Spring 5-28-2015

Fast Semivariogram Computation Using FPGA Architectures

Yamuna Sri Lagadapati

FAST SEMIVARIOGRAM COMPUTATION USING FPGA ARCHITECTURES

by

YAMUNA SRI LAGADAPATI

A thesis submitted in partial fulfillment
of the requirements for the degree of
Masters of Science in Electrical Engineering
Department of Electrical Engineering

Mukul Shirvaikar, Ph.D., Committee Chair

College of Engineering and Computer Science

The University of Texas at Tyler
May 2015

The University of Texas at Tyler
Tyler, Texas

This is to certify that the Master's thesis of

YAMUNA SRI LAGADAPATI

has been approved for the thesis requirements on
April 1st, 2015
for the Master of Science in Electrical Engineering

Approvals:

Thesis Chair: Dr. Mukul V. Shirvaikar, Ph.D.

Member: Dr. Hector Ochoa, Ph.D.

Member: Dr. Ron J. Pieper, Ph.D.

Chair and Graduate Coordinator: Dr. Hassan El-Kishky, Ph.D.

Dr. James K. Nelson, Jr., Ph.D., P.E.,
Dean, College of Engineering.

ACKNOWLEDGEMENTS

I would like to express my special appreciation and thanks to my advisor Professor Dr. Mukul Shirvaikar, for being a tremendous mentor. I am grateful to him for his exceptional guidance, encouragement, support and patience to complete my thesis successfully. I would also like to thank my committee members, Professor Dr. Ron J. Pieper, Professor Dr. Hector Ochoa, for serving as my committee members even at hardship. I also want to thank them for letting my defense be an enjoyable moment, and for your brilliant comments and suggestions.

A special thanks to my parents and family for their constant encouragement and financial support to pursue my master of science in Electrical Engineering and for all of the sacrifices that you've made on my behalf. I would also like to thank all of my friends who supported me in writing, and encouraged me to strive towards my goal. At the end I would like express appreciation to my beloved husband Sundeep who spent sleepless nights with and was always my support in the moments when there was no one to answer my queries.

I would like to thank each and every person for their encouragement and support to complete my thesis work successfully.

TABLE OF CONTENTS

List of Tables iv

List of Figures v

Abstract vii

Chapter One Introduction 9

1.1 Introduction to Texture 9

1.2 Various Texture Analysis Techniques 10

1.3 Objective and Framework 12

1.4 Organization of Thesis 12

Chapter Two Past Work 14

Chapter Three Technical Background 18

3.1 Semi-variogram 18

3.1.1 Range (L) 19

3.1.2 Sill Variance (C) 19

3.1.2 Nugget Variance (C_0) 20

3.2 Different Types of Semivariogram Models 22

3.3 Field Programmable Gate Arrays: 25

Chapter Four Methods and Experimental Procedures 28

4.1 Architecture Version 1 (Not Pipelined): 29

4.2 Architecture Version 2 (Pipelined): 32

4.3 Architecture Version 3 (Pipelined and Hardware Reutilization): 34

4.4 Image Buffer and Counter Module 34

4.4.1 Architecture Version 1 34

4.4.2 Architecture Version 2 36

4.5 Distance Module 38

4.5.1 Architecture Version 1 38
4.5.2 Architecture Version 2 39
4.6 Difference Module 40
4.6.1 Architecture Version 1 40
4.6.2 Architecture Version 2 40
4.7 Sorting Module 42
4.7.1 Architecture Version 1 42
4.7.2 Architecture Version 2 43
4.8 Variogram Calculation Module 44
Chapter Five Results and Design Implementation 46
5.1 Image Buffer Implementation Results 46
5.2 Distance Module Implementation Results 48
5.3 Difference Module Implementation Results 50
5.4 Sorting module Implementation results 52
5.5 Variogram Calculation Module Implementation Results 54
5.5.1 Implementation for Architecture Version 1 54
5.5.2 Implementations for Architecture Version 2 for a 10 × 10 Image 55
5.5.3 Implementations for Architecture Version 2 for a 20 × 20 Image 56
Chapter Six Discussion and Conclusion 61
6.1 Conclusion 61
6.2 Future Work 62
References 63
Appendix A VHDL Code 65

LIST OF TABLES

Table 2.1 Summary of All Previous Studies 17

Table 3.1 Stochastic Parameters of Experimental Variogram 20

Table 3.2 Calculation of Difference of Pixels in the 1-D Array to Compute Variogram for Lag Distance h=1 21

Table 3.3 Number of Computations for Variogram Calculation for $P = M \times N$ Image . 24

Table 5.1 Computational Time Comparison for Matlab Code, Non-Pipelined and Pipelined Architecture Versions of Implementations. 58

Table 5.2 Verification of the Semivariogram Values Computed Using the Specified Algorithm for a 10×10 Image. 58

Table 5.3 Verification of the Semivariogram Values Computed Using the Specified Algorithm for a 20×20 Image. 59

Table 5.4 Device Utilization and Synthesis Report 60

LIST OF FIGURES

Figure 3.1 Characteristics of Semivariogram 20

Figure 3.2 Types of Semivariogram Models 23

Figure 3.3 Xilinx XUPV5-LX110T Development Kit with Virtex 5 FPGA..... 27

Figure 4(a) Cropped Image of the Hip Used for Experiments..... 29

Figure 4(b) Original Scanned Image of the Hip Used for Experiments 29

Figure 4.1(b) Block Diagram of Architecture Version 1 (Non Pipelined) 31

Figure 4.2(a) Timing Diagram of Architecture Version 2 (Pipelined) 32

Figure 4.2(b) Block Diagram of Architecture Version 2 (Pipelined) 33

Figure 4.4.1 (a) Block Diagram of Counter Module (Not Pipelined)..... 35

Figure 4.4.1 (b) Block Diagram of Image Buffer Module (Not Pipelined)..... 36

Figure 4.4.2 (a) Block Diagram of Counter Module (Pipelined)..... 37

Figure 4.4.2 (b) Block Diagram of Image Buffer Module (Pipelined)..... 37

Figure 4.5.1 Block Diagram of Distance Module (Not Pipelined) 38

Figure 4.5.2 Block Diagram of Distance Module (Pipelined) 39

Figure 4.6.1 Block Diagram of Difference Module (Not Pipelined)..... 40

Figure 4.6.2(a) Block Diagram of Difference Module (Pipelined) 41

Figure 4.6.2(b) Block Diagram for Skipping Algorithm 42

Figure 4.7.1 Block Diagram of Sorting Module (Not Pipelined) 43

Figure 4.7.2 Block Diagram of Sorting Module (Pipelined) 44

Figure 4.8 Block Diagram of Variogram Module 45

Figure 5.1 (a) Implementation Results for Image Buffer (Not Pipelined)..... 46

Figure 5.1 (b) Implementation Results of 10 × 10 Image Buffer (Pipelined) 47

Figure 5.1 (c) Implementation Results of 20 × 20 Image Buffer (Pipelined) 47

Figure 5.2 (a) Implementation Results for Distance Module (Not Pipelined)..... 48

Figure 5.2 (b) Implementation Results of 10 × 10 Image Distance Module (Pipelined) . 49

Figure 5.2 (c) Implementation Results of 20 × 20 Image Distance Module (Pipelined) . 49

Figure 5.3 (a) Implementation Results for Difference Module (Non-Pipelined) 50

Figure 5.3 (b) Implementation Results of 10 × 10 Image Difference Module (Pipelined) 51

Figure 5.3 (c) Implementation Results of 20 × 20 Image Difference Module (Pipelined) 52

Figure 5.4 (a) Implementation Results for Sorting Module (Non-Pipelined)..... 53

Figure 5.4 (b) Implementation Results for Sorting Module (Pipelined)..... 54

Figure 5.5.1 .Implementation Results for Architecture Version 1..... 55

Figure 5.5.2 Implementation Results for Architecture Version 2 for a 10 × 10 Image.... 56

Figure 5.5.3 Implementation Results for Architecture Version 2 for a 20 × 20 Image.... 57

ABSTRACT

FAST SEMIVARIOGRAM COMPUTATION USING FPGA ARCHITECTURES

Yamuna Sri Lagadapati

Thesis Chair: Mukul Shirvaikar, Ph. D.

The University of Texas at Tyler
May 2015

The semivariogram is a statistical measure of the spatial distribution of data, and is based on Markov Random Fields (MRFs). Semivariogram analysis is a computationally intensive algorithm that has typically seen applications in the geosciences and remote sensing areas. Recently, applications in the area of medical imaging have been investigated, resulting in the need for efficient real time implementation of the algorithm. A semi-variance, $\gamma(h)$, is defined as the half of the expected squared differences of pixel values between any two data locations with a lag distance of h. Due to the need to examine each pair of pixels in the image or sub-image being processed, the base algorithm complexity for an image window with n pixels is $O(n^2)$.

Field Programmable Gate Arrays (FPGAs) are an attractive solution for such demanding applications due to their parallel processing capability. FPGAs also tend to operate at

relatively modest clock rates measured in a few hundreds of megahertz. This thesis presents a technique for the fast computation of the semivariogram using two custom FPGA architectures. A modular architecture approach is chosen to allow for replication of processing units. This allows for high throughput due to concurrent processing of pixel pairs. The current implementation is focused on isotropic semivariogram computations only. The algorithm is benchmarked using VHDL on a Xilinx XUPV5-LX110T development Kit, which utilizes the Virtex5 FPGA. Medical image data from MRI scans are utilized for the experiments. Implementation results of the first architecture shows that a significant advantage in computational speed is attained by the architectures with respect to Matlab implementation on a personal computer with an Intel i7 multi-core processor. It is also observed that the massively pipelined architecture is nearly 100 times faster than the non-pipelined architecture.

CHAPTER ONE

INTRODUCTION

Medical image processing is a rapidly advancing area with many modalities like X-ray, Magnetic resonance imaging (MRI), and ultra-scanning etc.,. Automated processing of medical images can provide medical personnel invaluable assistance in the diagnostic process. Large amounts of data are used to represent a typical image, so the analysis of an image needs a large amount of memory and can take more time. In order to reduce the amount of data, an image is typically processed to generate a set of features. Feature extraction is a primitive type of pattern recognition, and it is very important to extract information from an image and may involve features such as color, shape, and texture. Features can be used to extract quantitative information about an image or for the tasks such as searching, retrieval, and storage. Features are divided into different classes based on the kind of properties they describe. Proper selection of features is critical to aid diagnosis using medical imaging.

1.1 Introduction to Texture

Texture is an important feature, which quantifies gray level differences (contrast), over a defined size of area where change occurs (window), and directionality. It plays an important role in human vision and in image classification. Pictures of flowers, walls, water, or patterns on a fabric or single objects are distinguished according to their texture. The observation of texture depends on certain conditions such as light, angle, distance, or other environmental effects. Texture features contain not only the visual characteristics information, but also the characteristics which cannot be visually differentiated. “Texture” as it is used in this context refers to the visual effect produced by the spatial distribution of pixel value variation over relatively small areas.

Texture is the pattern of information or arrangement of the structure found in a picture, which uses features in the analysis and interpretation of images. There are two types of texture based on spatial frequency, namely, fine and coarse. Fine textures have high spatial frequencies or a high number of edges per unit area. Coarse textures have low spatial frequencies or a small number of edges per unit area. Texture analysis on radiographs is a common way to investigate bone microarchitecture. Stochastic parameters range, nugget and sill are calculated from the semivariogram plot can be used to represent spatial variations in bone image. As the lag distance increases, it is suggested that the bone is more dissimilar on average. Correlation length describes the degree of smoothness or roughness in the map. A relatively larger correlation length implies a smooth variation, whereas a smaller correlation length corresponds to acute changes over the spatial domain. The semi-variance converges to the sum of the nugget variance and the sill variance when the separation distance (h) approaches infinity.

1.2 Various Texture Analysis Techniques

Texture analysis is generally a difficult problem due to the diversity and complexity of natural textures. Texture features should use a minimal amount of resources while being able to accurately describe the underlying phenomena of interest of the data. There are four different types of texture features: statistical, structural, model based, and transform based.

Statistical texture features can be obtained using the higher order statistics of pixel gray levels. First order statistical features measure the probability of observing a gray value in the image at a randomly chosen location. First order statistics can be computed from the histogram of pixel intensities in the image. These depend only on individual pixel values, and not on the interaction or co-occurrence of neighboring pixel values. The average intensity in an image is an example of the first order statistic. Gray level co-occurrence matrix [GLCM] is the second statistical texture analysis introduced by Haralick, et al [1]. This technique is commonly used in texture analysis, because it provides a large set of features for each sample and it can be assumed that at least one of these features reflects the small variation of texture. Several statistical parameters can be extracted from the

GLCM, to quantify the spatial relationship between pixels within the area under investigation.

