

7-26-2021

## Hardware Acceleration in Image Stitching: GPU vs FPGA

Joshua David Edgcombe

Follow this and additional works at: <https://scholarworks.gvsu.edu/theses>



Part of the [Hardware Systems Commons](#)

---

### ScholarWorks Citation

Edgcombe, Joshua David, "Hardware Acceleration in Image Stitching: GPU vs FPGA" (2021). *Masters Theses*. 1018.

<https://scholarworks.gvsu.edu/theses/1018>

This Thesis is brought to you for free and open access by the Graduate Research and Creative Practice at ScholarWorks@GVSU. It has been accepted for inclusion in Masters Theses by an authorized administrator of ScholarWorks@GVSU. For more information, please contact [scholarworks@gvsu.edu](mailto:scholarworks@gvsu.edu).

Hardware Acceleration in Image Stitching: GPU vs FPGA

Joshua D. Edgcombe

A Thesis Submitted to the Graduate Faculty of

GRAND VALLEY STATE UNIVERSITY

In

Partial Fulfillment of the Requirements

For the Degree of

Master of Science in Engineering  
Electrical & Computer Engineering

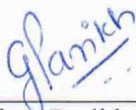
School of Engineering

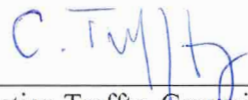
August 2021

## Thesis Approval Form



The signatories of the committee members below indicate that they have read and approved the thesis of Joshua David Edgcombe in partial fulfillment of the requirements for the degree of Master of Science in Engineering, Electrical and Computer Engineering.

 07-19-2021  
Dr. Chirag Parikh, Thesis committee chair Date

 07-19-2021  
Dr. Christian Trefftz, Committee member Date


 07-19-2021  
Dr. Nabeeh Kandalaft, Committee member Date

Accepted and approved on behalf of the  
Padnos College of Engineering and Computing

  
Dean of the College

7/19/21  
Date

Accepted and approved on behalf of the  
Graduate Faculty

  
Associate Vice-Provost for the Graduate School

7/26/2021  
Date

© Copyright by Joshua D. Edgcombe 2021

All Rights Reserved

## **DEDICATION**

First and foremost, I would like to dedicate this paper to my Lord and Savior Jesus Christ. He is my strength and my rock; my life would not be what it has been without Him.

## **ACKNOWLEDGMENTS**

I would like to thank my committee, including Dr. Parikh, Dr. Trefftz, and Dr. Kandalaft for their guidance and help throughout the process of developing and writing this thesis. Additionally, I would like to thank all the professors and faculty that have helped and allowed me to get to this point. Notably, Dr. Dunne, Dr. Ward, and Dr. Brakora for their assistance in achieving admission to the master's program.

I would also like to thank DornerWorks for working with me throughout my time in the master's program. Not only did they provide me time and space to balance full-time work and thesis studies, but they also provided me access to phenomenal engineers with decades of experience in industry that were always willing to answer questions or give direction. The entire DornerWorks FPGA team was extremely helpful during the FPGA design and development process.

Finally, I would like to thank my family for always being there for me regardless of how difficult balancing my responsibilities became. I could not ask for a better and more reliable support structure than the family I was given.

## **ABSTRACT**

### **Hardware Acceleration in Image Stitching: GPU vs FPGA**

Image stitching is a process where two or more images with an overlapping field of view are combined. This process is commonly used to increase the field of view or image quality of a system. While this process is not particularly difficult for modern personal computers, hardware acceleration is often required to achieve real-time performance in low-power image stitching solutions. In this thesis, two separate hardware accelerated image stitching solutions are developed and compared. One solution is accelerated using a Xilinx Zynq UltraScale+ ZU3EG FPGA and the other solution is accelerated using an Nvidia RTX 2070 Super GPU. The image stitching solutions implemented in this paper increase the system's field of view and involve the end-to-end process of feature detection, image registration, and image mixing. The latency, resource utilization, and power consumption for the accelerated portions of each system are compared and each systems tradeoffs and use cases are considered.

AUGUST 2021

JOSHUA D. EDGCOMBE, B.S.E., GRAND VALLEY STATE UNIVERSITY

M.S.E., GRAND VALLEY STATE UNIVERSITY

Directed by: Professor Chirag Parikh

## TABLE OF CONTENTS

ACKNOWLEDGMENTS .....	5
ABSTRACT.....	6
TABLE OF CONTENTS.....	7
LIST OF TABLES .....	10
LIST OF FIGURES .....	11
INTRODUCTION .....	13
1.1 Problem Statement.....	13
1.2 Scope.....	14
1.3 Layout of Thesis .....	15
Literature Review.....	17
2.1 Field Programmable Gate Arrays .....	17
2.2 Graphical Processing Units.....	20
2.3 Image Stitching .....	22
2.4 Feature Detection .....	23
2.4.1 Harris-Stephens Corner Detectors .....	24
2.4.2 Scale-Invariant Feature Transform (SIFT) .....	25
Scale-Space .....	26
Feature Detection.....	27
Feature Description.....	29
2.4.3 Speeded Up Robust Features (SURF) .....	31
Integral Image .....	32
Box Filters.....	33
Feature Detection.....	35
Feature Description.....	39
2.5 Image Registration [Homography] .....	43
Homography Calculation .....	44
2.5.1 Assumption Method.....	45
2.5.2 SVD Method .....	46
2.6 Mixing/Blending .....	46
Implementation .....	48
3.1 Development Environments.....	48
3.1.1 FPGA .....	48
3.1.2 GPU.....	49



3.2 General Design Decisions .....	49
3.2.1 Image Preprocessing .....	50
3.2.2 SURF Parameters .....	51
Dominant Orientation Parameters .....	52
Feature Description .....	53
3.2.3 Matching Parameters .....	54
3.2.4 Homography Parameters .....	54
3.2.5 Image Mixing .....	55
3.3 FPGA Design .....	55
3.3.1 Feature Detection Subsystem .....	57
Integral Image .....	59
Filtering .....	61
Non-maximal Value Suppression (NMS) .....	66
3.3.2 Feature Description .....	69
3.4 GPU Design .....	70
3.4.1 SURF Feature Detection .....	72
Integral Image .....	73
Filtering .....	74
Non-maximal Value Suppression (NMS) .....	76
3.4.2 Feature Description .....	79
Results & Discussion .....	81
4.1 Tests .....	81
4.1.1 Latency .....	81
4.1.2 Power .....	82
4.1.3 Resources .....	82
4.2 Test Methods .....	83
4.2.1 FPGA .....	83
Latency .....	83
Power .....	84
Resources .....	84
4.2.2 GPU .....	84
Latency .....	84
Power .....	85
Resources .....	86
4.3 Results .....	86
4.3.1 Latency .....	86
FPGA .....	86
Integral Image .....	87
SURF Filtering .....	87
Non-maximal Value Suppression .....	89
GPU .....	89
Integral Image .....	90
SURF Filtering .....	90
Non-maximal Value Suppression .....	91
4.3.2 Power .....	92

FPGA .....	92
GPU .....	93
4.3.3 Resources .....	94
FPGA .....	95
Integral Image .....	96
Filtering.....	96
NMS .....	97
GPU .....	97
Integral Image .....	98
Filtering.....	98
NMS .....	99
4.4 Summary .....	99
4.4.1 Latency.....	99
4.4.2 Power .....	100
4.4.3 Resources .....	100
Future Work.....	102
5.1 Potential Design Improvements.....	102
Conclusion .....	103
6.1 Summary of Work.....	103
6.2 Summary of Results.....	103
APPENDIX A: Figures.....	104
Bibliography .....	109

## LIST OF TABLES

Table	Page
Table 1. SURF Implementation Parameters .....	52
Table 2. FPGA SURF Filter Constants.....	65
Table 3. FPGA SURF Implementation Comparisons.....	87
Table 4. FPGA SURF Filtering Latencies .....	89
Table 5. FPGA NMS Latencies .....	89
Table 6. GPU Filtering Latencies .....	91
Table 7. GPU NMS Latencies .....	92
Table 8. Summary of Test Results.....	101
Table 9. GPU Dxx Filtering Latencies .....	106
Table 10. GPU Dyy Filtering Latencies .....	107
Table 11. GPU Dxy Filtering Latencies .....	107
Table 12. GPU SURF Filtering Latencies .....	107

## LIST OF FIGURES

Figure	Page
Figure 1. FPGA Architecture.....	18
Figure 2. GPU Architecture .....	20
Figure 3. SURF Feature Point Detection .....	22
Figure 4. Image Registration (Feature Point Matching) .....	23
Figure 5. Stitched Image Output Example.....	23
Figure 6. Scale-Space Example .....	27
Figure 7. SIFT Scale-Space Implementation .....	28
Figure 8. Non-Maximal Value Suppression .....	29
Figure 9. SIFT Feature Point Descriptor.....	31
Figure 10. Integral Image Calculation Example .....	32
Figure 11. First Order Gaussian Box Filter Comparison ( $D_x$ ) .....	34
Figure 12. Second Order Gaussian Box Filter Comparison ( $D_{yy}$ ) .....	34
Figure 13. Second Order Gaussian Box Filter Comparison ( $D_{xy}$ ) .....	34
Figure 14. $D_{xx}$ Box Filter .....	35
Figure 15. $D_{yy}$ Box Filter .....	36
Figure 16. $D_{xy}$ Box Filter .....	36
Figure 17. $D_x$ Box Filter.....	37
Figure 18. $D_y$ Box Filter.....	37
Figure 19. SURF Descriptor Local Region.....	42
Figure 20. FPGA Feature Detection High-Level Architecture.....	56
Figure 21. FPGA – SURF Feature Detection Subsystem .....	59

Figure 22. Integral Image Calculation Code.....	61
Figure 23. Integral Image Value Calculation Function .....	62
Figure 24. $D_{yy}$ Filter Value Calculation Processes .....	63
Figure 25. Signed to Floating Point Standard Logic Vector Conversion Function .....	64
Figure 26. SURF Filtered Value Calculation Blocks.....	66
Figure 27. NMS Calculation and Output Assignment .....	67
Figure 28. Floating Point Comparison Function.....	68
Figure 29. FPGA 32-bit Feature Point Row, Column, and Octave Encoding .....	69
Figure 30. GPU SURF Algorithm Dataflow Diagram.....	72
Figure 31. GPU Integral Image Calculation Kernel .....	74
Figure 32. GPU Accelerated $D_{xx}$ Filter Kernel .....	75
Figure 33. GPU SURF Filter Value Kernel .....	76
Figure 34. GPU NMS Comparison Kernel .....	78
Figure 35. FPGA Power Estimates .....	93
Figure 36. Power Usage With GPU Accelerated Program Running .....	94
Figure 37. Post-synthesis resource utilization .....	95
Figure 38. Post-implementation resource utilization .....	96
Figure 39. FPGA Top Level Block Diagram (Full).....	104
Figure 40. FPGA Top Level Block Diagram (Left) .....	105
Figure 41. FPGA Top Level Block Diagram (Right) .....	106
Figure 42. Nvprof Profiling of GPU Accel.....	108

## **CHAPTER 1**

### **INTRODUCTION**

Two hardware accelerated implementations of an image stitching algorithm are developed and compared in this thesis. The image stitching algorithm is composed of three main components: feature detection and description, feature matching, and image registration. The feature detection and description portion of the algorithm finds sufficiently unique points in an image in a repeatable manner and describes them in a robust and repeatable way. The feature matching portion of the algorithm matches features from one image to similar features in a second image. The image registration process converts the two separate images to a common coordinate system. These three steps make up the core of the hardware accelerated image stitching algorithm used in the image stitching solutions developed.

Looking at the two hardware accelerated image stitching solutions, the latency, resource utilization, and power consumption are analyzed and compared for the accelerated portions of the image stitching process mentioned above. Using the results of these individual comparisons, the full systems are compared and their tradeoffs discussed.

#### **1.1 Problem Statement**

Due to the computational complexity of the image stitching process, power consumption can be high to achieve real-time performance in an image stitching solution. While power is not a concern for all designs, low power solutions are often required for embedded or edge computing solutions. In the past, computationally complex image

processing has been performed using graphical processing units (GPUs), but due to advances in field programmable gate array (FPGA) technology these tasks have become more viable with lower power consumption FPGA devices. The goal of this thesis is to use two common methods of hardware acceleration, FPGAs and GPUs, to develop image stitching solutions used to increase the field of view of a system and then compare them.

## 1.2 Scope

In this thesis, two hardware accelerated image stitching solutions are implemented and their performance compared. The two methods of hardware acceleration being compared are field programmable gate arrays (FPGAs) and graphical processing units (GPUs). The speeded up robust features (SURF) algorithm [1] is used for the feature detection and description portions of the image stitching process. An exhaustive comparison of two linked lists is used to perform feature point matching. Random sample consensus (RANSAC) is used for matched point selection to calculate the homography matrix for the image registration process. Image mixing is performed using the nearest neighbor method.

While the image stitching algorithms and methodology were kept as similar as possible between the two hardware acceleration methods, design decisions specific to each implementation are noted where they occur. Differences in the algorithms or methodology are only present where hardware limitations prevent the use of the original algorithm or methodology.

The factors to be compared are latency, resource utilization, and power consumption. Latency will be measured from the time the first pixel enters the feature detection portion of the system to the time the first possible feature point location can be

output from the feature detection portion of the system and will be measured in clock cycles. Resource utilization will look at the amount of resources required for the hardware accelerated portion of each implementation and will be measured in percentage of total resources utilized. The effects of the percentage of total resource utilization on each type of hardware will also be discussed. Power consumption will be measured for each solution in watts.

This thesis only covers the portions of the image processing pipeline that are necessary for hardware acceleration of the image stitching algorithm. Implementation details specific to the image processing pipelines for each implementation such as image encoding or decoding, pixel value preprocessing functions such as gamma correction or pixel debayering, camera initialization values and processes, DMA component specifics, and output components such as DisplayPort encoding or windowing systems used will be excluded from description and analysis in this paper to retain a focus on the accelerated image stitching algorithm performance in both systems.

### **1.3 Layout of Thesis**

The next section of this thesis will cover the literature and background material used as a basis for the implementation of the systems. A quick introduction to FPGAs, GPUs, and the image processing algorithms used in this thesis will be covered at a theoretical level in this section. By the end of this section, the reader will have all the knowledge necessary to understand this thesis on an abstract theoretical level.

The third section will cover the implementation details and design decisions made for each form of hardware acceleration used in this paper. This section will include deeper explanations of the algorithms used to implement the theory discussed in the



previous section as well as implementation details for each form of hardware acceleration. Diagrams will be used to help visualize the high-level organization of the systems on both forms of hardware acceleration. Additionally, the methods used for measuring the performance metrics on each form of hardware acceleration will be covered in this section.

The fourth section will cover the results of the tests performed as well as general comments on the performance of each system. The effects of the differences in the implementation specific design decisions will be considered and different use cases will be discussed in this section.

The final section will discuss potential improvements for future versions of each of the systems and include a discussion of the future of the technologies used in these implementations. With a general understanding of what is included in this thesis, we can now start reviewing the technologies and theory behind the algorithm used for both implementations.

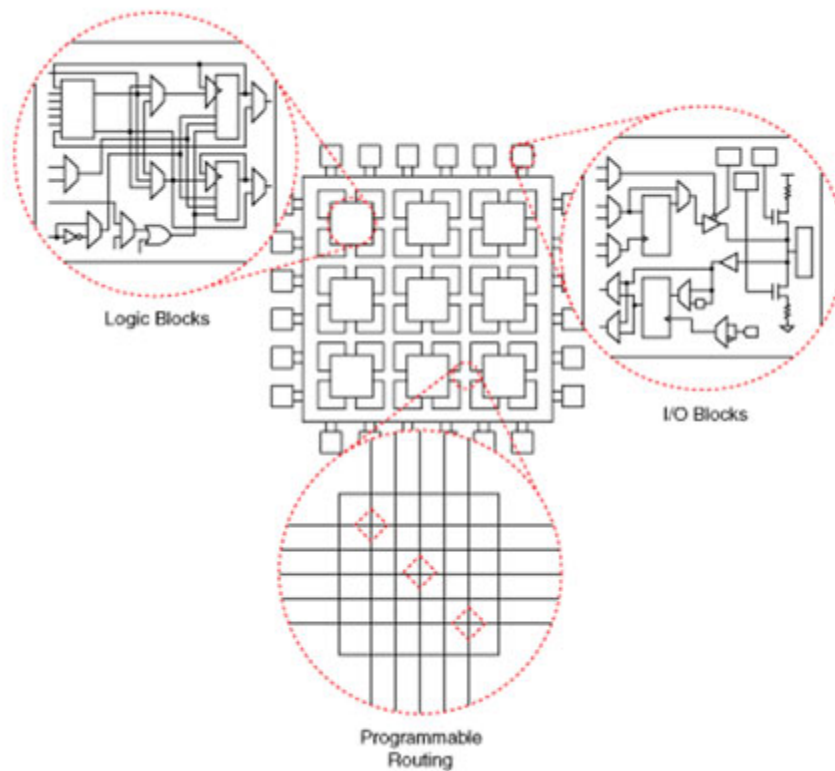
## CHAPTER 2

### LITERATURE REVIEW

#### 2.1 Field Programmable Gate Arrays

Field programmable gate arrays (FPGAs) are a type of integrated circuit in which the physical circuitry can be modified using a hardware description language (HDL) such as VHDL or Verilog. The first FPGAs available for purchase were produced by Xilinx in 1984 [2] and were sold as a great solution for achieving near application specific integrated circuit (ASIC) performance while retaining the programmability of software. Depending on the requirements of the system, FPGA implementations are commonly targeted to increase the speed or reduce the power consumption of the device.

On a high level, the basic architecture of an FPGA involves the connection of input and output pins to functional cores using programmable routing lines. There are a variety of functional components that are available depending on the specific FPGA being considered, but only the most common will be discussed and considered in this paper. A graphical representation of the high-level architecture of an FPGA can be seen in Figure 1 below [3].



**Figure 1. FPGA Architecture**

While the names used to reference the pieces inside an FPGA differ between manufacturers, three main components are common in nearly all modern FPGAs. The three main components common to nearly all modern FPGAs are the memory, logic, and math components. The memory available to an FPGA generally includes on-chip memory, such as block ram (BRAM), look-up tables (LUTs), and registers. Depending on the system, an FPGA can also be given access to off-chip memory such as DDR or SRAM modules. The logic performed on an FPGA uses LUTs and flip-flops to implement any arbitrary logical function that can be defined in HDL. Due to the necessity of floating-point mathematic operations in many modern systems, most modern FPGAs also include floating-point specific algebra blocks. The floating-point algebra blocks available on Xilinx devices are called digital signal processing (DSP) slices and are

named DSP slices due to how common floating-point operations are in digital signal processing applications. Modern FPGAs also often include peripherals such as serializer-deserializers (SERDES) for high-speed communication and DDR that can be used on device specific pins. While these three types of modules can generally be found on any modern microcontroller, the big advantage an FPGA provides is the ability to electronically connect these blocks together using HDL into an extremely low latency and time-efficient system.

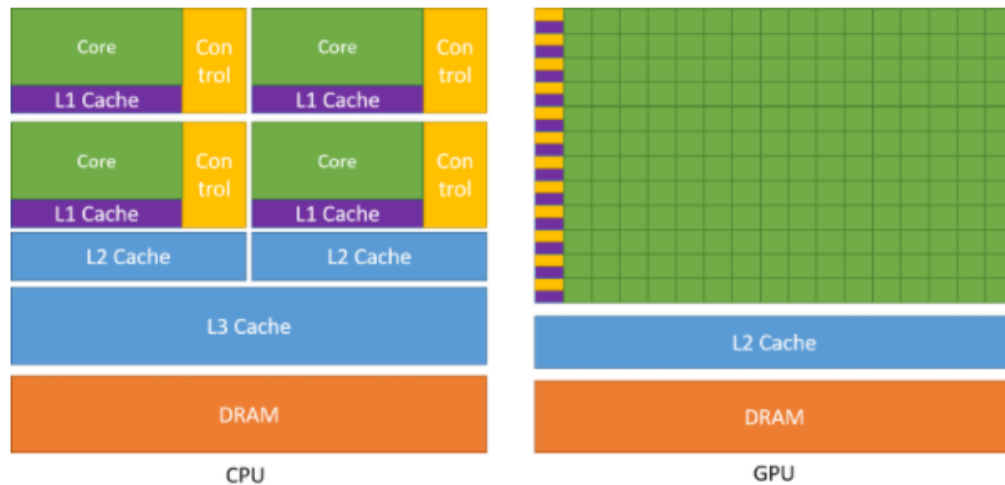
The ability to electronically connect these blocks together in an arbitrary manner as defined by HDL provides the programmability aspect of FPGAs while providing near ASIC performance. Complex operations can be broken up into many clock cycles to achieve higher clock frequencies and throughput. This is design technique known as pipelining. Additionally, the number of clock cycles necessary to perform certain logical and mathematical operations can be reduced using custom logic, but may result in a lower clock speed for the system. The FPGA also provides cheap and easy parallelization of custom logic operations. Each operational pipeline can be duplicated as many times as the FPGA's resources allow and can all simultaneously access data in a single clock cycle without affecting the performance of the other pipelines if a common data source is used.

