FFT accelerated sobel edge detection on Cyclone V SoC

by

Peter Nguyen

B.S., Kansas State University, 2023

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Mike Wiegers Department of Electrical and Computer Engineering
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2024

Approved by:

Major Professor
Don. M Gruenbacher

Copyright

© Peter Nguyen 2024.

Abstract

With the increase of processing power in parallel computing, there has been a massive amount of research being done on implementing machine learning and artificial intelligence in real systems. This implementation has led to the development of computer vision systems. Computer vision systems are systems that utilize machine learning and neural networks to help computers understand the visual world. Neural networks are a subset of machine learning that aims to mimic the human brain in computers. Computer vision systems typically use Graphics Processing Units to accomplish this task. In computer vision systems, the most typical neural networks used are Convolutional Neural Networks (CNNs). CNNs utilize convolution in two-dimensional space to help computers identify important characteristics in images and videos, such as humans, objects, edges, etc.

This thesis focuses on two main aspects of computer vision systems. The first focus is on proving that convolutions done in CNNs can be replaced with a frequency transformation for a potential decrease in computation time. The second focus is on creating a fully portable System on Chip-Field Programmable Gate Array (SoC–FPGA) design that will implement the frequency transformation-based CNNs. This thesis accomplishes these two goals through a SoC-FPGA design that applies frequency transformation-based convolution, resulting in images identical to traditional two-dimensional convolution with performance similar to current generation CPUs.

Table of Contents

List of Figures

List of Tables

Acknowledgements

The author would like to thank Dr. Gruenbacher for the opportunity to pursue this thesis and for providing his time and knowledge. The author would also like to thank committee members Dr. Guo and Dr. Scoglio for providing their time and support.

Chapter 1 - Introduction

1.1 Motivation

The concept of machine learning and artificial intelligence has been around since as early as the 1960s [1]. The bottleneck back then was the limited technology that prevented researchers from implementing the ideas into real systems [1]. With the recent exponential advancement of processing power in central processing units (CPUs) and graphical processing units (GPU's), the ability to implement the concepts of machine learning is now possible. Machine learning is mainly implemented by utilizing a multitude of GPUs that perform all the necessary calculations. This process proves to be very power-intensive and would be unviable in a low-power setting, such as for use in mobile devices. One possible solution to implementing machine learning in low-power systems is to utilize a Field Programmable Gate Array (FPGA) device. An FPGA is an integrated circuit that can be reconfigured, which provides developers with the ability to create multiple prototypes of a design using one device. FPGAs have the capability of doing parallel calculations like GPUs with the benefit of less power consumption [2].

1.2 Prior Work

A Convolutional Neural Network (CNN) is a deep neural network used for image processing systems. Specifically, CNNs utilize two-dimensional convolutions to determine characteristics of an image, such as edges and borders, and send those characteristics to a fully connected layer for the network to apply linear regression to obtain the desired result. CNNs are heavily used by developers to create computer vision systems. As images get larger, the computation needed for two-dimensional convolutions gets larger, resulting in longer processing time. As shown in [3], a Fast Fourier Transform (FFT) can be used in place of two-dimensional convolutions for faster computations. This is because FFTs can implement the convolution

through matrix elementwise multiplication. Multiple papers have been published showing how FFT based convolutions can be used to speed up CNNs on multiple platforms. The rest of this section will go over prior work, which provides clear results showing improvements in FFT-based CNNs over traditional CNNs.

In the first paper, Nair et al. [4] focus on implementing an FFT into a U-Net. U-Net is a CNN proposed by Ronneberger et al. [5] for use in biomedical image segmentation. In terms of speed, the conclusion made by Nair et al. [4] states that training time was improved by approximately 30%. The accuracy (intersection over union was used) showed itself to be much better, with a 36% improvement. Nair et al. [4] have contributed by showing how they were able to improve speed and accuracy by implementing an FFT into a fully convolutional neural network.

Liang et al. [6] proposed an FFT algorithm for speeding up convolution layers in many well-known CNN architectures that are faster than the state-of-the-art (paper released in 2019). The architecture used for the FFT algorithm is built on the overlap-and-save method. This method involves overlapping tiles on the input when applying the FFT, then separating and saving the outputs. Compared to overlap and add, where the input is split into tiles without overlap and then overlapped and added at the output stage, the author states overlap-and-add has disadvantages in terms of memory usage, as the overlap in the output layer results in shared addresses when adding. This leads to data dependency issues. Liang et al. [6] then outline the main architecture used, which focuses on how to efficiently move the input and filter data, as on-chip memory is incapable of storing it all. The FFT is applied in the processing engine design. The difference in Liang et al. [6] implementation of the FFT compared to others is how Liang et

al. [6] use the Hermitian symmetry of the inputs and outputs to reduce memory needs and the number of multiplications needed.

Abtahi et al. [7] focus on comparing CNNs with FFT-based CNNs on multiple platforms. The comparison between direct convolution and FFT convolution is done on the Xilinx Zynq 7020 FPGA using SPARCNet accelerator. The SPARCNet accelerator, proposed by Page et al. [8], is a hardware accelerator used to deploy Sparse Convolutional Neural Networks (SPARCNet). SPARCNet was created to be used in situations where platforms require low power usage [8]. Results for the FPGA implementation show that the FPGA using FFT convolution is faster than doing direct convolution and more energy efficient. For the CPU implementation, the ARM Cortex-A53 (found in RPi3B) was used. Abtahi et al. [7] implement direct convolution and FFT convolution on the CPU platforms and conclude that FFT-convolution is faster than direct convolution. The final platform tested by Abtahi et al. [7] was the NVIDIA TX1 Jetson GPU. Abtahi et al. [7] run the same experiment, implementing direct convolution and FFT convolution and comparing the results. In all three cases, Abtahi et al. [7] prove that there exist FFT convolution algorithms that will run faster than direct convolution.

The three papers above prove that FFT-based CNNs can be faster than traditional CNNs. The other aspect of this thesis deals with implementing FFT-based CNNs on low power hardware. Typically, neural networks have been implemented on GPUs for their ability to do parallel calculations, leading to faster computation times than CPUs. One issue with GPUs is that they are very power-intensive. For fields that utilize low-power mobile platforms, it is typically infeasible to use GPUs [9]. This leads to a need for an alternative platform. This is where FPGAs become viable. As stated in the previous section, FPGAs are capable of parallel calculations that GPUs can do with the added benefit of using less power [2].

1.3 Main Contributions

The work documented in this thesis made two contributions to the field of computer vision.

1. Provide a portable hardware accelerator design for FFT-based convolution for low-cost Altera SoC-FPGA devices.
2. Prove that computation time for CNNs can be decreased by replacing convolution with FFT-based convolution.

1.4 Thesis Organization

This thesis is organized into five chapters. This first chapter focused on the main motivation for this thesis, an examination of prior work, the main contributions of this thesis to advancing computer vision, and the organization of this thesis. Chapter two provides descriptions of convolutions and Fourier transformations and how they are applied in two-dimensional space. The descriptions are provided to help readers refresh their memory of those concepts. This chapter also describes bilinear interpolation, a concept that it necessary to understand an important aspect of the SoC FPGA design. Chapter three introduces the proposed SoC FPGA design, with details on the hardware and software architectures. Chapter four discusses the complexity between CNN and FFT with MATLAB and the results obtained from the proposed SoC FPGA design compared to MATLAB. Chapter four also includes an analysis of system timing, ways that timing can be improved, and alternative platforms that could be utilized. Chapter five ends with the conclusion and future work.

Chapter 2 - Background Knowledge

This chapter focuses on providing the necessary background knowledge to understand what this thesis is trying to accomplish. The math involving two-dimensional convolution and two-dimensional Fourier transformations will be reviewed. The theory behind CNNs will also be explained along with the Fourier Transform based CNN. Finally, the last concept that will be briefly discussed is Bilinear Interpolation.

2.1 Two-Dimensional Convolution

Convolution in one-dimensional space involves multiplying two functions and finding the integral of the given product. For multiplication, one function is flipped across the y-axis and then shifted slowly across the other function. The flip is always applied to the sliding function to ensure that the beginning of each function overlaps. Mathematically, the equation for convolution is the following:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(x)g(t-x)dx$$

where *t* represents the current interval in the convolution. Figure 2.1 provides a visual explanation of the process.

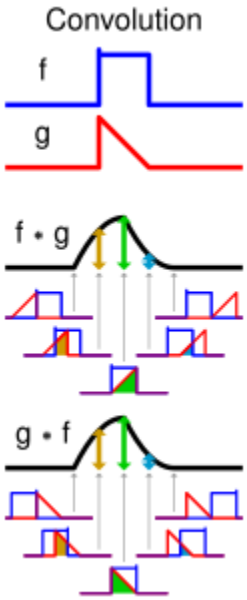

Figure 2.1 Visual representation of one-dimensional convolution [10]

This thesis focuses on how two-dimensional convolution is applied to images. Given an image (input matrix), a kernel (odd-dimension square matrix, typically 3x3) slides over the original image, multiplying overlapping values, and then summing all the products. The two-dimensional convolution usually starts with the center of the kernel overlapping with the top left corner of the image. The kernel then slides over the image horizontally, one pixel at a time. Once the center of the kernel overlaps with the top right corner of the image, the kernel then moves back to the left side of the image and down one pixel vertically, repeating the process, moving one full row at a time and then moving down one pixel until the center of the kernel has reached every pixel in the image. Figure 2.2 gives a visual explanation of this process.

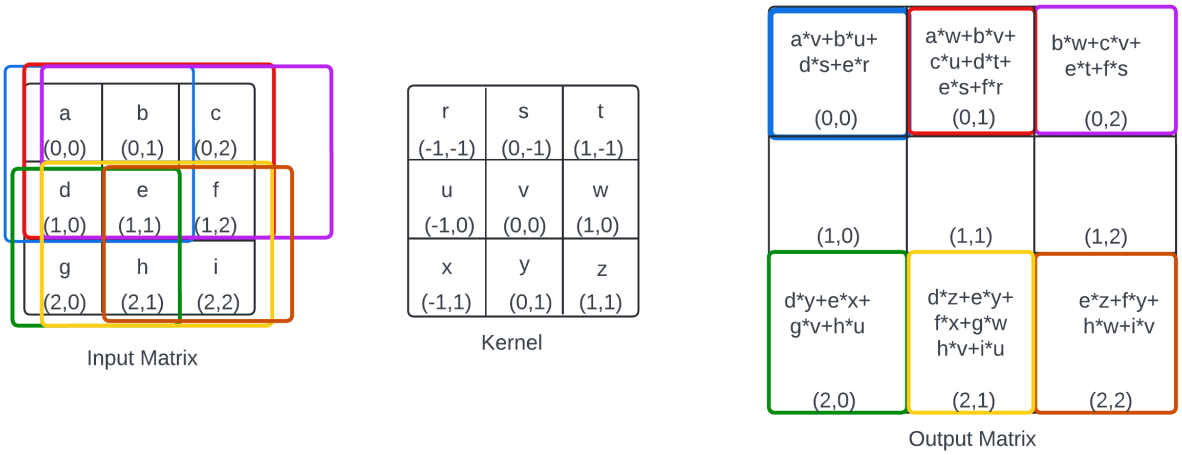

Figure 2.2 Visual representation of two-dimensional convolution

Mathematically, the equation for two-dimensional convolution is:

$$y[i,j] = \sum_{m=-1}^{1} \sum_{n=-1}^{1} h[m,n] \cdot x[i-m, j-n]$$

where y is the output matrix, h is the kernel, x is the input matrix, i and j are the index for the output matrix, and m and n represent the index scheme for the kernel where the center is (0,0) as illustrated in Figure 2.1. In the equation, function x has shifts in the i and j positions to account for the kernel in Figure 2.2 needing to be flipped to hold true to convolution. The order of complexity for the two-dimensional convolution is:

$$O(M * N * k^2)$$

where M and N are the size of the input matrix and k is the size of the kernel.

2.2 Two-Dimensional Fourier Transform

When applying filters to images through convolution, larger image sizes and larger filter sizes will increase the time complexity of convolution. Convolution in the time or spatial domain can be optimized by processing the function in the frequency domain, a procedure called fast

convolution. Convolution in the time domain corresponds to multiplication in the frequency domain. The Discrete Fourier Transform (DFT) is used to transform the function from time domain to frequency domain. The DFT is a mathematical technique used to transform a time domain function into a frequency domain function. By applying the Fourier transform to an image and a filter, multiplication is carried out instead of convolution. This can result in quicker calculations, leading to a decrease in processing time in the convolution layer of the Convolutional Neural Network. The Fast Fourier Transform (FFT) is an optimized algorithm to speed up the computation done by the DFT and is used in this paper. Applying the FFT to a two-dimensional space (an image) is a straightforward process. First, the FFT is applied to the rows of the image. After the first FFT is complete, a second FFT is applied to the columns of the image resulting from the first FFT. This process is also applied to the filter. Once both the image and the filter have been transformed, elementwise multiplication is performed between them. The last step to get the final image is to apply a two-dimensional inverse FFT. This process is done the same way as the two-dimensional FFT. The process for both applying an FFT and Inverse FFT (IFFT) to an image and kernel will be referred to as the Frequency Transform.