Probability models have gained wide acceptance, because of their modelling power and expressiveness. These models pose the problem of texture analysis in a statistical setting, which allows a wide range of well-established theories and methodologies in mathematical statistics to be introduced into texture modelling. In particular, Markov random fields (MRFs), which describe a texture in terms of spatial geometry and quantitative strengths of inter-pixel statistical dependency. The semivariogram is a method that can be applied to two dimensional plain-projection images and is based on Markov random fields (MRF).

Semivariogram analysis is a computationally intensive algorithm that has typically seen applications in the geosciences and remote sensing areas [2]. Recently, applications in the area of medical imaging have been investigated, resulting in the need for efficient real time implementation [3]. The semivariogram is a plot of semi variances for different lag distances between pixels. It is commonly represented as a graph that shows the variance in measure with distance between all pairs of sampled locations. Such a graph is helpful to build a mathematical model that describes the variability of the measure with location. It is a property used to express the degree of relationship between pixels of an image. The semivariance value typically increases with the lag distance converging to a constant limit called the “sill”. The sill of a model can be used to describe the variability as well. The variance gradually increases till a threshold is reached in the distance of separation. This threshold is called a “range”. Once the distance between two points is beyond range, the variance becomes independent of the distance and maintains a constant value. Thus, the inverse of the range can be used to measure variability. When the variogram is extrapolated to zero distance, the variance reaches a non-zero value called a nugget. The value increases rapidly at low lags, and then progresses linearly. Strictly speaking, this value should be zero when the distance between two points is zero. However, some factors such as sampling error may cause dissimilar values for samples at locations close to each other.

The semivariogram displays the average change of a property and relation between a pair of pixels with changing lags. It can be used as a descriptor of second-order statistics within the image and hence provide a quantitative measure of texture. The determination of the spatial variability of field parameters is usually based on the concept that sampled values at nearby locations are more similar than those from further apart. Measurements from the field are usually gathered as point data, such as an individual plant. Geostatitical analysis methods can be used to interpolate the measurements to create a continuous surface map or to describe its spatial pattern. As a powerful tool in geostatistics, variogram describes the spatial dependence of data and gives the range of spatial correlation, within which the values are correlated with each other, and beyond which they become independent. The effect of sampling on the accuracy of sample variogram was studied from independently generated random fields and from experimental data. Variogram has been estimated and investigated in a wide range of remote sensing applications.

1.3 Objective and Framework

This work intends to implement the semivariogram calculation with an FPGA module taking advantage of its re-configurability characteristics. The aim is to have faster calculation times using design techniques to implement parallelism and pipelining, which is not possible with dedicated DSP designs. The FPGA Kit used in this project is the Xilinx XUPV5-LX110T Development Kit, which utilizes the Virtex5 FPGA. A VHDL test bench was designed to verify the functionality followed by synthesizing the design, real hardware, and developing test applications to verify functionality and performance of the design.

1.4 Organization of Thesis

This thesis is divided into six chapters. Chapter 2 gives a brief study about the previous works, which are related to the current study. Chapter 3 explains the technical terms (variogram, parameters like range, sill, and nugget), and describes the FPGA board used in the current study. Chapter 4 describes the architectures implemented and experimental procedures to find the semivariogram values. Chapter 5 lists and analyzes the simulation

and implementation results for the architectures, computing the variogram values. Chapter 6 consists of the conclusion and future improvements.

CHAPTER TWO

PAST WORK

A brief literature survey reveals past attempts at implementing algorithms such as the variogram on FPGA hardware to increase its speed. Marcotte, proposed two programs based on the Fast Fourier Transform (FFT) algorithm to compute direct, cross, and pseudo-cross variogram for the fast computation of variogram [4]. This paper shows how to compute an experimental variogram for data that are on a regular grid (like the pixels of an image). The grid does not need to be complete; any missing data on the grid is simply represented as NaN. The computation use the FFT which is orders of magnitude faster than the usual approach based on scanning each available pair of data in turn. The computation of variograms using the FFT is done by defining two indicator matrices which have ones at each data point and zeros elsewhere. These two indicator matrices are used in process of calculating the variogram. The computation is performed at lower cost by shifting the data matrix, and the data pairs are formed by overlapping the original matrix and shifted matrix. The FFT approach is shown to be faster than the spatial approach for calculating the variograms especially when repetitive variogram computations are required.

Most other work appears to focus on the fast computation of gray-level co-occurrence matrices (GLCM). It is worth examining these approaches since the GLCM also involves comprehensive pixel-pair data extraction. GLCM is computed using the frequency of occurrence of pixel pair values in the image. Roumi proposed an FPGA-based architecture for parallel computation of symmetric co-occurrence matrices [5]. Symmetrical algorithms are faster than non-symmetrical, and also a hardware implementation consumes less area and less power compared to a software implementation. He proposed an FPGA architecture which is capable of calculating GLCMs in parallel for four different distances in four directions. The approach improves

by a factor of 2 to 4 the processing time for simultaneous computation of sixteen co-occurrence matrices. The feature calculation operation has two steps. In the first step, mean, contrast, entropy, and dissimilarity are calculated by four different Processing Elements (PEs). Processing elements contain multipliers and adders that execute in parallel. Furthermore, for increasing the computation speed, RAM is used for the calculation of the log function in the entropy measure. In the second step, the angular second moment, variance, and correlation are calculated. Further, he found that Virtex-XCV2P30 has a better throughput than Virtex4 and Virtex5.

Harshavardhan *et al* implemented a novel FPGA-based architecture for real-time extraction of four GLCM features by dividing the architecture in two stages, a pre-processing stage, and the feature extraction block [6]. The first stage prepares input data for processing by the feature extraction block while the second combines both software and hardware to calculate GLCM features. The hardware module is implemented on a Xilinx FPGA using Verilog. This module consists of the control unit which coordinates the functionality of the FPGA, by generating the signals which synchronize the other units, a memory controller and a feature calculation unit capable of reading GLCMs, for extracting the required features and storing them into the on-card memory. Thus in this paper image features have been extracted using different algorithms specified with architectural models with internal modules represented.

Haralick texture feature extraction algorithms can be divided into two parts: calculation of the co-occurrence matrices and calculation of texture features using the calculated co-occurrence matrices which are computationally intensive. Akoushideh *et al* proposed a parallel FPGA architecture to calculate co-occurrence matrices and thirteen texture features [7]. In the proposed architecture, in order to improve performance, first, the co-occurrence matrix is computed then all thirteen texture features are calculated in parallel using computed co-occurrence matrix. The proposed architecture has been implemented on Virtex 5 fx130T-3 FPGA device. Results show a speedup of 421 yields over a software implementation on Intel Core i7 2.0 GHz processor. In order to improve the performance, 3 texture features contrast, mean and sum of entropy are computed instead of 13 using ranking of Haralick's features based on their important role in texture

classification. Evaluation results showed a performance improvement of 4849 yields compared to software implementation on Intel Core i7 2.0 GHz processor. In addition, the clock divider technique was applied with parallel implementations on a cell processor. Experimental results show that using 16 processing elements in parallel provided speedups of up to 10 times the non-parallelized implementation.

Girisha *et al*, presented an FPGA architecture for Gray Level Co-occurrence Matrix (GLCM) to increase the speed of computation [8]. The GLCM architecture was implemented using Verilog hardware description language. The design was focused on GLCM hardware realization. This paper does not include the Haralick feature extraction of the image; it calculates the GLCM for four different angles 0°, 45°, 90° and 135°for a given intensity of an image. Later, they proposed a novel system for texture feature extraction of video frames using GLCM using hardware [9]. The properties or features extracted from normalized symmetrical GLCM are Energy or Angular second moment, correlation, homogeneity and contrast. The proposed architecture involves extracting frames from video which are resized to 8×8 image and scaled down to 8 tone image creating the GLCM. Then making the obtained matrix symmetric, then normalizing it, and finally extracting the required features.

Wielgosz has proposed a FPGA Architecture for Kriging Image Interpolation in three steps: finding a basis for interpolation, constructing a variogram matrix and computing coefficients and interpolation [10]. Image interpolation is the basis for quality shaping of the image. An effective interpolation mechanism is obtained by implementing it on FPGA. The architecture aims to decrease the overall latency in FPGA. Constructing the variogram matrix involves implementing it in a pipelined fashion but the architecture is not optimal in terms of image data distribution.

Table 2.1 shows the various research studies in the literature arranged chronologically, that were aimed at improving the speed of pixel-pair statistical computations. The current study involves hardware implementation of the semivarioram texture analysis procedure, to increase its computational speed compared to a purely software implementation.

Table 2.1 Summary of All Previous Studies

Author, year	**Significance of study**
Denis Marcotte [4], 1996	FFT approach is shown to be faster than the spatial approach for calculating the variograms.
M. Roumi [5], 2009	GLCM calculation in parallel on FPGA for four different distances in four directions.
M. Harshavardhan [6], 2014	Proposed a hardware based FPGA architecture for real time extraction of four GLCM features.
Akoushideh [7], 2012	Implemented FPGA architecture to calculate co-occurrence matrices and thirteen texture features.
A.B. Girisha [8], 2013	Proposed FPGA architecture for GLCM to increase the speed of computation.
M. Wielgosz [10], 2013	Proposed FPGA Architecture for Kriging Image Interpolation

CHAPTER THREE

TECHNICAL BACKGROUND

The main goal of variogram analysis is to construct a statistical plot that best estimates the autocorrelation structure of the underlying stochastic process. The variogram is described through several parameters namely the nugget effect, sill and range. This chapter describes the technical terms like range, sill and nugget that are used in this study to describe the characteristics of semivariogram. A description of the FPGA hardware is also provided. The techniques, which have been implemented in this study to obtain the required parameters, are also described in following sections.

3.1 Semi-variogram

The theoretical variogram is a function describing the degree of spatial dependence of a spatial random field. It is defined as the variance of the difference between field values at two locations across realizations of the field. The physical distance between the two locations is known as the lag distance or lag, and is denoted as h. The semivariogram, denoted as $\gamma(h)$, displays the average change of a property with changing lags locations in the image and is usually related to the application domain under consideration. A property such as the difference between the gray-level values of the two locations can serve as a measure of textural parameters or second-order statistics of the random field. The experimental variogram is calculated by averaging one half the differences squared of the pixel values over all pairs of observations with the specified lag distance and orientation. The relation between a pair of pixels that are lag distance h apart can be given by the average variance of the difference between all such pairs and is expressed as follows:

$$\gamma(h) = \frac{1}{2m}\sum_{i=1}^{m}[z(x_i) - z(x_i + h)]^2 \qquad (1)$$

where *m* is the total number of pixel pairs used in the computation of the sum. The value of *m* is governed by the geometrical properties of the region of interest in the image used for the statistical analysis and the value of *h*. Given the same region, larger values will result in lower values of *m* since the dipole connecting the pixels in the pair may not lie within the region of interest for certain pixels.

There are two types of semivariograms: isotropic and anisotropic. The semivariogram value that depends only on the magnitude of the lag vector, not the direction, and the empirical semivariogram can be computed by accumulating data pairs separated by the appropriate distances, regardless of direction. Such a semivariogram is described as omnidirectional or an isotropic semivariogram. The semivariogram where the property shows different autocorrelation structures in different directions is anisotropic semivariogram.

There are a few constraints for a semivariogram model to be able to represent. These include:

(a) a monotonic increase with increasing lag distance from the ordinate

(b) an asymptotic maximum, or 'sill'

(c) a positive intercept on the ordinate, or 'nugget'

(d) periodic fluctuation, or a 'hole' and anisotropy.

The stochastic parameters range, sill and nugget variance are described below.

3.1.1 Range (L)

The distance where the model first flattens out is known as the range or correlation length. Sample locations separated by distances closer than the range are spatially auto-correlated. It also describes the degree of smoothness or roughness in an image. A relatively large correlation length implies a smooth variation, whereas a small correlation length corresponded to rapid variations over the spatial domain.

3.1.2 Sill Variance (C)

The value that the semivariogram model attains at the range (the value on the y-axis) is called the sill. Usually the variogram value levels off at this semivariance value. It can be used to refer to the "amplitude" of certain component of the semi-variogram.

3.1.2 Nugget Variance (C_0)

According to theory, the semi-variogram value at the zero separation distance (lag = 0) should be zero. However, at an infinitesimally small separation distance, the semivariogram often exhibits a nugget effect, which is some value greater than 0. For example, if the semivariogram model intercepts the y-axis at 2, then the nugget is 2. The nugget represents variability at distances smaller than the typical sample spacing, including measurement error. Variation at micro scales smaller than the sampling distances will appear as part of the nugget effect.

Table 3.1 Stochastic Parameters of Experimental Variogram

Parameter	Meaning
Range	Lag distance at which model flattens out
Sill variance	The value attained at the range
Nugget variance	Variability at distances and measurement errors

The following diagram shows the plot of experimental semivariogram with the stochastic parameters marked.