The areas that FPGAs can excel in are power consumption, latency, and parallelization. It is worth noting that FPGAs work best on streams of data. The main disadvantage of using an FPGA is the level of expertise required to include one in a system and the relatively low number of qualified and available engineers.

## 2.2 Graphical Processing Units

Graphical processing units (GPUs) are a type of integrated circuit designed specifically for accelerating common graphical operations. While different algorithms and hardware were used to accelerate graphical operations prior to GPUs, the first official GPU was produced for sale by Nvidia in 1999 [4]. The first GPUs were sold as a device capable of offloading graphical processing from the CPU and accelerating operations common to computer graphics.

One of the main types of operations GPUs excel at performing is the floating-point algebraic operation, particularly when performed on large sets of data. As such, GPUs tend to provide a large amount of high-speed memory resources with many smaller distributed GPU-specific floating point and logical operation processing cores. A high-level graphical representation of the architecture of a modern Nvidia GPU can be found in Figure 2 below [5].



**Figure 2. GPU Architecture**

The large memory space is commonly used for image or video frame buffers for the frames being rendered or otherwise manipulated prior to rendering when used for

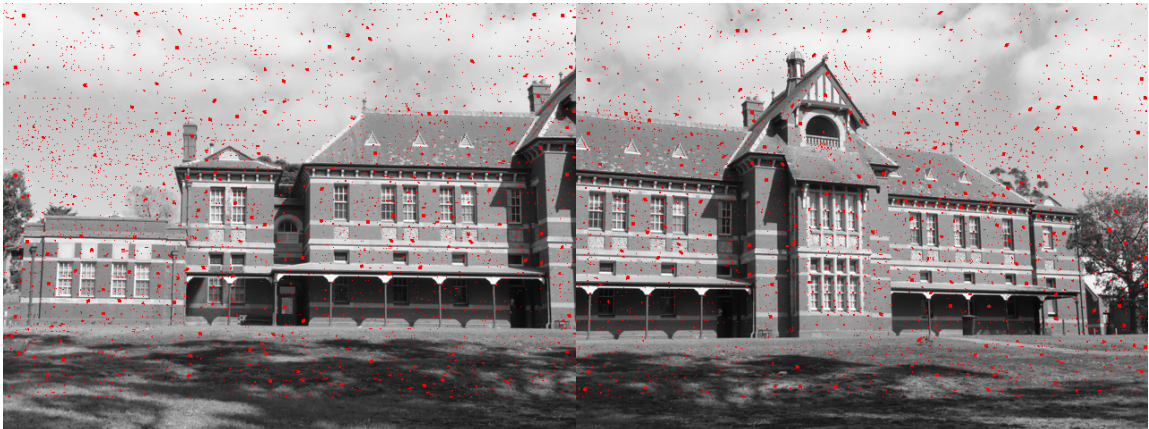
image processing. GPUs are built around the concept of single-instruction, multiple data (SIMD) hardware acceleration. In order to utilize this form of acceleration, latency is introduced when initializing the memory space or transferring data between the host and the GPU device. While latency is introduced when transferring data or initializing memory, performing mathematical and logical operations on the data in the memory space is highly accelerated.

One of the big disadvantages of GPUs is that the actual integrated circuit chips are not as readily available as FPGAs for hardware engineers to get their hands on. Without the ability to buy the individual chips, full personal computer systems generally need to be built to utilize the hardware acceleration that can be gained using the GPU. This increases the cost, reduces the customizability, and increases the power consumption of the system. With that being said, GPUs are relatively easy to obtain with PCI-E connectivity for applications where power is not a central concern.

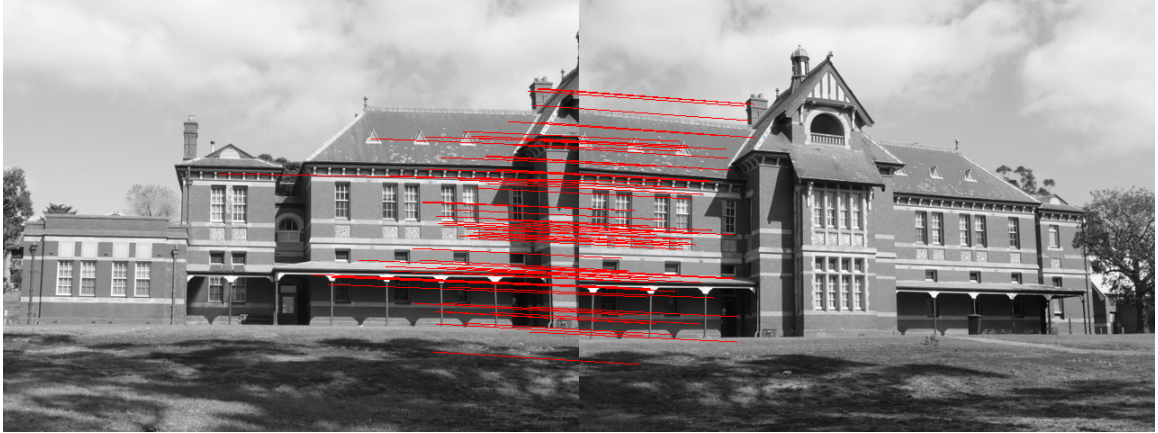
GPUs are extremely good at applying a series of instructions to an exceptionally large data set in parallel. This advantage makes GPUs very well suited for large linear algebra operations. Another advantage of GPUs is the accessibility of libraries that provide a relatively easy to use programming interface. While there is a slight learning curve for using these libraries and understanding their design principles, learning to write a program to be accelerated using a GPU is much more accessible to a programmer than learning to write HDL for an FPGA.

## 2.3 Image Stitching

Image stitching is the process of merging together two or more images to create a single image that combines the overlapping field of view of all images to increase the field of view or image quality of a system. In this case, two or more images are being stitched together to create an image mosaic using multiple image sources. The three main steps involved in the image stitching process are feature detection, image registration, and image mixing. Feature detection is the process of finding unique points in an image and describing them in a robust and repeatable manner. Image registration is the process of using the feature points from the feature detection process to map two images to a common coordinate system. Image mixing is the process of combining the two or more images mapped to a common coordinate system in the image registration step into a single output image or video. An example of the steps involved in stitching together two images can be seen in the figures below.



**Figure 3. SURF Feature Point Detection**



**Figure 4. Image Registration (Feature Point Matching)**



**Figure 5. Stitched Image Output Example**

## **2.4 Feature Detection**

Feature detection is the process of finding unique points in an image and describing them in a robust and repeatable manner. This means that the performance of a feature detection algorithm can be determined and compared by how likely it is to find the same point in multiple images regardless of differences in lighting, rotation, translation, scale, or other forms of optical noise or transformations. The more likely an algorithm is to find and describe the same location in two or more perspectives of the



same scene with a similar, or the same, feature description, the better the performance of the algorithm.

There are a variety of feature detection algorithms available in the field of computer vision and each has their own tradeoffs. For this project, the Speeded Up Robust Features (SURF) algorithm was used because it is known for having a good balance between performance and speed of execution. Originally, the Scale-Invariant Feature Transform (SIFT) algorithm [6] was planned to be used, but resource issues arose during development of the FPGA accelerated solution.

During development of the FPGA accelerated portion of the thesis, floating point resource availability required the switching of the algorithm to a less resource hungry algorithm. With performance comparable to SIFT and a significant reduction in resource usage, SURF was an obvious replacement candidate. In order to understand how SURF functions and where it came from, a quick look at Harris Corner Detectors and the SIFT algorithm is appropriate.

#### **2.4.1 Harris-Stephens Corner Detectors**

The Harris-Stephens Corner Detector [7] was one of the first feature detection algorithms used in computer vision. This algorithm starts from the basic idea that the autocorrelation function applied to a region can be used to determine if a pixel is sufficiently different than neighboring pixels. The algorithm works by using the partial derivatives of the autocorrelation function to find pixel locations with high x and y partial derivatives indicating a locally unique point. This algorithm is rotation and translation invariant, but only includes the feature point detection process. Although no feature point description is associated with this algorithm, it is possible to use the SIFT or SURF

feature description algorithms with the Harris Corner Detector algorithm. A deeper explanation of this algorithm can be found in Appendix B.

While the Harris-Stephens Corner Detection algorithm provided a strong method for finding useful features in an image, it was not scale-invariant which is a significant issue for many modern applications of image processing. To achieve scale invariance, one of the best performing modern feature detection algorithms incorporates the concept of scale-space filtering and is called the SIFT algorithm.

#### **2.4.2 Scale-Invariant Feature Transform (SIFT)**

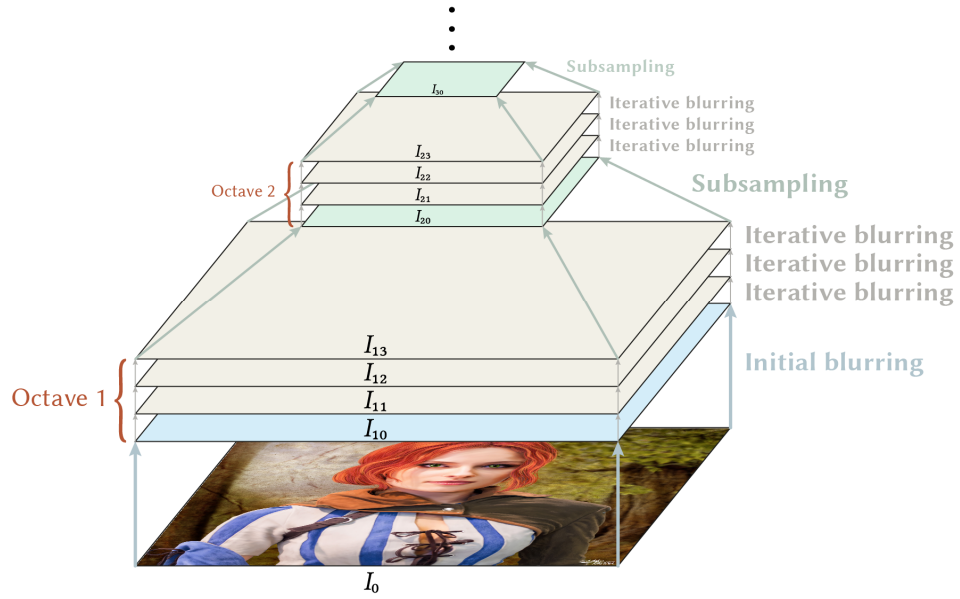
SIFT is one of the best performing feature detection algorithms currently available but requires a significant amount of processing to be performed. SIFT, like SURF, can be broken down into two main components: feature detection and feature description. Feature detection is the process of finding distinct points in one or more images with high repeatability. Feature description is the process of describing the detected feature points in a way that they can be matched to other feature points that have similar regional or local pixel values. As an example, describing a point on a car tire using the SIFT feature point description process should be described similarly to, or the same as, other descriptions of the same location on a car tire in other pictures containing car tires regardless of lighting, rotation, blur, scale, etc.

While there are many factors to be considered in these algorithms, the most commonly considered transforms are scale, rotation, and translation invariance. When running multiple images through the feature detection portion of the algorithm, the same points should be found in each image regardless of how the image is rotated, shifted, or how far away the image is taken from any point of interest. Additionally, feature points

detected in regions of overlapping fields of view should be described similarly in all involved images. While rotation and translation invariance has been achieved in previous feature detection algorithms, achieving scale-invariance in the SIFT algorithm is accomplished using scale-space.

### **Scale-Space**

The concept of scale-space is used to achieve scale-invariance in the SIFT algorithm. Scale-space was originally discussed by Andrew Witkin in 1983 [8]. The scale-space representation of an image can be used to detect objects or features in a scale-invariant manner. The method approximates how human biological vision works to recognize images at various distances or scales. A scale-space set of images is built by creating a set of images with a gaussian blur applied of up to  $\sigma=2$ . This set of successively blurred images is known as an octave. The next octave is created by subsampling the image blurred to an effective  $\sigma=2$  value down to a scale of half and thus reducing the loss of information during subsampling to an optimal level. A visualization of the gaussian blur image pyramid used in scale-space can be seen in Figure 6 below [9].



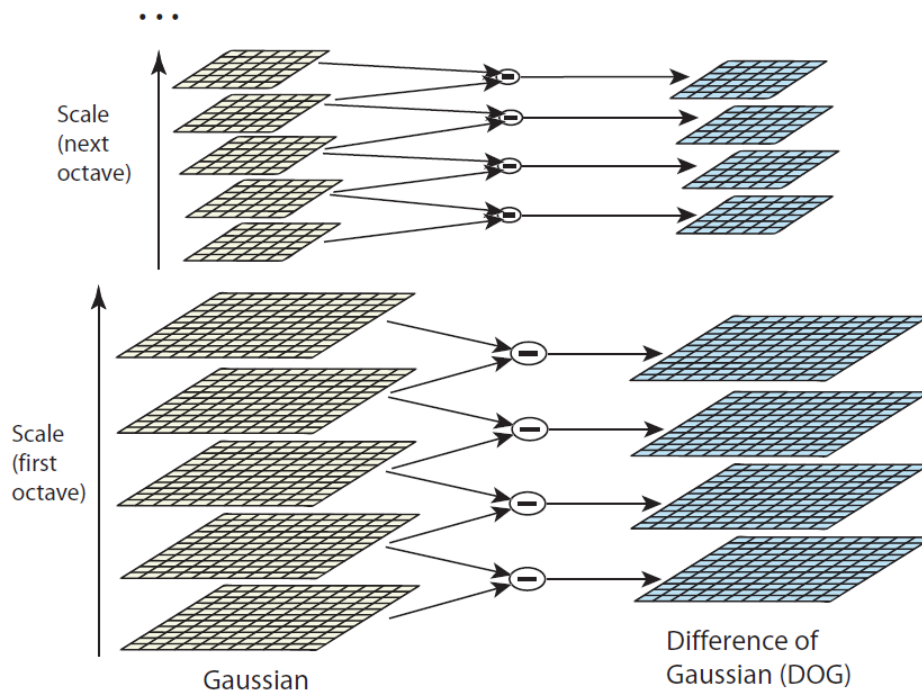
**Figure 6. Scale-Space Example**

This concept can be used for any number of octaves and filtered levels within the octaves that the algorithm designer desires. The gaussian blur filters being applied to each octave can be either applied iteratively or in parallel to result in an overall gaussian blur level that reduces loss of information between octaves due to subsampling to an optimal level.

## Feature Detection

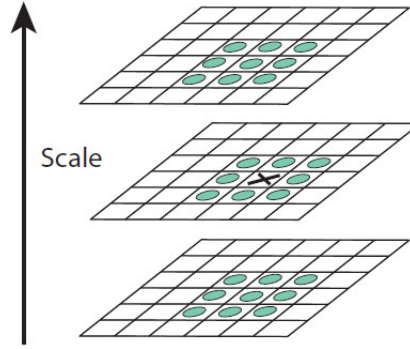
SIFT accomplishes scale invariance by using a gaussian filtered image pyramid, otherwise known as scale-space. After the image is filtered for all filtered levels of the selected octaves, a difference of gaussian (DoG) calculation is made between adjacent filter levels within the same octave and a non-extrema suppression filter is used to select distinct points from the DoG images. The non-extrema value suppression filter compares the DoG values at one gaussian filter level with the DoG values associated with the neighboring DoG values immediately above and below in the scale-space image pyramid. The SIFT image pyramid can be seen graphically in Figure 7 below [10]. The difference

of gaussian is used as a blob detection technique and approximates the Laplacian of the gaussian.



**Figure 7. SIFT Scale-Space Implementation**

The non-extrema suppression looks for both local maxima and local minima. The only pixels compared are the eight pixels on the current DoG level immediately surrounding the current pixel location in question, the 3x3 DoG area directly above, and the 3x3 DoG area directly below. This results in a 3x3x3 cube determining if the middle pixel value is the local extrema. This can be seen visually in Figure 8 below [10]. This same principle is applied for all desired image scales and octaves with two exceptions per octave. The two DoG layers excluded per octave are the first and last DoG layer in each octave due to the lack of both an upper and lower DoG for comparison. After the feature points have been detected and their scale and location recorded, the feature points are then described.



**Figure 8. Non-Maximal Value Suppression**

### Feature Description

The SIFT algorithm includes the definition of a 128-value feature point descriptor. The feature point's location, scale, and orientation must first be adjusted to be as accurate as possible prior to classification. The interpolated location of the feature point is found using a method for fitting a 3D quadratic function to the feature point and its surrounding sample points [6]. Once the exact location of the feature point is found, the dominant orientation must be calculated.

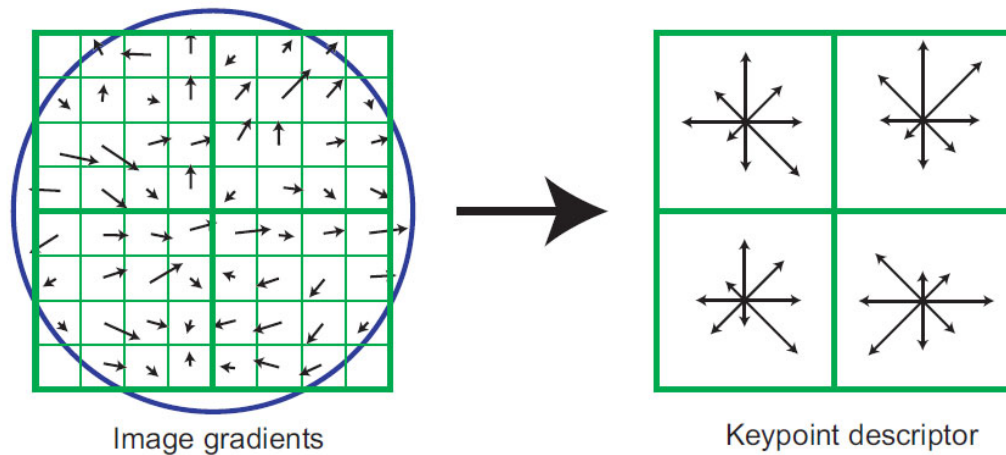
The dominant orientation of the feature point is found using the magnitude  $m(x,y)$  and orientation  $\theta(x,y)$  of the points local to the feature point. The magnitudes of these local points are separated by angle into a 36-bin histogram with each bin representing a ten-degree angle of the full 360 degrees possible. The magnitudes in each bin are then summed to calculate the final bin values. The scale of the feature point is used to determine which Gaussian filtered image,  $L$ , to use for these calculations. The formulas used for the magnitude and orientation calculations can be found below.

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2} \quad (\text{Eq. 2.4.2.1})$$

$$\theta(x, y) = \tan^{-1} \left( \frac{L(x, y+1) - L(x, y-1)}{L(x+1, y) - L(x-1, y)} \right) \quad (\text{Eq. 2.4.2.2})$$

The directional bins associated with the largest summed magnitude and the summed magnitudes of the two neighboring directional bins are used as three points to fit a parabola against to determine a more accurate peak orientation. If no dominant orientation can be found due to multiple equal maximum valued directional bins, feature points will be created using the same location and scale, but with each of the different maximum valued orientations found. Once the dominant orientation for the feature point is determined, the descriptor can be calculated

To be used as a SIFT feature point descriptor, a 4x4 matrix of eight-orientation histograms are used to describe a feature point. The 4x4 matrix is created using the location, scale, and dominant orientation from the previous step and using the gradient values of the points local to the feature point. Each histogram represents a 4x4 sample subregion where the sample spacing is determined using the scale of the feature point. The histogram is the summation of magnitudes with orientation values nearest to one of the eight orientations, or nearest 45 degrees, used to describe the region. A scaled down graphical example of what the conversion to a matrix of histograms from a local subsampled region looks like in practice can be seen in Figure 9 below [10].



**Figure 9. SIFT Feature Point Descriptor**

While the figure above uses only a 2x2 array of eight-orientation histograms, the actual algorithm uses a 4x4 array of eight-orientation histograms for the regions surrounding the feature point resulting in a 128-value feature point descriptor. The subregions used to generate the 4x4 array of eight orientation histograms have a dimension of 4x4 sample locations resulting in a total sample region of 16x16 samples surrounding the feature point. All sample points in the 16x16 sample region have their gradient magnitudes binned by orientation, summed with similarly oriented samples in the same subregion, and simplified down to the 4x4 feature point descriptor array. A gaussian filter is applied to the 16x16 full sample region prior to calculating the histogram for all regions. Additionally, the descriptors are normalized to reduce the effect of low contrast regions on the descriptor.

### **2.4.3 Speeded Up Robust Features (SURF)**

The SURF algorithm functions very similarly to the SIFT algorithm but uses several approximations and a few tricks. The core speed-up provided by SURF is gained



through the use of box filters instead of exact gaussian filters. These box filters approximate the gaussian filters used in SIFT, but do not result in a significant feature detection performance hit. Even though there is not a significant detection performance disadvantage to using SURF, the box filters allow calculation of a pixel's filtered value using significantly fewer calculations. However, in order to use box filters, an integral image must be created first.