One important note in applying two-dimensional FFTs to images is the necessary use of padding. The convolution used in the CNN applies zero padding around the borders. This padding is used to prevent the convolution from wrapping around the image, leading to distortion. In terms of the FFT, when taking the FFT of the image, it is necessary to pad the image with zeros to prevent distortion. Another important aspect when applying FFT-based convolution for images is shifting the kernel. Before applying the FFT to the kernel, the kernel needs to be shifted so that the center pixel (0,0) is in the top left corner. The image of the kernel was shown previously in Figure 2.2. This is to mimic how the convolution process starts with the

center of the kernel being placed onto the top left value of the input matrix. The kernel also needs to be padded with zeros to match the size of the input matrix or else elementwise multiplication will not work due to mismatched matrix sizes. Figure 2.3 provides a visual for how two-dimensional FFT and two-dimensional inverse FFT are applied in this thesis.

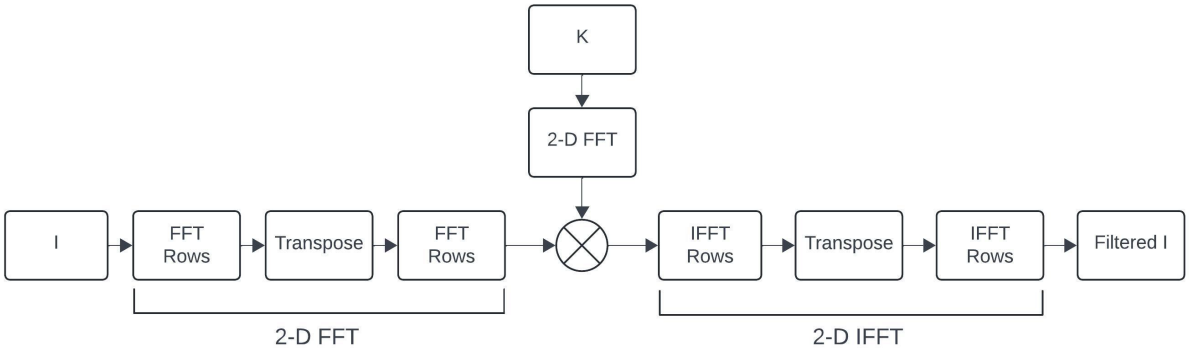

Figure 2.3 Frequency Transform

The equation for the two-dimensional FFT is:

$$Y_{p+1,q+1} = \sum_{j=0}^{M-1} \sum_{k=0}^{N-1} e^{-2\pi ijp/M} e^{-2\pi ikq/N} X_{j+1,k+1}$$

where M and N are the dimensions of matrix X, p and q are indices that run from 0 to M-1, and q and k are indices that run from 0 to N-1,

The order of complexity for the two-dimensional FFT is:

$$O(MN \log(MN) + MN)$$

where M and N are the size of the input matrix.

2.3 CNN

The main purpose of this thesis was to accelerate the convolutional layer of CNNs. With that in mind, it was only necessary to understand how convolution works in two-dimensional

space. However, CNN will briefly be described here to give readers a better idea of how this thesis can be integrated into the full system.

CNN is a deep learning algorithm used to train a system to identify or recognize notable features in an image. CNNs accomplish this by taking in images, applying multiple rounds of convolution and pooling layers, and then sending the data to a fully connected layer. Figure 2.4 shows the main flow of CNNs.

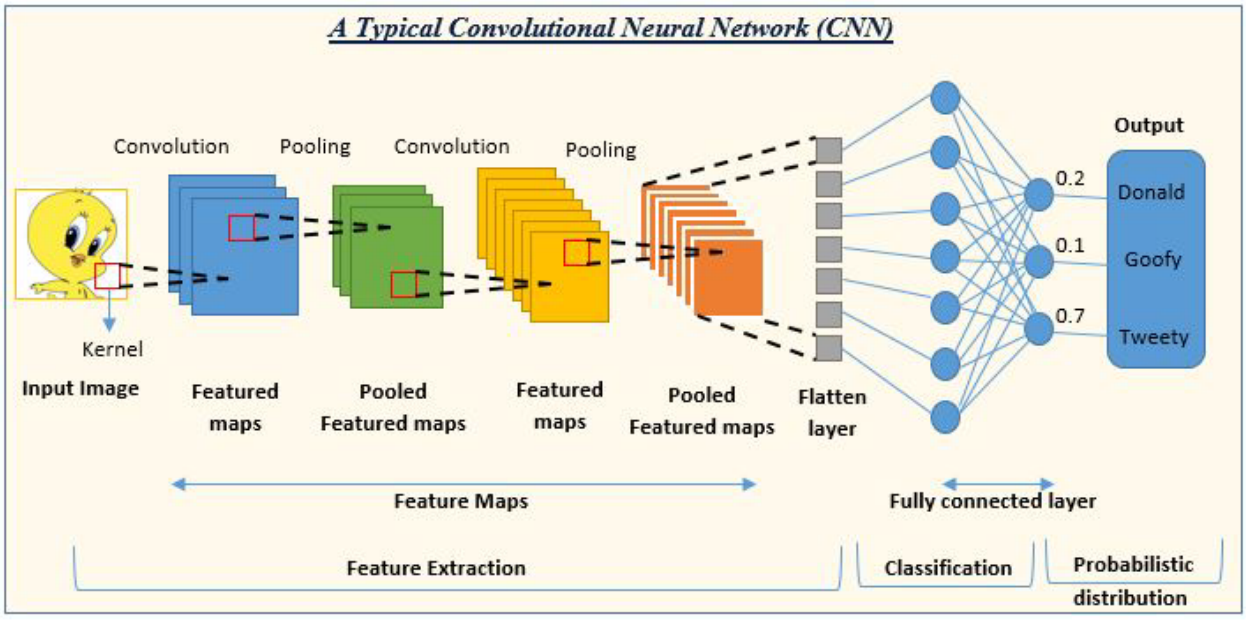

Figure 2.4 Typical Flow of CNNs [11]

In the convolution layer, convolution is applied between the image and a kernel. This process was described in section 2.1 Two-Dimensional Convolution. After passing through the convolution layer, the pooling layer is applied to the convolved image. The image size is scaled down in the pooling layer while maintaining its most important features. This is done to decrease the computational time of subsequent rounds of convolution-pooling layers. Figure 2.5 gives one example of how pooling is applied in a CNN.

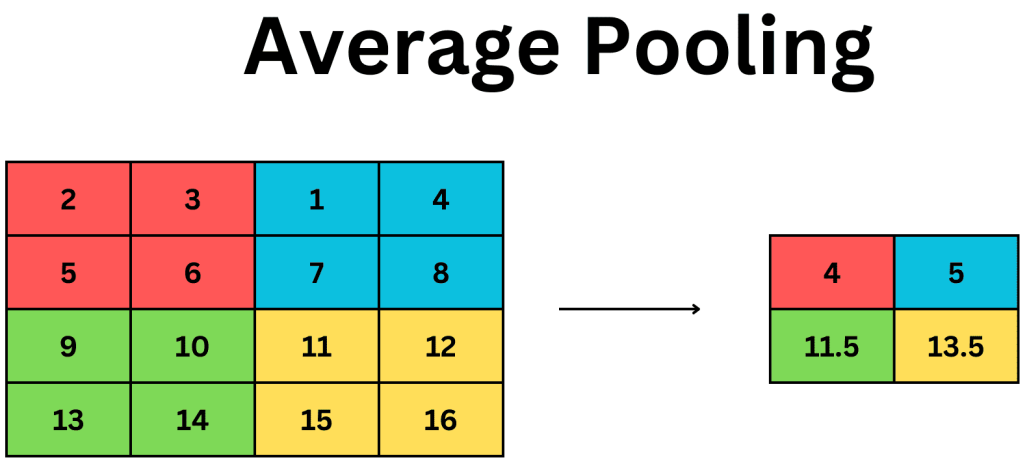

Figure 2.5 Visual Representation of pooling in CNN [12]

After the desired number of convolution-pooling layer rounds have been met (as set by the user), the data is then sent to the fully connected layer. The fully connected layer contains a traditional neural network. In this layer, the image is given a classification. This is done through forward propagation through the data, applying linear regression, obtaining weights, then applying back propagation to update the weights to gain more accuracy. As more images are given to a CNN, the fully connected layer will give each image a classification, splitting the images into groups defined by the user (such as images containing cats and images not containing cats).

2.4 Bilinear Interpolation

When decreasing the size of an image (in terms of width and height), it is important to preserve as much of the original data as possible. The best way to accomplish this is to maintain the aspect ratio of the image when decreasing its size. Maintaining the aspect ratio ensures that

the image is neither stretched nor warped. One way to accomplish this, and as used in this paper, is through bilinear interpolation. Bilinear interpolation is the mathematical process of applying multiple linear interpolations to a two-dimensional grid. Linear interpolation works by taking two known points, drawing a straight line through them, and using any point that falls within the straight line. Figure 2.6 gives a visual of how linear interpolation works.

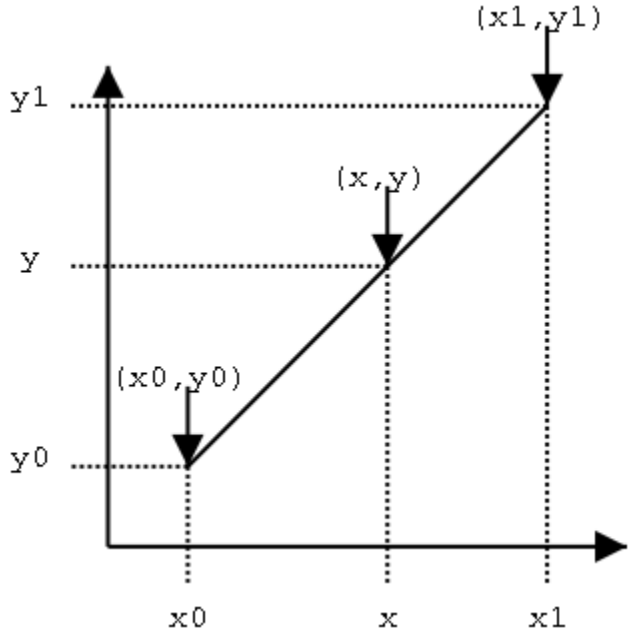

Figure 2.6 Visual Representation of linear interpolation [13]

To extend this to a two-dimensional grid (bilinear interpolation), at least four points must be known. For an image, these four points will be four pixel values that form a small square grid. In this square grid, linear interpolation will first be applied to the top left and right pixels and bottom left and right pixels respectively. After finding those two pixels, linear interpolation will then be applied between them to get the final pixel value that will occupy that square grid. This process gets repeated for the whole image for the specified parameters to scale the image to. Figure 2.7 shows a visual on the explanation provided for bilinear interpolation.