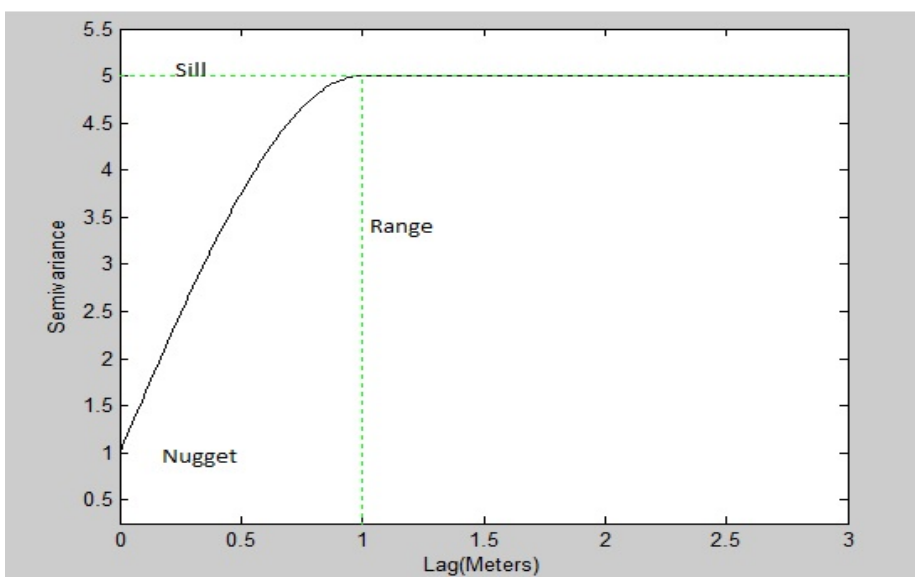


Figure 3.1 Characteristics of Semivariogram

In order to understand the computation of the semivariogram for an image, it is helpful to look at an example with some image data. Table 3.2 shows the pixel arrangement in a one dimensional array, where x_i denotes the position of the pixel values $z(x_i)$. The difference of the pixel values is calculated in the third row and the obtained values are squared in the fourth row for a lag distance $\mathrm{h} = 1$. It should be noted that a two-dimensional image can be arranged as a vector for computation of the isotropic variogram as long as the position information is retained for each location.

Table 3.2 Calculation of Difference of Pixels in the 1-D Array to Compute Variogram for Lag Distance h=1

x_i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$z(x_i)$	28	25	27	25	21	25	11	16	11	17	27	26	25	26	21
Pixel Difference	3	-2	2	4	-4	14	-5	5	-6	-10	1	1	-1	5	
Square of pixel difference	9	4	4	16	16	196	25	25	36	100	1	1	1	25	

The following steps have to be followed to calculate the isotropic semivariogram value,:

1. Calculate the Euclidean distance and sort the pixel pairs according to the lag distances.
2. Calculate square of the difference between all pixel values with distance *h* as computed in Table 3.2.
3. Determine the number of pixel pairs *m*, for the example shown in Table 3.2 the number of pairs of pixels with distance 1 is *m*=14.
4. Add all the square of differences for a particular lag distance *h*. Summing all the squares of differences computed in Table 3.2 we get, $\sum_{i=1}^{m}[z(x_i) - z(x_i + h)^2]$ =459
5. Compute the final semivariogram value using the Equation 1.

For the given example variogram value can be computed to $\gamma(h = 1) = 459/(2*14) = 32.8$. Similarly, the values for different lag distances can be computed to be:

$\gamma(h = 2) = 44.3$; $\gamma(h = 3) = 76.5$; $\gamma(h = 4) = 110.8$; $\gamma(h = 5)$=108.9; $\gamma(h = 6) = 124.8$; $\gamma(h = 7) = 89.9$

As can be noted from the example, the semivariogram values tend to rise rapidly and then converge to a value known as the sill.

3.2 Different Types of Semivariogram Models

The semivariogram plot can be fitted with various models described by exponential or hole-effect parametric equations to obtain a description of the texture in the region of interest [17]. The appropriate model is chosen by matching the shape of the curve of the experimental variogram to the shape of the curve of the mathematical function. The function must therefore be mathematically defined for all real lag distances (h).

Geostatistical Analyst provides the following functions to model the empirical semivariogram [17]:

- Circular
- Spherical
- Tetraspherical
- Pentaspherical
- Exponential
- Gaussian
- Rational Quadratic
- Hole Effect
- K-Bessel
- J-Bessel
- Stable

The selected model influences the prediction of the unknown values, particularly when the shape of the curve near the origin differs significantly. The steeper the curve near the origin, the more influence the closest neighbors will have on the prediction. As a result, the output surface will be less smooth. Each model is designed to fit different types of phenomena more accurately.

The nugget model represents the discontinuity at the origin due to small-scale variation. The spherical model actually reaches the specified sill value, at the specified range. The exponential and Gaussian approach the sill asymptotically; practical range is the distance at which the semivariance reaches 95% of the sill value. These three models are shown below:

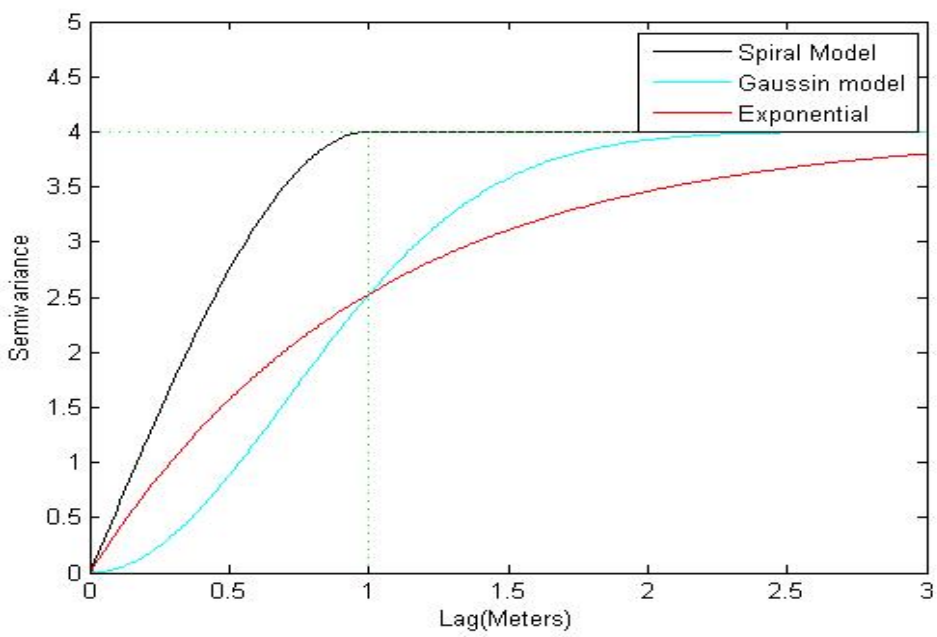


Figure 3.2 Types of Semivariogram Models

The Gaussian model, with its parabolic behavior at the origin, represents very smoothly varying properties. The spherical and exponential models exhibit linear behavior at the origin, appropriate for representing properties with a higher level of short-range variability.

It is a good idea to study the computational complexity of the semivariogram algorithm for the isotropic computation case. For $P = M \times N$ pixels in the image there are $P(P -$

$1)/2$ unique pixel pairs. Calculating the variogram value includes the computation of the distance and difference between all the pixel pairs, and summation of squares of differences. The distance calculation has $3 \times (P(P-1)/2)$ additions and multiplications each, whereas the difference calculation between the pixel pairs and the square of the differences has $P(P-1)$ additions. Final variogram calculations have $P(P-1)/2$ additions and nearly $(P-1)/4$ multiplications. Hence, the variogram value can be computed with $3 \times P(P-1)$ sums and nearly $3 \times (P(P-1)/2)$ multiplications. The computational complexity of the variogram calculation can be shown as $O(n^2)$ where n is the number of pixels in the image region [10].

Table 3.3 Number of Computations for Variogram Calculation for $P = M \times N$ Image

Parameters	Number of Computations for $P = M \times N$ image
Unique pixel pairs	$P(P-1)/2$
Number of additions for the calculation between the pixel values.	$3 \times (P(P-1)/2)$
Multiplications for lag distance calculation between the pixel values.	$3 \times (P(P-1)/2)$
Additions for calculating the difference between the pixel values.	$P(P-1)$
Additions for calculating variogram value using the equation (1)	$P(P-1)/2$
Multiplications for calculating variogram value using the equation (1)	$(P-1)/4$
Total number of additions required for variogram calculation.	$3 \times P(P-1)$
Total number of multiplications required for variogram calculation.	$3 \times (P(P-1)/2)$

For example, for $P = M \times N$ $(M = 10, N = 10)$, the total number of pixel pairs is 4950 assuming distinct pixels, and the total number of summations would be 29,700 and the total number of multiplications would be 14,850. Similarly for $P = M \times N$ $(M = 20, N = 20)$, the total number of pixel pairs is 79800 and the total number of summations 478800 and multiplications are 239400. The value is particularly large even for a small images of 10×10 and 20×20 pixels.

As seen by the computational breakdown, a large part of the burden exists due to the comprehensive analysis of pixel pair data. Practical applications in the medical domain involve regions of interest with around 50×50 pixels, resulting in an exponentially larger number of computations: approximately 187 million sums and 95 million multiplications. This large number of computations makes the semivariogram calculation extremely time intensive. It is worthwhile investigating methods to speed up the computation in order to make the semivariogram a practical technique for clinical usage. For this reason, FPGA architectures have been proposed to reduce the computational time for calculating the variogram values.

3.3 Field Programmable Gate Arrays:

A Field-Programmable Gate Array (FPGA) is a reconfigurable device with the ability of being reprogrammed to satisfy particular requirements. It consists of basic elements combined to form highly optimized and high performance systems. The two major manufacturers are Xilinx and Altera, though their FPGAs have strong differences at the architectural level, their basic operation and functionality is the same. Input/Output Blocks (IOBs) usually serve as a way to interconnect the silicon package pins carrying external signals to the internal Configurable Logic Blocks (CLBs). Another crucial element Switch Matrix (SM) interconnects all elements inside the FPGA. The SM and CLBs are programmable by the user and their configuration is stored in individual Static Random Access Memory (SRAM) cells inside each element. The CLBs allow for combinatorial or sequential logic to be generated inside the FPGA. A single CLB is typically comprised of slices, each of them including several Look Up Tables (LUTs), carry logic and storage units. An LUT unit can have multiple inputs; the most common is

a four input LUT. However, five and six input LUTs can also be found. With an N-input LUT, any Boolean logic function with N-inputs can be implemented.

In order to program an FPGA with a desired logic design one has to follow three steps which are escribed below:

1. The design is first created using a hardware description language such as Verilog or VHSIC Hardware Description Language (VHDL).
2. This is then utilized in generating a low-level device specific bit stream file.
3. The final step is to download the bit stream onto the configuration memory of the FPGA, which individually defines the behavior of the CLBs and corresponding interconnects needed to implement the design.

FPGAs have gained popularity in recent years because the present line of commercial FPGAs are capable of integrating powerful embedded processors and several common intellectual property cores that provide a complete system-on-chip solution. FPGAs that contain dedicated multiplier blocks are particularly suited as co-processors for computation-intensive applications such as digital signal processing.

A good example is the Virtex 5 that is the latest in the Virtex series of FPGAs and is used for the current experiment. The device shown in Figure 3.3 is built on a 65 nm process and includes 288 dedicated DSP blocks, each block consisting of a 25 bit x 18 bit multiplier and a 48 bit adder/subtractor/accumulator. Other features include 4 embedded Ethernet MAC blocks capable of implementing 1000 Base-X (Gigabit Ethernet) implementation. Figure shows the FPGA Kit used in this project is the Xilinx XUPV5-LX110T Development Kit, which utilizes the Virtex5 FPGA.