### Integral Image

The integral image is used in the SURF algorithm as an intermediary step that greatly increases the performance of the SURF algorithm in later steps. The value of every pixel in an integral image is the sum of the current pixel's intensity and all pixel intensities above and to the left of the current pixel. To demonstrate this visually, an example of a 3x3 block of pixel values can be seen in the figure below. The left side 3x3 block represents the pixel values at each location and the right side 3x3 block represents the integral image values.

		Columns							
		0	1	2			0	1	2
Rows	0	1	2	3	0	1	1 (1)	3 (2+1)	6 (3+3)
	1	4	5	6	1	1	5 (4+1)	12 (5+5+3-1)	21 (6+12+6-3)
	2	7	8	9	2	2	12 (7+5)	27 (8+12+12-5)	45 (9+27+21-12)

**Figure 10. Integral Image Calculation Example**

An important aspect of creating an integral image is the use of an appropriate number of bits to prevent overflow from occurring during the integral image calculation process. Equation 2.4.3.2 can be used where  $N$  represents the number of bits required to store each integral image value,  $n$  represents the number of bits used to store each normalized pixel's intensity value, and  $M$  represents the total number of values to be summed (the number of pixels in an image).

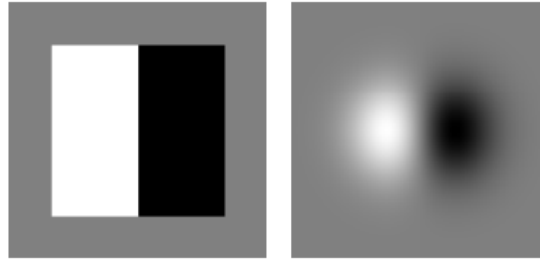
$$N = n + \lceil \log_2(M) \rceil \quad (\text{Eq. 2.4.3.1})$$

Once the integral image has been calculated, the SURF algorithm uses the integral image for the filtering process.

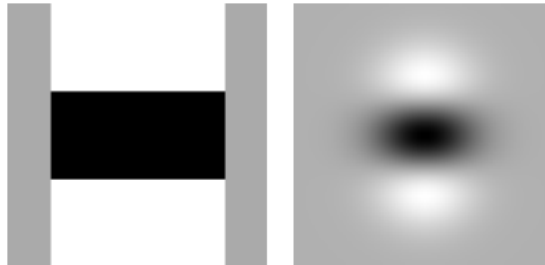
### **Box Filters**

Box filters are used by the SURF algorithm as an approximation to first and second order gaussian filters once an integral image has been calculated. Box filters provide a speed up by allowing the application of filters to be performed with significantly fewer calculations than a typical gaussian filter. While gaussian filters need to apply a different floating point multiplier to all pixels, box filters only need to use the integral image values of a filter's corner locations and don't require any floating point arithmetic.

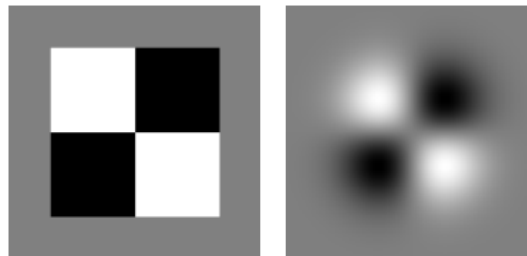
As a comparison of what the integral image will compute versus what the gaussian filters will compute, figures have been included below [10]. In Figure 11, the box filter only requires reading 6 pixel values and performing 7 integer arithmetic operations. In Figure 12, the box filter only requires reading 8 pixel values and 8 integer arithmetic operations. In Figure 13, the box filter only requires reading 9 pixel values and performing 12 integer arithmetic operations.



**Figure 11. First Order Gaussian Box Filter Comparison ( $D_x$ )**



**Figure 12. Second Order Gaussian Box Filter Comparison ( $D_{yy}$ )**



**Figure 13. Second Order Gaussian Box Filter Comparison ( $D_{xy}$ )**

Due to the use of box filters, the down sampling and filtering used in SIFT is not required to apply the filters to different scales of the image. Instead of down sampling and filtering, the size of the box filters being applied is changed using a filter width value. Only needing to change the filter width value allows the application of filters at various scale-space levels to be applied in parallel on the same base image. This greatly reduces the processing and memory resources required for this algorithm.

## Feature Detection

For feature detection in SURF, three box filters are used on the integral image to approximate second order gaussians and two box filters are used on the integral image to approximate first order gaussians. The dimensions of the filters vary based on the filter width parameter  $L$  chosen. Examples of the box filters used in the SURF algorithm can be seen in the figures below.

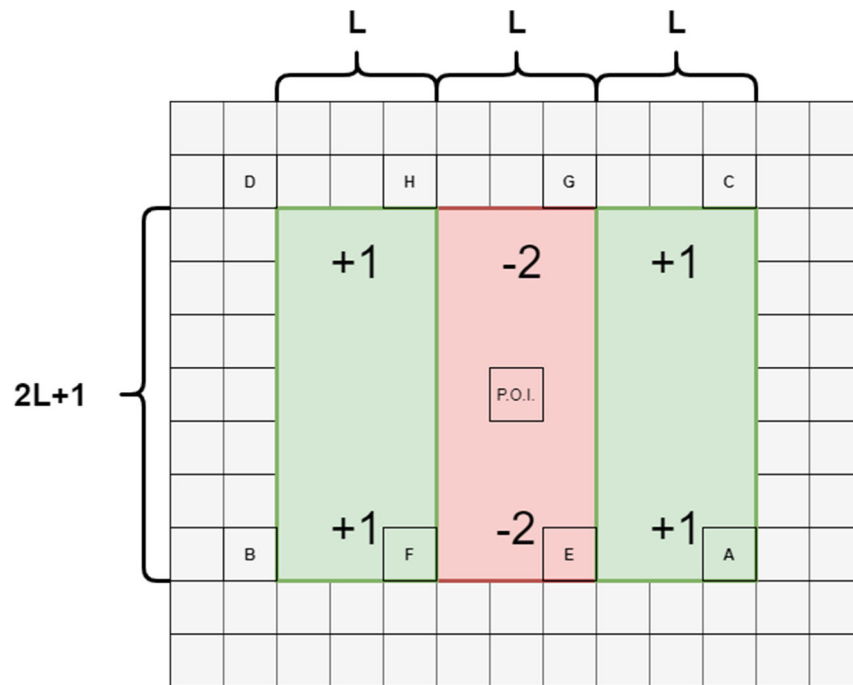


Figure 14.  $D_{xx}$  Box Filter

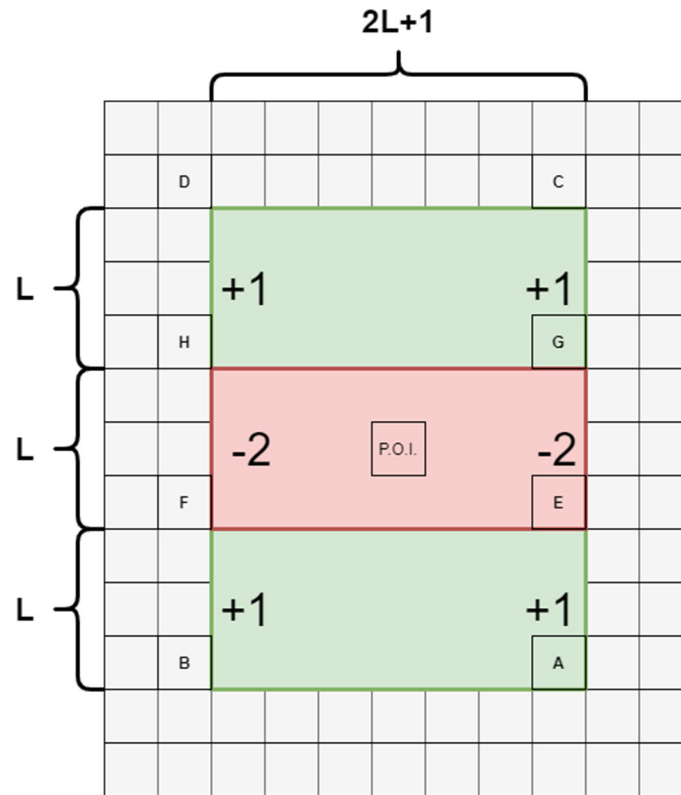


Figure 15.  $D_{yy}$  Box Filter

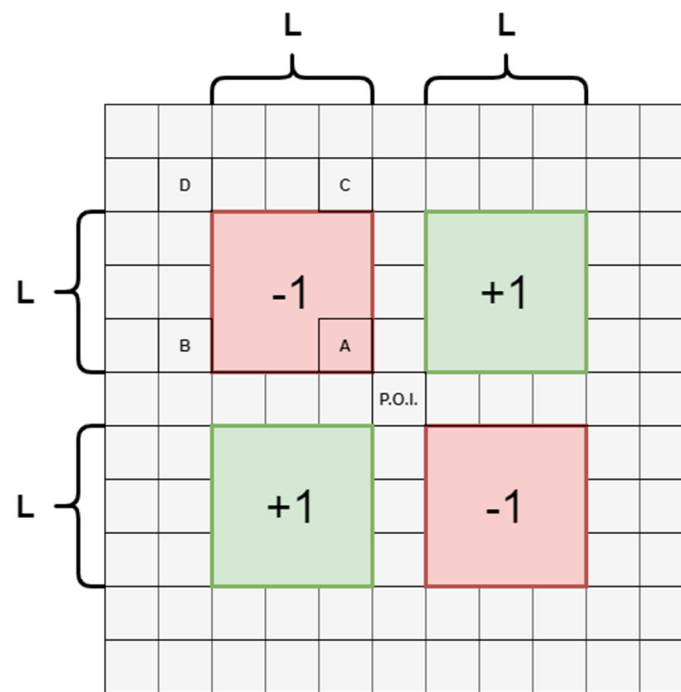


Figure 16.  $D_{xy}$  Box Filter

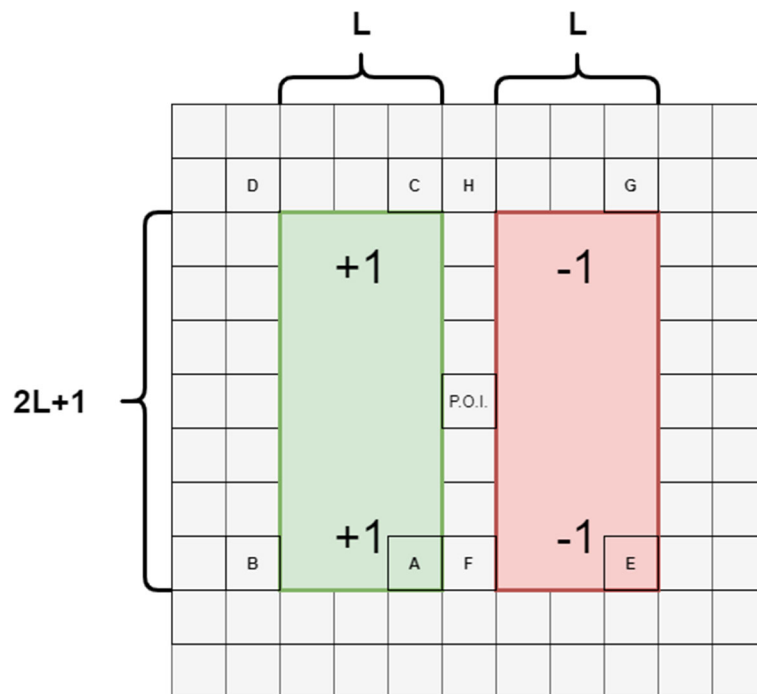


Figure 17.  $D_x$  Box Filter

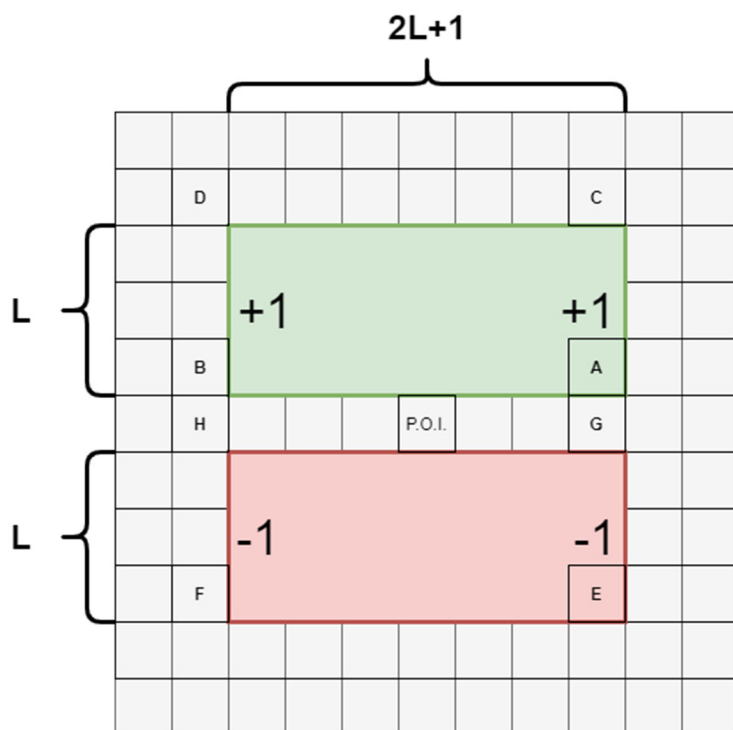


Figure 18.  $D_y$  Box Filter

Assuming the letters A through H act as variables to represent the values of the integral image at the location specified in the figures above for  $D_{xx}$ ,  $D_{yy}$ ,  $D_x$ , and  $D_y$ , the value of the filters can be found using the equations below.

$$D_{xx} = (A - B - C + D) - 3 * (E - F - G + H) \quad (\text{Eq. 2.4.3.2})$$

$$D_{yy} = (A - B - C + D) - 3 * (E - F - G + H) \quad (\text{Eq. 2.4.3.3})$$

$$D_x = (A - B - C + D) - (E - F - G + H) \quad (\text{Eq. 2.4.3.4})$$

$$D_y = (A - B - C + D) - (E - F - G + H) \quad (\text{Eq. 2.4.3.5})$$

The second order box filters are used to calculate the Determinant of Hessian for a specified filter width L and applied to the integral image using equation 2.4.3.6 below. In this equation the box filters for a specified filter width L for the integral image u are represented as  $D_{xx}^L(u)$ ,  $D_{yy}^L(u)$ , and  $D_{xy}^L(u)$ . The coefficient  $\omega$  is specific to the filter length L and can be calculated using eq. 2.4.3.7.

$$DoH^L(u) := \frac{1}{L^4} \left( D_{xx}^L(u) * D_{yy}^L(u) - \left( \omega * D_{xy}^L(u) \right)^2 \right) \quad (\text{Eq. 2.4.3.6})$$

$$\omega = \sqrt{\frac{2L-1}{2L}} \quad (\text{Eq. 2.4.3.7})$$

Prior to calculating the final SURF filtered value, the values of the box filters must be scale-normalized. Scale-normalization is performed by multiplying the box filter values by constants specific to the filter width. Equation 2.4.3.8 is the constant used for the  $D_{xx}$  and  $D_{yy}$  filtered values. Equation 2.4.3.9 is the constant value used for the  $D_{xy}$  values.

$$6L(2L - 1) \quad (\text{Eq. 2.4.3.8})$$

$$4L^2 \quad (\text{Eq. 2.4.3.9})$$

After the filtered pixel values have been calculated, non-maximal value suppression (NMS) is performed. NMS in SURF, similar to SIFT, is performed on all

filtered pixel values within an octave excluding the first and last filters in the octave. The first and last filters within an octave are excluded because they do not have filters both above and below them for filtered value comparison. The NMS process also uses a minimum value threshold to ensure that noise in low contrast regions is not selected for feature points. Once NMS has been performed, scale-space interpolation is performed to find the exact location and scale at which the feature point should be located.

Scale-space interpolation is used to find the exact location and scale at which the feature point should be located. This process also drops points that are not likely to be a good fit as a feature point. Once this process is complete, the scales and locations of feature points found can be considered continuous values. After the list of feature points has been found, the feature points must be described.

### **Feature Description**

After a feature point has been found to be sufficiently unique, it must be described. The description process works in two steps. First, the dominant orientation of the feature point is found using box filters for first order gaussian derivative approximations. Once the dominant orientation is found, a 64-value descriptor is calculated by separating the region surrounding the point into sixteen five-by-five subsections. Four values are calculated for each of the sixteen subsections resulting in the 64-value descriptor.

In order to achieve scale-invariance, a scale factor  $\sigma_k$  is used to determine the spacing between samples used for the feature point description process. The equation for the scale factor can be seen in equation 2.4.3.10 where  $o$  is the zero-based octave,  $i$  is the zero-based filter level within the octave, and  $L$  is the filter width parameter.



$$\sigma_{k(L)} = \frac{1.2}{3} (2^o * i + 1) \approx [0.4L_k] \quad (\text{Eq. 2.4.3.10})$$

The first step in the feature description portion of SURF is determining the dominant orientation of the feature point. The dominant orientation of the feature point is found by binning the magnitudes calculated using the first order gaussian approximation box filters on the integral image based on their orientation. The magnitudes of each sample point are weighted using the distance the sample point is from the feature point using a gaussian function and only values within a  $6\sigma_k$  radius from the feature point's location are considered. The range of orientation angles stored in each bin is determined by the algorithm designer. The weighted magnitudes stored in each bin are then summed to find the final value associated with each bin. This process can be better understood by examining how the process works for a single feature point.

For a feature point  $k$ , each sample point within a  $6\sigma_k$  radius from the feature point at an iterative spacing of  $\sigma_k$ , the first order box filters are applied to the integral image. The  $x$  and  $y$  components found from the previous step are then multiplied by a gaussian weighting value calculated using a standard deviation of  $2\sigma_k$  and centered on the feature point. The values found at this point in the process can be represented by equation 2.4.3.11 where  $D_x$  and  $D_y$  are the box filters approximating the first order Gaussian derivative in the  $x$  and  $y$  directions, respectively.  $u(x,y)$  is the integral image and  $G$  is the Gaussian function. The  $k$  subscript is used to represent values associated with the feature point being iterated through and is used for determining the appropriate Gaussian weighting value to apply to the sample point of interest.

$$\phi_k(x, y) := \begin{pmatrix} D_x \\ D_y \end{pmatrix} \circ u(x, y) \bullet G\left(\frac{x-x_k}{2\sigma_k}, \frac{y-y_k}{2\sigma_k}\right) \quad (\text{Eq. 2.4.3.11})$$

After calculating and weighting the x and y components for each point within the radius of interest, the magnitudes and directions for each point in the  $6\sigma_k$  radius region are found using equations 2.4.3.12 and 2.4.3.13 below.

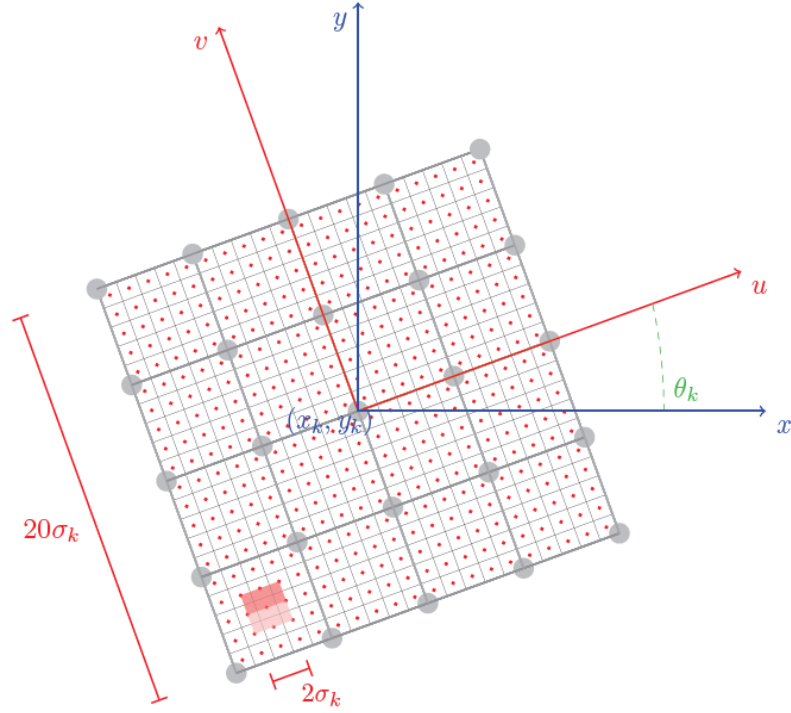
$$m(x, y) = \sqrt{x^2 + y^2} \quad (\text{Eq. 2.4.3.12})$$

$$\phi(x, y) = \tan^{-1} \left( \frac{y}{x} \right) \quad (\text{Eq. 2.4.3.13})$$

Unlike the one stage of histograms used in SIFT for determining dominant orientation, the SURF algorithm uses a second binning step. The second binning step sweeps the angles between 0 and  $2\pi$  at intervals decided on by the algorithm implementer and associates each possible dominant orientation angle with a second stage bin. The second bins store the sum of magnitude values of first stage bins that represent an angle within the range  $\pm \frac{\pi}{6}$  of the angle associated with the second stage bin being calculated. The second stage bin with the greatest magnitude is chosen as the dominant orientation.