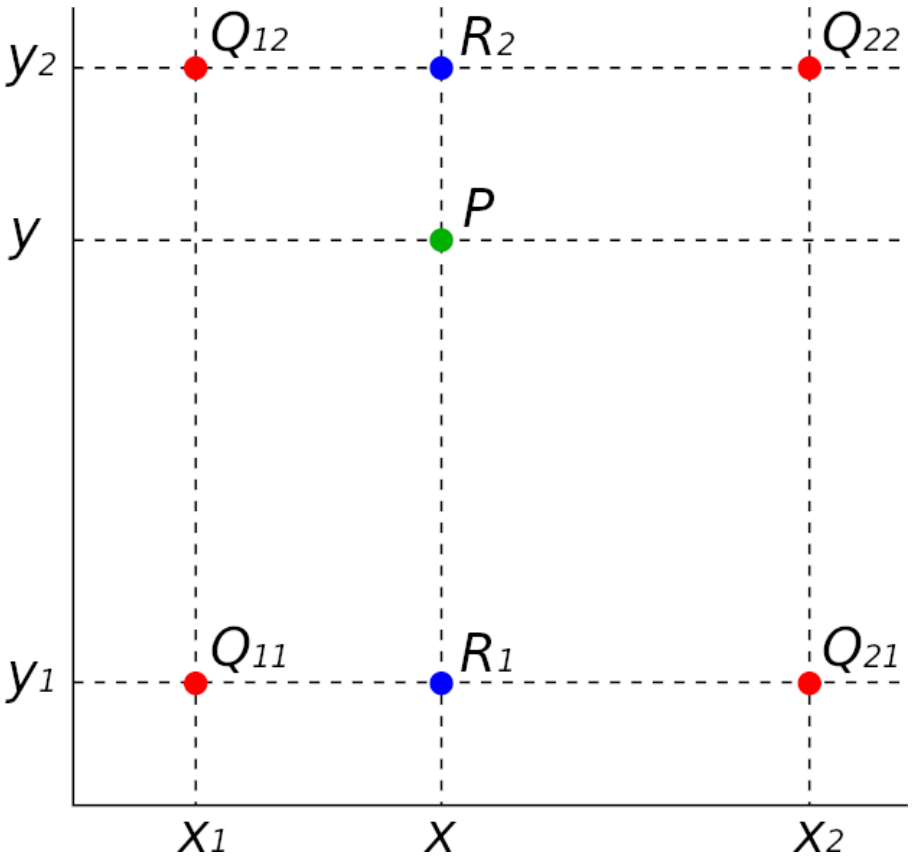

Figure 2.7 Visual representation of bilinear interpolation [14]

Chapter 3 - Proposed SoC Design

In the previous chapter, the necessary background knowledge was given to help readers obtain a basic understanding of the theory used. For this chapter, the focus will be on the main design used to implement the theory. The design was implemented on the Terasic DE1-SoC board which contains the Cyclone V SoC with extra peripherals connected to it. The Cyclone V SoC was chosen for its dual processor capability, low cost, and accessibility. A USB webcam was connected to the Hard Processing System (HPS) for gathering images. The layout of the design is shown below in Figure 3.1.

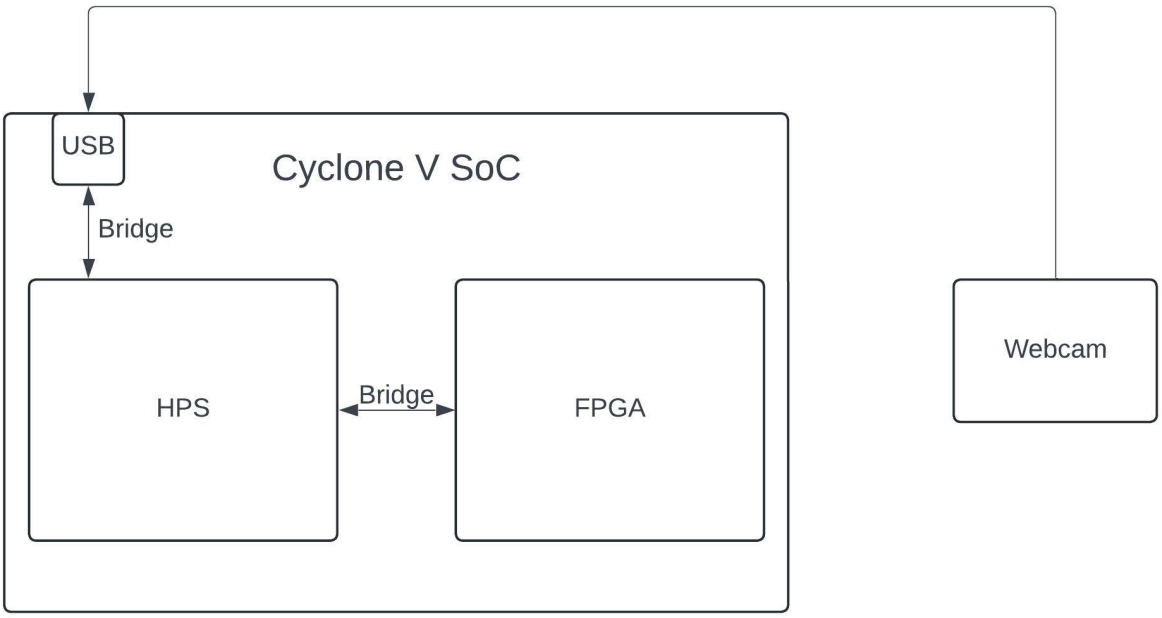

Figure 3.1 Block diagram of physical layout of SoC-FPGA design

The USB webcam captures images when prompted by the user. The image is converted to grayscale then rescaled in the HPS before being sent over to the FPGA. The FPGA side will obtain the gray and rescaled image data and apply an FFT-based convolution before sending the

processed image data back to HPS. Finally, the HPS will convert the data image into a file format that can be viewed by users. The process for transferring data from the HPS to FPGA and FPGA to HPS is done through two different methods. The first method utilizes SDRAM while the second method utilizes SRAM. Using larger images will require the SDRAM while using smaller images can be done on the SRAM. Both methods will be expanded upon in later sections of this chapter.

3.1 Cyclone V SoC Architecture

The Cyclone V SoC Architecture consists of the DE1-SoC Computer design provided by the Intel University Program. As shown in Figure 3.2, the DE1-SoC Computer provides the necessary bridge and connection to the various components and peripherals of the FPGA and HPS. The design also specifies the address space for all the components in the Quartus platform designer. In Quartus, the platform designer is a system integration tool that allows a high level of abstraction between connecting parts. Utilizing the platform designer, users can connect advanced Hardware Description Language (HDL) components (such as IP cores) through a graphical interface. When connecting components, the platform designer automatically creates the interconnected logic between the components, so users do not have to create the HDL themselves. Using this design allows for faster implementation of projects that deal with the interfacing between the FPGA and HPS.

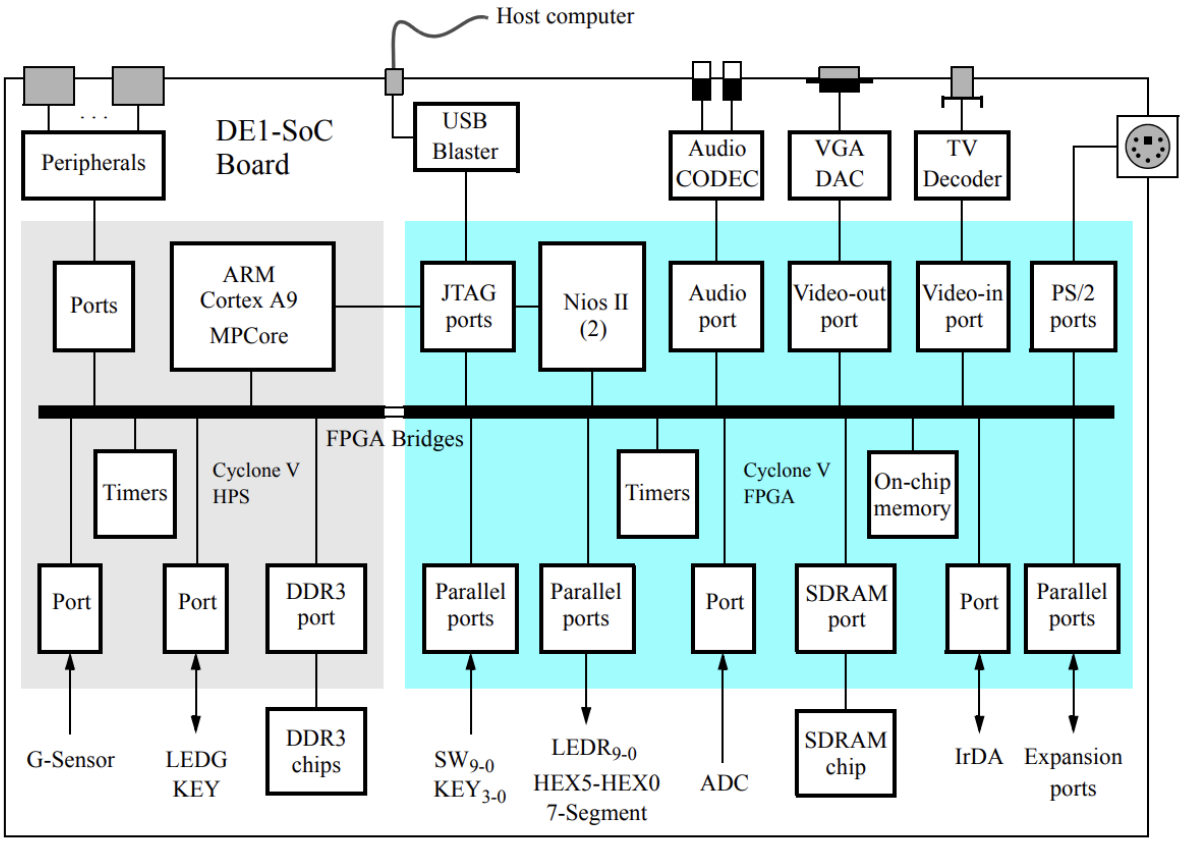

Figure 3.2 Intel DE1-SoC Computer block diagram [15]

3.2 USB Webcam

The USB webcam used was the Logitech C270 webcam. This webcam was chosen because its drivers were already integrated into the Linux version on the DE1-SoC board, so no extra code was needed to access the camera. Since the USB webcam was connected to the HPS side of the DE1-SoC board, C code was used to retrieve the data from the webcam, manipulate it, and then send it to the SDRAM for the FPGA to process. To access the data from the webcam FFmpeg [16] was used. FFmpeg is a free and open-source software project that allows users to handle video and audio streams and files. In the C code, a command line call using FFmpeg was used to obtain one single frame from the webcam video stream. The image obtained from FFmpeg was then converted to BMP file format to ensure all data from the image was kept. The

C Code then takes the BMP image and converts it to grayscale using the NTSC formula for converting RGB images to grayscale. Once the image is fully grayscale, bilinear interpolation is used to rescale the original image size from 640x480 to 128x128. After scaling down the image, the C Code then sends the image over to the FPGA side through the SDRAM bridge. For this implementation, the C Code also does matrix transposition and frequency domain multiplication. The matrix transposition was done in the C Code since there were concerns with not having enough logic elements on the FPGA side to implement the design. The frequency domain multiplication, on the other hand, was concerned with speed and data space. There was not enough space on the FPGA side to hold all the data of the filter values and the time it would take to move filter values was inefficient compared the multiplication on the HPS side. The final bit of C Code involved taking the data of the fully FFT convolved image and creating the BMP file through custom code integrating C FILE data type. Figure 3.3 shows the general flow of the coding.

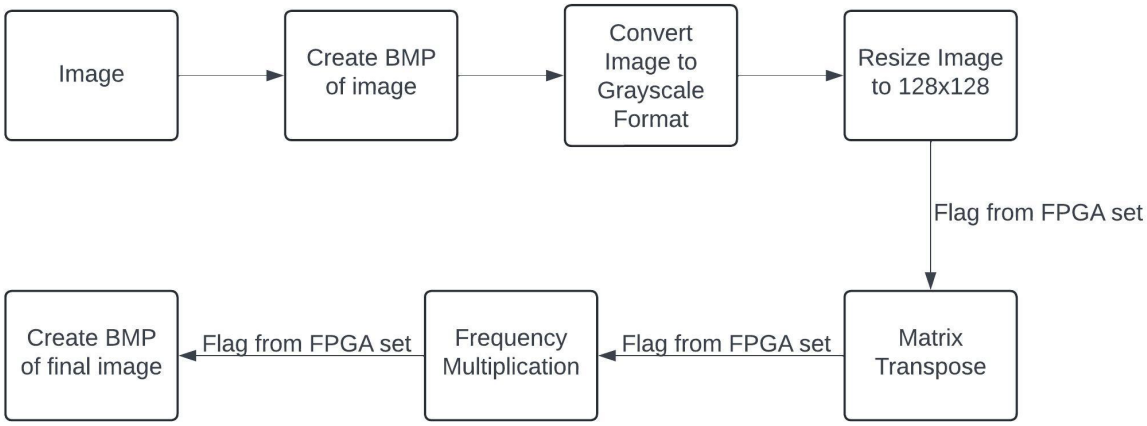

Figure 3.3 Flowchart for HPS C code design

3.3 FFT Core

The first significant challenge was implementing an FFT in the FPGA. Creating a custom FFT core was not the objective of this thesis, so locating an appropriate core was the first task. Initially, the plan was to implement the FFT IP core provided by Intel. However, when attempting to use the IP core with the Intel provided DE1-SoC computer design, there were issues with using multiple IP cores without having a full license for the Quartus software. An open-core FFT was used to handle this issue. Provided by Gisselquist Technology [17], the FFT core allows users to create FFT's with many different parameters, such as length, input and output sizes, twiddle factors, etc. The data format of the FFT is a fixed point, with the real and imaginary values sharing one data bus. Figure 3.4 shows the block diagram of the FFT core. The biggest benefit of using this open-core FFT was that it was fully pipelined. The FFT was verified through ModelSim using a testbench and through SignalTap to ensure it was synthesizable and the behavior was consistent with the ModelSim simulation. One issue found with the FFT was the scaling factor used. For this design, the FFT was created with a length of 256. From the previous chapter, the FFT can be scaled by applying $\frac{1}{n}$ to either the FFT or IFFT or applying $\frac{1}{\sqrt{n}}$ to both. Regardless of which scaling factor is used, the overall scaling done to the FFT and IFFT should be $\frac{1}{n}$. Two issues were found while looking at the core through ModelSim and SignalTap. The first issue was that the scaling factor was off. Instead of $\frac{1}{n}$ being applied to the FFT, the scaling factor was off by a power 2. The second issue was that when creating the IFFT, the same scaling factor was applied. This meant that the scaling factors being applied to both the FFT and IFFT were incorrect. This was easily fixed by shifting the input and output data appropriately before/after sending/receiving the data, respectively.