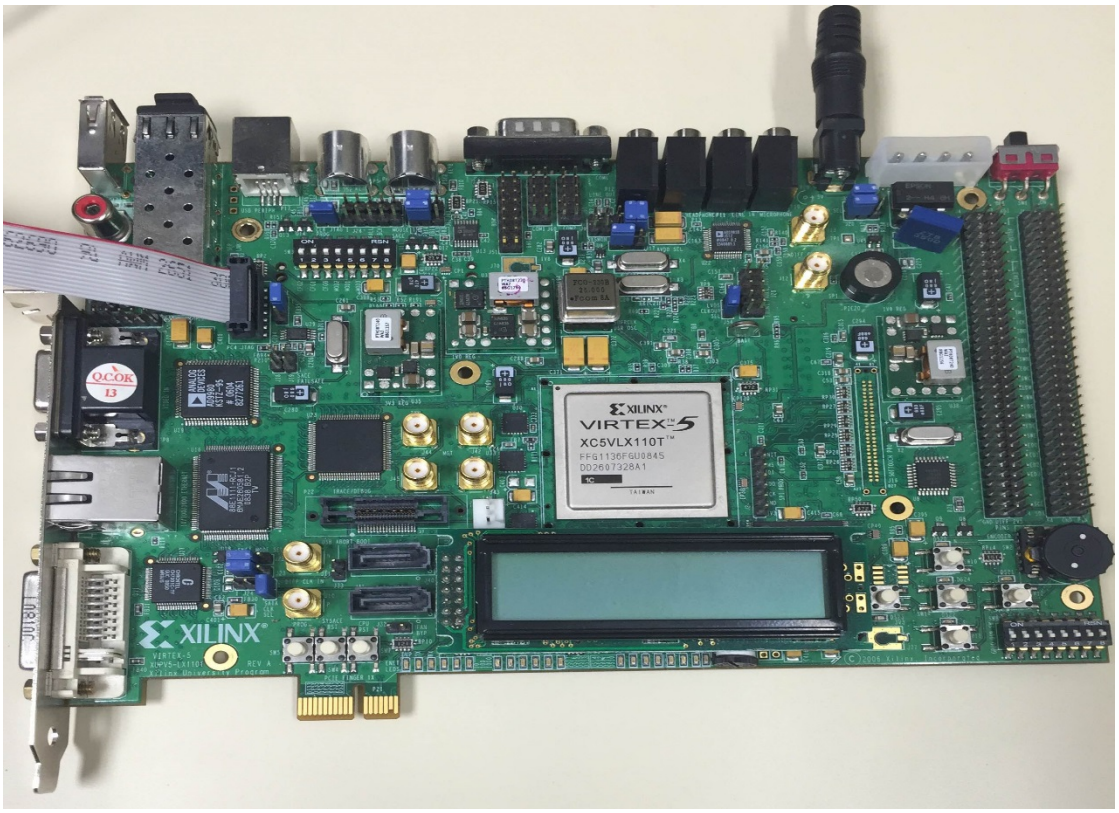


Figure 3.3 Xilinx XUPV5-LX110T Development Kit with Virtex 5 FPGA

CHAPTER FOUR

METHODS AND EXPERIMENTAL PROCEDURES

In this study, two architectures were designed and implemented in succession on the FPGA. Architecture version 1 (non-pipelined) was initially implemented and tested. Architecture version 2 (pipelined) was subsequently introduced to improve upon the performance of the former. Both architectures were programmed in VHDL, and consist of four different modules with each module performing different tasks as described below:

1. Distance module: Calculates the lag distance between pixels.
2. Difference module: Calculates the difference between pixel values.
3. Sorting module: Sorts the pixel values based on distance and maintains a count of the number of pixel pairs.
4. Semivariogram module: Calculates the final semivariogram value.

In addition to these two architectures, a third architecture was proposed, but Architecture Version 3 (Pipelined and Hardware Reutilization) could not be fully incorporated because of the huge complexity in the circuit and hardware limitations. Because for the complex bigger circuits in FPGAs, the components and the wires connecting them are limited moreover time-consuming also matters.

Two-dimensional medical image data of specimen from MRI scans are utilized for the experiments. This image is obtained using Dual energy X-ray absorptiometry operating in a fan beam mode. Figure 4(a) is used as the input which has been cropped from the original scanned image of hip as shown in Figure 4(b).

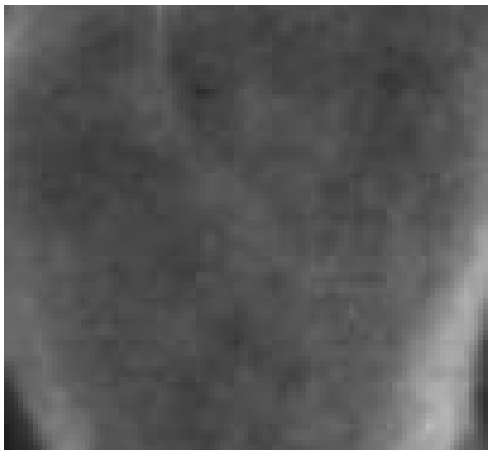


Figure 4(a) Cropped Image of the Hip Used for Experiments

Figure 4(b) Original Scanned Image of the Hip Used for Experiments

The following sections describe the algorithm implementation and digital logic design specification.

4.1 Architecture Version 1 (Not Pipelined):

The block diagram and description of the basic architecture is shown in the Figure 4.1(b). Initially the image pixel values are loaded into the image buffer. Then the counters for the two image buffers start incrementing and the image data is loaded into the difference

module. The difference module computes the absolute difference between two pixel values. The corresponding counter values are loaded into the distance module concurrently. The distance module calculates the Euclidean distance between the two corresponding pixel values. The distance and difference data is then accumulated in the sorting module which sorts the differences indexed by the lag distances and calculates the number of pixel pairs for each distance. These sorted values are passed to the final module which computes the semivariogram value for different lag distances. The final module is the output multiplexer that displays the output values one at a time. Figure 4.1(a) shows the timing diagram for the architecture version 1.

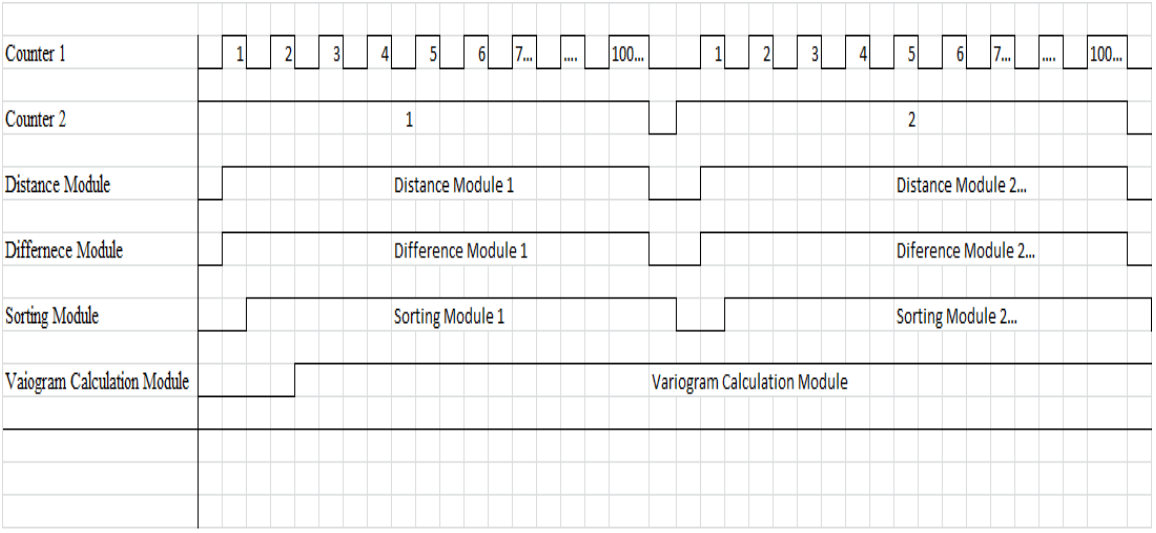


Figure 4.1(a) Timing Diagram of Architecture Version 1 (Non Pipelined)

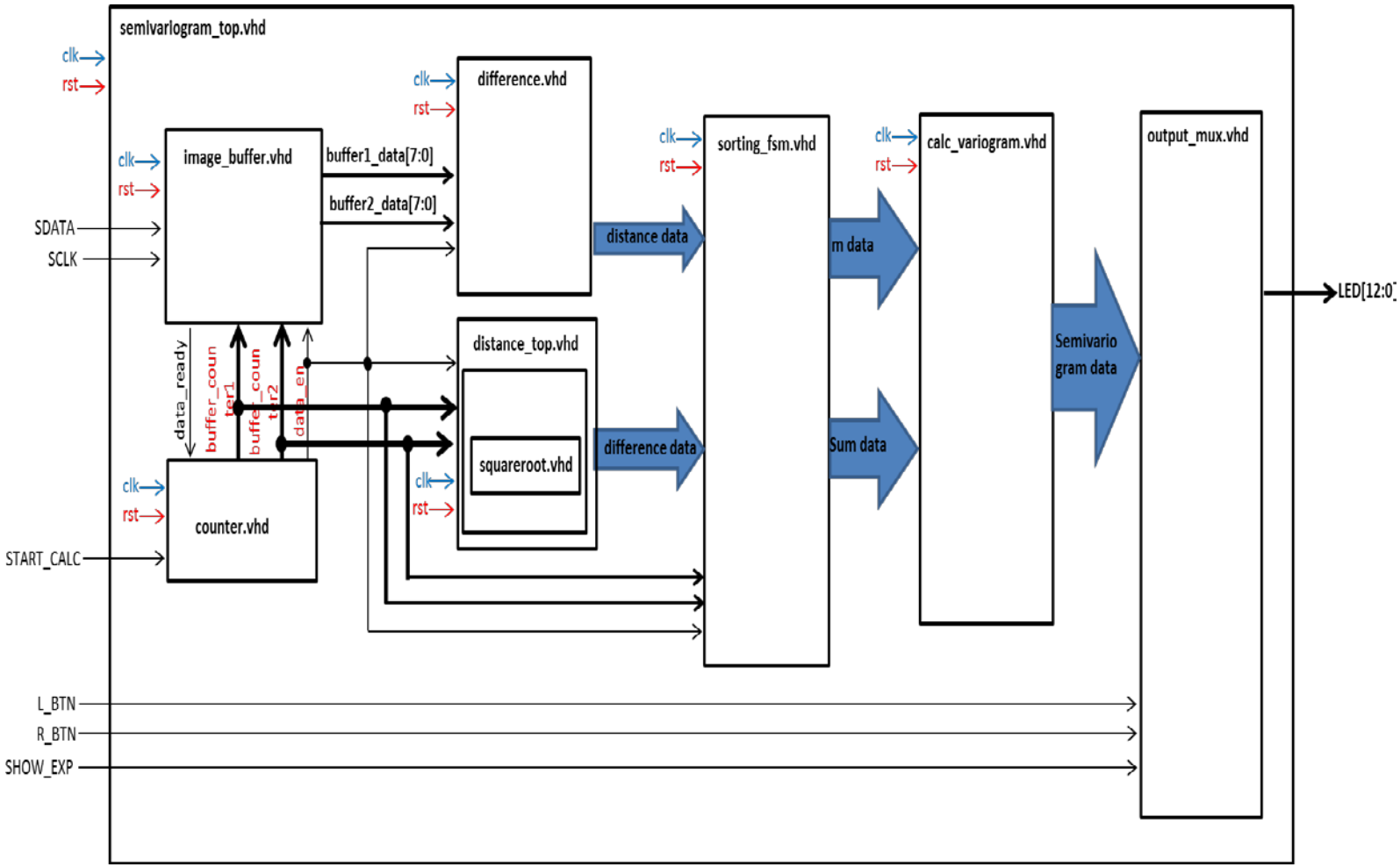


Figure 4.1(b) Block Diagram of Architecture Version 1 (Non Pipelined)

4.2 Architecture Version 2 (Pipelined):

Figure 4.2(b) shows the block diagram for the architecture version 2 which is pipelined to improve performance. This architecture has the distance and difference modules replicated multiple times (99 times for 100 pixel values and 399 times for the 400 pixels) for the tested implementation. Each of the replicated distance sub-modules computes the Euclidean distances between pixels. The difference sub-modules calculate differences for a fixed pixel with all the pixel values pair with. For example, the first distance and the difference modules calculate distances and differences between pixel 1 and pixels 1 to N.

The sorting module has the same functionality as the sorting module in the first architecture version, with the exception of that it has 33 sub modules in 10×10 image version and 25 sorting sub modules for the 20×20 image versions that are pipelined. Each sub-module in the 10×10 image handles 3 distance or difference values, whereas 16 distance and difference values are handled in each of the 25 sorting modules in the 20×20 image. The output number of pixel pairs (m) and sum of the sub-modules are then added to find the number of pixel values and sum values for each distance, which is then passed to variogram module for final calculations. Figure 4.2(a) shows the timing diagram for the architecture version 2.