Without the first stage of binning, calculating the summed magnitude associated with any angle  $\pm \frac{\pi}{6}$  would require examining every sample's orientation for every angle considered for dominant orientation. The two-stage binning process reduces the number of algebraic steps associated with calculating the total magnitude within a range of angles. After the dominant orientation has been determined, the 16x4 SURF descriptor vector can be computed.

Unlike SIFT, SURF does not use histograms for the feature point descriptor vector. The region surrounding a feature point is sampled as a  $20\sigma_k \times 20\sigma_k$  region which is divided into  $4\sigma_k \times 4\sigma_k$  subregions, each containing 25 samples, and oriented to the previously found dominant orientation of the feature point as shown in Figure 19 below [10].



**Figure 19. SURF Descriptor Local Region**

The gradient components are computed by applying the first order box filters to each of the points in the  $20\sigma_k \times 20\sigma_k$  region using the integral image with a sample spacing of  $\sigma_k$ . The results are then rotated using a rotation matrix to correct the results to the dominant orientation of the feature point. The rotation matrix and the differential equations to be applied to the region can be found below.

$$R_\alpha = \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix} \quad (\text{Eq. 2.4.3.14})$$

$$\begin{pmatrix} d_x(u, v) \\ d_y(u, v) \end{pmatrix} := R_{-\theta_k} \begin{pmatrix} D_x^{L_k} \\ D_y^{L_k} \end{pmatrix} u(x, y) * G_1 \left( \frac{u}{3.3}, \frac{v}{3.3} \right) \quad (\text{Eq. 2.4.3.15})$$

Once all the differential function values have been calculated for the full  $20\sigma_k \times 20\sigma_k$  region of interest, four values must be calculated for each of the 16 subregions. The four calculated values for each subregion are the sum of the first order filters as applied to each of the points in the subregion and the sum of the absolute values

of the first order box filters applied to the subregions. The four values are combined into a vector for each subregion and ultimately combined to form a 64-value descriptor for the feature point. An example subregion descriptor vector can be found in the equation below for the subregion (i,j) within the group of  $4\sigma_k \times 4\sigma_k$  subregions of interest.

$$\mu_k(i, j) = \begin{pmatrix} \sum_{(u,v) \in R_{i,j}} d_x(u, v) \\ \sum_{(u,v) \in R_{i,j}} d_y(u, v) \\ \sum_{(u,v) \in R_{i,j}} |d_x(u, v)| \\ \sum_{(u,v) \in R_{i,j}} |d_y(u, v)| \end{pmatrix} \quad (\text{Eq. 2.4.3.16})$$

The full SURF feature point descriptor is created by concatenating all subregion descriptor vectors. The full SURF feature point descriptor is then normalized using  $l^2$  normalization. This step makes the SURF feature point descriptor less susceptible to linear contrast changes.

## 2.5 Image Registration [Homography]

A homography matrix is a matrix which converts the points in two images containing an overlapping field of view to a common coordinate system. The algorithm for calculating the homography matrix requires at least four pairs of matched feature points to be calculated. The matched points are coordinates in two images that correspond to the same physical location in the scene shared by both images. One of the core requirements of homography is that the four selected points must be located on a single plane within the scene.

The issue with homography is that while it maps a common plane from one image to a similar plane in another image, it does not consider the depth of other objects in the scene. This can result in distortion of objects in the scene which are closer or further away than the plane formed by the four matched pairs being used to calculate the

homography matrix. A better stitching operation can be performed using the extrinsic camera parameters for the image sensors as well as knowledge of the depths of all objects in the scene but requires more processing to be performed and knowledge which is not always known prior to image processing.

### Homography Calculation

The homography matrix calculation starts with a matrix using the coordinates of one of the four matched pairs of feature points. In this matrix, the row and column locations of the first feature point of the matched pair are  $x_i, y_i$  while the row and column locations of the second feature point in the matched pair are  $x'_i, y'_i$ . Using these coordinates, a matrix can be constructed to represent the matched pair as seen in Eq. 2.5.1 below.

$$A_p = \begin{bmatrix} -x_i & -y_i & -1 & 0 & 0 & 0 & x_i x'_i & y_i x'_i & x'_i \\ 0 & 0 & 0 & -x_i & -y_i & -1 & x_i y'_i & y_i y'_i & y'_i \end{bmatrix} \quad (\text{Eq. 2.5.1})$$

With four sets of matched feature points, the vectors are row appended to each other to get an 8x9 matrix as seen in the figure below.

$$A_m = \begin{bmatrix} A_{p_1} \\ A_{p_2} \\ A_{p_3} \\ A_{p_4} \end{bmatrix} \quad (\text{Eq. 2.5.2})$$

When looking at this matrix as a set of equations, the set of equations is underdetermined by one equation. The homography matrix however only has eight degrees of freedom with the ninth value only representing a scaling factor. It is worth noting that while the scaling value is generally not zero, in special cases it can be zero. Using the full matrix containing all the matched pairs, the homography matrix used to

convert the two images to a common coordinate system can be found. The homography matrix is computed using the matrix above in one of two ways, the assumption method or the singular value decomposition (SVD) method.

### **2.5.1 Assumption Method**

The assumption method appends a ninth row to the matrix above with all zeros except the last column which contains a value of one. This effectively sets the variable associated with the ninth column, which represents a scaling factor, to one in the system of equations. Forcing the matrix to be critically constrained makes solving for the variables a trivial operation. Converting the new matrix to upper triangular form and using reverse substitution, all the variable values which correspond to the values of the homography matrix can be found.

The advantage of using the assumption method is that it requires less computations and is computationally simpler. These are important factors to keep in mind when working in a resource constrained environment like an FPGA or embedded environment.

The disadvantage of the assumption method is that it is incapable of handling transformations that result in a scale value of zero. A common example of this would be when only rotation is involved in the transformation. In this case, the resulting homography matrix would not accurately model the transformation necessary to find a common coordinate system between the two images.

### **2.5.2 SVD Method**

The singular value decomposition (SVD) method uses SVD to find the homography matrix. SVD must be used because the eigenvector associated with the smallest eigenvalue of the system of equations represents the homography matrix with the lowest error that can be used to fulfill the system of equations. The smallest eigenvalue in this case is the eigenvalue closest to zero, not the most negative value. SVD must be used because the matrix is non-square.

The advantage of the SVD method is that it will always return the best approximation of the homography matrix no matter what the transformation involves. This means that a homography matrix can always be calculated and works for all cases.

The disadvantage of the SVD method is that it requires more computations and is more computationally complex. This method is best to be used when compute resources are not constrained or are plentiful.

## **2.6 Mixing/Blending**

There are a variety of image mixing algorithms available with the end goal of determining the output value of each pixel. Generally speaking, there are two main methods of combining the pixel values of two or more images: blending and nearest neighbor selection.

The first method involves the blending together of pixels using weighted values based on how far each pixel is from the location of the pixel being determined. After applying a transformation matrix like a homography matrix, the coordinate locations of pixels in an image do not necessarily line up with exact integer locations ( (1,0), (1,1), (1,2), etc.) so the output pixel values at each location use neighboring pixels from both

images to determine an appropriate output pixel value. Due to the necessity of calculating the distance from the nearby transformed pixel locations to the current pixel value being calculated, this method requires more arithmetic operations to be performed (resulting in a higher run time), but generally results in a more accurate combination of the images.

The second method, called the nearest neighbor method, simply selects the pixel value of the pixel in the image with a location nearest to the output pixel being determined. This method only uses the nearest pixel if it is within a reasonable distance of the output pixel value being determined. The nearest neighbor method runs much quicker because it does not have the additional step of weighting pixel values based on their distance from the pixel value being calculated. This method generally results in a sufficient quality output frame and has a greatly reduced runtime when compared to the blending method.

Now that the theory and basic pipeline architecture have been covered, the implementation can be considered. While the two forms of hardware acceleration differ greatly, the implementations are kept as similar as possible to the algorithm outlined above. The development environments and libraries used for each implementation will also be presented. Finally, any differences that exist between the implementations and their effects on the performance of each system will be discussed.



## **CHAPTER 3**

### **IMPLEMENTATION**

#### **3.1 Development Environments**

Before diving into the implementation, it is important to consider the development environments used for each implementation. The development environments used can significantly influence the performance and quality of builds for each type of hardware acceleration. Because of these effects, a description of the development environments used for each implementation, and the hardware they were implemented on, has been included in this section.

##### **3.1.1 FPGA**

The most significant piece of the FPGA build environment is the version of integrated development environment (IDE) used. FPGAs generally require the use of vendor-specific IDEs to utilize their FPGAs and the version of IDE used, depending on the vendor, can greatly affect builds. For this project, the Digilent Genesys ZU-3EG development board was used for implementation. The development board uses a Xilinx FPGA so the Xilinx IDE, Vivado, was required to be used. For this project, Vivado 2019.1 was used due to licensing requirements for IP that was planned to be used for this thesis. The IP that was expected to be used was removed late in the development process, but the use of Vivado 2019.1 continued throughout the remainder of the thesis.

### **3.1.2 GPU**

For GPU acceleration of the algorithm, the IDE tends to be much less important than the build tools and libraries used. For general development and build automation, Visual Studio 2019 was used with CMake 3.18.4. OpenCV version 4.5.0-76 was used for image storage and image operations performed on the host computer. OpenCV was chosen because it provides a quick means of retrieving, modifying, and displaying or saving images.

The GPU accelerated implementation used an Nvidia 2070 Super. Nvidia was chosen as the vendor to be used because of the well documented CUDA library available at the time of development. The version of the CUDA compiler, known as nvcc, and its associated library are probably the most significant aspects of the GPU development environment. nvcc version 11.1.74 was used for the GPU compiler. Now that a general description of the development environment has been provided, the general design decisions made for the implementations can be considered.

## **3.2 General Design Decisions**

In order to reduce the duplication of documentation and descriptions of portions of each implementation that are shared between the projects, a general design decision section has been included. In this section, constants that are used in both implementations as well as concepts that are applied similarly or the same between the projects are introduced and covered. This reduces duplication of information in the document and provides a central location for parameter specifics for both implementations. This section will start with the image preprocessing portion of the implementations and work all the way through to the image mixing portion of the pipeline.

### 3.2.1 Image Preprocessing

Prior to running the image through the feature detection portion of the image processing pipeline, the image must first be preprocessed. First, all pixels must be converted from the three value RGB colored pixel representation associated with each pixel to a single pixel intensity value associated with each pixel. For both implementations, an 8-bit average of the three color components was used to calculate the pixel intensity value.

The pixel intensity values can be normalized throughout the entire image to reduce the effects of low contrast images on feature detection. The normalization process involves finding the maximum and minimum pixel intensity values in an image and scaling all pixel intensity values in the image to reflect the maximum possible range. The normalization process was not performed for either of the acceleration methods because it requires the system to know all pixel intensity values in a system prior to normalization. This cannot be assumed for both implementations.

Methods for approximating the maximum and minimum pixel intensity values for a frame were considered, but the image stitching pipeline worked sufficiently well without the normalization step so normalization was not implemented. The equation used for normalization is included below as equation 3.2.1.1 where  $n$  is the number of bits used to store each pixel intensity value,  $i_{min}$  is the minimum pixel intensity value in the image,  $i_{max}$  is the maximum pixel intensity value in the image,  $i$  is the pixel intensity value being normalized, and  $I$  is the normalized pixel intensity value.

$$I = (2^n - 1) * ((i - i_{min}) / (i_{max} - i_{min})) \quad (\text{Eq. 3.2.1.1})$$

An integral image can be calculated directly from the pixel intensity image, but performing the SURF filtering process on the integral image calculated directly from the pixel intensity image results in an unusable region of filtered values around the border of the image. This can be solved by either padding the integral image out or excluding the unusable region from the feature detection process. Both methods were tested and a significant difference was not observed in the performance of the algorithm using either of the two methods. The exclusion region method was chosen and implemented on both forms of acceleration due to ease of implementation.

### **3.2.2 SURF Parameters**

The portion of the image stitching pipeline that was accelerated for both implementations was the SURF feature detection algorithm. The SURF implementation used in this thesis uses three octaves with three filter widths per octave. The filter width values  $L$  and the coefficient values  $\omega$  used in the filter calculations for both implementations can be found in the table below. The filter width value  $L$  is used to determine the dimensions of the box filters used in the calculation of the SURF filtered values. The  $\omega$  value functions as a coefficient within the SURF filtered value calculation. The organization of octaves, filters, and their corresponding  $\omega$  coefficient can be found summarized in Table 1 below.

**Table 1. SURF Implementation Parameters**

Octave	L	$\omega(L)$
1	3	0.9129
1	5	0.9487
1	7	0.9646
2	5	0.9487
2	9	0.9718
2	13	0.9806
3	9	0.9718
3	17	0.9852
3	25	0.9900

Following SURF filtering, non-maximal value suppression is performed. For non-maximal value suppression, each pixel is compared with its 26 neighboring pixels in addition to a minimum hessian threshold value. A hessian threshold of 1000.0 was used for both implementations. Sample spacings of 1, 2, and 4 were used for octaves 1, 2, and 3 respectively when performing NMS. Once NMS has been performed, the remainder of both pipelines is performed in software without hardware acceleration.

### **Dominant Orientation Parameters**

When determining the dominant orientation of the feature point for feature point description, a two-step process was used. First, the  $D_x$  and  $D_y$  values for all points within a  $6\sigma_k$  radius were calculated at an interval of  $\sigma_k$ . The  $D_x$  and  $D_y$  values were used to calculate a magnitude and orientation associated with each sample point. The magnitude

values were multiplied by a gaussian filter function reducing the effects of values further away from the point of interest prior to first stage binning. The magnitude associated with each point was then grouped into 20 different equal range angular regions based on orientation. The values stored in the bins were summed which resulted in 20 angular region bins which stored the sum of all magnitudes whose dominant orientation corresponded to that angular region.

After all the magnitudes of the points within the  $6\sigma_k$  radius were categorized into their associated angular region bins, these bins were used to find the dominant orientation. To find the dominant orientation, 20 different angles were considered. For each angle being considered, all bins whose range of angles were within  $\pm \frac{\pi}{6}$  of the current orientation being examined were summed. Once all 20 different orientations had summed the angular region bins whose regions fell within the  $\pm \frac{\pi}{6}$  range of angles, the orientation with the highest summed bin value was selected as the dominant orientation.

### **Feature Description**

The remainder of the feature description process is performed using relatively similar code for both systems. Once the row-column location, scale, and dominant orientation for each feature point was found and stored, the feature point is described using the feature point description process described in section 2.4.3. Other than the number of bins used in the dominant orientation calculation, the feature point description process has no parameters specific to either implementation to be described and an in-depth explanation of its functionality has been excluded from this section as such.

### **3.2.3 Matching Parameters**

Matching is performed using a brute force method of matching one list of feature points to another. The error is calculated between each point on one list and all points on the second list. The lowest two errors are tracked for each feature point so that relative error matching can be performed. If the lowest error is lower than the second lowest error multiplied by a relative coefficient less than one, the point is treated as a match. The relative matching value of 0.15 was found to work well during testing and used for both implementations.

### **3.2.4 Homography Parameters**

Homography is calculated using four sets of matched feature locations and the assumption method. This means that an 8x9 matrix is constructed using the four sets of matched feature locations and a ninth row is appended with all zeros and a final column value of one. This matrix is solved by converting it to an upper triangular matrix and using substitution to solve for the homography matrix values. The resulting homography matrix is then normalized using the scale value in the homography matrix. The scale value in the homography matrix is the value in the last row and column of the matrix.

Once a homography matrix is found, it is applied to the location of all feature point matches from the secondary image used in the homography calculation. Error is calculated using the sum of the Euclidean distances between primary image locations and transformed secondary image locations for all matched feature points. Assuming more than four matched pairs of feature points were found, the homography calculation is performed using four randomly selected matched feature point pairs, applied to the secondary image feature point locations, and the error is calculated. The homography

matrix with the lowest error is then accepted as the best homography matrix. The repetition of these calculations and the selection of the homography matrix with the lowest associated error is how RANSAC was implemented with 2000 iterations. Once an appropriate homography matrix has been found, image mixing can be performed.

### **3.2.5 Image Mixing**

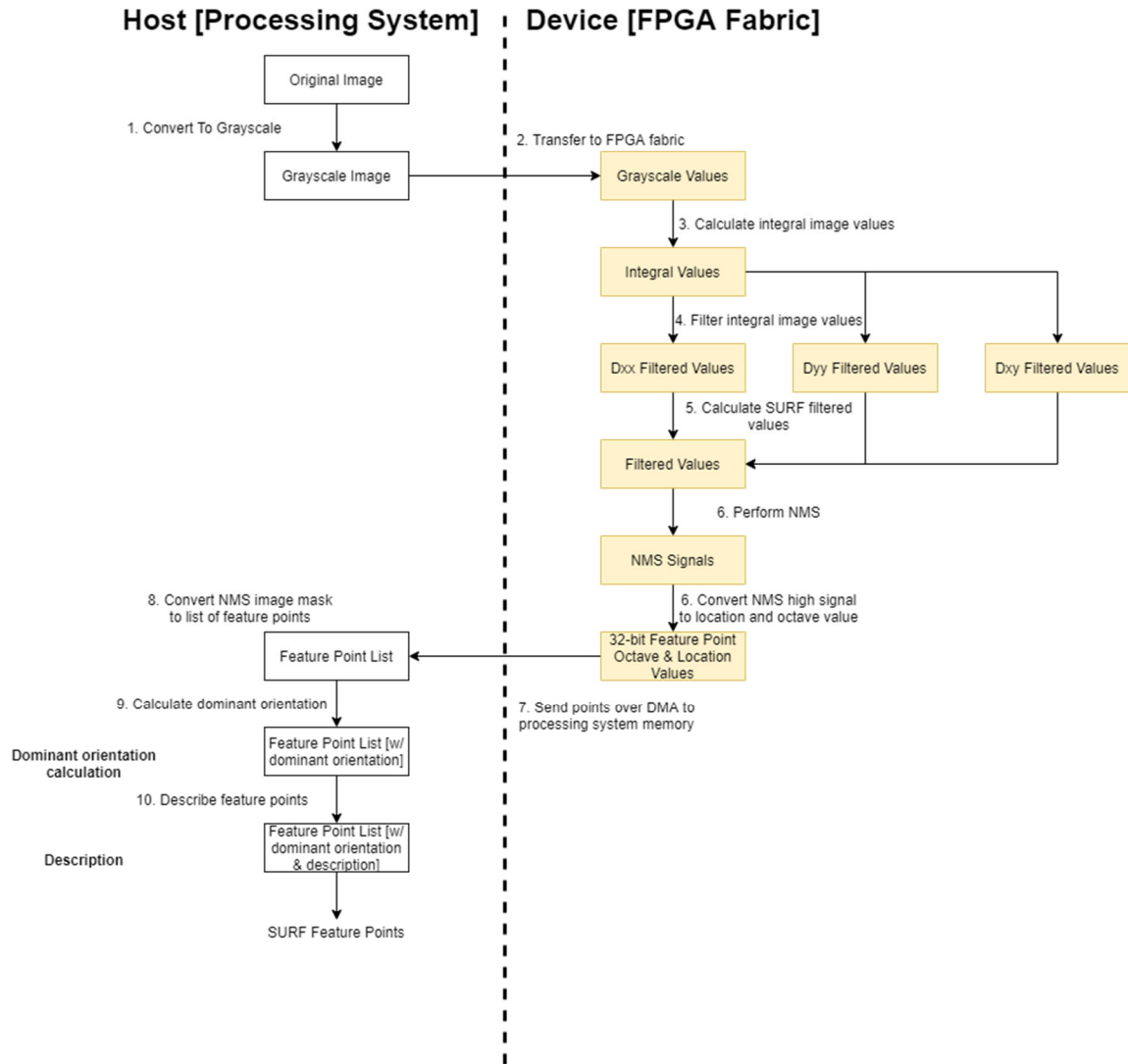
To perform image mixing, the primary image was centered within the output frame and the secondary image was transformed using the best homography matrix found. The nearest neighbor method of image mixing is used so the pixel values of the image centered in the output frame always take precedence in the output frame. The pixel values associated with the transformed image then fill in the output frame outside of the non-transformed image. Both implementations use OpenCV to apply the homography matrix to the secondary image, mix the two images together, and output the mixed image to a new window.

## **3.3 FPGA Design**

For the FPGA accelerated portion of the thesis, acceleration was used for image preprocessing and the feature detection portion of the pipeline. The choice to accelerate only this portion of the pipeline is mainly due to the large resource utilization required for parallelizing the application of all the different SURF filters being applied to a common set of data. Once the feature point locations have been found, they are passed with the calculated integral image to the systems DDR memory via DMA. The remainder of the processing performed by the image stitching pipeline is performed using the FPGA's



integrated hardcore ARM processing subsystem. A high-level architecture of the FPGA feature detection portion of the pipeline can be seen in the figure below.