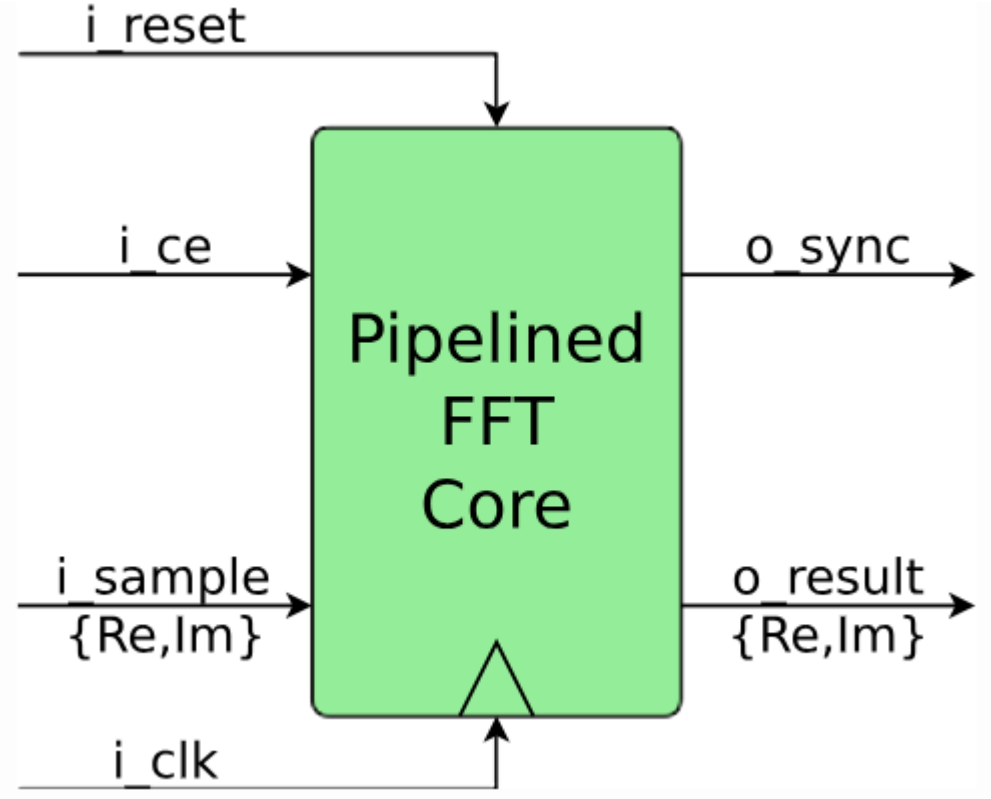

Figure 3.4 Block Diagram of FFT Core [17]

3.4 SDRAM

One of the most significant challenges for the design was handling 279,936 bytes of data that needed to be moved between the FPGA and HPS. Initially, the plan was to handle the data transfer with Programmed Input–Outputs (PIO), but this was quickly dismissed as there were not enough logic elements in the FPGA to handle the full amount of data. With PIOs being unviable, the next solution was looking at storage options accessible by both the FPGA and HPS. Looking back at Figure 3.2, on the FPGA side, there is an external SDRAM chip. This SDRAM chip provides 64 MB of data and has a component that is already available in the Quartus platform designer with address spaces mapped out. This component is already connected to the AXI master, allowing the HPS to communicate with the SDRAM.

To access the SDRAM on the FPGA, the External Bus to Avalon Bridge (EBAB) IP component provided by Intel's university program was used. This IP component allows for an external peripheral and the Avalon Switch Fabric to connect to each other through a master-slave relationship, with the peripheral acting as the master and the Avalon Switch Fabric acting as the slave. The functional diagram for this component is shown in Figure 3.5, and Figure 3.6 shows the timing diagram of the signals from the master to the Avalon Switch Fabric. With this configuration, the SDRAM is used as the peripheral and can handle reading or writing data from the FPGA and HPS side through the Avalon Switch Fabric. The SDRAM was configured to send data with a 32-bit width to work in tandem with the FFT core. Using the SDRAM with the SoC-FPGA design operating with a 50 MHz clock resulted in the design taking 12,541,264 total clock cycles, with the FPGA taking 8,961,264 clock cycles and the HPS taking 3,580,000 clock cycles. The full breakdown of the total clock cycles will be provided in Chapter 4 and the method for obtaining the clock cycles will be discussed in Section 3.5.

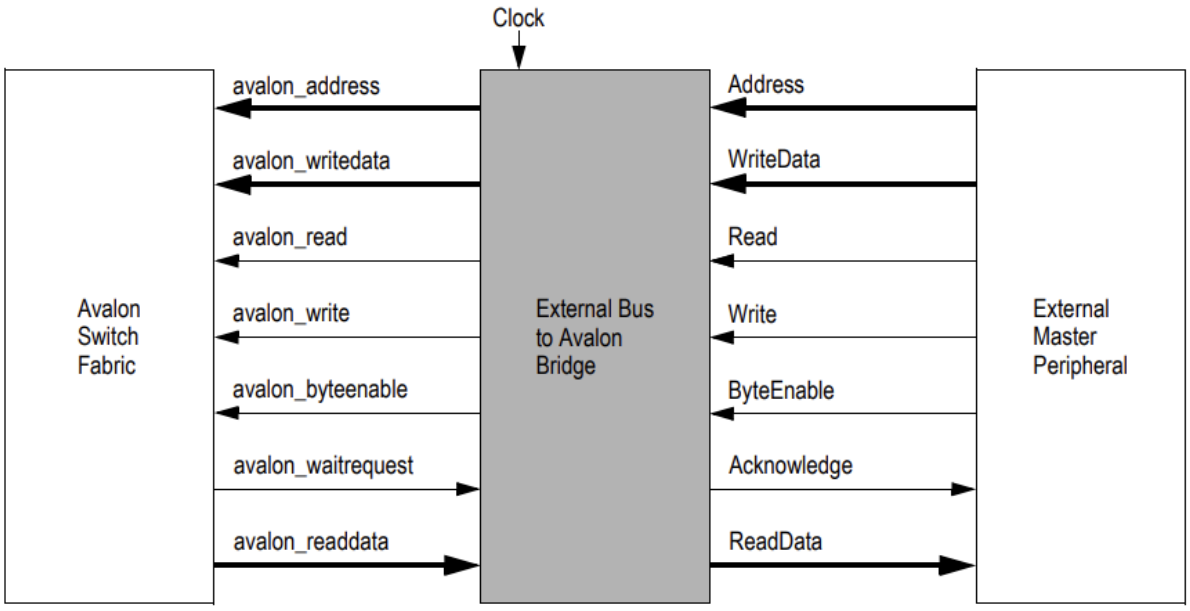

Figure 3.5 Block Diagram of the EBAB IP [18]

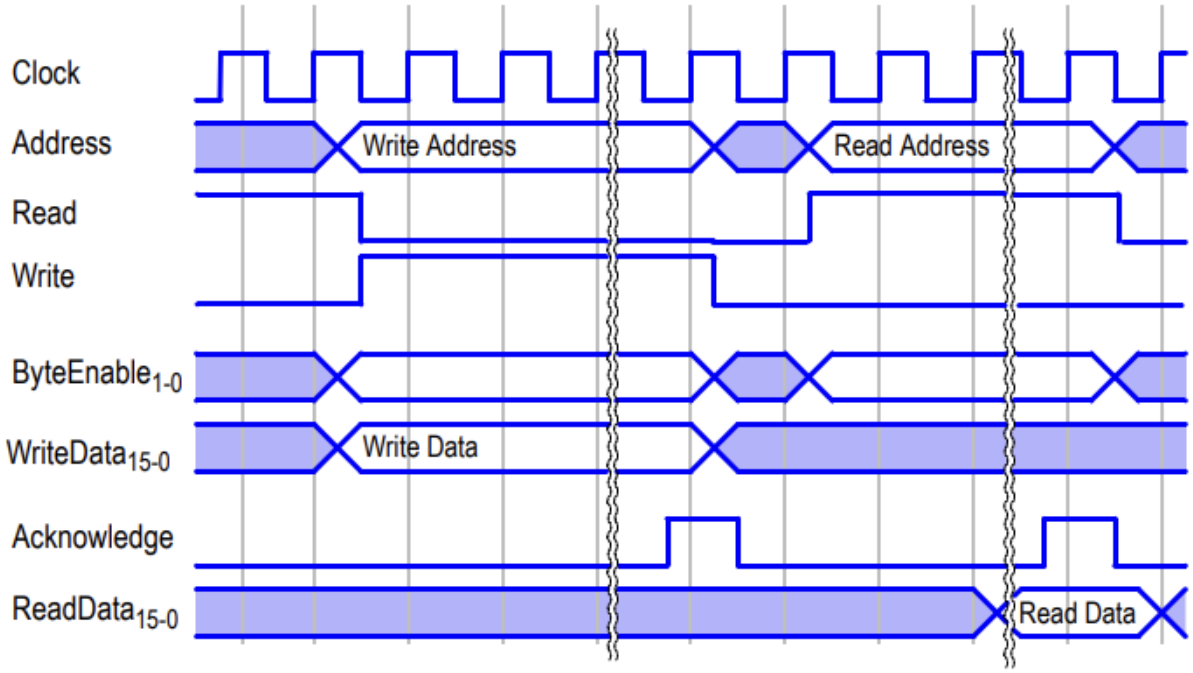

Figure 3.6 Timing diagram for the EBAB IP [18]

3.5 SDRAM State Machine

This section shows the state machine implemented in the FPGA side to control data flow from SDRAM to FFT and IFFT modules. Figure 3.7. shows the state machine.

The SDRAM state machine consists of the following 13 states:

1. EnableRead
2. Idle
3. MemReadEnable
4. MemReadAck
5. MemWriteEnable
6. MemWriteAck
7. FFT
8. INVFFT

9. Transpose
10. WriteHPS
11. FreqMult
12. HPS_Last
13. GetClockCycles

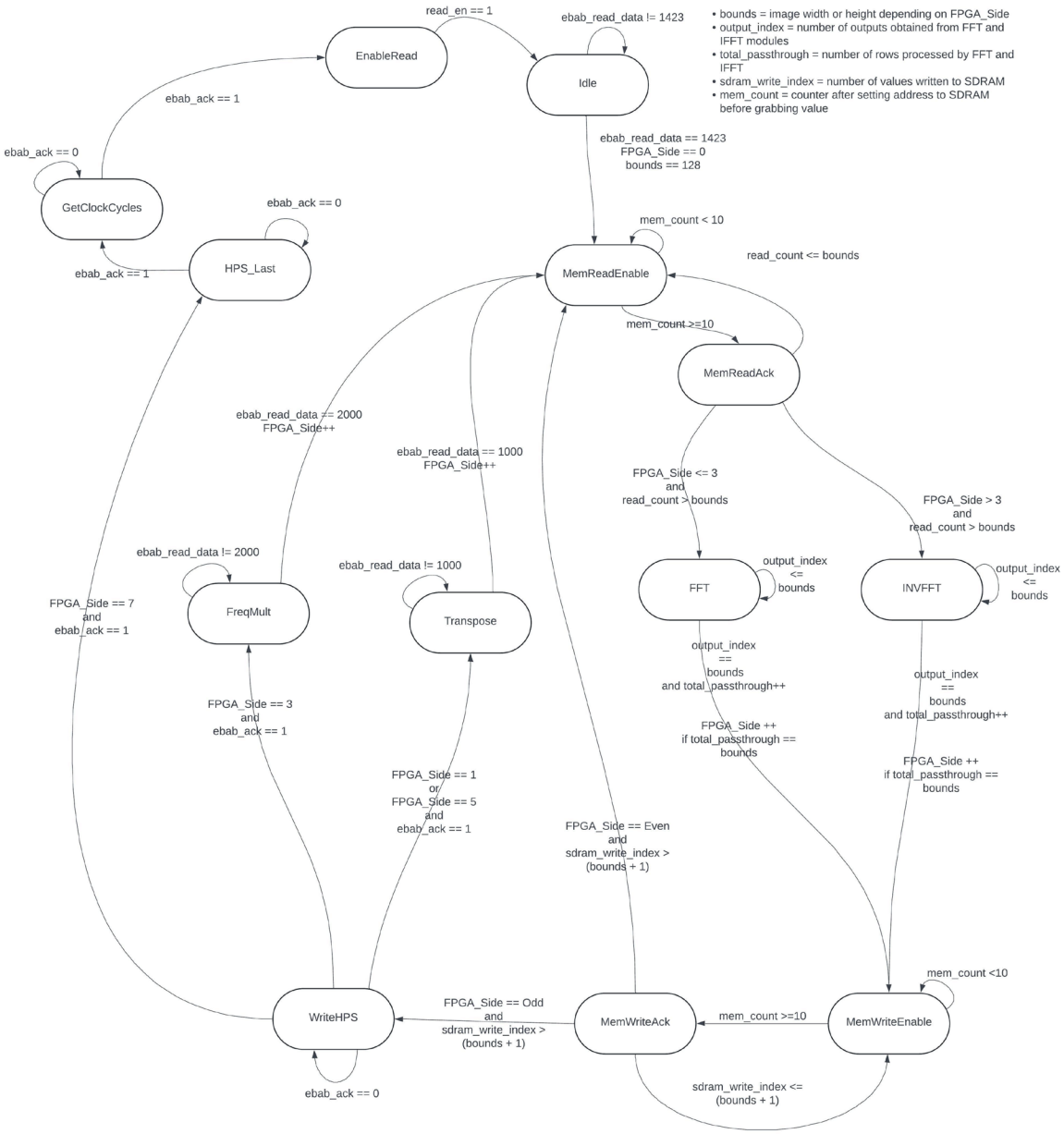

Figure 3.7 State Machine of SoC-FPGA design utilizing SDRAM

For ease of reading, the MemReadEnable and MemRead Ack states will be referenced together as MemRead and the MemWriteEnable and MemWriteAck will be referenced together as MemWrite.

As mentioned earlier in section 3.4, this section will provide the method for finding the clock cycles. To find the clock cycles, a register was incremented at the beginning of every state. This gave the total clock cycles of the design. To find the individual clock cycles of each state, the register was incremented only at the beginning of a desired state, the value was recorded, then the register would increment at the beginning of the next desired state, repeating this process until each state's clock cycles were recorded.

One of the most important registers used in the state machine is labeled as "FPGA_Side." This register keeps track of how many times the state machine has passed through states FFT, INVFFT, Transpose, and FreqMult. Table 3.1 provides the values of FPGA_Side and what they represent.

Table 3.1 FPGA_Side register with associated values for SDRAM state machine

FPGA_Side Value	0	1	2	3	4	5	6	7
Representation	Initial Value	FFT has completed processing full image first time (rows)	Image Matrix has been transposed	FFT has completed processing full image second time (columns)	Frequency Multiplication between image and filter	INVFFT has completed processing full image first time (rows)	Image Matrix has been transposed	INVFFT has completed processing full image second time (columns)

The state machine consists of two big loops. The first is the following state transition: MemRead → FFT → MemWrite → MemRead. This loop repeats until the image has been

transformed. The second loop is the same transition, with the difference being replacing the FFT state with INVFFT. Both loops must be used twice, once for the rows of the images and again for the columns of the image. To keep from making duplicate FFT and INVFFT states, FPGA_Side keeps track of how many times the data has fully been processed through FFT or INVFFT and when the data of the image has been transposed. Using this tracking register, the number of redundant states can be reduced from the state machine. The rest of this section will focus on descriptions of each state.

The EnableRead state sets the read enable flag to high. The read enable flag is used to set the FPGA to continuously read from the SDRAM. This state starts the process, enabling the flag then immediately transitioning to the next state.

In the Idle state, all registers are set to initial values. Before the registers are initialized, the state is constantly reading from address space h03938700 in the SDRAM until the value in the address is 1423. The address and value in the address were arbitrarily chosen, however, the reason behind doing this is to ensure the FPGA does not start applying an FFT-based convolution to the image until the HPS fully processes the image. Once the state reads in the chosen value, the registers are initialized, and the state transition occurs.

The MemRead states handle reading all the data of the grayscale and rescaled image from the SDRAM. Figure 3.6 shows the timing for reading from a master device. These states read in the data one row of the image at a time and store it into a two-dimensional register. Reading the data one row at a time was done due to the limitations of how much data the two-dimensional register could store without using all the FPGA logic elements and with the FFT core set to only process one row or column of the image at one time. Once a full row is read, the state will transition to either FFT or INVFFT depending on the FPGA_Side register value.

The next two states, FFT and INVFFT, function the same, with the difference being the module being called. Both states send the values from the two-dimensional register in MemRead to their respective modules to be transformed. As soon as the modules start outputting data, this state takes that data and stores it back into the two-dimensional register. Due to memory constraints, only a singular two-dimensional register is used, with values updating in a pipelined fashion. Figure 3.8 shows a visual for how the singular two-dimensional register is used. Once the modules fully transform one row of data, the state sets a write flag then transitions to MemWrite. These two states are reached repeatedly, transforming one row of data at a time, storing the transformed data, then state transitioning to MemWrite until all the original image data has been processed.

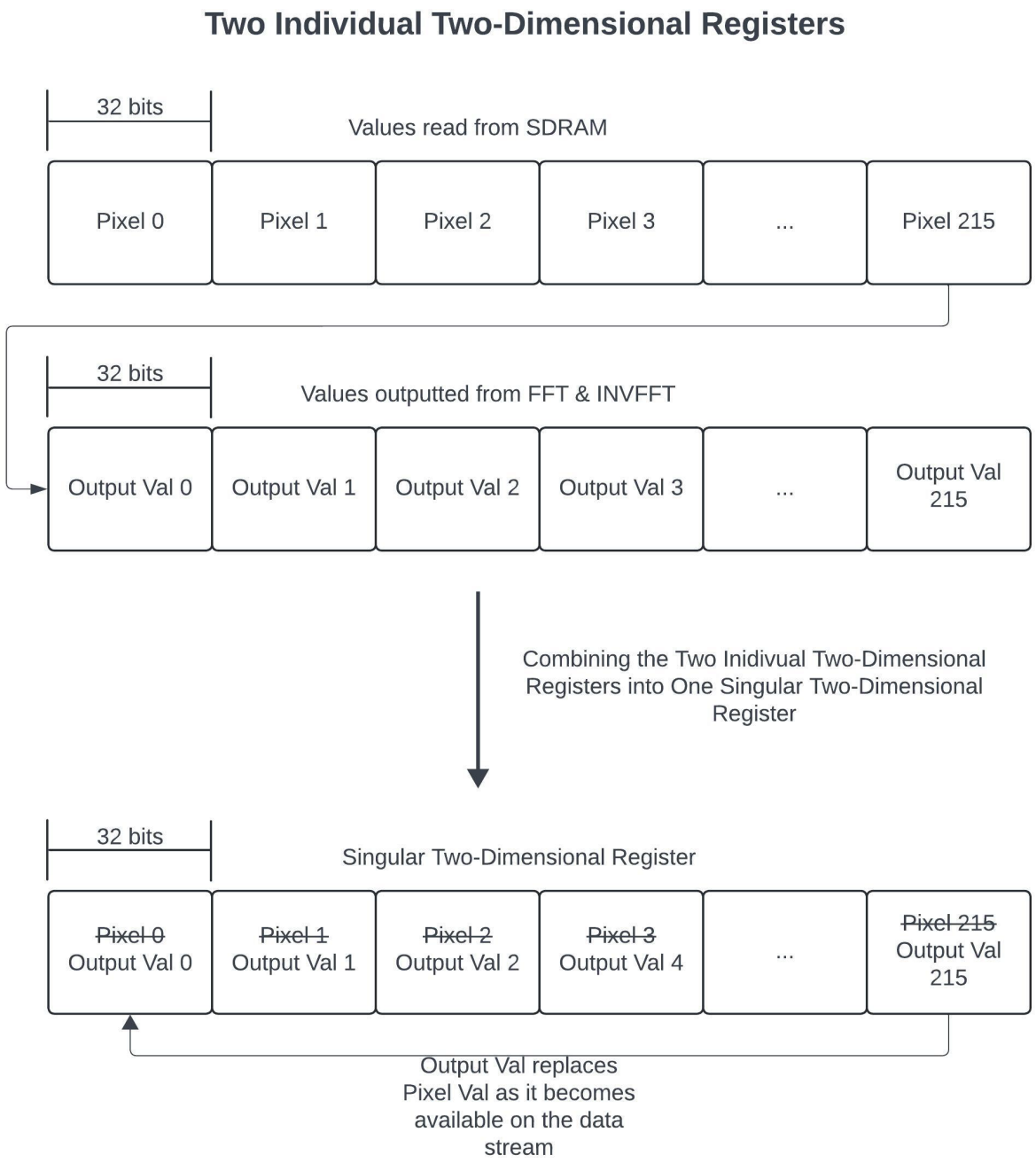

Figure 3.8 Visual representation for the two-dimensional register

In MemWrite, the data from the two-dimensional register is written to the SDRAM. After writing the data from the register, the state will transition to either MemRead or WriteHPS based on the value of FPGA_Side. If transitioning to WriteHPS, this state will set a register to a

decimal value for WriteHPS to write to SDRAM depending on the value of FPGA_Side and set the read enable flag high. In this case, FPGA_Side will signify whether the HPS side will be handling matrix transposition, frequency multiplication, or creating the filtered final image.

The WriteHPS state writes the register value from MemWrite to the SDRAM. Once the value is written, the state will transition to either Transpose, FreqMult, or HPS_Last depending on the value of FPGA_Side.

The next two states: Transpose and FreqMult, function similarly to one another. These states continuously read from the SDRAM until a value at a specified address is set. Once this value is set, the states will then change the starting address value that MemRead will use to read from SDRAM. The state then transitions to MemRead from either state.

HPS_Last writes a value to the SDRAM at a specific address. This value acts as a flag for the HS, signifying that the HPS side can now pull data from the SDRAM and create the final image. Once the value is written to SDRAM, the state transitions to GetClockCycles.

The last state, GetClockCycles, writes the number of clock cycles taken to get through the state machine. When this value is written to the SDRAM, the state transitions to EnableRead and the process repeats when the user inputs a new image.

3.6 SRAM

While the SoC-FPGA design using the SDRAM worked correctly (shown in results in Chapter 4), the number of clock cycles taken was far too large. When reviewing the DE1-SoC computer (from Figure 3.2), it was noted that the bridge was also connected to the FPGA On-Chip memory. In the Quartus platform designer, the FPGA On-Chip memory component was labeled as On-Chip SRAM and provided users with 256 KB of memory organized as 64K x 32 Bits. The SRAM was originally used by the DE1-SoC computer as the memory for the pixel

buffer for the VGA system but was repurposed to use as general memory to transfer data between HPS to FPGA and FPGA to HPS. Since the SRAM is located on the FPGA itself, data transfers in the FPGA do not require a separate component like the SDRAM. Using the SRAM with the SoC-FPGA design operating with a 50 MHz clock resulted in 1,337,417 total clock cycles, with the FPGA taking 697,936 clock cycles and the HPS taking 639,481 clock cycles. The full breakdown of the total clock cycles will be done in Chapter 4 and the method for obtaining the clock cycles will be discussed in Section 3.5. Figure 3.9 and Figure 3.10 provide the waveforms for SRAM read and write signals.