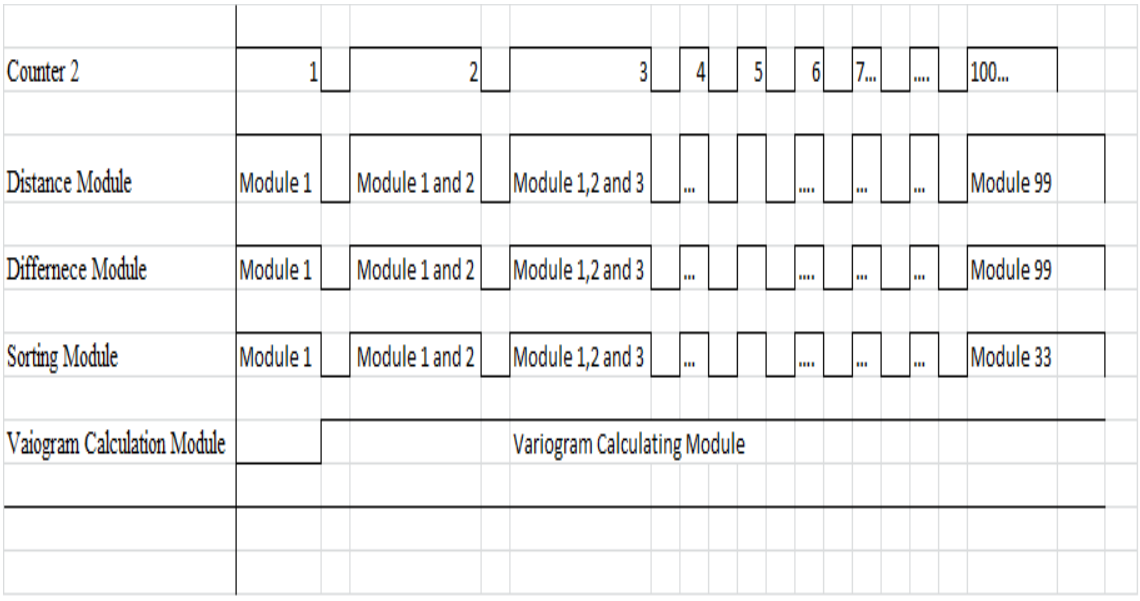


Figure 4.2(a) Timing Diagram of Architecture Version 2 (Pipelined)

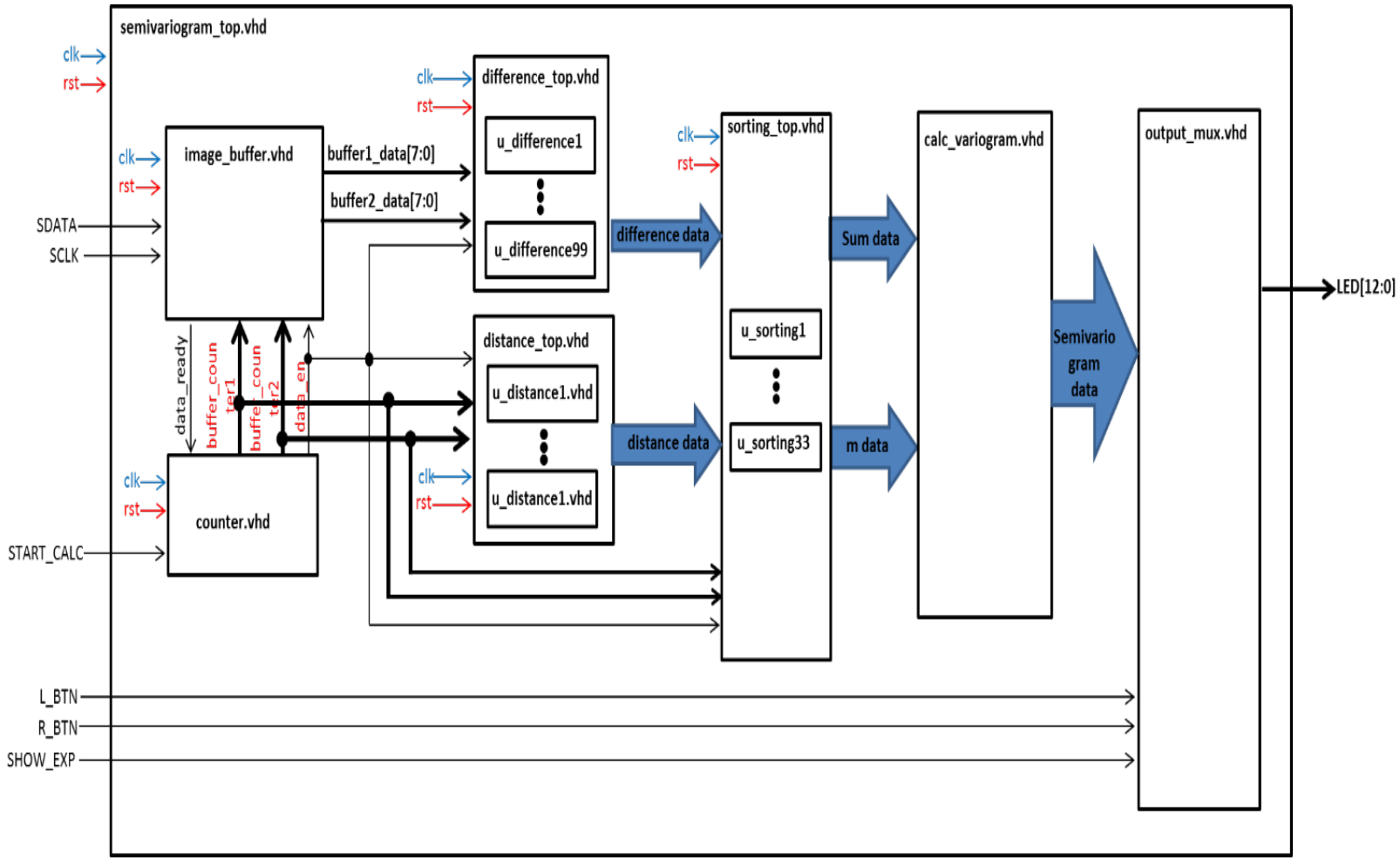


Figure 4.2(b) Block Diagram of Architecture Version 2 (Pipelined)

4.3 Architecture Version 3 (Pipelined and Hardware Reutilization):

The use of an FPGA provides a substantial amount of freedom for design. Yet, the hardware designer must be much more aware of availability of resources and of limitations than the software developer. In this thesis a new architecture for 25×25 and 10×10 image sizes was attempted for hardware reutilization. This architecture was actualized using the loops for 100 distance and difference sub-modules. The architecture is so designed that it erases the pixel data from the difference and distance sub-modules once they have been utilized, and the new values are loaded into the counter and image buffer. This method of implementation not only reduces the complexity of the architecture by reducing the number of sub-modules but can also be used for bigger image sizes. Due to the timing and memory size limitations inside the FPGA, the circuit could not be synthesized and implemented on the FPGA. Timing is a concern that usually does not occur in software, but pops up unavoidably in circuit design. A detailed description of each block is provided in the following sections.

4.4 Image Buffer and Counter Module

4.4.1 Architecture Version 1

Image data loading is one of the challenging tasks for any architecture with comprehensive pixel-based computations. One of the major bottlenecks is that data has to be presented to the modules in pairs. Popular approaches like the crossbar switch and ring buffers can be used to load the data into the processing modules. The crossbar switch has a matrix with $M \times N$ cross-points or places where the "bars" cross [11]. At each cross point is a switch, which when closed, connects one of M inputs to one of N outputs. It has the disadvantage of large implementation size due to the number of connections. A ring buffer is a data structure that uses a single and fixed buffer connected end-to-end [12]. Ring buffers can be used to present the serialized data in a tight sequence to the modules. The linear ring buffer faces a disadvantage due to large memory size and access time. Thus the architecture has been implemented with two single array image buffers to load the data into all the sub-modules. Image data is preloaded into the buffers to allow for testing access by an algorithm on hardware.

Image pixel data are first loaded into the two image buffers as data vectors. If the image is of size N × N then the pixel vector is of size N^2 (for the experiments N = 10 and 20 have been used). The counters are incremented when the data enable is high, similar to nested "for loops" in order to load the pixel data pairs into the distance and difference modules. Counter 1 starts from 1 and increments every time the second counter counts up to the last pixel value N starting from 1. Registers are used to store the pixel values and the two multiplexers present these values to the sub-modules in sequence. The first multiplexer presents pixel data to output buffer 1 and the second one multiplexes pixel data registers into output buffer 2, based on the current values of input buffer counter 1 and input buffer counter 2.

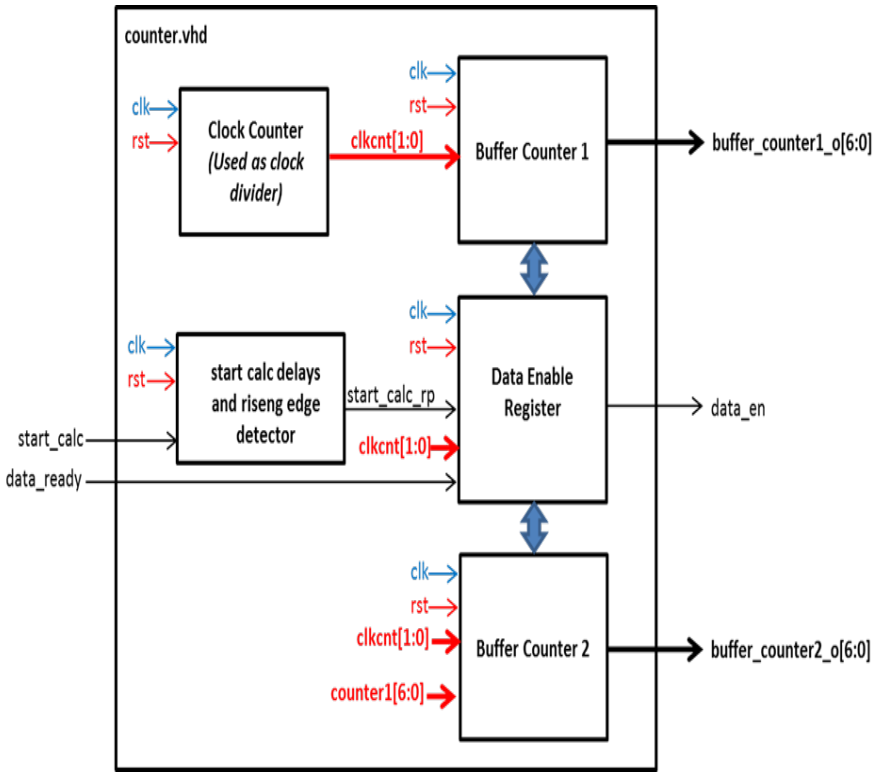


Figure 4.4.1 (a) Block Diagram of Counter Module (Not Pipelined)

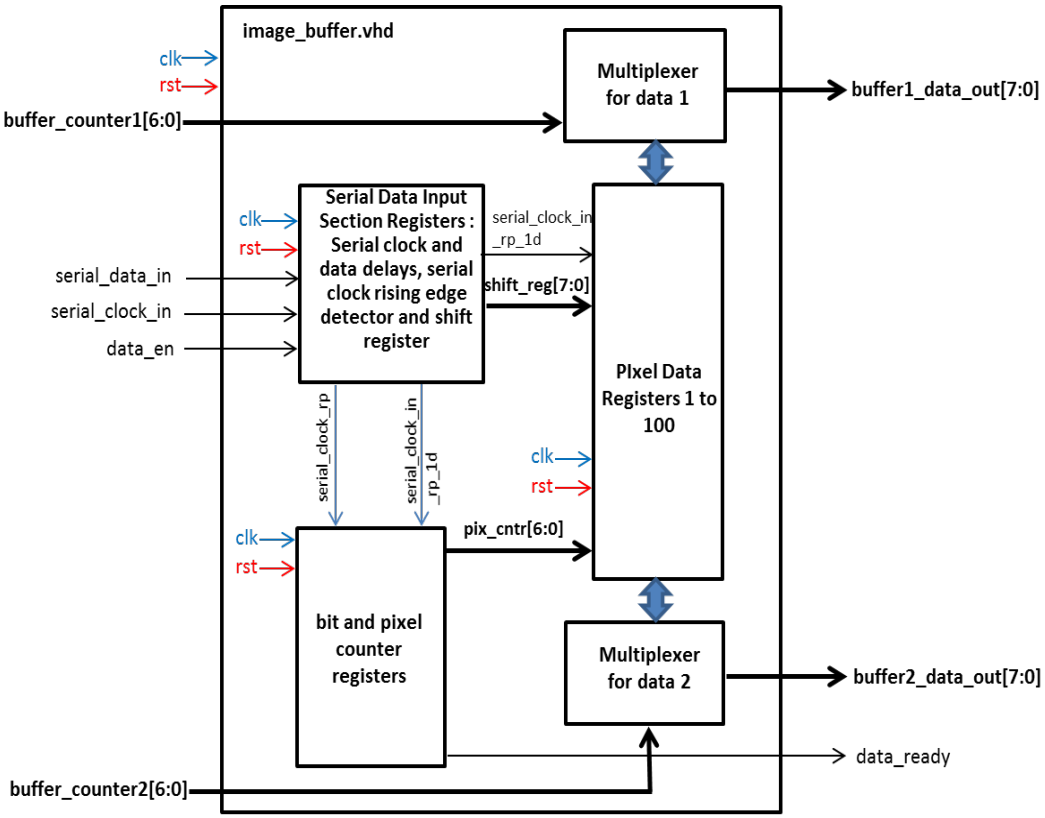


Figure 4.4.1 (b) Block Diagram of Image Buffer Module (Not Pipelined)

4.4.2 Architecture Version 2

Since the pixel data loading process can be time-consuming, the second architecture addresses this process in a brute force fashion. The counter and the image buffer for architecture version 2 (pipelined) have just one multiplexer which presents the data from second counter and image buffer 2 to all the distance and difference sub-modules. The counter 1 data and the buffer 1 pixel data are replicated directly in all the sub-modules whenever the counter increments. The replication of the data takes place simultaneously whenever the image buffer 2 and the counter 2 data are loaded in the corresponding sub-modules. For example, in the first difference sub-module replication of image buffer values takes place during loading of the pixel 1 value from the second image buffer. Figure 4.3.2 (a), 4.3.2 (b) shows the counter and image buffer for the architecture version 2 (pipelined).