**Figure 20. FPGA Feature Detection High-Level Architecture**

Once the processor receives the feature point locations and their associated filter widths along with the integral image, feature point description can occur. Feature points are appended to a linked list and the feature points found in the exclusionary region around the border of the image are discarded. The number of linked lists for the feature points retained during processing is equal to the number of image sources used. Once all

the feature points have been described for at least two input images, the feature point lists are supplied to the feature matching code.

Feature matching is performed using a brute force comparison of all points between the two linked lists of feature points using a relative matching criterion. The brute force comparison process tracks the two lowest feature point description errors using two variables. When the two lowest errors fulfill the relative matching criteria after iterating through all possible matches, the feature point associated with the lowest error and the feature point being considered are considered a match. A match is indicated using a nonzero pointer value stored in the linked list struct associated with each feature point. Once all feature point matches have been found, the homography calculation is performed using RANSAC.

After the homography matrix is found, it is supplied to the host computer via a UART serial communication link. The mixing of the two images is then performed on the host computer using OpenCV and displayed to the user in a new window.

### **3.3.1 Feature Detection Subsystem**

The accelerated feature detection portion of the FPGA image stitching pipeline can be broken down into three steps: integral image calculation, filtering, and non-maximal value suppression. The integral image calculation takes in the image intensity value associated with each pixel and outputs the associated integral image value for that pixel. The filtering portion takes as input the integral image values and outputs SURF filtered values. The non-maximal value suppression portion takes as input the SURF filtered values and outputs a signal that goes high when a local maximum value is found.

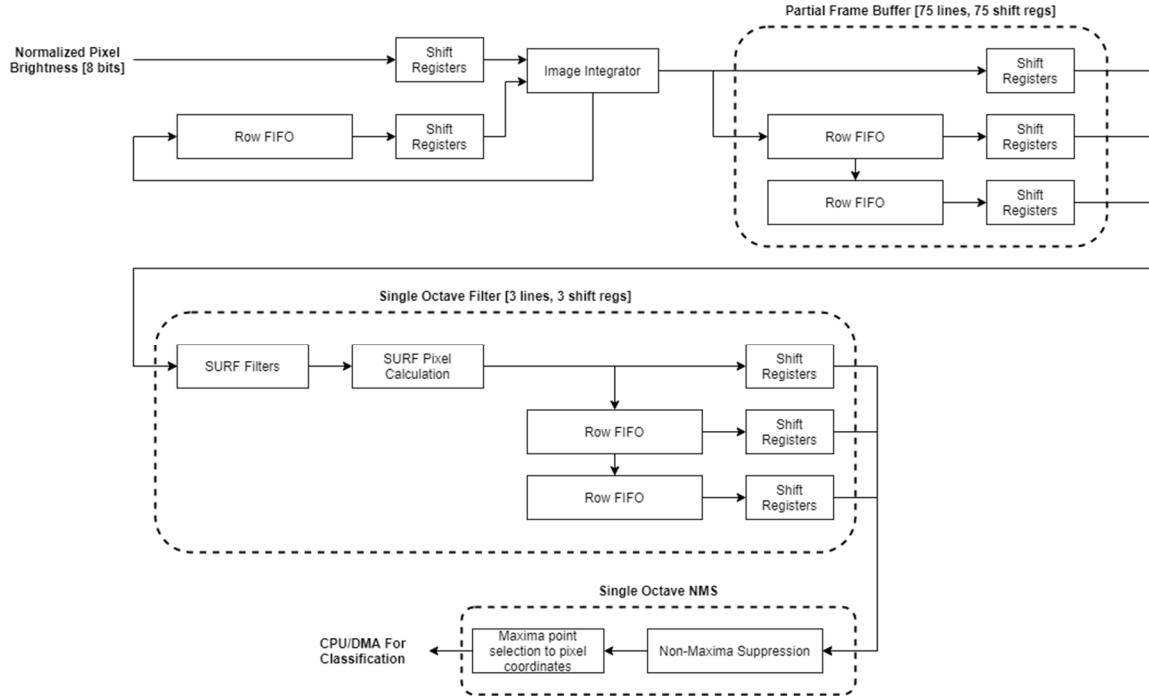
These three steps make up the hardware accelerated portion of the FPGA image stitching pipeline.

For the integral image calculation to work on the stream of pixel intensity values, the pixel intensity of the location being calculated and the integral image values of the pixels to the left, above, and diagonally above and to the left are required. A single row buffer is used to store the integral image values from the row above the current row with two shift registers to store the integral image values directly above and diagonally above and to the left. These input values are used by the image integrator block to calculate and output the integral image values to the partial frame buffer.

The SURF filtering portion of the pipeline uses the partial frame buffer to calculate SURF filtered values. The partial frame buffer provides an area of 75 by 75 integrated image pixels. The partial frame buffer values are used by the SURF second order partial derivative box filters. There are seven unique SURF filter widths used to constitute the nine filters split between the three SURF octaves. The outputs of the filters are then used to calculate the SURF filtered pixel value in the SURF pixel calculation block. There is one SURF pixel calculation block per unique filter width.

The outputs from the SURF pixel calculation blocks are put into shift registers, grouped by octave, and used for non-maximal value suppression. The NMS block compares the values supplied to it and outputs a high value signal when a maximum value has been found and low value signal otherwise. Using the output signal from the NMS block, the detected feature point's pixel location is then found using the maxima point selection to pixel coordinates block. The feature point location and filter octave are formatted into a 32-bit value and supplied to the CPU over DMA. A figure showing the

high-level architecture of the FPGA accelerated implementation with hardware-specific resources and details can be seen below.



**Figure 21. FPGA – SURF Feature Detection Subsystem**

## Integral Image

When calculating the integral image, we do not need to iterate through all pixels above and to the left of the current pixel to calculate the current pixel's integral image value. The current pixel's integral image value can be calculated by adding together the pixel intensity of the current pixel and the integral image pixel values directly above, directly to the left, and diagonally above and to the left of the current pixel. The integral image pixel value diagonally above and to the left needs to be subtracted because that area is double counted between the integral image pixels directly above and directly to the left. This method allows the integral image pixel values to be calculated using a stream of pixel intensity values. The equation for an integral image's value at any

location (x,y) can be found using equation 3.3.1.1 where I represents the integral image and i represents the pixel intensity image.

$$I(x, y) = i(x, y) + I(x - 1, y) + I(x, y - 1) - I(x - 1, y - 1) \quad (\text{Eq. 3.3.1.1})$$

Now that the math and approach have been presented, the code used to implement the integral image calculation on the FPGA can be considered. An enable signal was used to allow enabling and disabling the calculation process when non-valid pixels are supplied to the component. The code performs unsigned addition and subtraction based on the location of the pixel. The four cases for a pixel location are first row, first column, first row and first column, or any other location. The row and column for the pixel location being considered are tracked using unsigned registers row\_r and col\_r. The location registers are updated for every valid pixel supplied to the integral image calculation component. The code for the process used to implement the integral image calculation can be seen in the figure below.

```

-- Calculate the integral pixel value
calc: process(clk_in)
begin
    if rising_edge(clk_in) then
        -- if reset(rst_in) then
        if rst_in = '0' then
            integ_pixel_r <= (others => '0');

        else
            if enable_r = '1' then
                -- if first row
                if row_r = 0 then
                    -- If first col
                    if col_r = 0 then
                        integ_pixel_r <= current_pixel_r;

                    -- First row, not first col
                    else
                        integ_pixel_r <= std_logic_vector(unsigned(current_pixel_r) +
                                                                    unsigned(integ_pixel_r));

                    end if;

                -- If not first row
                else
                    -- Not first row, first col
                    if col_r = 0 then
                        integ_pixel_r <= std_logic_vector(unsigned(current_pixel_r) +
                                                                    unsigned(upper_pixel_r));

                    -- Not first row, not first col
                    else
                        integ_pixel_r <= std_logic_vector(unsigned(current_pixel_r) +
                                                                    unsigned(integ_pixel_r) +
                                                                    unsigned(upper_pixel_r) -
                                                                    unsigned(diagonal_pixel_r));

                    end if;
                end if;
            end if;
        end if;
    end if;
end process;

```

**Figure 22. Integral Image Calculation Code**

## Filtering

Filtering for the SURF feature detection algorithm is performed on the FPGA using a 75x75 pixel frame buffer region. Filtering is always performed for the pixel at row 37, column 37 within the frame buffer simultaneously for all unique filter widths. This allows a single pixel location counter to be used for all filtered values when converting from a positive non-maximal value suppression result to a row-column pixel

location. The  $D_{xx}$ ,  $D_{yy}$ , and  $D_{xy}$  filter values are calculated using signed arithmetic in HDL. The  $D_{xx}$ ,  $D_{yy}$ , and  $D_{xy}$  filter values are then converted to floating point values and supplied to a component which utilizes the DSP resources on the FPGA to perform floating point arithmetic implementing equation 2.4.3.6 to calculate the SURF filtered value. None of the pieces in the filtering algorithm are blocking, but rather are pipelined and introduce a fixed latency of 37 clock cycles from integral image pixel input to SURF filtered value output.

The  $D_{xx}$ ,  $D_{yy}$ , and  $D_{xy}$  filters are applied in a two-step process. First, the integral values for all subregions of the filter being applied are calculated using a function. Second, the subregion integral values are added or subtracted as appropriate for the filter being applied and supplied to the filter output port. Example code for the  $D_{yy}$  filter calculation processes and the integral image region calculation function can be seen in the figures below.

```
function find_integral_area(
    top_left      : std_logic_vector;
    top_right     : std_logic_vector;
    bottom_left   : std_logic_vector;
    bottom_right  : std_logic_vector
) return unsigned is

    variable retval : unsigned(top_left'range);

begin

    retval := unsigned(bottom_right) -
              unsigned(top_right) -
              unsigned(bottom_left) +
              unsigned(top_left);

    return retval;

end function;
```

**Figure 23. Integral Image Value Calculation Function**

```

calc_first_stage: process(clk_in)
begin
    if rising_edge(clk_in) then
        stage_one_top_r <= find_integral_area(
            top_left    => color_data_top_left_in,
            top_right   => color_data_top_right_in,
            bottom_left => color_data_mid_upper_left_in,
            bottom_right => color_data_mid_upper_right_in
        );

        stage_one_mid_r <= find_integral_area(
            top_left    => color_data_mid_upper_left_in,
            top_right   => color_data_mid_upper_right_in,
            bottom_left => color_data_mid_lower_left_in,
            bottom_right => color_data_mid_lower_right_in
        );

        stage_one_bot_r <= find_integral_area(
            top_left    => color_data_mid_lower_left_in,
            top_right   => color_data_mid_lower_right_in,
            bottom_left => color_data_bottom_left_in,
            bottom_right => color_data_bottom_right_in
        );
    end if;
end process;

calc_second_stage: process(clk_in)
begin
    if rising_edge(clk_in) then
        color_data_out <= std_logic_vector(signed(stage_one_top_r) +
                                            signed(stage_one_bot_r) -
                                            shift_left(signed(stage_one_mid_r), 1));
    end if;
end process;

```

**Figure 24.  $D_{yy}$  Filter Value Calculation Processes**

Once the  $D_{xx}$ ,  $D_{yy}$ , and  $D_{xy}$  filter values have been calculated, those values are converted to 32-bit floating point values using a function. The function used for converting 32-bit signed values to 32-bit floating-point values can be seen in the figure below.



```

-- Signed version of float conversion
function signed_to_float_slv(val: std_logic_vector) return std_logic_vector is
    variable val_v : std_logic_vector(31 downto 0);

    variable highest_set_bit_v : integer := 0;
    variable left_shift_v      : integer := 0;

    variable retval_v      : std_logic_vector(31 downto 0);

    variable mantissa_v : unsigned(22 downto 0);
    variable exponent_v : unsigned(7 downto 0) := to_unsigned(128, 8);
    variable sign_v      : std_logic := '0';

    variable zero_v      : std_logic_vector(val'left downto val'right) := (others => '0');

begin
    val_v := val;

    -- Corner case: zero
    if val = zero_v then
        retval_v := (others => '0');
    else
        -- If negative number
        if val(val'high) = '1' then
            -- Perform two's complement
            val_v := std_logic_vector(unsigned(not val) + to_unsigned(1, val'length));

            -- Set sign bit high
            sign_v := '1';
        end if;

        for x in val'length-1 downto 0 loop
            left_shift_v := left_shift_v + 1;

            if val_v(x) = '1' then
                exit;
            end if;
        end loop;

        -- Shift to be left aligned and store as mantissa
        mantissa_v := shift_left(unsigned(val_v), left_shift_v)(31 downto 9);

        -- Calculate exponent
        exponent_v := exponent_v + to_unsigned(val'left - left_shift_v, 8);

        -- Use number of shifts to determine exponent
        retval_v := sign_v & std_logic_vector(exponent_v) & std_logic_vector(mantissa_v);
    end if;

    return retval_v;
end function;

```

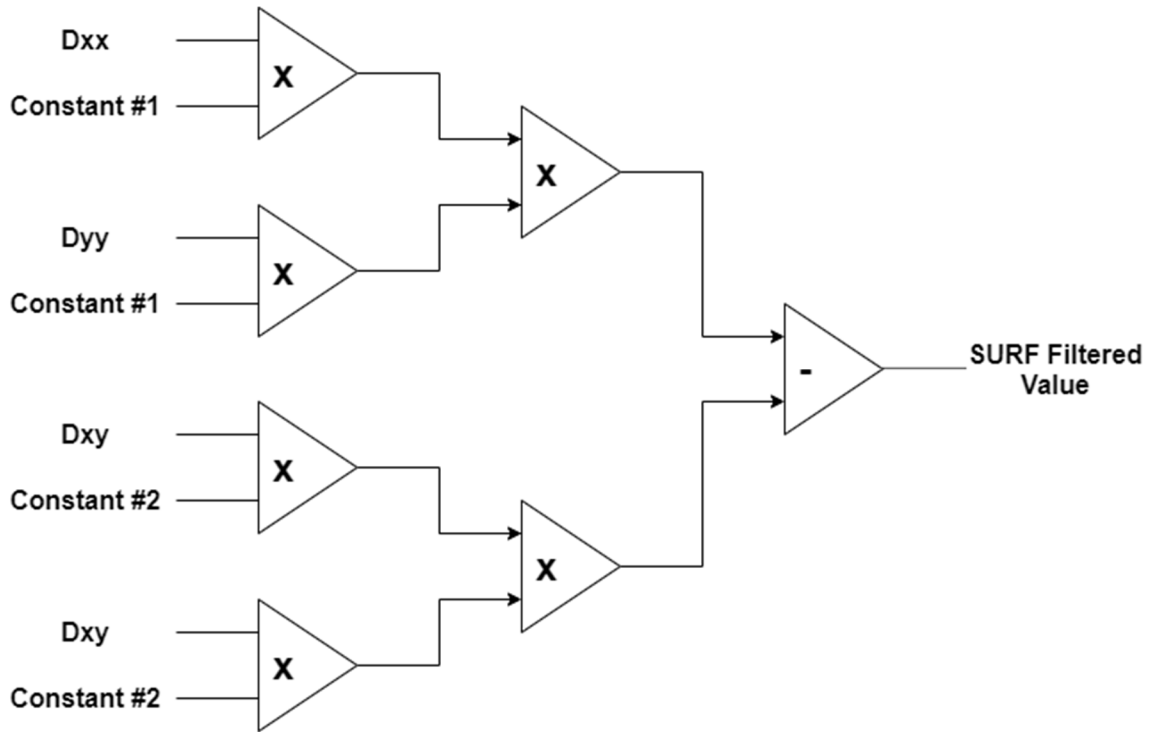
**Figure 25. Signed to Floating Point Standard Logic Vector Conversion Function**

Once the  $D_{xx}$ ,  $D_{yy}$ , and  $D_{xy}$  filter values have been converted to 32-bit floating point values, the floating-point values can be supplied to the SURF filtered value calculation block. The SURF filtered value calculation block utilizes the FPGAs DSP resources to perform the floating-point algebra required to calculate the SURF filtered

value. The constants used are dependent on the filter width and are summarized in the table below. Due to how lengthy the code is for declaring and connecting these blocks together, a block diagram representation of how these DSP blocks are organized can be seen in the figure below.

**Table 2. FPGA SURF Filter Constants**

<b>Filter Width</b>	<b>Constant #1 [Hex]</b>	<b>Constant #1 [Decimal]</b>	<b>Constant #2 [Hex]</b>	<b>Constant #2 [Decimal]</b>
3	0x3e2aaaab	1/6	0x3dc511a3	$\frac{0.9129}{\sqrt{90}}$
5	0x3dcccccd	1/10	0x3d6c7b90	$\frac{0.9487}{\sqrt{270}}$
7	0x3d924925	1/14	0x3d28ea8c	$\frac{0.9636}{\sqrt{546}}$
9	0x3d638e39	1/18	0x3d036117	$\frac{0.9718}{\sqrt{918}}$
13	0x3d1d89d9	1/26	0x3cb5e8e5	$\frac{0.9806}{\sqrt{1950}}$
17	0x3cf0f0f1	1/34	0x3c8b1b82	$\frac{0.9852}{\sqrt{3366}}$
25	0x3ca3d70a	1/50	0x3c3d2fa6	$\frac{0.9899}{\sqrt{7350}}$



**Figure 26. SURF Filtered Value Calculation Blocks**

### **Non-maximal Value Suppression (NMS)**

The NMS portion of the pipeline is passed the SURF filtered values found in the SURF filtering step and uses a 9x9 buffer region of SURF filtered values. The 9x9 buffered region uses the value found at row 5, column 5 as the center of the NMS calculation. The 9x9 buffered region is required for the different sample spacings used for different filter octaves.

There are three NMS components used in the design which represent the three filter octaves used in the algorithm. When a local maximum is found in a filter octave that is above the minimum hessian threshold, the output signal takes on a high value. The three maximum value output lines from the three NMS components are combined into a three-bit NMS active bus. Row and column counters are used to track the location of the current NMS output's corresponding pixel location. When the NMS active bus does not

equal zero, the row-column location of the pixel and the NMS bus are combined into a 32-bit value and sent to memory via DMA.

For the 32-bit floating-point value comparison, a function was written to reduce the latency of the process. While the difference of the two numbers that are being compared could be computed using DSP resources on the FPGA, the 27 comparisons needing to be performed for each of the 3 octaves would have required more DSP resources than were available. The function used for floating point comparisons and the first octave's NMS code utilizing it can be seen in the figures below.

```
with sample_spacing select
    max_value_out <= spacing_one when 1,
    spacing_two when 2,
    spacing_tre when 4,
    '0' when others;

spacing_one <= greatest_value(layer_1_row_1_vals(5), layer_0_row_0_vals(4))
    and greatest_value(layer_1_row_1_vals(5), layer_0_row_0_vals(5))
    and greatest_value(layer_1_row_1_vals(5), layer_0_row_0_vals(6))
    and greatest_value(layer_1_row_1_vals(5), layer_0_row_1_vals(4))
    and greatest_value(layer_1_row_1_vals(5), layer_0_row_1_vals(5))
    and greatest_value(layer_1_row_1_vals(5), layer_0_row_1_vals(6))
    and greatest_value(layer_1_row_1_vals(5), layer_0_row_2_vals(4))
    and greatest_value(layer_1_row_1_vals(5), layer_0_row_2_vals(5))
    and greatest_value(layer_1_row_1_vals(5), layer_0_row_2_vals(6))
    and greatest_value(layer_1_row_1_vals(5), layer_1_row_0_vals(4))
    and greatest_value(layer_1_row_1_vals(5), layer_1_row_0_vals(5))
    and greatest_value(layer_1_row_1_vals(5), layer_1_row_0_vals(6))
    and greatest_value(layer_1_row_1_vals(5), layer_1_row_1_vals(4))
    and greatest_value(layer_1_row_1_vals(5), layer_1_row_1_vals(6))
    and greatest_value(layer_1_row_1_vals(5), layer_1_row_2_vals(4))
    and greatest_value(layer_1_row_1_vals(5), layer_1_row_2_vals(5))
    and greatest_value(layer_1_row_1_vals(5), layer_1_row_2_vals(6))
    and greatest_value(layer_1_row_1_vals(5), layer_2_row_0_vals(4))
    and greatest_value(layer_1_row_1_vals(5), layer_2_row_0_vals(5))
    and greatest_value(layer_1_row_1_vals(5), layer_2_row_0_vals(6))
    and greatest_value(layer_1_row_1_vals(5), layer_2_row_1_vals(4))
    and greatest_value(layer_1_row_1_vals(5), layer_2_row_1_vals(5))
    and greatest_value(layer_1_row_1_vals(5), layer_2_row_1_vals(6))
    and greatest_value(layer_1_row_1_vals(5), layer_2_row_2_vals(4))
    and greatest_value(layer_1_row_1_vals(5), layer_2_row_2_vals(5))
    and greatest_value(layer_1_row_1_vals(5), layer_2_row_2_vals(6))
    and greatest_value(layer_1_row_1_vals(5), hessian_threshold_in);
```

**Figure 27. NMS Calculation and Output Assignment**

```

function greatest_value(val_0 : std_logic_vector; val_1 : std_logic_vector) return std_logic is
    variable sign_0 : std_logic;
    variable sign_1 : std_logic;
    variable exp_0 : unsigned(7 downto 0);
    variable exp_1 : unsigned(7 downto 0);
    variable mantissa_0 : unsigned(22 downto 0);
    variable mantissa_1 : unsigned(22 downto 0);
    variable retval : std_logic := '0';
    variable sign_check : boolean := false;
    variable exp_check : boolean := false;
    variable mant_check : boolean := false;
begin
    sign_0 := val_0(31);
    sign_1 := val_1(31);
    exp_0 := unsigned(val_0(30 downto 23));
    exp_1 := unsigned(val_1(30 downto 23));
    mantissa_0 := unsigned(val_0(22 downto 0));
    mantissa_1 := unsigned(val_1(22 downto 0));

    if sign_0 = '0' then
        if sign_1 = '1' then
            retval := '1';

        else
            -- If higher exponent
            if exp_0 > exp_1 then
                retval := '1';

            -- If lower exponent
            elsif exp_1 > exp_0 then
                retval := '0';

            -- Equal exponent, need to check mantissa
            else
                if mantissa_0 > mantissa_1 then
                    retval := '1';

                else
                    retval := '0';

                end if;
            end if;
        end if;

    -- Negative sign
    else
        if sign_1 = '0' then
            retval := '0';

        -- Same sign, check exponent
        else
            -- If smaller exponent, return true
            if exp_0 < exp_1 then
                retval := '1';

            elsif exp_0 > exp_1 then
                retval := '0';

            -- Equal exponents, check mantissa
            else
                if mantissa_0 < mantissa_1 then
                    retval := '1';

                else
                    retval := '0';

                end if;
            end if;
        end if;
    end if;

    return retval;
end function;

```

**Figure 28. Floating Point Comparison Function**

### 3.3.2 Feature Description

The features are first converted from the format which DMA used to store the points in memory to the feature point structure used in software. The format used to store the points in memory by DMA can be seen in the figure below. Bits 31 through 29 represent a bitmask whose values are high when a positive NMS value has been detected in their octave, otherwise they are zero. The remainder of the 32-bits are used to store the feature point's row and column pixel location as unsigned integers.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Oct 2	Oct 1	Oct 0	Row													Column															

**Figure 29. FPGA 32-bit Feature Point Row, Column, and Octave Encoding**

The DMA peripheral system used by the FPGA's processing system tracks the number of bytes that have been transferred in the last DMA transaction. Using the number of bytes transferred in the last DMA transaction allows the software to determine how many feature points were detected once the feature detection process has completed. Once the feature detection process is complete and the number of feature points detected is known, the feature description process can begin.