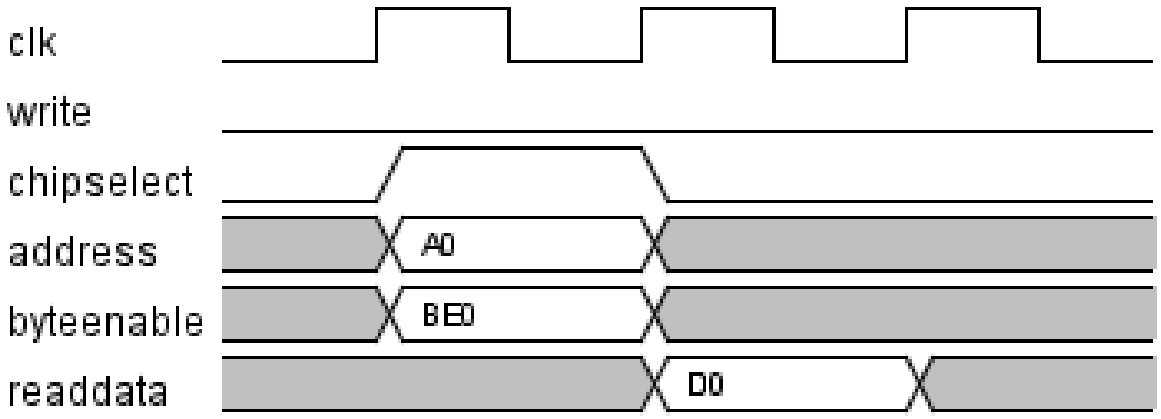

Figure 3.9 Waveform for SRAM read (from Intel)[19]

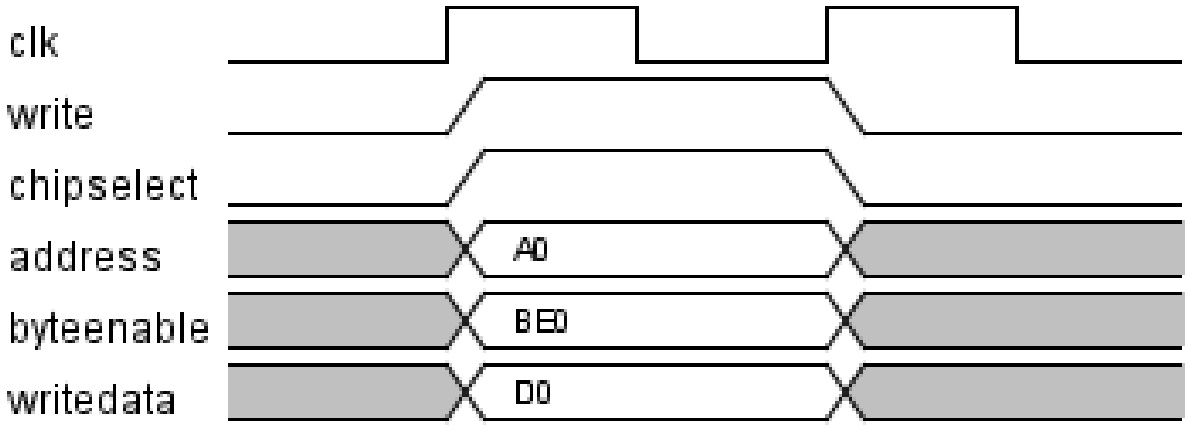

Figure 3.10 Waveform from SRAM write (from Intel)[19]

3.7 SRAM State Machine

The SRAM state machine, shown in Figure 3.8, was built with the SDRAM state machine as the foundation. Many of the states in the SRAM state machine have the same behavior as the SDRAM state machine (such as the FFT and INVFFT states) with changes consisting of edits to some of the existing states and the removal of states that were not required to make the SRAM work. Figure 3.11 shows the SRAM state machine.

The SRAM state machine consists of the following states:

1. Idle
2. MemReadEnable
3. MemReadWait
4. MemReadAck
5. MemWriteAck
6. FFT
7. INVFFT
8. WriteHPS
9. Transpose
10. FreqMult
11. HPS_Last

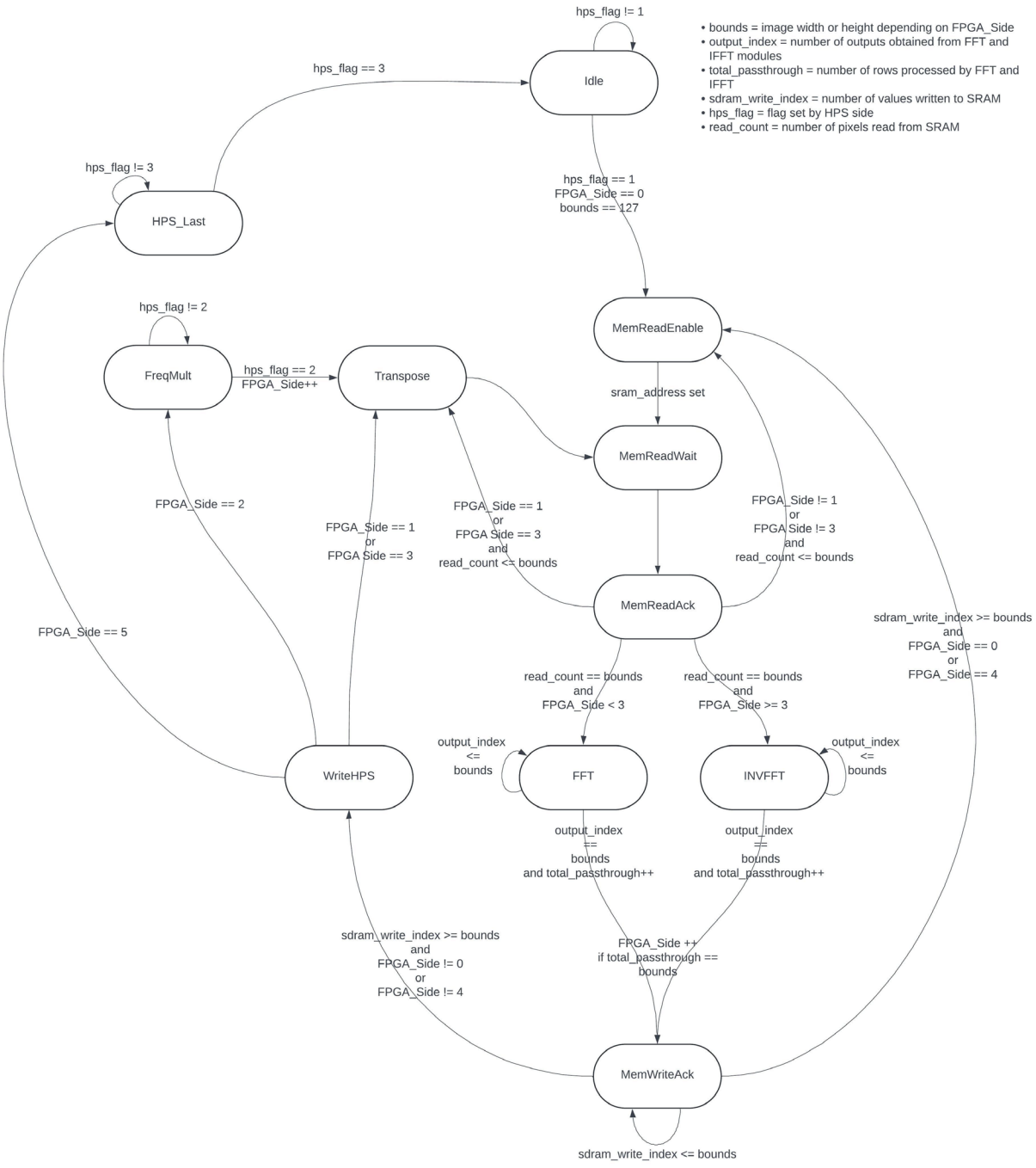

Figure 3.11 SRAM State Machine

The singular two-dimensional register in the SDRAM state machine and concepts like the two big loops behave the same as they do in the SDRAM State Machine, with the descriptions in

Section 3.5. The rest of this section looks at what changes were made in the SRAM state machine compared to the SDRAM state machine.

The first change deals with the FPGA_Side register. For the SRAM SoC-FPGA design, matrix transpose was accomplished on the FPGA instead of the HPS. How this was done will be described later in the section. Table 3.2 shows the FPGA_Side register and its associated values.

Table 3.2 FPGA_Side register with associated values for SRAM state machine

FPGA_Side Value	0	1	2	3	4	5
Representation	Initial Value	FFT has completed processing full image first time (rows)	FFT has completed processing full image second time (columns)	Frequency Multiplication between image and filter	INVFFT has completed processing full image first time (rows)	INVFFT has completed processing full image second time (columns)

In terms of states, the only major change occurs in the Transpose state. In the SDRAM state machine, the Transpose state waited for a flag set from the HPS side since transposing the matrix was done on the HPS. In the SRAM state machine, to accomplish matrix transposing, the addressing scheme for accessing the memory was changed. When transposing in the FPGA, instead of incrementing the address by 1 after every memory read, the address is incremented by the following equation:

$$address = (total\_count) + (256*address\_increment)$$

where address is the address to access memory, total_count is how many columns of the matrix have been accessed, address_increment is a counter that increments by 1 after every memory read, and 256 is the value of the row or column of the matrix multiplied by 2 since the real and imaginary values are stored separately instead of grouped together. To better understand how this equation works, Figure 3.12 shows how the equation accesses a matrix.

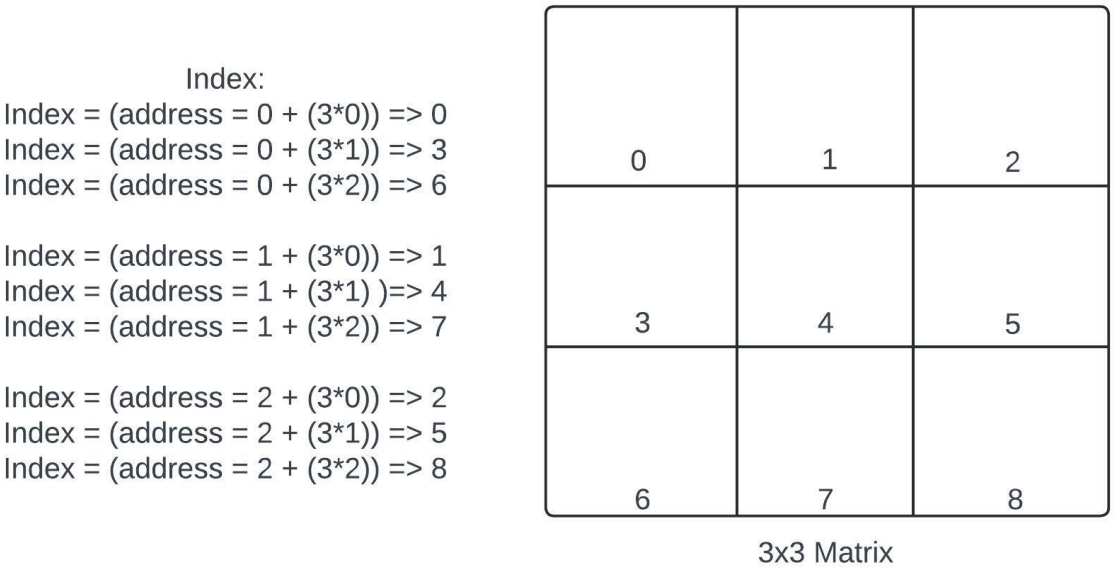

Figure 3.12 Matrix Access using Matrix Column Addressing Equation

The next major change comes from the addition of the MemReadWait State and the removal of the EnableRead, MemWriteEnable, and GetClockCycles states. The MemReadWait state is used to delay obtaining the value from SRAM after setting the SRAM address by 1 clock cycle. This is done due to the timing of the read waveforms as referenced in Figure 3.9.

Both EnableRead and GetClockCycles were removed due to using PIOs to set flags between the HPS and FPGA and to also keep track of the number of clock cycles while MemWriteEnable was removed due to the SRAM write waveform being able to set the address and write the data to the address in the same clock cycle, as shown in Figure 3.10. As for the other states, only minor changes were made, such as WriteHPS using different values to represent flags, but the core behavior of the remaining states is the same.

Chapter 4 - Analysis

This chapter focuses on the analysis of the FPGA design. The analysis includes a comparison of 2-D Image Convolution vs 2-D Fast Fourier Transform based image convolution for varying kernel sizes, the images produced from the design compared to MATLAB, the timing analysis of the FPGA for the SDRAM and SRAM, possible ways to increase the performance of the FPGA design, and how the results could be extrapolated to other SoC platforms. To improve readability, the process of taking the 2-D FFT of an image, multiplying it by a kernel in frequency domain, and then applying 2-D Inverse FFT to obtain the final image will be referred to as the 2-D Frequency Transform.

4.1 Convolution vs Frequency Transform

In general, signals are typically processed faster when applying a Fourier transform to them instead of using convolution for significantly larger signals. This theory holds true when applying a filter to an image. Applying a larger filter to a larger image leads to less computation time when using the Frequency Transform over convolution. Figure 4.1 shows plots of the complexity for two-dimensional Frequency Transforms and two-dimensional convolutions for varying sized kernels and matrices.

