed-point pixel value for the convolved image. Visually, the output of the filter produces a blurred version of the input image.

The kernel stride varies for each pyramid level in the Difference of Gaussian pyramid. It is calculated using the formula **`stride = 2^(level/2)`**. A kernel stride greater than one results in a down-sampled image. For example, a kernel stride of two halves the original image because only every other row and column is computed. Strides are implemented by asserting a pixel valid signal only if the pixel calculated is valid for that level's stride. Thus, a filter running with a kernel stride of one would output a valid signal constantly until all pixels are computed, but a stride of two would output a valid signal for every other row and column. This pixel valid signal, along with the output pixels is connected to the input of the half-wave rectification module for difference computing.

hw_rectify.sv

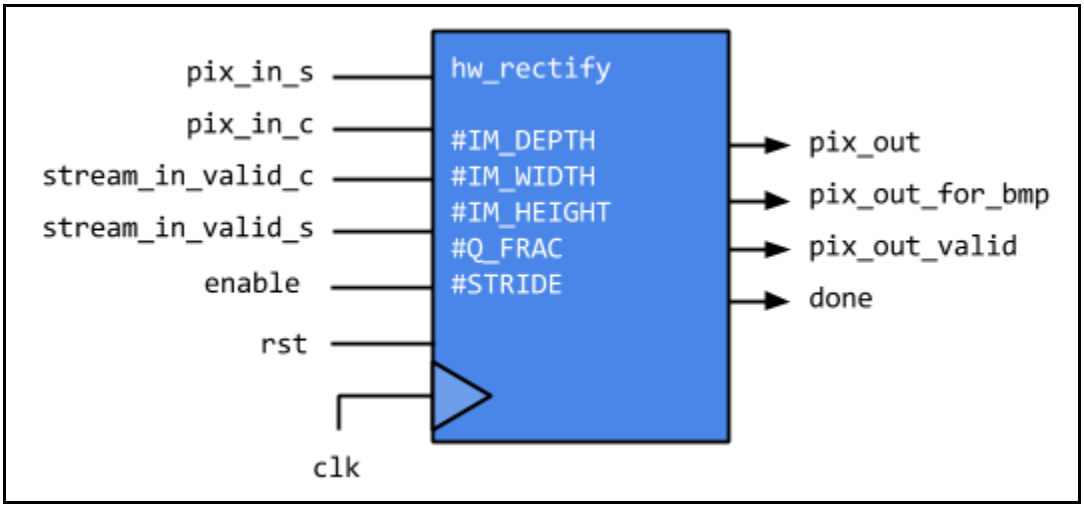

Figure 14: Ports and parameters for hw_rectify.sv

The hardware rectification module performs the Difference of Gaussians computation to produce a pixel stream of the final output image. One module is instantiated for each pyramid level. Again, the image bit depth, width, height, number of fractional bits, and kernel stride are given as parameters. Each module takes in the pixel stream produced from the center and surround kernel Gaussian filters within a level, in addition to their corresponding pixel valid signals.

Within each level, the surround kernel used in the Gaussian filter is always larger than the center kernel. Because of this, the center kernel does not need as many pixels stored the filter's pixel accumulator to begin convolution of the first pixel. Figure 15 illustrates this difference, which results in out-of-sync output pixel streams produced by the center and surround convolutions. The rectification module accounts for this difference by using a counter to keep track of how many pixels have been accumulated for each filter. When the first pixel of the surround filter is received from the filter module, the subtraction can begin.