A single feature point description requires, at a bare minimum, 64 floating-point values which equates to roughly 1 Mb of data per 4000 points. In addition to the 64 floating-point values, metadata used for the matching and homography calculation processes are stored with each feature point, increasing the amount of memory required for each described feature point. With the images tested generally finding over 4000 feature points, the memory requirements can add up quickly. With memory resources available to the FPGA being relatively small, a linked list is used to store described feature points so that the system is not required to attempt to allocate large portions of

contiguous memory when using malloc. Once space has been allocated for all feature points and they have been initialized with the appropriate data, the process of performing scale-space interpolation is skipped and the dominant orientation is found.

The FPGA version does not use scale-space interpolation, but rather performs filtering on every pixel that is encountered allowing detection of feature points that would normally be located between sample locations for any octaves greater than one. While this does not allow the exact peak of a feature point to be found when it exists between filter levels, this technique was found to work sufficiently well for most cases. Additionally, scale-space interpolation was not used in the FPGA version because it requires the use of SURF filtered values which are not saved to memory by the feature detection hardware.

Dominant orientation and feature point description use largely identical code to that which is used by the GPU solution. Any parameters used in the dominant orientation and feature point description portions of the algorithm can be found in section 3.2.2 above.

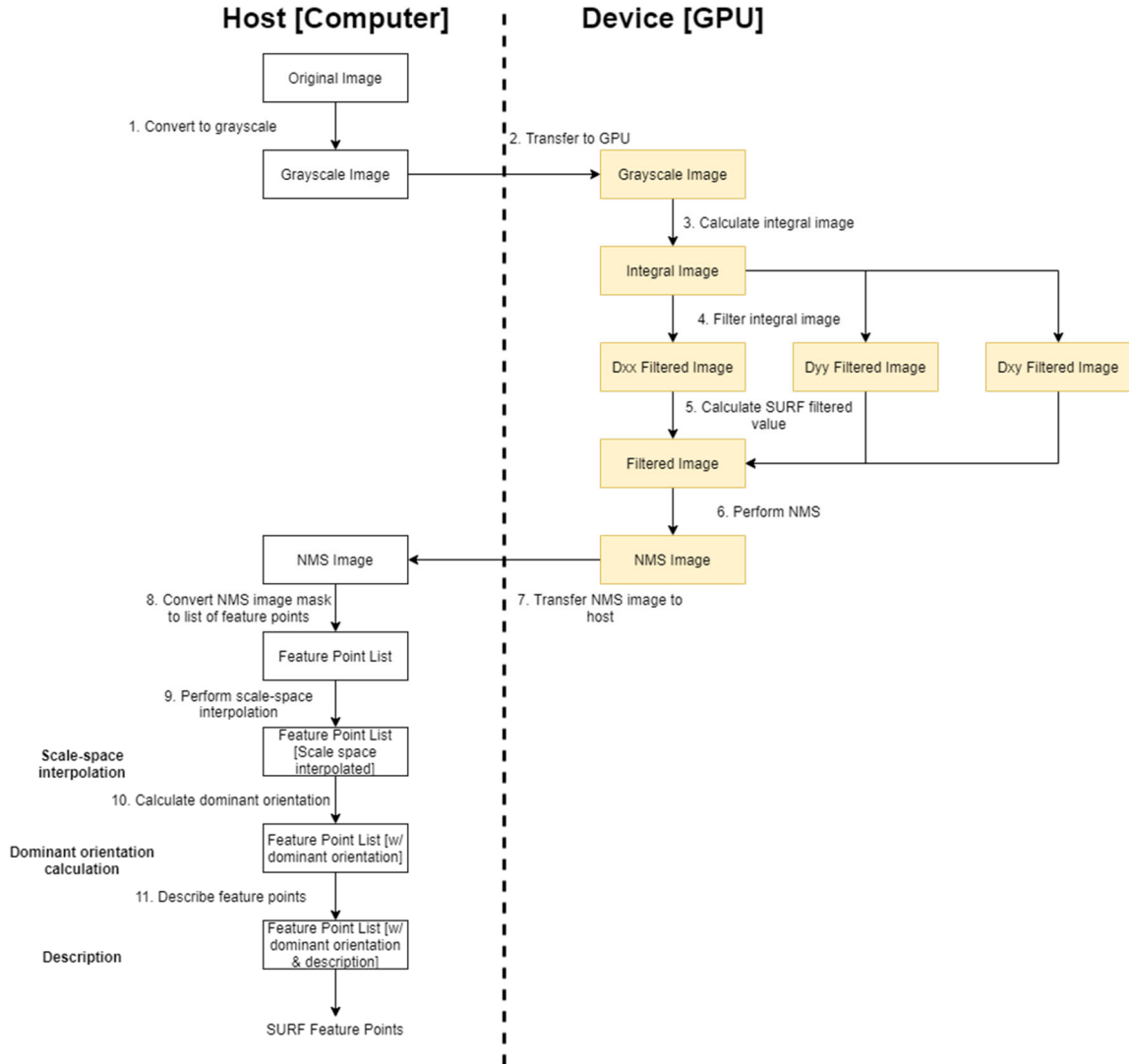
### **3.4 GPU Design**

For the GPU design, the CUDA library was used for accelerating the algorithm on an Nvidia graphics card and OpenCV was used to read the input images and output the stitched image. The program was written in C++ with most of the code being C compliant. The SURF algorithm, feature point matching algorithm, and homography calculation algorithm were written from scratch originally as a proof-of-concept implementation to be executed on a CPU. Writing these algorithms from scratch also allowed for easier porting to the GPU-based hardware acceleration. Additionally, some of

the code written for the proof-of-concept CPU version of the program was used in the FPGA implementation.

The portion of the pipeline that was accelerated on the graphics card was the feature detection portion of the pipeline. This includes the integral image calculation, SURF image filtering, and non-maximal value suppression. The NMS image the GPU calculates is a mask image which gets converted to a list of feature points on the host computer so that it can be used by the remainder of the image stitching pipeline. A graphical representation of the high-level architecture of the GPU implementation can be seen in the figure below.





**Figure 30. GPU SURF Algorithm Dataflow Diagram**

### 3.4.1 SURF Feature Detection

For image preprocessing, grayscale conversion is performed on the host computer to keep implementations consist between the two systems. The grayscale image is then transferred to the GPU from the host computer. Once the grayscale image is stored in shared memory on the GPU, the integral image calculation can be performed.

## **Integral Image**

The integral image calculation, when implemented similarly to the FPGA, is particularly slow on the GPU due to the dependence on calculated integral image values surrounding the pixel of interest. Additionally, profiling the GPU version of the algorithm is particularly difficult as the number of calls to the integral image kernel is relatively high. The number of calls to the integral image calculation kernel for the GPU version of the algorithm is equal to the number of rows or number of columns in the image, whichever is greater. Additionally, the logic for determining which integral image values can be calculated, based on which values were previously calculated, is rather difficult. For these reasons, the straightforward approach of summing all appropriate pixel intensity values for the current integral image value was chosen.

The integral image portion of the pipeline functions by allocating a thread for every pixel in the image. Each thread then sums all of the pixel intensity values above and to the left of the current pixel as well as the pixel intensity value of the current pixel. The integral image kernel uses the pixel intensity image stored in shared memory for the calculation. The kernel is supplied a pointer to the pixel intensity image shared memory, a pointer to allocated shared memory for the output values, the number of rows in the image, and the number of columns in the image. The code for the integral image calculation kernel implemented for the GPU can be seen in the figure below.

```

/**
 * Creates an integral image from orig_img and returns it.
 *
 * @param orig_img A normalized single channel image matrix
 * @return An integral image matrix
 */
__global__ void gpu_integral_img(int *out, float *orig_img, int num_rows, int num_cols) {
    int iter_offset = blockIdx.x * blockDim.x + threadIdx.x;

    int row_off = iter_offset / num_cols;
    int col_off = iter_offset % num_cols;

    // If out of image bounds
    if(row_off < 0 || row_off >= num_rows ||
        col_off < 0 || col_off >= num_cols) {
        return;
    }

    int intensity_offset = 0;
    int sum = 0;

    for(int col_i=0; col_i <= col_off; col_i++) {
        for(int row_i=0; row_i <= row_off; row_i++) {
            // Calculate offset
            intensity_offset = col_i + row_i * num_cols;

            // Add to sum
            sum += (unsigned int)orig_img[intensity_offset];
        }
    }

    out[iter_offset] = sum;
}

```

**Figure 31. GPU Integral Image Calculation Kernel**

## Filtering

Considering hardware acceleration of the SURF algorithm, an obvious candidate for acceleration is the calculation of the SURF filtered values for all pixels. The filtered values ( $D_{xx}$ ,  $D_{yy}$ , and  $D_{xy}$ ) are calculated for all pixels in the image and only rely on the integral image for their calculation. This means that once the integral image is calculated and present in the device's memory, all three filter values can be calculated in parallel for all pixels simultaneously assuming the hardware resources are available. Once the three filter values have been calculated for all pixels in the image, the final filtered value can be calculated for all pixels in the image in parallel as well.

For each of the second order filters being applied, the kernels are supplied a pointer to shared memory for the output values, a pointer to shared memory for the integral image, and the filter width of the filter being applied. The row and column location are calculated using the thread and threadblock ids supplied to every kernel created on the GPU. The  $D_{xx}$  box filter kernel has been provided as an example in the figure below.

```

/*****
 * Calculate the Dxx Box filter at filter width L
 *****/
global__ void dxx_box_filter(box_filt_t *out, intensity_image_t *img, filt_width_t L) {
    intensity_image_t a, b, c, d, e, f, g, h;
    box_filt_t filt_val;

    int row = blockIdx.x;
    int col = threadIdx.x;

    int padded_offset = SURF_MAT_PADDED_BASE + row*SURF_MAT_PADDED_STRIDE + col;
    int output_offset = row * NUM_COLS + col;

    a = img[padded_offset + DXX_A_PADDED_OFFSET(L)];
    b = img[padded_offset + DXX_B_PADDED_OFFSET(L)];
    c = img[padded_offset + DXX_C_PADDED_OFFSET(L)];
    d = img[padded_offset + DXX_D_PADDED_OFFSET(L)];

    e = img[padded_offset + DXX_E_PADDED_OFFSET(L)];
    f = img[padded_offset + DXX_F_PADDED_OFFSET(L)];
    g = img[padded_offset + DXX_G_PADDED_OFFSET(L)];
    h = img[padded_offset + DXX_H_PADDED_OFFSET(L)];

    filt_val = (a - b - c + d) - 3*(e - f - g + h);

    out[output_offset] = filt_val;
}

```

**Figure 32. GPU Accelerated  $D_{xx}$  Filter Kernel**

Once the  $D_{xx}$ ,  $D_{yy}$ , and  $D_{xy}$  filtered values have been calculated, the final SURF filtered value can be calculated. The SURF filtered value kernel is supplied a pointer to shared memory on the device for the output value; pointers to the shared memory for the  $D_{xx}$ ,  $D_{xy}$ , and  $D_{yy}$  filtered values; the filter width of the filter being applied; and the  $\omega$  coefficient used for calculating the SURF filter value. The kernel finds the row and column location of the SURF filtered pixel being calculated using the kernel's thread and

threadblock ids. The code for the kernel used to perform the SURF filtered value calculation on the GPU has been included in the figure below.

```
__global__ void filter(float *out, box_filt_t
                    *dxx, box_filt_t *dyy,
                    box_filt_t *dxy, filt_width_t L,
                    float coeff) {
    int row = blockIdx.x;
    int col = threadIdx.x;

    int pixel_offset = row*NUM_COLS + col;

    float filt_val;

    // Calculate normalization coefficients
    float nxx = sqrt(6.0*L*(2.0*L-1.0));
    float nxy = sqrt(4.0*L*L);

    // Apply scale-normalization
    float Dxx = dxx[pixel_offset] / nxx;
    float Dyy = dyy[pixel_offset] / nxx;
    float Dxy = dxy[pixel_offset] / nxy;

    // Calculate SURF filter value
    filt_val = (Dxx*Dyy-coeff*(Dxy*Dxy));

    // Assign to output matrix
    out[pixel_offset] = filt_val;
}
```

**Figure 33. GPU SURF Filter Value Kernel**

### Non-maximal Value Suppression (NMS)

The non-maximal value suppression process references the appropriate SURF filtered value images calculated in the previous step and stored in shared GPU memory. Using those SURF filtered value images, NMS is performed, and an output image is created where non-zero values are used to represent local maximums.

The non-maximal value suppression kernel is supplied a pointer to shared memory for the NMS mask output values; pointers to shared memory for the SURF filtered values for the current filter width, the filter width immediately above the current octave, and the filter width immediately below the current octave; the sample spacing of

the current octave; and the minimum hessian threshold. The output pixel offset is calculated using the kernels thread and threadblock ids. The code used for the NMS mask calculation kernel is included in the figure below.

```

/*****
* Perform non-maximal value suppression
*****/
__global__ void nms(nms_t *out, float *d_filt_one, float *d_filt_two,
                  float *d_filt_three, int sample_spacing, float hessian_threshold) {

    int row = blockIdx.x;
    int col = threadIdx.x;

    int pixel_offset = row*NUM_COLS + col;

    nms_t retval = 0;

    float value = d_filt_two[pixel_offset];

    if(row == 0 && col == 0 && sample_spacing == 1) {
        printf("Found point.\n");
    }

    if(row < sample_spacing || row > NUM_ROWS-sample_spacing-1 ||
       col < sample_spacing || col > NUM_COLS-sample_spacing-1) {
        return;
    }

    if(
        value > hessian_threshold &&
        value > d_filt_one[pixel_offset - (sample_spacing*NUM_COLS) - sample_spacing] &&
        value > d_filt_one[pixel_offset - (sample_spacing*NUM_COLS)] &&
        value > d_filt_one[pixel_offset - (sample_spacing*NUM_COLS) + sample_spacing] &&
        value > d_filt_one[pixel_offset - sample_spacing] &&
        value > d_filt_one[pixel_offset] &&
        value > d_filt_one[pixel_offset + sample_spacing] &&
        value > d_filt_one[pixel_offset + (sample_spacing*NUM_COLS) - sample_spacing] &&
        value > d_filt_one[pixel_offset + (sample_spacing*NUM_COLS)] &&
        value > d_filt_one[pixel_offset + (sample_spacing*NUM_COLS) + sample_spacing] &&

        value > d_filt_two[pixel_offset - (sample_spacing*NUM_COLS) - sample_spacing] &&
        value > d_filt_two[pixel_offset - (sample_spacing*NUM_COLS)] &&
        value > d_filt_two[pixel_offset - (sample_spacing*NUM_COLS) + sample_spacing] &&
        value > d_filt_two[pixel_offset - sample_spacing] &&
        value > d_filt_two[pixel_offset + sample_spacing] &&
        value > d_filt_two[pixel_offset + (sample_spacing*NUM_COLS) - sample_spacing] &&
        value > d_filt_two[pixel_offset + (sample_spacing*NUM_COLS)] &&
        value > d_filt_two[pixel_offset + (sample_spacing*NUM_COLS) + sample_spacing] &&

        value > d_filt_three[pixel_offset-(sample_spacing*NUM_COLS) - sample_spacing] &&
        value > d_filt_three[pixel_offset-(sample_spacing*NUM_COLS)] &&
        value > d_filt_three[pixel_offset-(sample_spacing*NUM_COLS) + sample_spacing] &&
        value > d_filt_three[pixel_offset- sample_spacing] &&
        value > d_filt_three[pixel_offset] &&
        value > d_filt_three[pixel_offset+ sample_spacing] &&
        value > d_filt_three[pixel_offset+(sample_spacing*NUM_COLS) - sample_spacing] &&
        value > d_filt_three[pixel_offset+(sample_spacing*NUM_COLS)] &&
        value > d_filt_three[pixel_offset+(sample_spacing*NUM_COLS) + sample_spacing]
    ) {
        retval = 1;
    }

    out[pixel_offset] = retval;
}

```

Figure 34. GPU NMS Comparison Kernel

After non-maximal value suppression is performed and an NMS mask image is found, the NMS mask image is transferred back to the host machine. Once the NMS mask image is transferred back to the host machine, the NMS mask image is used to generate a linked list of feature points. Scale-space interpolation is performed prior to appending each feature point to the linked list. After the linked list of feature points has been generated, the linked list is converted to a vector to make feature point accesses and modifications faster for the remainder of the pipeline. Additionally, it would allow the feature point list to be transferred to the GPU as a block of memory if any of the remaining steps utilized hardware acceleration.

### **3.4.2 Feature Description**

The description portion of the algorithm is performed on the host machine's CPU. The code used for feature point description is nearly identical on both the FPGA and GPU implementations. The GPU version includes scale-space interpolation because it performs NMS at sample intervals specific to the filters being used as specified in the original SURF paper [1]. Other than the addition of scale-space filtering, the only other difference between the two implementations is the modification of function calls to use the vectorized list of feature points. Once the feature points have been described, feature point matching can be performed. Feature point matching and homography calculation are performed using identical code on both implementations.

Now that a more in-depth view of both implementations has been presented, we can look at how the implementations perform when tested. In the next chapter, we will be presenting the results of the tests and comparing the performance of both



implementations. Once the performance of both systems has been considered, potential use cases and design improvements will be presented.

## **CHAPTER 4**

### **RESULTS & DISCUSSION**

#### **4.1 Tests**

After both systems had been implemented, their performance was compared. The systems were compared based on latency, power consumption, and resource utilization. Additionally, the effects of increasing the input image resolution on the latency and resource utilization of both systems was considered. Finally, a summary comparison of both systems was included.

##### **4.1.1 Latency**

The latency for both systems was measured for the hardware accelerated portion of the pipeline and is measured in clock cycles. For this test, the number of clock cycles are measured between the first valid intensity image pixel entering the system and the first potential feature point location being output from the system. On the FPGA these are the clock cycles between the first pixel intensity value entering the FPGA fabric over DMA and the first possible pixel location being sent back to the FPGA processing system over DMA. On the GPU, these are the clock cycles between the beginning of the integral image calculation and the beginning of the DMA transfer of the NMS image mask back to the host from the device. Additionally, how latency is affected when increasing the resolution of input images for both systems is considered.

One issue encountered when determining a fair comparison of latency values was the issue of different image communication links. The FPGA uses DMA to supply the image to the FPGA fabric while the GPU uses a PCI-e connection. While the number of

clock cycles can be used to eliminate a dependence on the clock frequencies used for each implementation, they are still not equivalent methods of transmitting the image to the system.