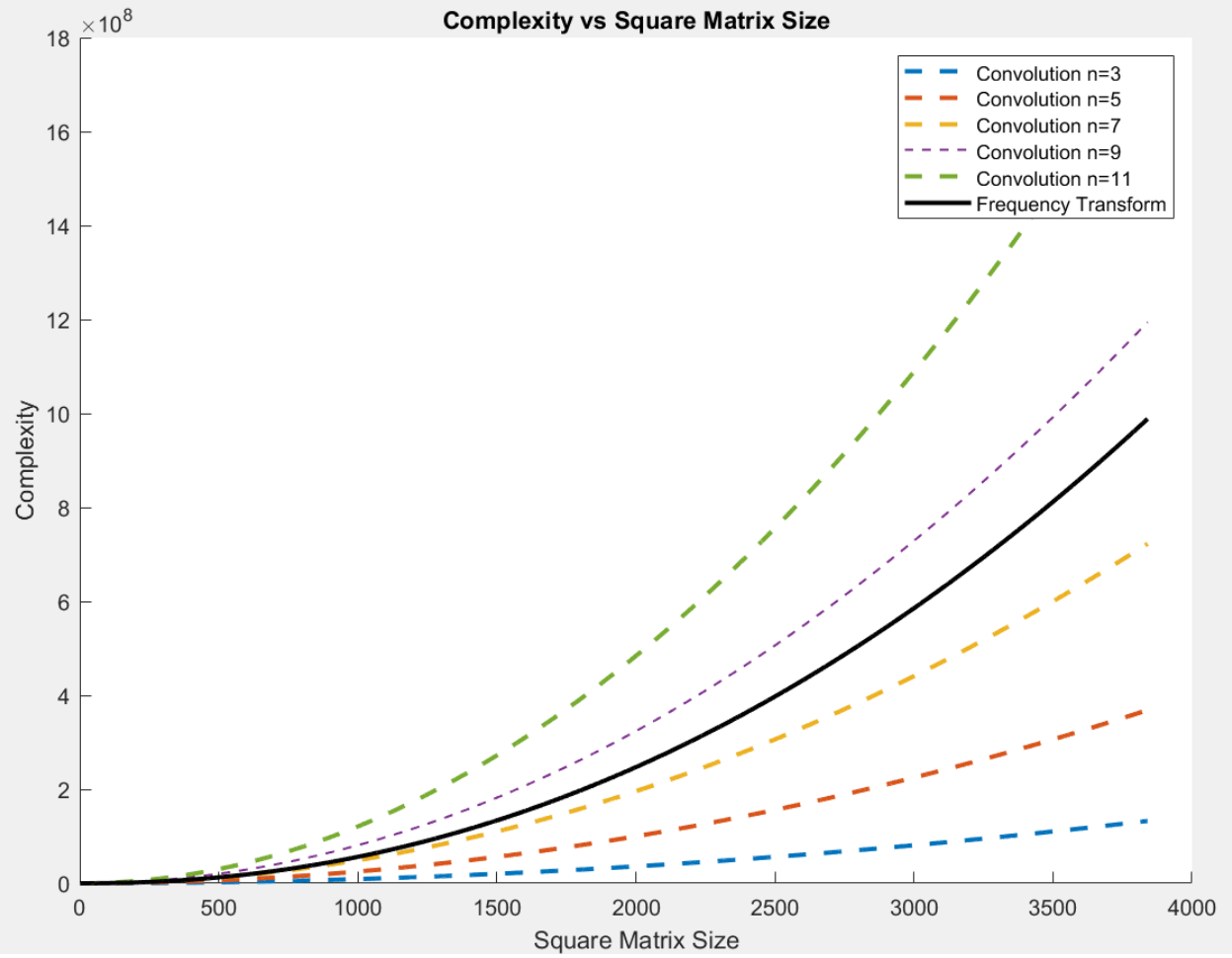

Figure 4.1 Plot of Complexity vs Square Matrix Size

From Figure 4.1, kernel size has a major effect on complexity. Increasing the kernel size leads to the two-dimensional Frequency Transform being advantageous over two-dimensional convolution in terms of complexity. This proves that the two-dimensional Frequency Transform can result in better performance for scenarios that require larger kernel sizes. Larger kernel sizes are advantageous for large sized images. Utilizing a larger kernel for a lager images results in faster computation time and less resources being used, as the larger kernel can traverse across the image quicker than a smaller kernel. Ding et al. [20] provides results to show improvement in CNN computation time and resource utilization for larger kernel sizes.

4.2 Image Results

Determining the accuracy of the edge detected images from the design involves two tests. The first test deals with visualization. To ensure the FPGA is properly applying edge detection, the webcam captured pictures of common images used in edge detection examples. These images were sent to MATLAB, where a two-dimensional image convolution and two-dimensional Frequency Transform was applied to them. Figures 4.2-4.4 show the original image compared to the MATLAB and SoC-FPGA images.

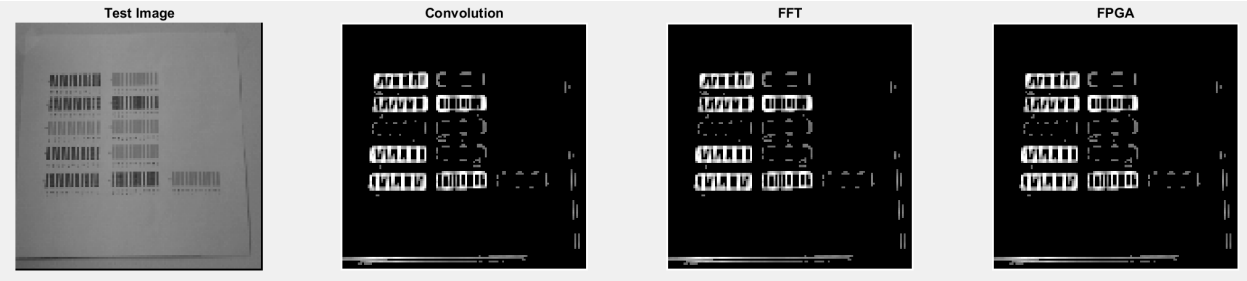

Figure 4.2 Image from printer alignment

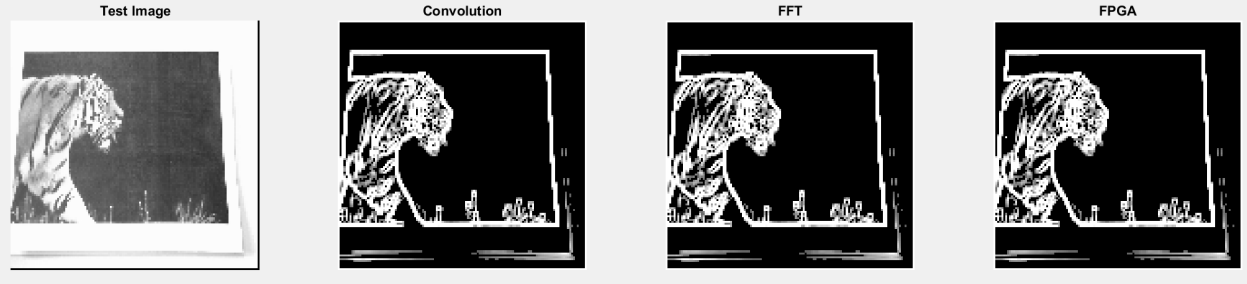

Figure 4.3 Tiger picture used in OpenCV edge detection example

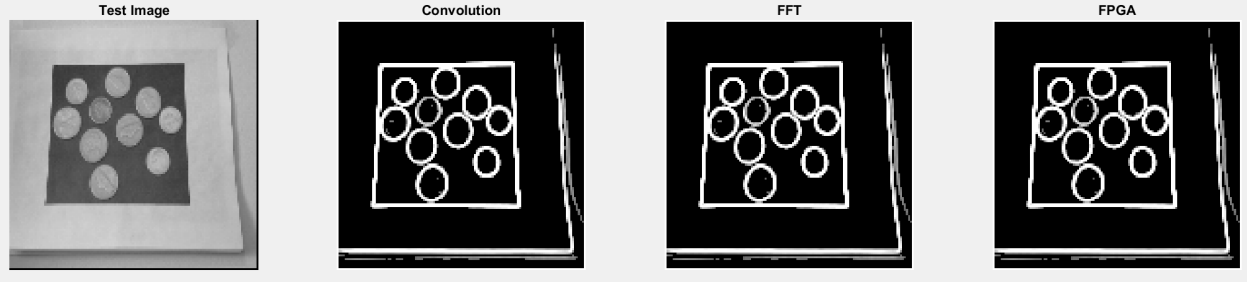

Figure 4.4 Coins on a similar color background

Based on the figures, the SoC-FPGA image is visually identical to the MATLAB resulting images. When looking at the actual image data, for these examples, the accuracy percentage was 100%. With the accuracy being 100% (no errors), this confirms that the SoC-FPGA produces edge detected images that are identical to the MATLAB results. To further confirm this, Figures 4.4 and 4.5 compare images obtained from the webcam capturing its surroundings.

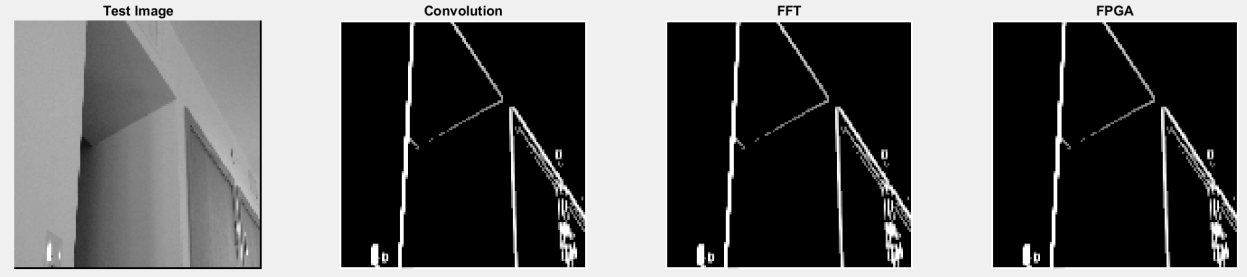

Figure 4.5 Image of wall with hallway and doors

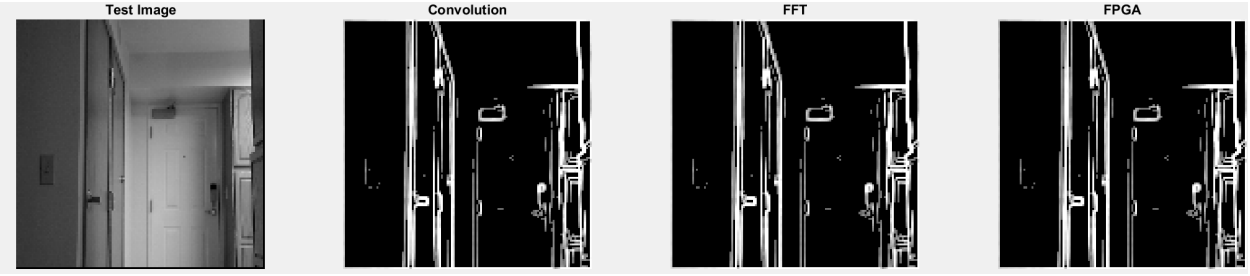

Figure 4.6 Image of doors and cabinets

4.3 Timing

The total time for the SoC FPGA SRAM implementation was noted in Chapter 3. This section serves to break down the total time based on how many clock cycles each state takes. The timing analysis will be done on both the SDRAM and SRAM implementations. This section will also analyze the scalability of the design based on the clock cycle breakdown.

To obtain the clock cycles for each state, a counter was incremented each time a state was accessed, including when a state would repeatedly loop through itself. Once the state machine reached the final state, the counter was written to memory, where the HPS reads in the value and

prints it out. To confirm that the clock cycle values the HPS read in were accurate, they were exported to the seven-segment display and viewed in Signal Tap on the FPGA. The three methods produced the same value, confirming that the clock cycle value written to memory was not corrupted. This method for finding clock cycles was utilized because the SoC FPGA design ran a state machine that updated on every positive edge of the clock signal used.

Table 4.1 provides the breakdown for the total time of the SDRAM implementation based on how many clock cycles each state takes and Table 4.2 provides the SRAM implementation. For the design, MemRead was executed 147,584 times and MemWrite was executed 131,072 times. The total time for the SoC FPGA SDRAM implementation is about 250 milliseconds while the total time for the SoC FPGA SRAM implementation is about 27 milliseconds. This time only encompasses the time it takes for the FPGA to apply a 2-D Frequency Transform and does not include the time the HPS side took to create the images from the webcam, process them, and send them to memory. Analyzing the SDRAM implementation from Table 4.1, much of the processing time was done through reading from and writing to the SDRAM and the frequency multiplication that was offloaded to the HPS side, which also includes the HPS reading from and writing to the SDRAM. As for the SRAM implementation, frequency multiplication accounted for 50% of the processing time. This is due to the frequency multiplication being done on the HPS side, as there were not enough memory blocks to create enough SRAM space to store the kernels (131,072 bits required) needed to process the frequency multiplication on the FPGA side.