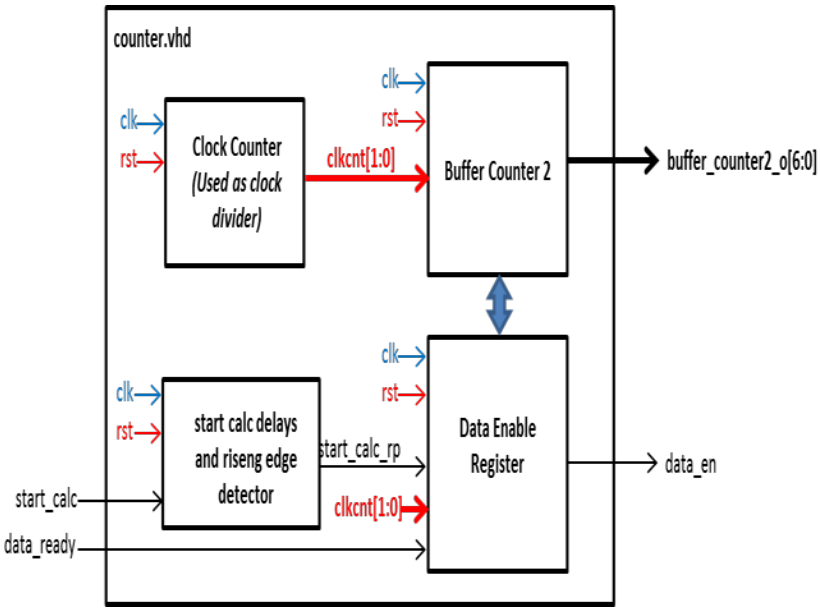


Figure 4.4.2 (a) Block Diagram of Counter Module (Pipelined)

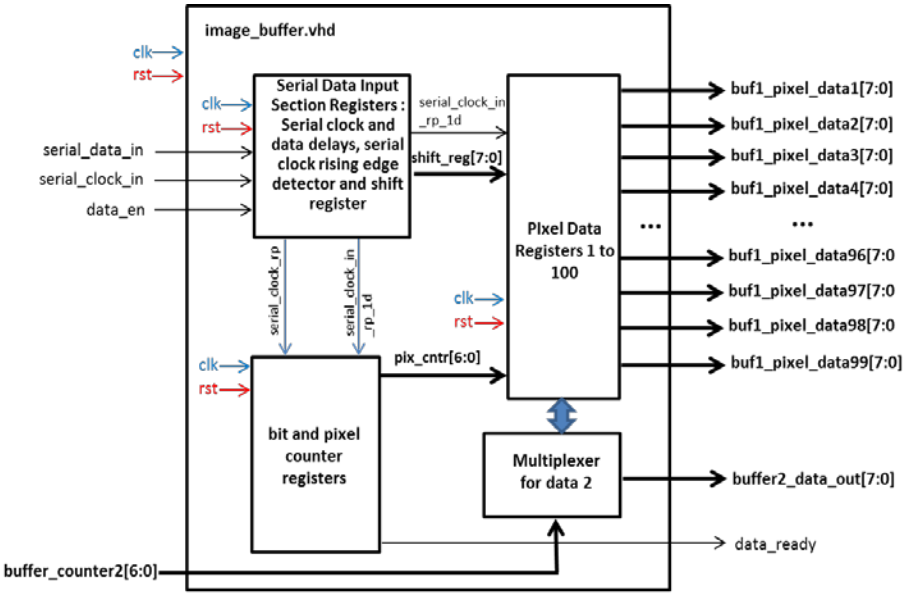


Figure 4.4.2 (b) Block Diagram of Image Buffer Module (Pipelined)

4.5 Distance Module

4.5.1 Architecture Version 1

The distance module calculates the Euclidian distance between two pixels in a pixel pair. The value from counter 2 is stored in the internal register of the distance module and the distance is calculated between this value and the remaining values loaded by the first counter. When the second counter increments the first counter is again loaded in the distance module. Depending on the counter values, the corresponding *x* and *y* position for the calculation of the distance are determined in the position 1 and position 2 decoder modules. The counter values are taken as the *y*-coordinates for the pixel values, and *x*-coordinate is taken as 1 for the first *N* values. Essentially for the next *N* values, the *x*-coordinate is taken as 2 and *y* coordinate as 1 to *N*. The coordinate values are stored in internal registers for data stability. These values are fetched and the values are multiplied and then added to get the square of the distance.

The final step is to calculate the square root of the obtained value using the square root sub-module. The designed sub-module selects square root of the input value depending on the logic designed in the module whenever it detects the input signal. The overall circuit inputs an 8-bit integer and outputs a 4-bit integer square root. This circuit uses the "entity" method. The output of the square root module is rounded up or down. This rounded output value is given as the input to the sorting module.

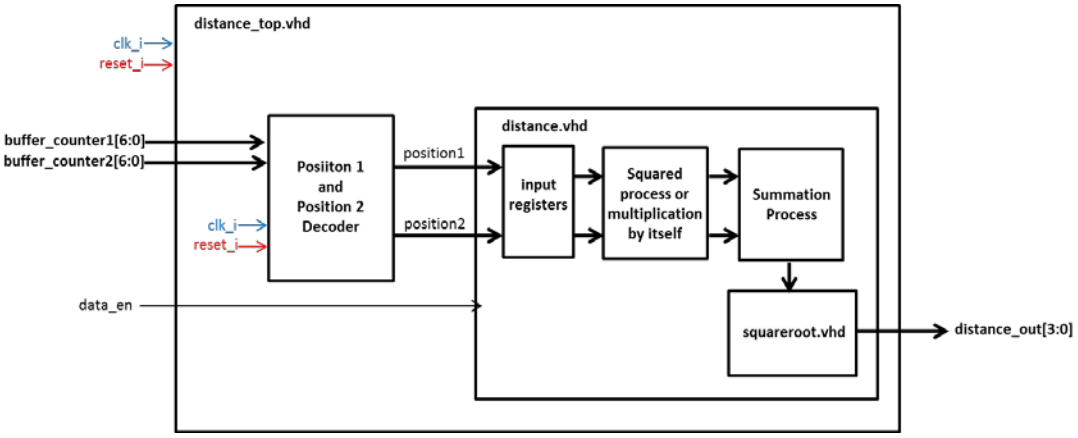


Figure 4.5.1 Block Diagram of Distance Module (Not Pipelined)

4.5.2 Architecture Version 2

In the architecture version 2, the distance module is replicated 99 times for 10×10 image and 399 times for the 20×20 image, each module with the identical function of parallel calculation of distance data. Each value of the counter 2 is loaded into its particular modules whereas the first counter values are directly replicated in all the modules. When the counter 2 increments, the first distance module calculates the distance between the first pixel value and all the pixels till *N*, where *N* is the last pixel value. In the meantime, module distance 2 starts calculating the distance for pixel 2 to pixel *N* once the module gets the input from the counter, and so on. Hence all the sub-modules calculate the distance between the two pixels in the parallel fashion by performing calculations at the same time in the corresponding modules. As soon as the modules complete the calculations they send the distance values to the sorting module without waiting for all the calculations to be completed.

During the distance calculation, the module is designed such that once the two buffer counter values are loaded in the distance module, the values are compared with the previous position values and the pixel values with repetitive position values are skipped to reduce computational time.

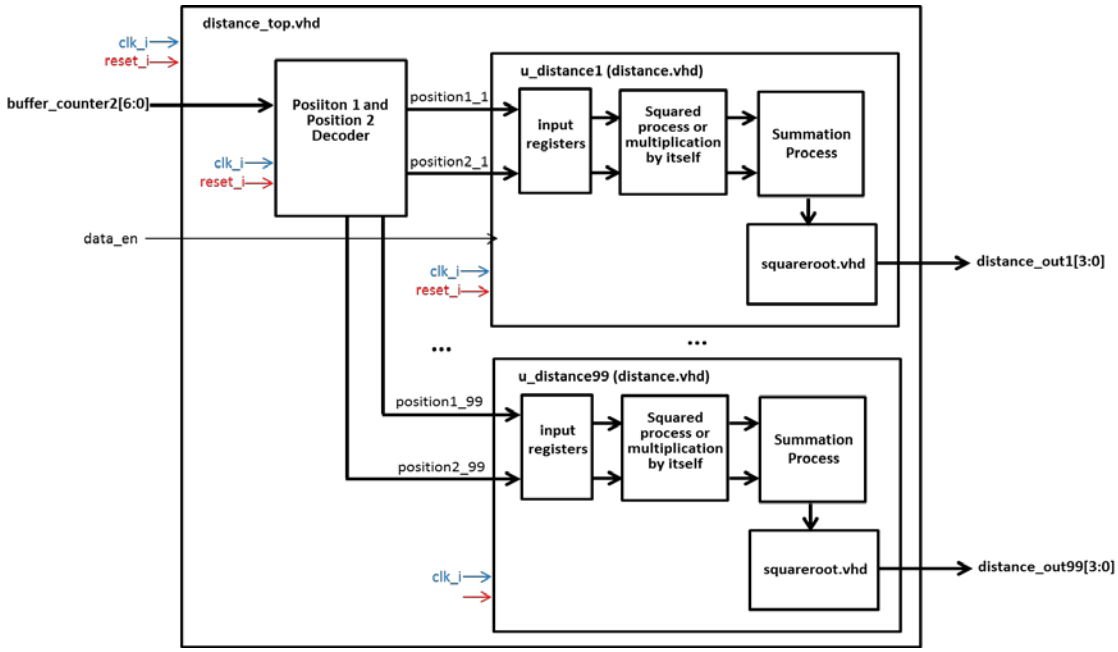


Figure 4.5.2 Block Diagram of Distance Module (Pipelined)

4.6 Difference Module

4.6.1 Architecture Version 1

The difference module handles computation of the absolute difference between two pixel gray levels. The pixel value from image buffer 2 is stored in the pixel 1 register in the difference module when the enable signal is high. The difference is calculated between this value and the remaining pixel values loaded by the first image buffer into the pixel 2 register. Image buffer 2 increments to the next value only when all the values of the second image buffer have been used for the difference calculation with the previous pixel value of the second image buffer. Whenever the image buffer 2 increments, the first image buffer starts loading all the values into the register. Once the two internal registers are stacked with the image pixel values, the module then checks if pixel 2 is higher than pixel 1. If it is higher, pixel 1 is subtracted from pixel 2, while if it is lower, pixel 2 is subtracted from pixel 1.

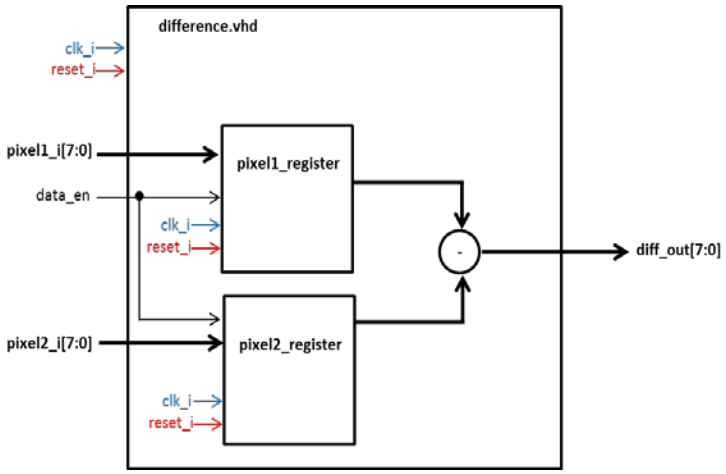


Figure 4.6.1 Block Diagram of Difference Module (Not Pipelined)

4.6.2 Architecture Version 2

In the architecture version 2, 99 modules for 10×10 image and 399 modules for the 20×20 image are replicated with the same function for the parallel calculation of differences between two pixel values. Each pixel value of the image buffer 2 is loaded into its particular module whereas the first image buffer is directly replicated in all the modules. When the image buffer 2 increments, the first module calculates the difference

between the first pixel value and all the pixels up to *N*, where *N* is the last pixel value. At the same time, module 2 calculates differences for pixel 2 to pixel *N* once it gets the second pixel value from the image buffer 2, and so on. All the modules hence start calculating in a parallel fashion, that is though the previous modules are calculating the difference between the pixel pairs, the image buffer 2 values are loaded and the corresponding submodules starts calculating the differences. As soon as the single pixel difference value has been computed in the sub-modules, it is given as the input to the sorting module without waiting for all the calculations to be completed.