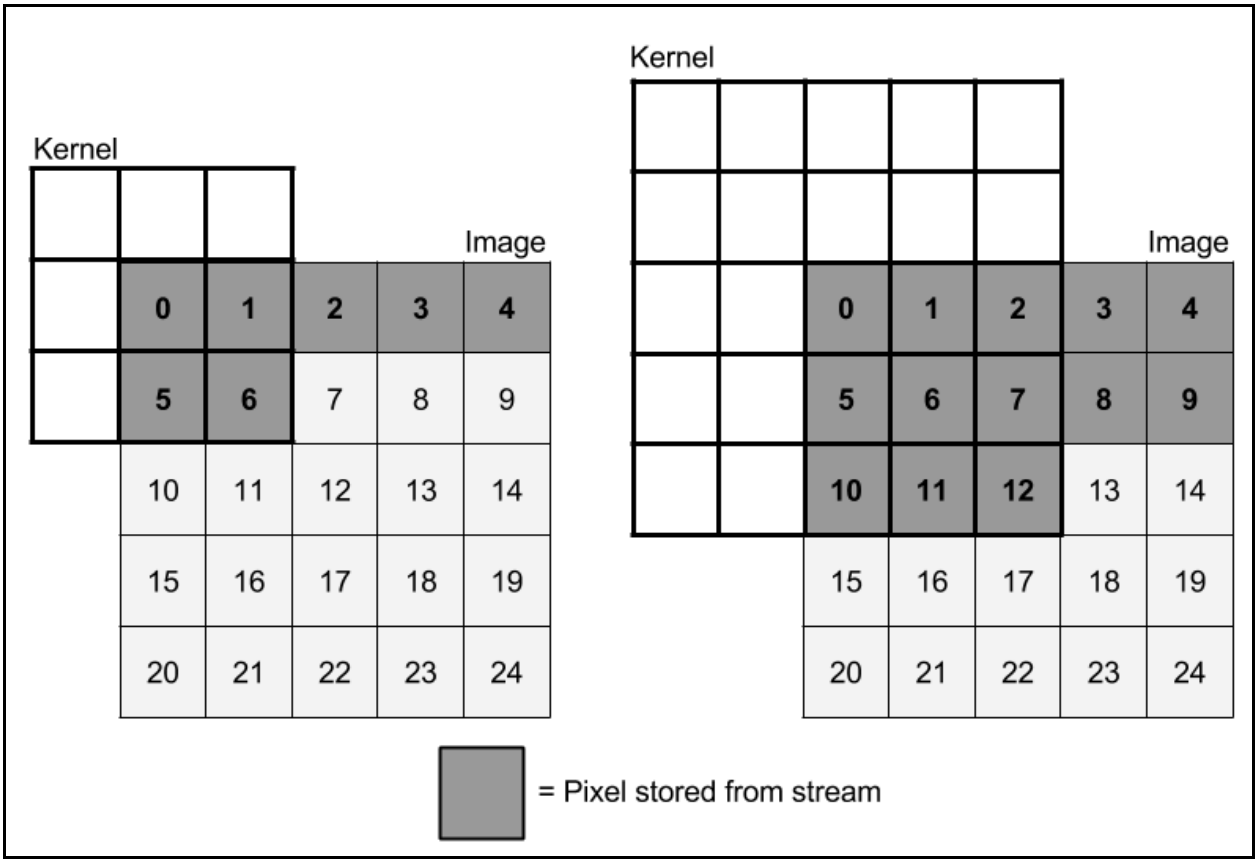

Figure 15: For the same image, a larger kernel size requires more pixels to be accumulated before the first pixel can be computed.

The outputs of this module consist of a pixel output stream in unsigned fixed point format specified by the **`Q_FRAC`** parameter along with a done signal. For kernel strides greater than one, a pixel valid signal is asserted if the computed pixel belongs in the down-sampled image. In addition, another pixel stream is outputted in integer format specified by **`IM_DEPTH`** so that the data can be converted and written to a bitmap file. The conversion is done by bit-shifting the computed difference to the right by **`Q_FRAC`** bits. When all pixels have been subtracted, the done signal is asserted.