With that being said, the FPGA begins processing pixels as soon as they are received. This leads to the FPGA outputting its first feature location before the full image has been sent through the system. The GPU system, on the other hand, must receive the full image prior to processing the image to find feature points. Due to this requirement, the size of the image will affect the latency of the GPU calculations. To equalize this out, the latency for the GPU implementation will be measured between the intensity image being present in memory and the feature point NMS mask calculation being completed.

#### **4.1.2 Power**

The maximum and minimum power consumption during runtime of both implementations is estimated using device specific tools. While these values are not exact for a final implementation, they provide a good baseline for what the power consumption of the part would be in system designed around this algorithm.

#### **4.1.3 Resources**

The memory resource utilization and the effects of increasing the image resolution for both systems are compared. Due to the differences in types of resources available on the FPGA and GPU, memory was chosen as a common type of resource shared between both systems that could be readily compared. For resource comparisons, memory usage for the integral image, filtering, and non-maximal value suppression portions of the pipeline are presented and considered. Additionally, the memory

resources used for each implementation are compared to the total memory resources available on each device.

## **4.2 Test Methods**

Before diving into the results, the test and analysis methods used in this chapter need to be outlined. The three factors tested and compared are latency, power consumption, and resource usage. Latency is measured in clock cycles and broken down into the integral image, filter calculation, and non-maximal value suppression portions of the pipeline. Power is measured in Watts and the maximum and minimum power consumption of each device is estimated. The memory resource usage in each implementation is estimated and broken down into the same three subsections as the latency tests. The methods used to test and analyze these factors are presented in the following sections.

### **4.2.1 FPGA**

#### **Latency**

To measure the latency on the FPGA, the Xilinx Vivado simulation tool was used. Pixel data was generated using a testbench and supplied to the SURF feature detection component in HDL. Using the simulation tool, the latency was found for the full SURF feature detection pipeline as well as the breakdown of latencies being introduced based on subsection. The three subsections examined are the integral image, filtered value calculation, and non-maximal value suppression portions of the feature detection algorithm.

## **Power**

Power consumption on the FPGA was estimated using Vivado's power estimation tool. Measuring the power consumption at the power supply was considered, but the board used for this thesis is a development board designed to demonstrate a variety of features that are not utilized in this thesis. This would have resulted in a grossly inaccurate power consumption measurement that would not be accurate for real world comparisons. For this reason, the Vivado power estimation tool was chosen for FPGA power estimations.

## **Resources**

The major memory resource consumers within the integral image calculation, filtering, and non-maximal value suppression portions of the pipeline on the FPGA are presented. The two main resources being considered are 32-bit registers and BRAMs used to perform the operations. The number of these resources required and how they scale are considered and compared to the GPU implementation.

### **4.2.2 GPU**

## **Latency**

Measuring the latency on the GPU version was done using Nvidia's profiling software. Two different profiling tools were used to better understand the latency breakdown of the GPU implementation. Nvidia Nsight Systems was used for the timeline functionality that provides the ability to see the order, latency, and number of kernel function calls at a high level. The Nvidia Nsight Compute tool was used to gain a more granular view of latency values for each of the three subsections being examined. The

latencies introduced between different kernels being executed were not included in the GPU latencies, only latencies incurred during the running of the kernels.

Due to the use of complex compilation tools and a scheduler for runtime, measuring latencies on the GPU is not as straightforward as measuring latencies on the FPGA. For functions that are called multiple times and should have relatively similar runtimes, the average is taken for all calls of that function as the expected latency. For kernels running in the pipeline which could operate in parallel, assuming the resources were available on the device, the longest average latency is used to represent that portion of the pipeline. For the NMS portion of the pipeline, the longest latency associated with any of the NMS kernel calls is used to represent that portion of the pipeline.

## **Power**

Power consumption on the GPU was estimated using Nvidia's system management interface (SMI) tool. Two commands were used to estimate the power consumption of the GPU while the algorithm was running. The first command, *nvidia-smi dmon -s p*, provides a rough estimate of the power consumption at one second intervals. The second command, *nvidia-smi --query-gpu=power.draw --format=csv --loop-ms=10*, samples at a rate of 100 samples per second and was used for a more granular view of power consumption during program runtime. As an absolute maximum value, the total power allowed to be pulled by a PCI-e connected device could be used. However, using the maximum power draw allowed by the appropriate PCI-e specification would not necessarily be representative of the system in a real world implementation and therefore will not be used.

## **Resources**

The major memory resources used by the integral image calculation, filtering, and non-maximal value suppression portions of the SURF feature detection pipeline are presented for the GPU implementation. The main memory resource considered is the amount of shared memory space on the GPU that is required for the frame buffers used by each portion of the pipeline. Due to the impact of the resolution of the image on the amount of shared memory utilized by the algorithm, only the number of full resolution frame buffers will be presented. The number of frame buffers required, and the effect they have on scaling, is considered and compared to the FPGA implementation.

## **4.3 Results**

### **4.3.1 Latency**

#### **FPGA**

The latency introduced during the feature detection process can be broken down into the integral image value calculation, the SURF filtering process, and the non-maximal value suppression process. The total amount of clock cycles introduced throughout the SURF feature detection process is 52,596 clock cycles. The latencies introduced by each section of the SURF feature detection algorithm will be expanded on in the sections below.

A comparison of the FPGA's SURF feature detection implementation to previous implementations can be seen in the expanded table below [11].

**Table 3. FPGA SURF Implementation Comparisons**

<b>SURF Version</b>	<b>Clock (MHz)</b>	<b>Resolution</b>	<b>FPS</b>	<b>Scales</b>	<b>Software/ Hardware</b>	<b>Descriptor</b>
Švab	100	1024x768	~10	8	S+H	No
Schaeferling	100	640x480	~2	8	S+H	Yes
Bouris	200	640x480	56	?	H	Yes
Fischer	125	640x480	406	1	H	Yes
Sledevič	25	640x480	60	6	H	Yes
Edgcombe	100	640x480	325	9	H	No

### **Integral Image**

On the FPGA, the integral image calculation only introduces a short latency. Due to input and output value buffering, the integral image component only introduces a latency of 2 clock cycles between being supplied a valid pixel intensity value and outputting the appropriate integral image value to the SURF filtering subsystem. Additionally, there exists a latency between valid pixel intensity values and valid integral image values output to the DMA of 7 clock cycles. The additional clock cycles added between intensity value input and integral value output to DMA are due to the logic required to format the integral image values into the AXIS stream format required for the DMA connection.

### **SURF Filtering**

The SURF filtering portion of the pipeline introduces latency for both the calculation process and partial frame buffer. The partial frame buffer used in the SURF



filtering process is a 75x75 pixel region. The latency introduced by filling this partial frame buffer is 47,435 clock cycles when using an image resolution of 640x480. The latency introduced due to filling a partial frame buffer can be calculated using equation 4.2.1.1 below where  $i_{cols}$  is the number of columns in the image,  $n_{rows}$  is the number of rows in the partial frame buffer,  $n_{cols}$  is the number of columns in the partial frame buffer, and  $L$  is the latency. Once the partial frame buffer has been filled with valid integral image values, the second order derivatives can be applied.

$$L = (n_{rows} - 1) * i_{cols} + n_{cols} \quad (\text{Eq. 4.2.1.1})$$

When applying the second order derivatives to the pixels, a latency of 3 clock cycles is introduced. Breaking down the clock cycles used for the second order derivatives: 1 clock cycle is used in the calculation of the subregion integrals, 1 clock cycle is used to sum the subregions according to the filter being applied, and 1 clock cycle is used to buffer the output of the second order derivatives. To further examine this breakdown, the code can be seen in the source code provided in section 3.3.1. Once the second order derivatives have been applied, the SURF filtered values can be calculated.

Once the second order derivatives have been calculated, the output values are converted to a 32-bit floating point representation so that they work with the FPGAs DSP resources. The conversion from signed 32-bit integer to 32-bit floating point format adds 1 clock cycle of latency. Once the values have been converted to a 32-bit floating point format, the values are passed through the DSP resources. The DSP resources are configured into a three-stage pipeline with two stages of multiplication and one stage of subtraction. The multiplication stages each introduce 8 clock cycles and the subtraction stage introduces 8 clock cycles as well. This results in the SURF filtered value

calculation introducing 24 clock cycles to the system. A summary of these latencies can be seen in the table below.

**Table 4. FPGA SURF Filtering Latencies**

<b>Purpose</b>	<b>Latency [Clock Cycles]</b>
Filling Integral Image Frame Buffer	47435
Second-Order Derivatives	3
Integer to Floating-Point Conversion	1
SURF Filtered Value Calculation	24

### **Non-maximal Value Suppression**

The NMS process introduces a significant latency for filling the 9x9 partial frame buffer with SURF filtered values, but the NMS calculation itself does not. Filling the 9x9 partial frame buffer with SURF filtered values adds a latency of 5129 clock cycles when using a 640x480 image resolution. The latency introduced when filling the partial frame buffer is calculated using equation 4.2.1.1 above. The NMS calculation and output only introduce 2 clock cycles of latency, one for comparison of all values and one for buffering the output. A table below is included to summarize the latencies introduced in the FPGA NMS calculation process.

**Table 5. FPGA NMS Latencies**

<b>Purpose</b>	<b>Latency [Clock Cycles]</b>
Filling NMS Frame Buffer	5129
NMS Comparison & Output	2

### **GPU**

The latency introduced on the GPU during the feature detection process is broken down into integral image value calculation, SURF filtering, and non-maximal value suppression. There are a few notes that should be considered prior to presenting the GPU

latency results. First, the preloading of the image into GPU accessible memory is not included in the latency measurements. This latency will scale with resolution and is a prerequisite to running the algorithm. Second, the time between kernel calls is not included in the GPU latencies being compared. The time between kernel calls should not be dependent on resolution but can add a significant number of latency clock cycles. Finally, the clock frequencies used on GPUs tend to be much higher than those used on FPGAs (sometimes an order of magnitude higher) so the realized latency in seconds may be lower for the GPU depending on the clock rates used for both systems. The total number of clock cycles of latency introduced by the GPU was measured to be 583,748. The breakdown of this number can be seen in the sections below.

### **Integral Image**

The integral image calculation step introduces a latency of 338,967 clock cycles. This number is such a significant portion of the latency introduced by the feature detection process because a thread is used for each of the pixels in the image. Each thread is responsible for adding all pixel intensity values above and to the left of the integral image pixel value being calculated. In addition to the number of threads and threadblocks required for this function being a limiting factor on the device, the number of additions is also significant. Luckily, the integral image calculation is only required to be performed once.

### **SURF Filtering**

The SURF filtering on the GPU can be broken down into two steps. The first step being the application of the second order derivatives to the image and the second step

being the SURF filtered value calculation. The first filtering step was found to introduce a latency of 23,355 clock cycles. When profiling, the second order derivatives were applied sequentially and grouped by octave, but assuming that the resources were available these operations could potentially be applied in parallel. For this reason, a best-case estimation is made with the assumption that a more powerful GPU may be able to apply all the filters in parallel.

The second step is the calculation of the SURF FPGA filtered values. This step was found to introduce a latency of 142,146 clock cycles. Like the first step, the second step could be applied for all filter widths in parallel assuming the GPU resources were available. For this reason, the latency for the second step will be assumed to be the average of all calls to this function during profiling. A summary of the latencies for both steps can be found in the table below.

**Table 6. GPU Filtering Latencies**

<b>Purpose</b>	<b>Latency [Clock Cycles]</b>
First Step Filtering	23355
Second Step Filtering	142146

### **Non-maximal Value Suppression**

For the NMS portion of the GPU pipeline, the longest latency in clock cycles was used to describe the latency it introduced. Due to the use of octave-based sample spacing for NMS, the first octave samples the most points and the last octave samples the least. The longest latency, found when running the first octave, was found to introduce a latency of 79,280 clock cycles to the pipeline. A table showing the latencies introduced when running the function for each of the octaves can be found below.

**Table 7. GPU NMS Latencies**

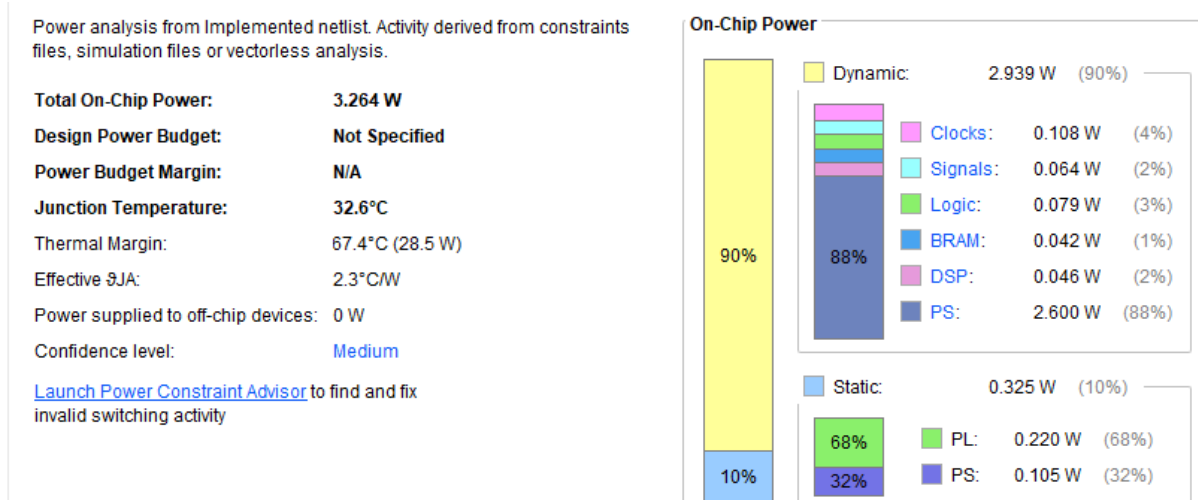
Octave	Time (us)	Clock Cycles
1	49.95	79280
2	9.73	14562
3	6.3	9054

#### 4.3.2 Power

A rough power estimation is performed for both implementations to estimate the power consumption of the devices while the algorithm is being run. The power estimation values are exactly that, estimations. These values are not necessarily meant to be true for all devices or all implementations. The estimated values are meant to provide a rough baseline for power consumption when using these devices, with the implementation discussed in this paper, in a larger system.

#### FPGA

The high-level power estimation for the FPGA found the device to have a low power consumption compared to the GPU. 3.264 W was the maximum estimated power consumption, and 0.325 W was the minimum estimated power consumption for the FPGA. This power consumption can be broken down into the static power consumption and the dynamic power consumption. The static power consumption is the amount of power consumed regardless of the operations that are occurring on the FPGA. The dynamic power consumption is the amount of power that can be consumed depending on the activity on the FPGA. The static power consumption was estimated to be 0.325 W and the dynamic power consumption was estimated to be 2.939 W. The power consumption summary provided by the Vivado power estimation tool can be seen in the figure below.



**Figure 35. FPGA Power Estimates**

## GPU

The power consumption measurements showed slightly higher power was required for the GPU implementation. Using the rough measurement command, the baseline power when idling the PC was found to be around 14 Watts. Using this same command while the accelerated program was running resulted in the GPU consuming around 36 Watts. This value was measured at an interval of 1 sample per second and the results can be seen in the figure below.

#	gpu	pwr
#	Idx	W
	0	14
	0	14
	0	14
	0	14
	0	13
	0	14
	0	14
	0	14
	0	14
	0	14
	0	14
	0	14
	0	13
	0	14
	0	13
	0	13
	0	13
	0	13
	0	13
	0	14
	0	13
	0	37
	0	36
	0	36
	0	36
	0	36
	0	36
	0	16
	0	14
	0	14
	0	14
	0	13

**Figure 36. Power Usage With GPU Accelerated Program Running**

Performing these power consumption measurements with more precise readings only solidified the previously measured values. The second power measurement performed on the GPU found a peak power consumption of 36.74 W while the baseline power consumption was found to be 12.61 W.

### 4.3.3 Resources

Memory resources were chosen as a common resource shared between the two forms of hardware acceleration which could be easily estimated based on the major

contributing factors found in each implementation. In the following sections, the memory resources will be broken down into the integral image, SURF filtering, and non-maximal value suppression portions of the pipeline and their resource usage compared. The comparisons also consider the relative availability of the resources used by each implementation when discussing their differences.

## FPGA

The total resource utilization table supplied by Vivado has been included in the figures below. With that being said, the analysis of memory resource utilization in this section will focus on the major contributing aspects of each portion of the pipeline being considered. Additional resources are required for each portion of the pipeline, but have been deemed insignificant due to their low contribution to total memory resource utilization and the lack of impact on image resolution scaling.

Name	CLB LUTs (70560)	CLB Registers (141120)	CARRY8 (8820)	F7 Muxes (35280)	F8 Muxes (17640)	Block RAM Tile (216)	DSPs (360)	GLOBAL CLOCK BUFFERS (196)	PS8 (1)
design_1_wrapper	41456	43417	1709	256	128	147.5	126	1	1
design_1_i (design_1)	41456	43417	1709	256	128	147.5	126	1	1
axi_dma_integ (design_1_i)	1189	1545	19	0	0	1	0	0	0
axi_dma_surf (design_1_i)	1082	1429	19	0	0	1	0	0	0
axi_dma_video (design_1_i)	1629	2135	28	0	0	2	0	0	0
axi_smc (design_1_i)	4330	6841	0	0	0	0	0	0	0
axis_data_fifo_0 (design_1_i)	61	57	0	0	0	1.5	0	0	0
axis_data_fifo_1 (design_1_i)	61	57	0	0	0	1.5	0	0	0
axis_data_fifo_2 (design_1_i)	61	57	0	0	0	1.5	0	0	0
ps8_0_axi_periph (design_1_i)	668	797	0	0	0	0	0	0	0
rst_ps8_0_100M (design_1_i)	19	40	0	0	0	0	0	0	0
SURF_v1_0_0 (design_1_i)	32056	30458	1643	256	128	139	126	0	0
zynq_ultra_ps_e_0 (design_1_i)	300	1	0	0	0	0	0	1	1

**Figure 37. Post-synthesis resource utilization**



Name	CLB LUTs (70560)	CLB Registers (141120)	CARRY8 (8820)	F7 Muxes (35280)	F8 Muxes (17640)	CLB (8820)	LUT as Logic (70560)	LUT as Memory (28800)	Block RAM Tile (216)	DSPs (360)	Bonded IOB (252)	PLL (6)	SYSMON4 (1)
design_1_wrapper	35858	41015	1652	256	128	6547	29744	6114	41015	137.5	126	3	1
design_1_j (design_1)	35858	41015	1652	256	128	6547	29744	6114	137.5	126	0	0	0
axi_dma_integ (design_1)	1016	1457	16	0	0	227	929	87	1	0	0	0	0
axi_dma_surf (design_1)	913	1341	16	0	0	228	828	85	1	0	0	0	0
axi_dma_video (design_1)	1372	1988	23	0	0	331	1254	118	2	0	0	0	0
axi_smc (design_1)	3660	5547	0	0	0	693	2678	982	0	0	0	0	0
axis_data_fifo_0 (design_1)	61	57	0	0	0	16	61	0	1.5	0	0	0	0
axis_data_fifo_1 (design_1)	61	56	0	0	0	17	61	0	1.5	0	0	0	0
axis_data_fifo_2 (design_1)	60	56	0	0	0	14	60	0	1.5	0	0	0	0
ps8_0_axi_periph (design_1)	624	760	0	0	0	188	555	69	0	0	0	0	0
rst_ps8_0_100M (design_1)	15	34	0	0	0	9	14	1	0	0	0	0	0
SURF_v1_0_0 (design_1)	28078	29719	1597	256	128	5182	23306	4772	129	126	0	0	0
zynq_ultra_ps_e_0 (design_1)	0	0	0	0	0	0	0	0	0	0	0	0	0

**Figure 38. Post-implementation resource utilization**

## Integral Image

The core contributing factor in the integral image calculation is the row buffer FIFO. FIFOs are implemented as logic surrounding BRAM resources that handle the write and read addresses as well as other control signals required to utilize BRAM resources as a FIFO. The row buffer requires as many BRAMs as are necessary to store a single row of the original resolution image. The BRAM used for the FIFO is the only significant memory resource used in the integral image calculation.

## Filtering

For the filtering portion of the FPGA, the main usage of resources is the 75x75 partial frame buffer. The partial frame buffer consists of a 75x75 matrix of 32-bit registers, for a total of 5625 32-bit registers. Additionally, 74 FIFOs are required to be used as row buffers. The number of BRAMs required for these row buffers is dependent on the number of columns in the original image resolution. With that being said, the 75x75 region is required to be a square region with dimensions equal to three times the largest filter width used in the algorithm. Unless the filter widths used in the algorithm change, the memory resources used in the filtering portion of the FPGA pipeline will only scale with the number of columns in the original image resolution.