During the difference calculation the module is so designed that the corresponding module starts calculating the differences between the fixed pixel value of image buffer 2, and the values after the fixed pixel value in the image buffer 1. This design skips the repetitive calculations between the pixel pairs to reduce the computational time. For example in the inference module 3 the fixed pixel value from the image buffer 2 is 3 and as discussed the image buffer 1 is replicated in this module. In this module the next value to the fixed pixel value in the replicated buffer is the fourth pixel value. Hence the module calculates the deference between 3 and all the pixel values from the fourth value to avoid the repetitive calculations. Figure 4.6.2 (b) shows the block diagram for the skipping algorithm.

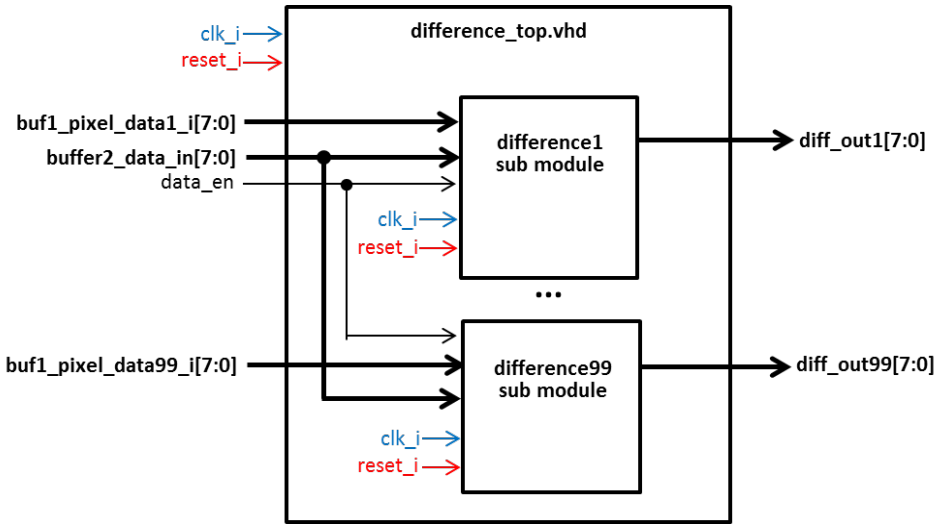


Figure 4.6.2(a) Block Diagram of Difference Module (Pipelined)

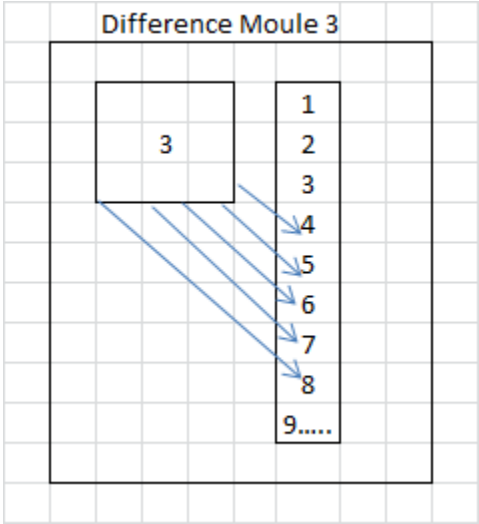


Figure 4.6.2(b) Block Diagram for Skipping Algorithm

4.7 Sorting Module

4.7.1 Architecture Version 1

After calculating the lag distances and the differences between the pixel pairs, the outputs from both the modules are connected to the sorting module. The sorting module sorts the differences by keying off the lag distances and computes the summation of the square of differences for each lag distance. The range starts from 1 to the maximum distance between the pixel pairs. It also calculates the number of pixel pairs for each distance. The block diagram below shows that the first part of the module is the change data detection circuit which actually detects the increment in counter 2. This data detection circuit is used to activate the corresponding sorting modules by the input signal which acts as an enable signal to the registers, initially set to 0. The increment of the counter indicates that the distance and the difference sub-modules have started the calculations and will be inputted to the corresponding sorting module.

The register which acts as the accumulator, checks the input distance value whenever the first counter value changes, if the distance is 1, then the first register $m1$ is incremented, if it is 2, second register $m2$ is incremented and so on. The registers calculating the sum adds the square of the input difference value to the current register value using the distance values as an offset. If the lag distance is 1 the square of the input difference gets added to the accumulator 1, if it is 2 the value gets accumulated in the second

accumulator, whenever the modules detect the signal from the data detection circuit. The counter register number corresponding to the offset is incremented and the square of the input difference value is added to the corresponding sum register. These sorted values are passed to the final module which computes the semivariogram value for different lag distances.

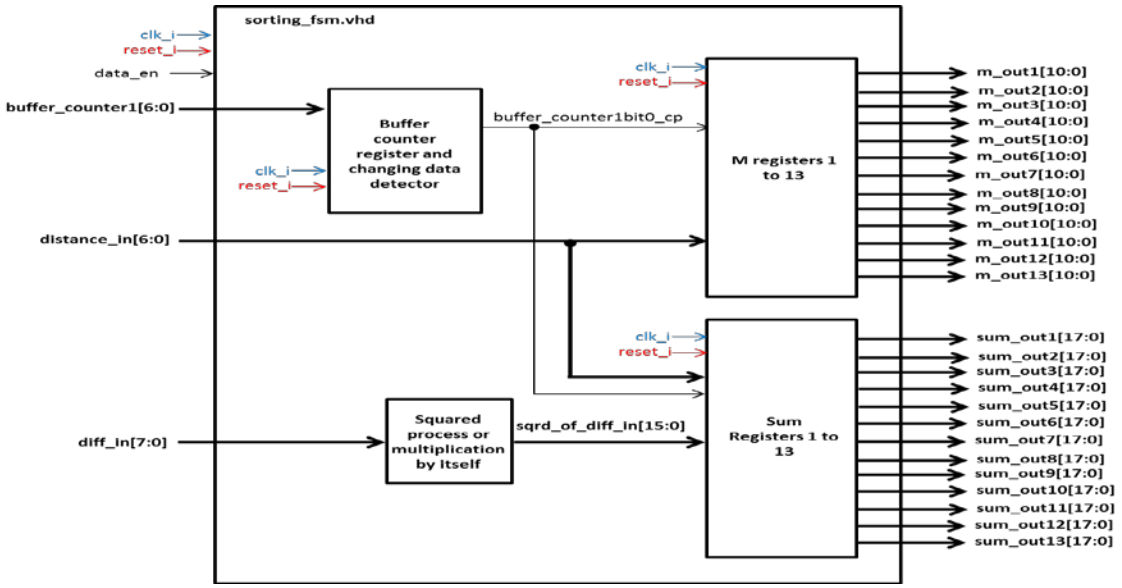


Figure 4.7.1 Block Diagram of Sorting Module (Not Pipelined)

4.7.2 Architecture Version 2

Implementation 2 performs the same sorting and summation processes to find total sum and number of pixel values for each lag distance as architecture 1. However in this implementation, 33 sub-modules (for 10×10 image) and 25 (for 20×20 image) sorting sub-modules have been replicated. Each sub-module in the smaller image handles 3 distance and difference values. The sorting module handles the output from three distance sub-modules (*distance_out1, distance_out2, distance_out3*) and output from three difference sub-modules (*diff_out1, diff_out2, diff_out3*). Similarly 25 sub-modules in the 20×20 image handles 16 difference and distance values. So the 33 sub-modules have been designed to handle the 99 sub-modules from the distance and difference modules whereas 25 sub-modules are designed to handle 400 values from distance and difference sub-modules. The output from these sub-modules is then used to find the total pixel

values and sum values for each lag distance. These number of pixel values (*m*) and summation values acts as the input to the final module for calculation

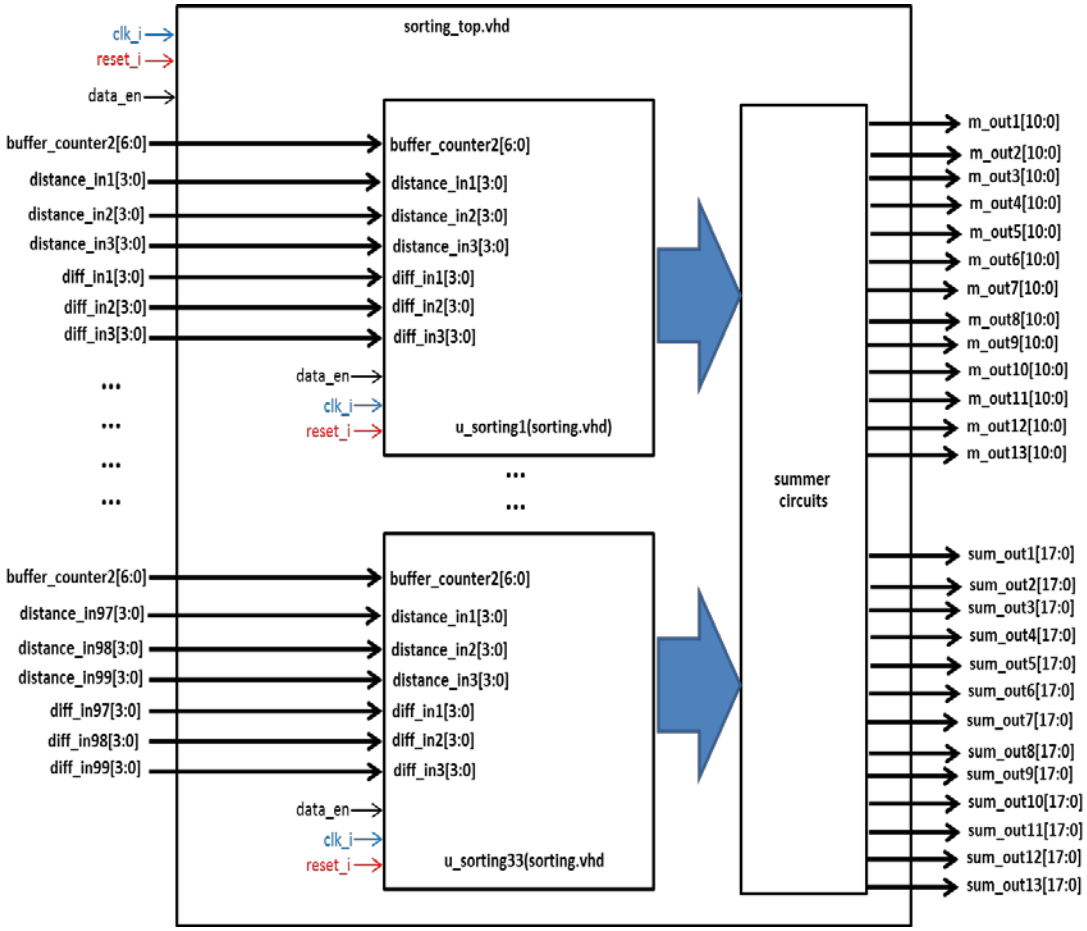


Figure 4.7.2 Block Diagram of Sorting Module (Pipelined)

4.8 Variogram Calculation Module

The final stage is the same for both the architectures. It calculates the semivariogram value for all the lag distances, using the division function. The number of pixels calculated in the sorting module and the square of the summation of difference of gray values is given as input to the variogram module. First, all the *m* data from the sorting module is multiplied by 2 by shifting bits to the left by one. The divider circuits for each input, from lag distance 1 to the maximum lag distance between pixel pairs are used to divide the sums with the number of pixel pairs for each lag distance. Each lag distance has a specified register where the corresponding semivariances are calculated. After the division process, the result is then rounded up or down to the nearest integer value. The

output multiplexer is used to display results on the LED display one at a time on the Virtex 5 FPGA board. The semivariogram values obtained are in the binary format.