stream_to_data.sv

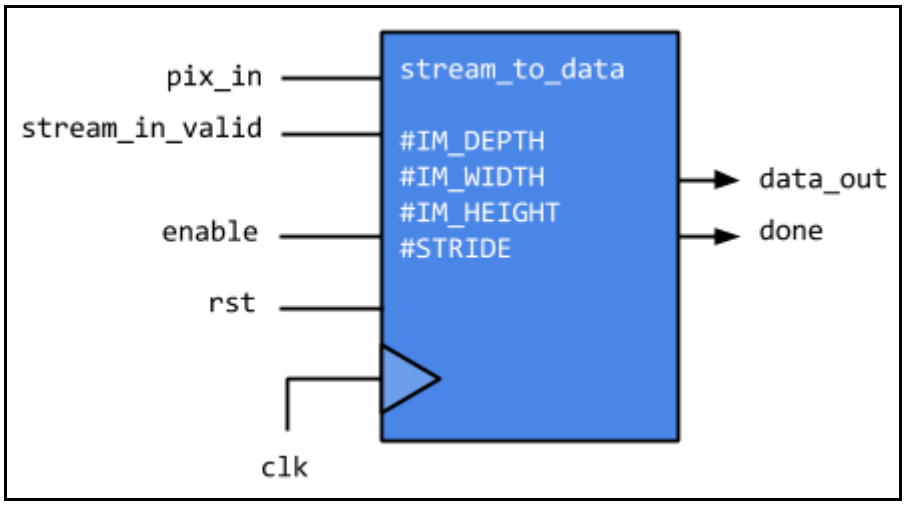

Figure 16: Ports and parameters for stream_to_data.sv

This module essentially reverses the functionality of **`data_to_stream.sv`**. Its inputs are the stream of integer pixel data from **`hw_rectify.sv`** and its valid signal signifying whether the pixel received is valid given the pixel stride. Parameters are passed for the image's depth, width, height, and the kernel stride used during convolution. Its outputs consist of a 2D array containing all the pixels streamed in and a done signal. In the design, it acts as a serial-in-parallel-out shift register, taking the integer pixel stream output from the **`hw_rectify.sv`** module and accumulating them into the 2D array. This 2D array is then sent to **`data_to_bmp.sv`** for writing to a bitmap file.

data_to_bmp.sv (simulation)

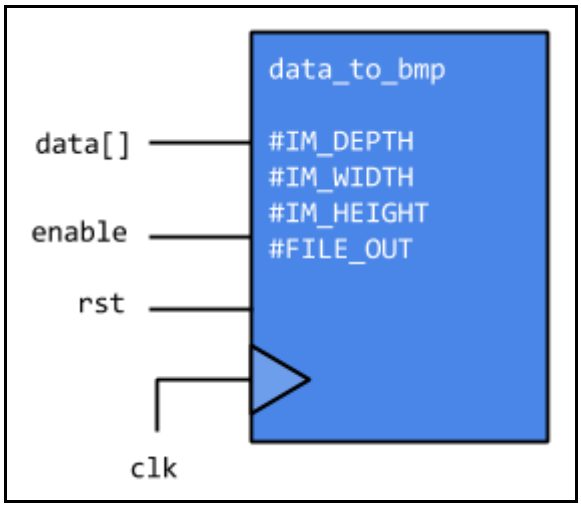

Figure 17: Ports and parameters for data_to_bmp.sv

For simulation purposes, this module takes in the 2D array of the convolved and rectified pixel values and writes it to a viewable bitmap file. The input to this module is the 2D array of pixel data, and parameters are taken for the image's bit depth, width, and height. To write the file, it constructs both the bitmap and DIB headers as shown in Table 1. The image size and pixel data size is calculated from the given width and height parameters. When writing pixel data to the file, the same value is written to the red, green, and blue channel to produce a grayscale image.

Chapter 4

Simulation & Results

ModelSim Simulation Setup

To verify the functionality of the Difference of Gaussians implementation, the SystemVerilog source files are compiled and run in ModelSim Altera Starter Edition 10.1e. The test bench serves as the top-level module in simulation and contains the module instantiations as well as the simulation sequence to be run.

Test Bench Module Instantiations

For simulation purposes, a single instantiation of the **`bmp_to_data`** module and **`data_to_stream`** module is performed. This serves to retrieve sample data from a bitmap residing in the simulation's path. To instantiate the necessary modules for each level, a **`generate`** loop is used. In the loop, a Gaussian filter module is instantiated for the center kernel and the surround kernel. A half-wave rectification module is also instantiated to take the difference between the two filters, creating one Difference of Gaussian pixel stream for each pyramid level. Finally, a **`stream_to_data`** and **`data_to_bmp`** module is instantiated to write the output of a chosen level back to a bitmap image.