## **NMS**

For the NMS portion of the pipeline, the only significant memory resource is the 9x9 partial frame buffer. The 9x9 partial frame buffer uses 81 32-bit registers and 8 FIFOs acting as row buffers. Similar to the filtering portion of the pipeline in the previous step, the memory resources used in the NMS portion of the pipeline only scale with respect to the number of columns in the original image resolution. The only other factor that would change the memory resources required by this portion of the pipeline would be the number of octaves used in the algorithm. With three octaves, sample spacings of 1,2, and 4 are used. Adding another octave would require a greater sample spacing and increase the amount of resources required for this portion of the pipeline. Now that we have considered the high-level memory resources utilized by the FPGA implementation, the memory resources used by the GPU implementation can be considered.

## **GPU**

For the GPU implementation, the main memory resource to be consumed is the shared memory available on the device. GPUs commonly have a significant amount of shared memory, often in the gigabytes of space, and can store many frames depending on the resolution of the frames being processed. The number of frames required for each processing step is what will be considered when analyzing memory resource consumption on the GPU. While other memory resources are available on the GPU, their utilization does not have a significant effect on the execution of the SURF implementation developed in this paper. Also, these other forms of memory are not significantly impacted by image resolution scaling.

At a minimum, the GPU implementation being run in parallel to reduce the latency of the system requires 22 full resolution frame buffers worth of space available during the filtering process. This number can be reduced at the expense of added latency, i.e., if the filters were applied sequentially or if intermediate frames were transferred to a different storage location.

### **Integral Image**

For the integral image portion of the GPU pipeline, a single full resolution frame buffer is required in shared memory in addition to the original intensity image frame buffer that is supplied. The intensity image frame buffer is referenced by all threads of this kernel to calculate the integral image values output to the integral image frame buffer. Since only a single frame buffer is used, the amount of memory consumed will scale directly with the number of pixels in the original image resolution.

### **Filtering**

For the filtering portion of the pipeline, four full resolution frame buffers are used per filter width. For this implementation, three of the frame buffers are used to store the results of the  $D_{xx}$ ,  $D_{yy}$ , and  $D_{xy}$  filtered values and the fourth frame buffer is used to store the SURF filtered values. If optimized, this portion of the pipeline can be performed with only three full resolution frame buffers per filter width if one of the frame buffers is reused to store SURF filtered values. This results in the total number of frame buffers required ranging from 21 to 36 frame buffers depending on whether only unique filter widths are processed and whether or not the fourth frame buffer reuses one of the first stage frame buffers. Using the best-case analysis, this section's memory resource usage

will scale at a rate of 21 times the area introduced by increasing the resolution of the original image.

## **NMS**

The NMS portion of the pipeline only adds a single new full resolution frame buffer per octave. This process results in a total of three full resolution frame buffers for the three octaves used in this algorithm. The number of NMS frame buffers could be reduced to a single full resolution frame buffer that uses a bitmask for each pixel to represent maximal values being found at each octave. However, this would either reduce the ability to safely run this process in parallel for all octaves due to potential write conflicts or increase the latency due to sequential execution. Additionally, the NMS frame buffers could be reduced in size to eliminate the exclusionary region around the border of the image, but this would only result in the elimination of a constant area region which would not affect the extra memory resources required when scaling the resolution of the images being processed. Using the best-case analysis, the NMS portion of the GPU pipeline scales with 3 times the number of new pixels required for the new resolution.

## **4.4 Summary**

### **4.4.1 Latency**

After performing the latency tests, the FPGA implementation was found to perform better with respect to latency times than the GPU implementation. Not only does the FPGA implementation have a lower pixel input to feature point location output high-level latency, the FPGA implementation was also found to perform better for all of the examined subsections of the pipeline. Additionally, the FPGA implementation handles

resolution scaling as good or better than the GPU for all examined subsections. These results were expected, it is common for FPGAs to be used for operations that require or desire low latency such as in networking applications.

#### **4.4.2 Power**

Comparing the two implementations, the FPGA was found to consume significantly less power than the GPU while running the algorithm and while in an idle state. Part of this difference can be attributed to the necessity of the GPU to accelerate other code for the computer system to which it is attached. If we subtract the baseline power consumed by the GPU however, the GPU is still consuming roughly 8 times more power than the FPGA when actively executing the algorithm.

#### **4.4.3 Resources**

When comparing the results for the memory resource usage, the amount of memory resources required for both implementations is comparable relative to the total available memory resources. However, the GPU performed as good or better than the FPGA when resolution scaling was considered for all examined subsections. While the amount of memory stored in bytes is greater on the GPU, the amount of storage is also significantly greater on the GPU. A summary of the test results can be seen in the table below.

**Table 8. Summary of Test Results**

	Implementation	
	Current	Scaling
Latency	FPGA	FPGA
Power	FPGA	N/A
Memory	FPGA	GPU

## **CHAPTER 5**

### **FUTURE WORK**

#### **5.1 Potential Design Improvements**

Regarding the work performed for the FPGA portion of this thesis, a great deal of completed work has been left out of the finished solution. The completed work during the development of this thesis included a full multi-image sensor input to display port output image processing pipeline. The image processing pipeline was originally implemented to receive images from two image sensors simultaneously and output the mixed image result to a monitor using a DisplayPort connection. This involved an AXIS switch for selecting the desired image stream, an image gate to allow only a single frame through the SURF feature detection logic at a time, image sensor configuration and initialization code for two OV5640 image sensors, DisplayPort output logic, and a custom DMA block to mix the two images using an arbitrary homography matrix. Additionally, HDL was written to accelerate the homography calculation portion of the image stitching process.

However, these pieces were excluded from the final implementation due to the difficulty of pulling together the SURF feature detection fabric and the rest of the image processing pipeline. Instead, the finished project used DMA to supply images to the fabric and to receive the feature point locations.

In future work, it would be beneficial to combine these features together to produce a low power image stitching product that could be used in edge computing applications. A single chip end-to-end real-time image stitching solution could be used for dynamic initialization of a multiple image sensor system or in the dynamic combination of image streams moving independently from each other.

## **CHAPTER 6**

### **CONCLUSION**

#### **6.1 Summary of Work**

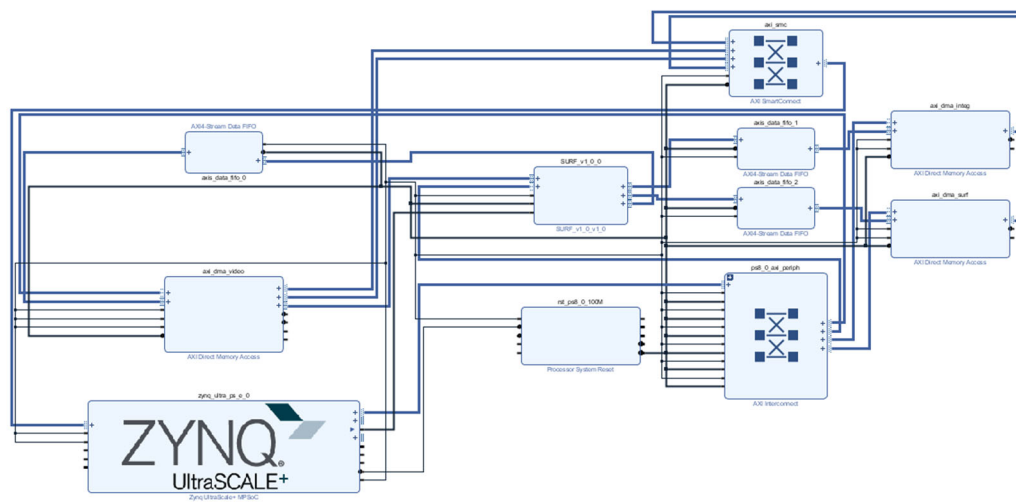
In this project, two implementations of an image stitching pipeline were developed on two different forms of hardware acceleration. The two forms of hardware acceleration used were an FPGA and a GPU. The development of the GPU version of the pipeline took roughly 6 weeks while the FPGA implementation took closer to 6 months. This difference was largely due to my lack of experience developing large systems on an FPGA prior to this project. Once both implementations had been developed, the implementations were compared based on latency, power consumption, and memory resource usage.

#### **6.2 Summary of Results**

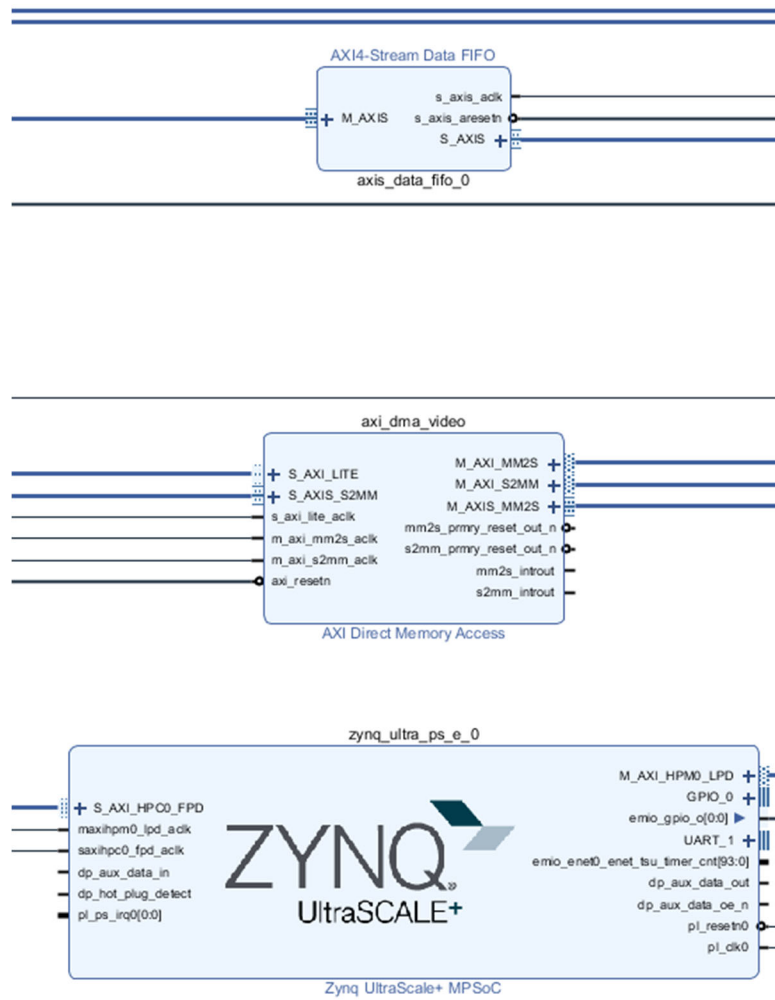
The tests were performed on only the accelerated portion of the image stitching pipeline. The tests showed that the FPGA had lower latency, the GPU had lower relative memory resource consumption, and the FPGA had lower power consumption. The GPU handles resolution scaling as good or better than the FPGA with regards to memory resource consumption. The FPGA handles resolution scaling as good or better than the GPU with regards to latency. If the project in this thesis were to be implemented and required large resolutions that may need to be increased later in the project, a GPU would likely be a better choice. If power and latency were important and the resolution could be constrained, an FPGA would likely be the better choice.



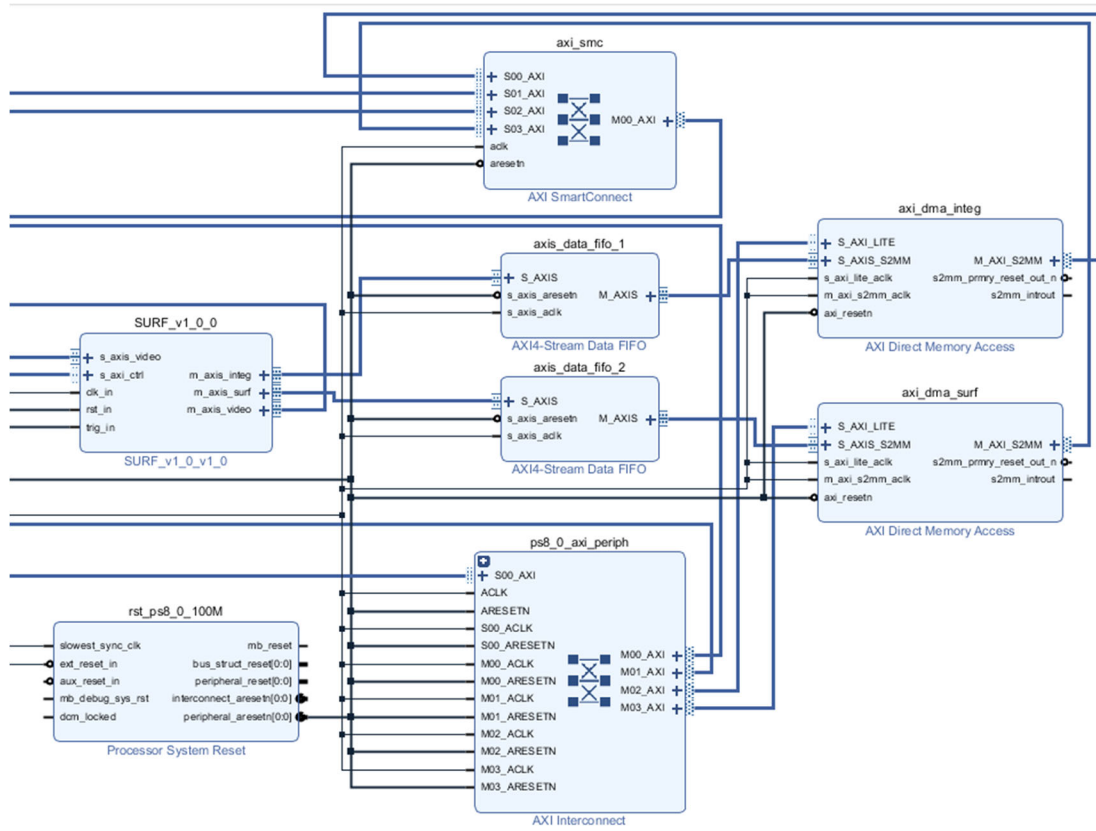
## APPENDIX A: FIGURES



### Figure 39. FPGA Top Level Block Diagram (Full)



**Figure 40. FPGA Top Level Block Diagram (Left)**



**Figure 41. FPGA Top Level Block Diagram (Right)**

**Table 9. GPU Dxx Filtering Latencies**

Filter Width	Time (us)	Clock Cycles
3	13.31	20761
5	13.02	17195
7	13.41	17413
9	13.12	20179
13	13.44	17302
17	11.71	17455
25	11.58	19963
Average	12.8	18610

**Table 10. GPU Dyy Filtering Latencies**

Filter Width	Time (us)	Clock Cycles
3	13.79	20706
5	13.79	18757
7	13.57	18582
9	13.63	18647
13	13.63	20464
17	12.38	18528
25	13.57	20289
Average	13.48	19425

**Table 11. GPU Dxy Filtering Latencies**

Filter Width	Time (us)	Clock Cycles
3	16.03	24908
5	14.34	24374
7	15.94	22405
9	16.26	24850
13	16.13	22418
17	16.13	22156
25	16.29	22374
Average	15.88	23355

**Table 12. GPU SURF Filtering Latencies**

Filter Width	Time (us)	Clock Cycles
3	89.89	145105
5	91.14	143713
7	90.34	143598
5	90.62	140559
9	89.95	139720
13	89.5	142109
9	88	140336
17	89.57	142051
25	89.66	142123
Average	89.86	142146

```

$ nvprof ./surf.exe

====SURF Frame One====
==19188== NVPROF is profiling process 19188, command: ./surf.exe
Performing NMS...
Found point.
Number of feature points: 993

====SURF Frame Two====
Performing NMS...
Found point.
Number of feature points: 1087

First error: 1832.050781
New best error: 271.513275
New best error: 163.327133
New best error: 90.082878
New best error: 79.920891
New best error: 75.783676
New best error: 64.065804
New best error: 61.614037
New best error: 56.425365
New best error: 56.409096
New best error: 52.332645
New best error: 51.755173

==My Homography==
| 0.59076 -0.00467 919.86212 |
| -0.23637 0.90246 352.40436 |
| -0.00031 -0.00005 1.00000 |

Matched points: 84
Waiting for any key...
==19188== Profiling application: ./surf.exe
==19188== Warning: 2 records have invalid timestamps due to insufficient device buffer space. You can configure the buffer space using
the option --device-buffer-size.
==19188== Profiling result:

```

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	62.22%	44.739ms	6	7.4565ms	25.472us	22.345ms	[CUDA memcpy HtoD]
	19.01%	13.671ms	4736	2.8860us	896ns	35.519us	gpu_integral_img(int*, float*, int, int, int)
	15.42%	11.091ms	84	132.03us	24.288us	933.01us	[CUDA memcpy DtoH]
	2.21%	1.5926ms	18	88.477us	87.455us	90.943us	filter(float*, int*, int*, int*, int, float)
	0.32%	228.22us	18	12.679us	12.448us	12.960us	dxv_box_filter(int*, int*, int)
	0.27%	197.47us	18	10.970us	9.5040us	14.880us	dxv_box_filter(int*, int*, int)
	0.25%	182.53us	18	10.140us	9.8240us	10.431us	dyy_box_filter(int*, int*, int)
	0.16%	112.61us	6	18.767us	4.0960us	45.823us	nms(int*, float*, float*, float*, int, float)
	0.13%	93.054us	2	46.527us	46.399us	46.655us	gpu_pad_img(float*, float*, int, int, int, int)
API calls:	73.41%	328.79ms	88	3.7362ms	11.000us	301.51ms	cudaMalloc
	12.19%	54.581ms	1	54.581ms	54.581ms	54.581ms	cuDevicePrimaryCtxRelease
	8.23%	36.874ms	90	409.71us	48.200us	4.7122ms	cudaMemcpy
	4.04%	18.112ms	4818	3.7590us	2.8000us	79.800us	cudaLaunchKernel
	1.84%	8.2359ms	78	105.59us	84.500us	182.50us	cudaFree
	0.28%	1.2645ms	8	158.06us	7.1000us	479.30us	cuModuleUnload
	0.01%	25.000us	1	25.000us	25.000us	25.000us	cuDeviceTotalMem
	0.00%	18.600us	101	184ns	100ns	1.2000us	cuDeviceGetAttribute
	0.00%	3.5000us	3	1.1660us	300ns	2.9000us	cuDeviceGetCount
	0.00%	2.3000us	2	1.1500us	200ns	2.1000us	cuDeviceGet
	0.00%	1.1000us	1	1.1000us	1.1000us	1.1000us	cuDeviceGetName
	0.00%	600ns	1	600ns	600ns	600ns	cuDeviceGetLuid
	0.00%	300ns	1	300ns	300ns	300ns	cuDeviceGetUuid

Figure 42. Nvprof Profiling of GPU Accel

## BIBLIOGRAPHY

- [1] H. Bay, A. Ess, T. Tuytelaars and L. Van Gool, "Speeded-Up Robust Features (SURF)," *Computer Vision and Image Understanding*, vol. 110, no. 3, pp. 346-359, 2008.
- [2] S. M. Trimberger, "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 318-331, 2015.
- [3] National Instruments, "Introduction to FPGA Resources - LabVIEW NXG 5.1 FPGA Module Manual," National Instruments, 27 February 2021. [Online]. Available: <https://www.ni.com/documentation/en/labview-fpga-module/latest/fpga-targets/intro-fpga-resources/>. [Accessed 14 May 2021].
- [4] Nvidia, "NVIDIA Launches the World's First Graphics Processing Unit: GeForce 256," 31 August 1999. [Online]. Available: [https://web.archive.org/web/20160412035751/http://www.nvidia.com/object/IO\\_20020111\\_5424.html](https://web.archive.org/web/20160412035751/http://www.nvidia.com/object/IO_20020111_5424.html). [Accessed 23 May 2021].
- [5] Nvidia, "Programming Guide :: CUDA Toolkit Documentation," Nvidia, 20 April 2021. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. [Accessed 13 May 2021].
- [6] M. Brown and D. Lowe, "Invariant Features from Interest Point Groups," in *British Machine Vision Conference*, 2002.
- [7] C. Harris and M. Stephens, "A combined corner and edge detector," in *In Proc. of Fourth Alvey Vision Conference*, 1988.
- [8] A. P. Witkin, "Scale-Space Filtering," in *The International Joint Conferences on Artificial Intelligence*, 1983.
- [9] J. Sellner, "Scale invariance through Gaussian scale space - Milania's Blog," 25 August 2016. [Online]. Available: [https://milania.de/blog/Scale\\_invariance\\_through\\_Gaussian\\_scale\\_space](https://milania.de/blog/Scale_invariance_through_Gaussian_scale_space). [Accessed 13 05 2021].
- [10] E. Oyallon and J. Rabin, "An Analysis of the SURF Method," *Image Processing On Line*, vol. 5, pp. 176-218, 2015.
- [11] T. Sledevic and A. Serackis, "SURF algorithm implementation on FPGA," in *2012 13th Biennial Baltic Electronics Conference*, 2012.
- [12] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91-110, 2004.
- [13] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1981.
- [14] J. Shi and C. Tomasi, "Good Features to Track," in *IEEE Conference on Computer Vision and Pattern Recognition*, 1994.

- [15] W. Förstner, "A Feature Based Correspondence Algorithm for Image Matching," in *Int. Arch. of Photogrammetry and Remote Sensing*, 1986.