Table 4.1 SDRAM Clock Cycle Breakdown by State

State	Clock Cycles	Time (ms)
MemRead	5,100,000	102.0000
MemWrite	3,640,000	72.8000
FFT	110,592	2.2118

IFFT	110,592	2.2118
WriteHPS	80	0.0016
Transpose	2,410,000	48.2000
Freq Mult	1,170,000	23.4000
Design Total	12,541,264	250.8253

Table 4.2 SRAM Clock Cycle Breakdown by State

State	**Clock Cycles**	**Time (ms)**
MemRead	279,808	5.5962
MemWrite	131,072	2.6214
FFT	110,592	2.2118
IFFT	110,592	2.2118
WriteHPS	80	0.0016
Transpose	65,792	1.3158
Freq Mult	639,481	12.7896
Design Total	1,337,417	26.748

Table 4.3 provides the time comparison of the design to the MATLAB implementations. One important note about the comparison is the CPU of the device running MATLAB. The device used for the MATLAB implementation contained an Intel i9-13900h CPU operating with a 5.4 GHz clock while the Cyclone V SoC is operating off a 50 MHz clock. The Intel CPU was also released one decade after the Cyclone V SoC, so the Intel CPU will have a clear advantage with an extra decade of design and upgrades, but the comparison still holds value in giving readers an idea of how to judge the FPGA-SoC performance.

Table 4.3 Time Comparison to MATLAB

Process	**Time (ms)**
MATLAB 2-D Convolution	0.0915

MATLAB 2-D Frequency Transform	0.3052
SoC FPGA 2-D Frequency Transform SDRAM	250.8253
SoC FPGA 2-D Frequency Transform On-Chip SRAM	26.748

4.4 Performance Increase Opportunities

This section looks at performance improvement opportunities to the FPGA design for both the SDRAM and SRAM implementation.

The SDRAM design was utilized for the potential to apply the FPGA 2-D FFT implementation to larger images. Accessing the SDRAM is currently done through the Intel External Avalon Bridge to Bus IP. This interface, allowing users to read and write to the SDRAM from both the FPGA and HPS, currently accounts for 70% of the total processing time for the SDRAM implementation. The biggest performance increase would be seen from implementing an alternative way to access the SDRAM. One potential way to increase the performance would be to implement a DMA controller. With a DMA controller attached to the SDRAM and both the FPGA and HPS, the time taken to read and write from the SDRAM can be decreased.

For both SDRAM and SRAM implementations, frequency multiplication is offloaded to the HPS, with the process taking 1,170,000 clock cycles, or a rate of 72 clock cycles per pixel (1,170,000 / (128*128)) for the SDRAM and 794,220 clock cycles, or a rate of 48 clock cycles per pixel (794,220/ (128*128)) for the SRAM. When moving to the FPGA, it would take 16,384 clock cycles to multiply all the values, as multiplication could be done at a rate of 1

multiplication per clock cycle, plus the amount of clock cycles required to read and write from the on-chip memory.

4.5 Alternative SoC Platforms

As mentioned in Chapter 3, the Cyclone V SoC was used due to the low cost of the board and the stock that the University had of this board. Due to being a low-cost board, the Cyclone V SoC contained limitations to the clock speed that the SoC-FPGA design can be run with, the amount of logic elements on the FPGA, the amount of on-chip memory on the FPGA, and the speed of the external SDRAM. Looking at alternative SoC FPGAs in the Intel portfolio (Cyclone V SoC is manufactured by Altera/Intel), the high-end devices are within the Agilex and Stratix series SoC FPGAs. For this section, the Cyclone V SoC will be compared to the Agilex 5 and Stratix 10. Table 4.4 provides comparisons between the Cyclone V SoC, Agilex 5, and Stratix 10 in terms of number of logic elements, SDRAM interface, amount of on-chip memory, and DSP blocks.

Table 4.4 SoC FPGA Comparisons

	Cyclone V SoC	Agilex 5	Stratix 10
Logic Elements	85,000	656,000	2,800,000
SDRAM Interface	DDR3	DDR4	DDR4
On-Chip Memory	4.45 MB	31.46 MB	229 MB
DSP Blocks	87	1,692	5,760

With the high number of logic elements and large on-chip memory, the Frequency Transform can be fully done on the FPGA, decreasing the clock cycles for the processes that were being offloaded to the HPS on the Cyclone V SoC.

Both the Agilex 5 and Stratix 10 contain DDR4 SDRAMs, which are significantly faster than the DDR3 SDRAM on the Cyclone V SoC. On top of having a faster SDRAM interface,

both the Agilex 5 and Stratix 10 share the SDRAM with both the FPGA and HPS. By having the SDRAM connected to both the FPGA and HPS side without the need for an external interface, the number of clock cycles required to read and write from the SDRAM decreases significantly.

For the Cyclone V SoC, the SoC FPGA design was run using a 50 MHz clock. With the Agilex 5 and Stratix 10, the speed of the clock can be increased. For the Stratix 10, the clock can be operated at 1 GHz [21]. As for the Agilex 5, depending on the series used, the clock can be operated at either 500 MHz or 600 MHz [22]. Table 4.5 shows the time it would take for the SDRAM and SRAM designs to run if they were operated on a 500 MHz clock. With the 500 MHz clock operating the design, the total time for the Frequency Transform is decreased by a factor of 8. Table 4.6 provides calculations with a 600 MHz clock and Table 4.7 provides a table with the potential time the design would take operating on the Stratix 10 1 GHz clock.

Table 4.5 SDRAM and SRAM total time using 500 MHz clock

	Clock Cycles	**Time (ms)**
SDRAM	12,541,264	25.0825
SRAM	1,337,417	2.6748

Table 4.6 SDRAM and SRAM total time using 600 MHz clock

	Clock Cycles	**Time (ms)**
SDRAM	12,541,264	20.9021
SRAM	1,337,417	2.2290

Table 4.7 SDRAM and SRAM total time using 1 GHz clock

	Clock Cycles	**Time (ms)**
SDRAM	12,541,264	12.5413
SRAM	1,337,417	1.3374

Chapter 5 - Conclusion

This thesis presents a portable hardware accelerator for FFT-based Sobel Edge Detection on a low-cost SoC-FPGA device. The state machine created in the FPGA fabric has the capability to be transferred over to other SoC-FPGA devices that utilize the Quartus software. The software written in the design is also fully portable to SoC-FPGA's that utilize a Linux distribution for the HPS side. This portability allows future users to adopt this design and improve upon it further.

The resulting images from the SoC-FPGA are identical to the ones produced by MATLAB, both visually and in terms of image data. This provides assurance that 2-D image frequency transforms can be used in place of convolution without the fear of loss in accuracy, and that the design will work correctly in its current iteration.

The SoC-FPGA design provided in this thesis serves as a good baseline for creating a system that can rival the speeds of current technology. The timing results for the design show the capability of a SoC-FPGA design that can rival the speed of an Intel i9-13900h CPU running MATLAB at 5.4 GHz. Some potential ways that the speed could be increased were shown in Chapter 4, Section 4.4, and Section 4.5. Other ways in which future users can improve the performance of the SoC-FPGA design are through creating custom FFT modules and using techniques such as overlap and add that were discussed in Chapter 1, section 1.2. With Kalb et al. [9] showing that GPUs are inefficient in low power environments and Qasaimeh et al. [2] showing that FPGAs have the least power consumption between GPUs and CPUs for high performance computing, this design can be adapted to a low power platform that will be more power efficient than using a GPU or CPU.

References

1. Fradkov, A. L. (2020). “Early history of machine learning”. *IFAC-PapersOnLine*, *53*(2), 1385-1390.

2. M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno and P. H. Jones, "Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels," *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, Las Vegas, NV, USA, 2019, pp. 1-8, doi: 10.1109/ICESS.2019.8782524.

3. Khetrapal, Pushkar. “Fast-CNN: Substitution of Convolution Layers with FFT Layers.” *Medium*, Analytics Vidhya, 4 Nov. 2020, medium.com/analytics-vidhya/fast-cnn-substitution-of-convolution-layers-with-fft-layers-a9ed3bfdc99a.

4. V. Nair, M. Chatterjee, N. Tavakoli, A. S. Namin and C. Snoeyink, "Optimizing CNN using Fast Fourier Transformation for Object Recognition," *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Miami, FL, USA, 2020, pp. 234-239, doi: 10.1109/ICMLA51294.2020.00046.

5. Ronneberger, O., Fischer, P., Brox, T. (2015). “U-Net: Convolutional Networks for Biomedical Image Segmentation.” In: Navab, N., Hornegger, J., Wells, W., Frangi, A. (eds) Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015. MICCAI 2015. Lecture Notes in Computer Science(), vol 9351. Springer, Cham. https://doi.org/10.1007/978-3-319-24574-4_28

6. Y. Liang, L. Lu, Q. Xiao and S. Yan, "Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs," in *IEEE Transactions on Computer-Aided Design of*

Integrated Circuits and Systems, vol. 39, no. 4, pp. 857-870, April 2020, doi: 10.1109/TCAD.2019.2897701.

7. T. Abtahi, C. Shea, A. Kulkarni and T. Mohsenin, "Accelerating Convolutional Neural Network With FFT on Embedded Hardware," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 9, pp. 1737-1749, Sept. 2018, doi: 10.1109/TVLSI.2018.2825145.

8. Adam Page, Ali Jafari, Colin Shea, and Tinoosh Mohsenin. 2017. “SPARCNet: A Hardware Accelerator for Efficient Deployment of Sparse Convolutional Networks.” *J. Emerg. Technol. Comput. Syst. 13*, 3, Article 31 (July 2017), 32 pages. https://doi.org/10.1145/3005448

9. T. Kalb *et al*., "TULIPP: Towards ubiquitous low-power image processing platforms," *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, Agios Konstantinos, Greece, 2016, pp. 306-311, doi: 10.1109/SAMOS.2016.7818363.

10. Wikimedia Foundation. (2024, October 8). “Convolution”. *Wikipedia*. https://en.wikipedia.org/wiki/Convolution

11. Sinha, S. (2023, April 4). “All about convolutions”. *Medium*. https://medium.com/nerd-for-tech/all-about-convolutions-331471b9e5e5

12. Arham, M. (n.d.). “Diving into the pool: Unraveling the magic of CNN pooling layers.” *KDnuggets*. https://www.kdnuggets.com/diving-into-the-pool-unraveling-the-magic-of-cnn-pooling-layers

13. BYJU. “Linear interpolation formula: Definition, formula, solved examples.” *Toppr*. (2020, March 31). https://www.toppr.com/guides/maths-formulas/linear-interpolation-formula/

14. Wikimedia Foundation. (2024, October 11). “Bilinear interpolation”. *Wikipedia*. https://en.wikipedia.org/wiki/Bilinear_interpolation

15. Intel. “Intel® FPGA University program DE1-SOC Computer Manual.” https://ftp.intel.com/Public/Pub/fpgaup/pub/Intel_Material/18.1/Computer_Systems/DE1-SoC/DE1-SoC_Computer_NiosII.pdf

16. FFmpeg. (n.d.). https://www.ffmpeg.org/

17. Gisselquist Technology, LLC. (2018, October 2). “An Open Source Pipelined FFT Generator.” https://zipcpu.com/dsp/2018/10/02/fft.html

18. Intel. “External Bus to Avalon® Bridge.” https://ftp.intel.com/Public/Pub/fpgaup/pub/Intel_Material/18.1/University_Program_IP_Cores/Bridges/External_Bus_to_Avalon_Bridge.pdf

19. Altera Corporation. “Avalon Memory Mapped Slave” Qsys Editor

20. Ding, X., Zhang, X., Zhou, Y., Han, J., Ding, G., & Sun, J. (2022, April 2). “Scaling up your kernels to 31x31: Revisiting large kernel design in cnns.” *arXiv.org*. https://arxiv.org/abs/2203.06717

21. Intel. “1.3. FPGA and SOC features summary.” https://www.intel.com/content/www/us/en/docs/programmable/683729/current/fpga-and-soc-features-summary.html

22. Intel. “AgilexTM 5 FPGA and SOC FPGA product overview.” https://www.intel.com/content/www/us/en/products/details/fpga/agilex/5.html