Test Bench Module Simulation Sequence

The simulation sequence runs in a **`while()`** loop, toggling the clock signal on every iteration and incrementing a counter to keep track of the number of cycles executed. After ensuring the image input's width and height are greater than the largest kernel size used (in this design this value is 47), it initiates a **`read_bmp_enable`** signal to enable the **`bmp_to_data`** module. This triggers the **`data_to_stream`** module to begin streaming pixels. Once pixels begin streaming, the Gaussian filter modules initialize and, after an initial period of accumulating pixels, begin streaming the convolved output pixels. Meanwhile, the half-wave rectification module reads the Gaussian filter pixel stream and performs the differencing. Once all **`done`** signals for the half-wave rectification module in each level are asserted, the image is finished processing and the simulation terminates.

During this simulation sequence, a conditional block is written that monitors the **`done`** signal of one of the half-wave rectification modules. When this signal is asserted, the conditional block asserts a **`write_bmp_enable`** signal and the **`data_to_bmp`** module writes the output of the chosen pyramid level to a file. Multiple instantiations of this conditional block and the relevant modules can be made to output some or all of the levels to bitmap files.

Simulation Results

The performance metric used to measure this design is simulated clock cycles. The simulation runs on Modelsim at a resolution of 1 picosecond. While the test bench executes the simulation sequence, it toggles the clock every 10 picoseconds. During the simulation run for

one bitmap file, the number of clock cycles is recorded upon completion of each level. Table 2 and Figure 18 show the results of three simulation runs for images of varying dimensions.

Level	Cycles taken to complete						
	48x48	64x64	72x72	100x100	128x128	256x256	512x512
0	2358	4166	5262	10106	16518	65798	262662
1	2358	4166	5262	10106	16518	65798	262662
2	2358	4166	5262	10106	16518	65798	262662
3	2407	4231	5335	10207	16647	66055	263175
4	2407	4231	5335	10207	16647	66055	263175
5	2505	4361	5481	10409	16905	66569	264201
6	2505	4361	5481	10409	16905	66569	264201
7	2750	4686	5846	10422	17550	67854	266766
8	2701	4621	5772	10329	17421	67597	266253

Table 2: Simulation cycles taken to complete each level of the Difference of Gaussians pyramid for images of various sizes.

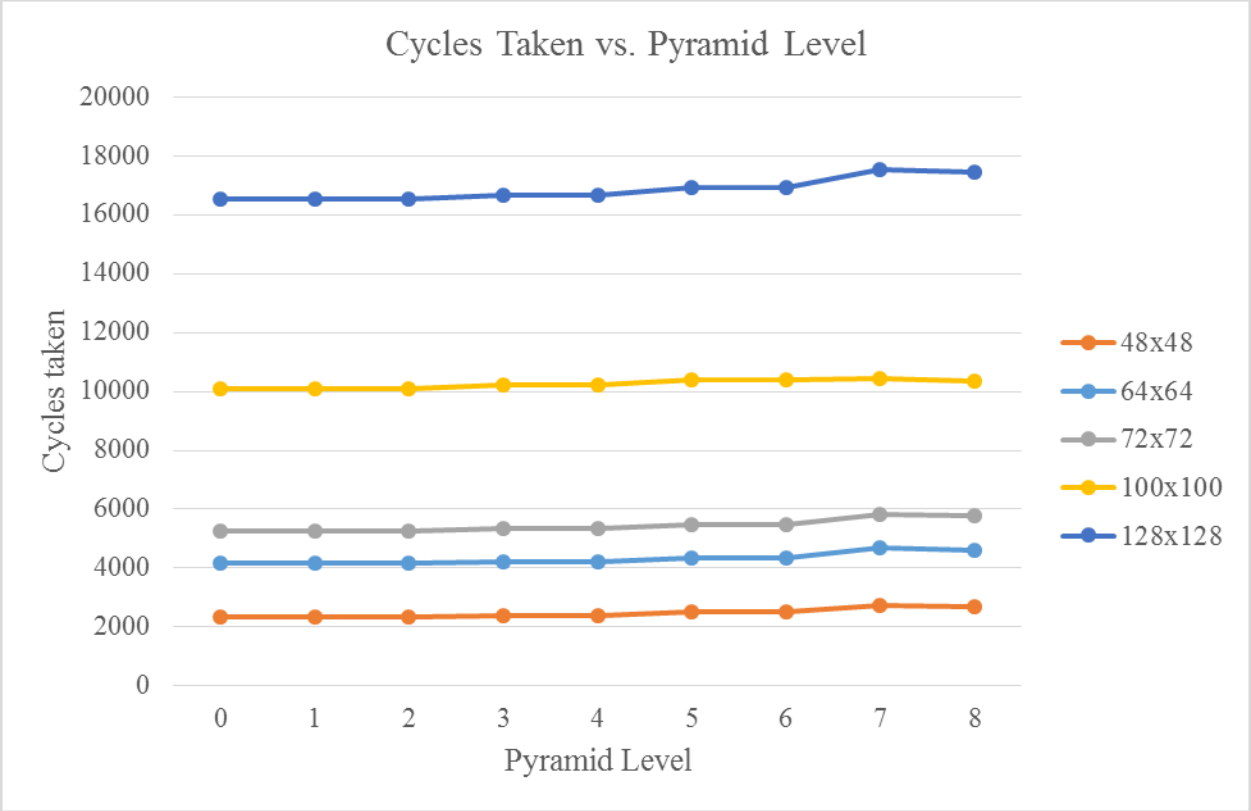

Figure 18: Graph of Table 2 (up to 128x128)

Because both the kernel stride and the kernel size generally increase for increasing pyramid levels, the number of cycles taken to complete each level is relatively constant for a given image. In fact, the eighth level completes before the seventh because the kernel stride is increased to 2^(8/2) = 16. Thus, increasing the number of levels to compute in the Difference of Gaussians pyramid does not have much impact the performance on the algorithm. Additionally, performance scales proportionally to the total number of pixels, as can be seen in Figure 19.

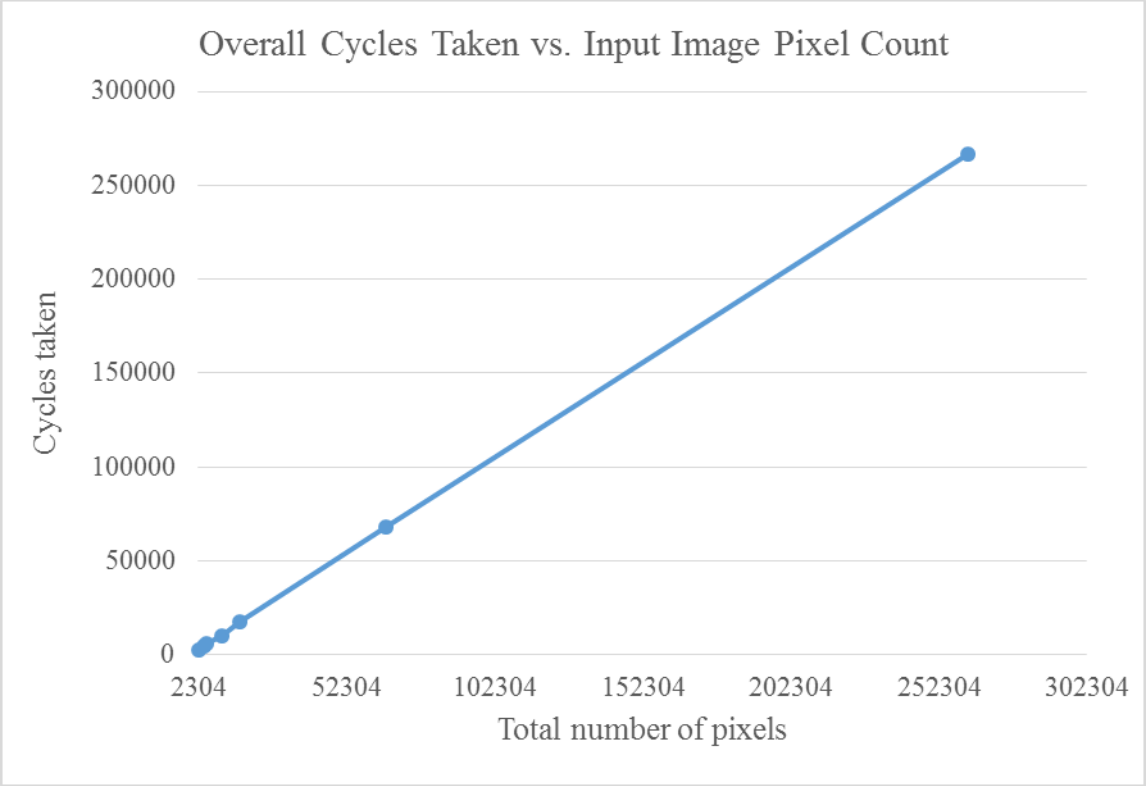

Figure 19: The total number of cycles taken to complete the entire Difference of Gaussians of varying dimensions.

Chapter 5

Conclusions

In this project, a parallelized, multi-resolution Difference of Gaussians Pyramid algorithm is implemented using SystemVerilog. Given a stream of pixel data, it successfully applies two Gaussian filters per level, one for a center convolution and another for the surround. The Difference of Gaussian Pyramid outputs are then successfully calculated by half-wave rectification. The final output is a pixel stream for each of the convolved images. This data can then be used to compute visual saliency of the original image.

Simulation results show that the parallelization used in the algorithm allows it to operate with high efficiency, producing an output pixel as quickly as an input pixel comes in, after some initialization time. Additionally, increasing the number of levels to be computed does not have a high impact on the algorithm's performance. The parallel structure of FPGAs can be effectively utilized using this algorithm.

When designing this algorithm, special consideration was given to parametrization. Throughout the design, the input image bit depth, width, height, Q format, number of levels, kernel sizes, and kernel sizes are all implemented as parameters. This makes the design highly modular. For example, any kernel can be used as an input to **`gauss_filter.sv`**. Additionally, the **`generate`** loop used in the top-level module makes adding or removing pyramid levels a trivial task. Splitting the design into individual modules also makes the structure of the design easy to visualize and understand.

Future Work

The algorithm is targeted to run on a cluster of PCIe boards containing high-performance Altera FPGA devices. Given the high rate of data being processed by the algorithm, interfaces and devices like PCI Express and QDRII + are chosen for use on the target hardware. Though the Difference of Gaussians algorithm is written with synthesizability in mind, implementing the design on actual hardware still needs to be performed and characterized.

The next logical step for this design is to generate the hardware for each of the Qsys components in the Altera Quartus environment, including the PCI Express DMA design, the QDRII+ SRAM controllers, and the Difference of Gaussians algorithm. The hardware components are then to be instantiated in a top-level module in Quartus and is compiled for the FPGA device on the board.

Further optimizations can be made on the design to adapt it for clustering of FPGAs. For example, each level can be split between FPGA devices, having one device dedicated to the computation of a single level of the Difference of Gaussians pyramid. Fully utilizing the resources made available by BittWare on its PCI Express FPGA boards can be explored in further detail.

WORKS CITED

Brown, Stephen, and Jonathan Rose. "Architecture of FPGAs and CPLDs: A Tutorial." *Architecture of FPGAs and CPLDs: A Tutorial* (n.d.): n. pag. *Department of Electrical and Computer Engineering*. University of Toronto, 12 Mar. 2000. Web. 9 Apr. 2015.

Delp, Edward J., and Leah J. Siegel. "Parallel Processing for Computer Vision." SPIE 336 (1982): 161-67. College of Engineering. Colorado State University. Web. 28 Jan. 2015.

Fine, Michael S., and Brandon S. Minnery. "Visual Salience Affects Performance in a Working Memory Task." *The Journal of Neuroscience* 29.25 (2009): 8016-021. The MITRE Corporation, 24 June 2009. Web. 29 Mar. 2015.

"FPGA (Field-Programmable Gate Array) Market Analysis By Application (Automotive, Consumer Electronics, Data Processing, Industrial, Military And Aerospace, Telecom) And Segment Forecasts To 2020." *Global FPGA Market Analysis And Segment Forecasts To 2020*. Grand View Research, May 2014. Web. 09 Apr. 2015.

Frintrop, Simon et al. (2010). "Computational Visual Attention Systems and their Cognitive Foundation: A Survey". ACM Transactions on Applied Perception (TAP), 7 (1).

"History of FPGAs." *Field Programmable Gate Array Chips: History*. Wayback Machine Internet Archive, 12 Apr. 2007. Web. 09 Apr. 2015.

Itti, Laurent. "Visual Salience." Scholarpedia. University of Southern California, Los Angeles, 2007. Web. 28 Jan. 2015.

Itti, Laurent, Christof Koch, and Ernst Niebur. "A Model of Saliency-Based Visual Attention for Rapid Scene Analysis." *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE* 20.11 (1998): 1254-259. *Department of Cognitive and Neural Systems*. Boston University. Web. 29 Mar. 2015.

Judd, Tilke, Frédo Durand, and Antonio Torralba. "A Benchmark of Computational Models of Saliency to Predict Human Fixations." *DSpace@MIT*. MIT Computer Science and Artificial Intelligence Laboratory, 1 Jan. 2013. Web. 29 Mar. 2015.

Lu, Shih-Lien. "The FPGA Paradigm." *Shih-Lien Lu' Home Page*. Oregon State University, 24 Sept. 2001. Web. 09 Apr. 2015.

Maxfield, Clive. "Xilinx Unveil Revolutionary 65nm FPGA Architecture: The Virtex-5 Family." *EE Times*. UBM Tech, 15 May 2008. Web. 09 Apr. 2015.

"Stratix V FPGAs." *Stratix V FPGAs*. Altera Corporation, n.d. Web. 09 Apr. 2015.

"This Month in Physics History." *This Month in Physics History*. Ed. Alan Chodos. American Physical Society, Nov. 2000. Web. 09 Apr. 2015.

ACADEMIC VITA

Brian Yu

EDUCATION

Bachelor of Science, Electrical Engineering — exp. May 2015
The Pennsylvania State University — University Park, PA
Schreyer Honors College | Dean's List (2012 – present)

Relevant Courses	Computer Networks	FPGA Design (Altera SoC & SystemVerilog)
	C++ & MATLAB	Computer Organization (MIPS & Verilog)
	Signals & Systems	Digital Logic Design (ABEL-HDL)
	Electromagnetics	LabVIEW & Microcontrollers

WORK EXPERIENCE

Electrical Engineering Intern, Motorola Solutions, Inc. — Schaumburg, IL
- Researched, proposed, and implemented a suite of tests for a software-defined radio module consisting of Xilinx FPGA SoC, mobile DDR, and RFIC. Tests were configurable for engineering design verification and production functional testing. Wrote programs in C and Verilog using the Xilinx SDK and Vivado. — May – Aug 2014

Electrical Systems Engineering Co-op, Gulfstream Aerospace Corp. — Savannah, GA
- Conducted systems testing and assisted FAA certification testing for Gulfstream's PlaneView avionics suite, including communication, navigation, and flight control systems. Performed ADS-B and SATCOM testing in lab facilities and on aircraft. — Aug – Dec 2013

Software Intern, Bentley Systems, Inc. — Exton, PA
- Created parametric 3D modeling applications in MicroStation with WinForms and C++/CLI. Used Microsoft Visual Studio 2010 with Mercurial package control. — Jun – Jul 2013
- Designed a collaboration website using SharePoint Designer and JavaScript. — Jul – Aug 2011
- Collected external financial statistics from websites and databases, using Microsoft Access and Excel to organize and share those statistics. — Jun – Aug 2010, Jul – Aug 2009

SKILLS & CERTIFICATION

- Certified LabVIEW Associate Developer (NI CLAD) — April 2013
- Proficient in Xilinx and Altera FPGA toolchains; Microsoft Office, Visual Studio, and SharePoint; Multisim; Altium; and Adobe Flash
- Experienced with C & C++, SystemVerilog, MATLAB, Lua, and HTML
- Exposed to working in a Linux environment and Android development

PROJECTS

Undergraduate Researcher, Applied Research Laboratory — Sep 2014 – present

- Writing image convolution algorithms on large-scale Altera FPGA clusters for use in real-time machine vision applications.

"Giftronome", personal project — Apr 2014 – present

- Wrote an applet that loops image sequences to a tempo based on playing music. Used Lua within the LÖVE 2D game engine along with Last.fm and Echo Nest APIs. Currently working on a port to Android devices.

frisbee.co.nr, personal project — May – Jun 2010

- Designed and maintained a website for high school's Ultimate Frisbee league. Used Flash, Javascript, and HTML. Implemented a text-based version for mobile.

ACTIVITIES

- Penn State Amateur Radio Club (Fall 2012 President) — 2011 – 2013
- Penn State Student Space Programs Laboratory (Member) — 2012 – 2013
- Penn State Campus Orchestra (Violinist) — 2012 – 2014