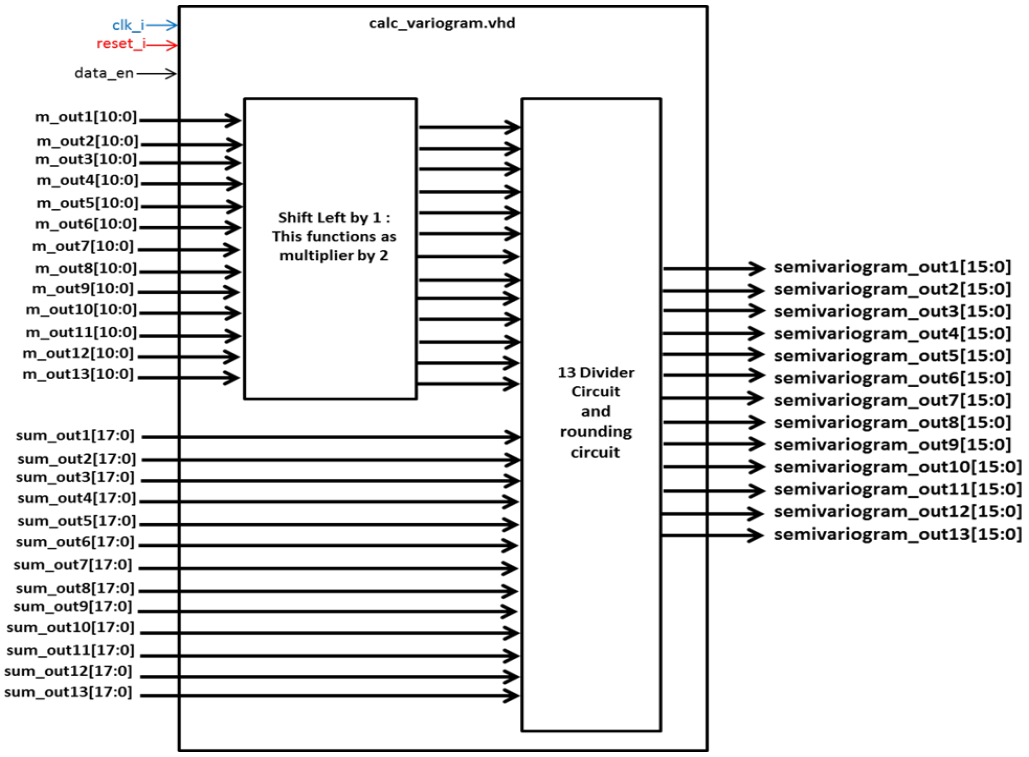


Figure 4.8 Block Diagram of Variogram Module

The aim of this thesis is to investigate the use of programmable logic devices (FPGAs) to accelerate the computation of GLCM features. The target hardware for this work is Xilinx XUPV5-LX110T based FPGA development hoard equipped with a Xilinx Virtex 5 FPGA. The bitstream generated for the whole architecture was dumped onto XUPV5-LX110T development Kit device of Xilinx Virtex 5 pro family.

CHAPTER FIVE

RESULTS AND DESIGN IMPLEMENTATION

Initially a MATLAB program was used to implement the algorithm and find the baseline variogram values. The design was then implemented on the FPGA board for a 10×10 image and a 20×20 image, implementation results of each modules of the designed two architecture are discussed in this chapter.

5.1 Image Buffer Implementation Results

The figures below are the simulation results showing internal signals of the image buffer module. For architecture 1 it is observed that the output data of image buffer 1 is updated in the *buffer1_data_out* signal for every change in counter 1. Similarly output data 2 is updated in the *buffer2_data_out* signal for each increment of the counter 2. These signals are given as the input to the difference module for difference computation. It can be observed that the image buffer 2 increments to the next only value only when all the values of the first buffer are passed to the difference module.

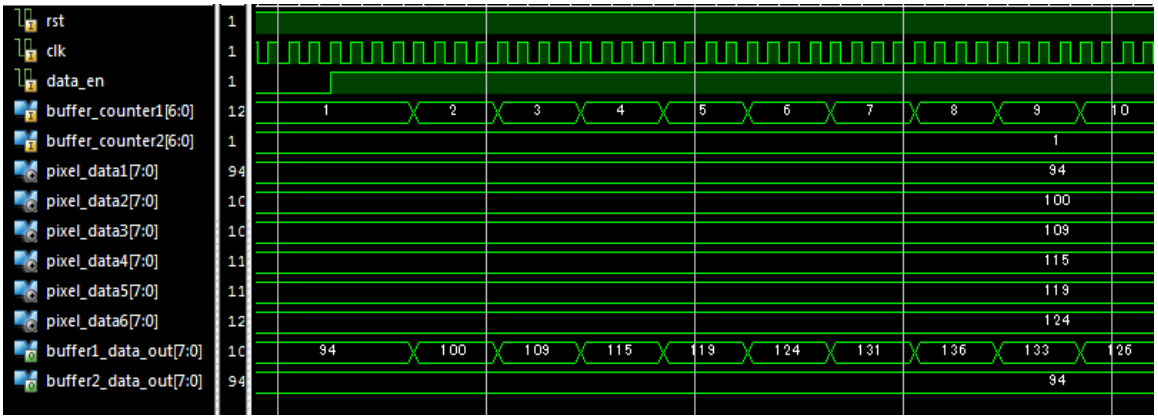


Figure 5.1 (a) Implementation Results for Image Buffer (Not Pipelined)

In architecture 2, there is only one image buffer whereas the other image buffer values are replicated in all the sub-modules for difference computation. The Figure 5.1 (b) and Figure 5.1 (c) shows the implementation results of the image buffer for the second

architecture for two different images. Implementation results show only one output signal *buffer2_data_out* which corresponds to image buffer 2. This data is loaded in the corresponding submodules for the difference calculations whenever the second counter (*buffer_counter2*) is incremented. Due to the replication of the image buffer 1 data in the submodules architecture version 2 is 100 times faster than the architecture version 1 for a 10 × 10 image.

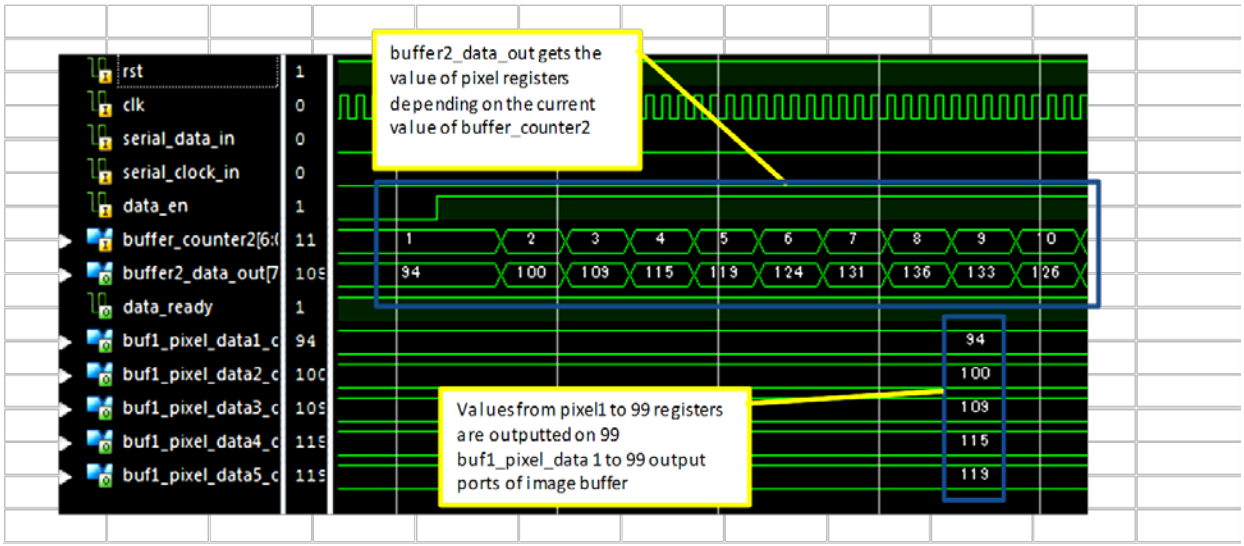


Figure 5.1 (b) Implementation Results of 10 × 10 Image Buffer (Pipelined)

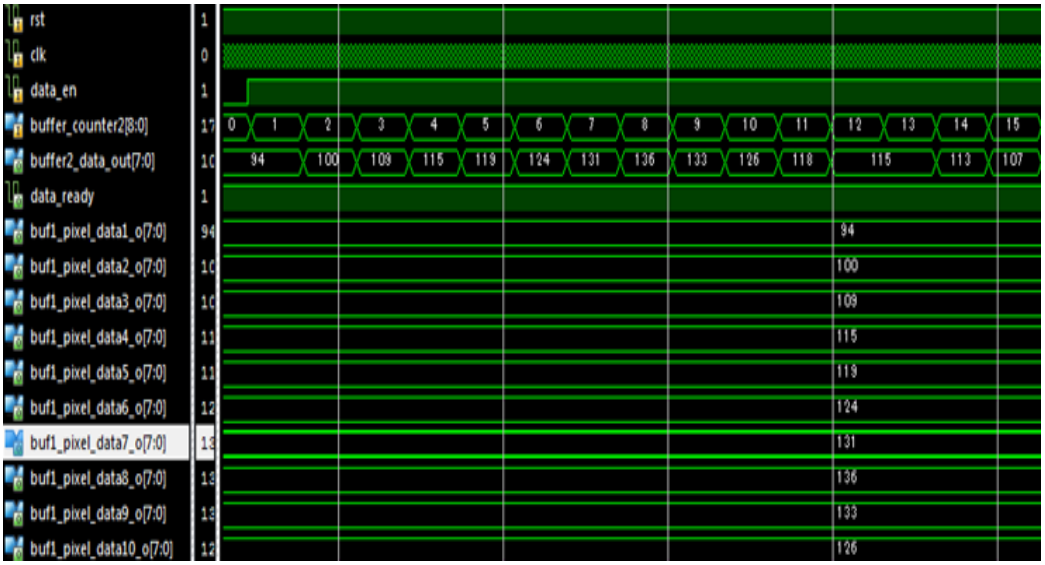


Figure 5.1 (c) Implementation Results of 20 × 20 Image Buffer (Pipelined)

5.2 Distance Module Implementation Results

The pixel data is loaded in the difference module while the corresponding counter values are inputted to the distance module. Figure 5.2(a) shows the Implementation results of the distance module of the first architecture, where the two counter values are used for calculating position. These position values are passed to the sub-modules through the *position1* and *position2* signals. After calculating the Euclidean distance, the values are outputted through only one *distance_out* signal as shown in the result. Sample calculations are shown in the result.

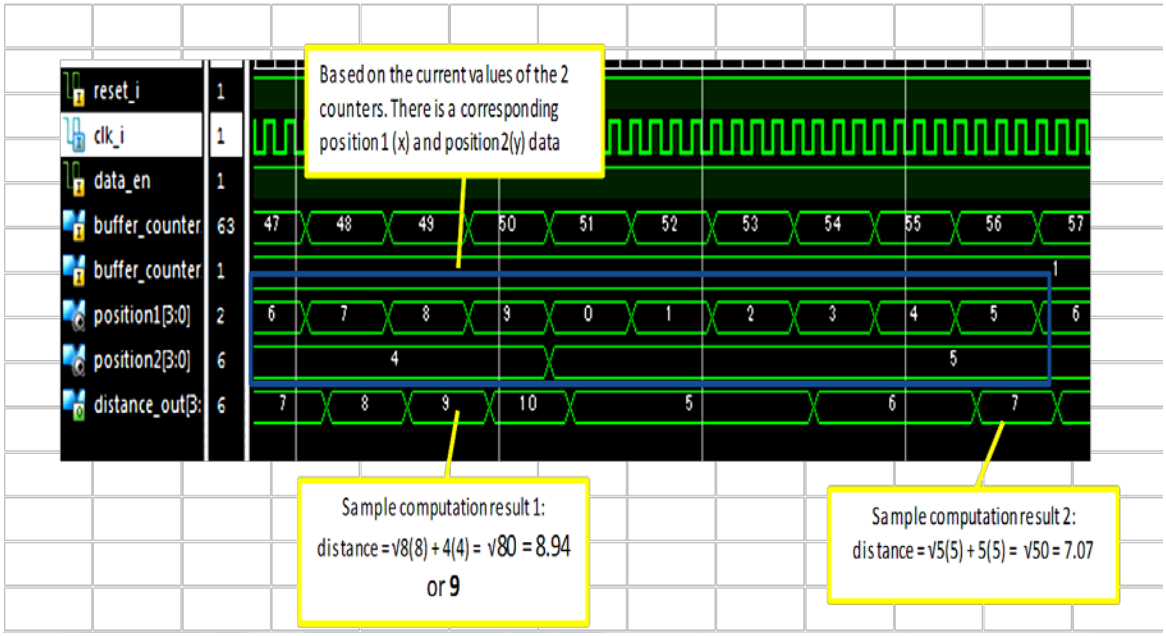


Figure 5.2 (a) Implementation Results for Distance Module (Not Pipelined)

The following results shows the simultaneous or parallel computations performed in the distance module since there are replicated sub-modules. It can be observed that as the counter values are updated, the position values are calculated with one clock cycle delay. After decoding the coordinate values, all the sub-modules start calculating the distance values.

Implementation results shows the distance values outputted from the sub-modules. There are 99 *distance_out* signals for a 10 × 10 image and 399 *distance_out* signals which are continuously updated with the distance values calculated in the submodules for all the pixel values as shown in the figure. Figure 5.2(b) and Figure 5.2(c) shows that all the position decoder modules and distance calculating modules with the square root

calculation work in parallel fashion. Distance and sorting modules are connected to each other though the *distance_out* signal. The distance values are given as the input to the sorting module where it acts as an offset or the sorting values.

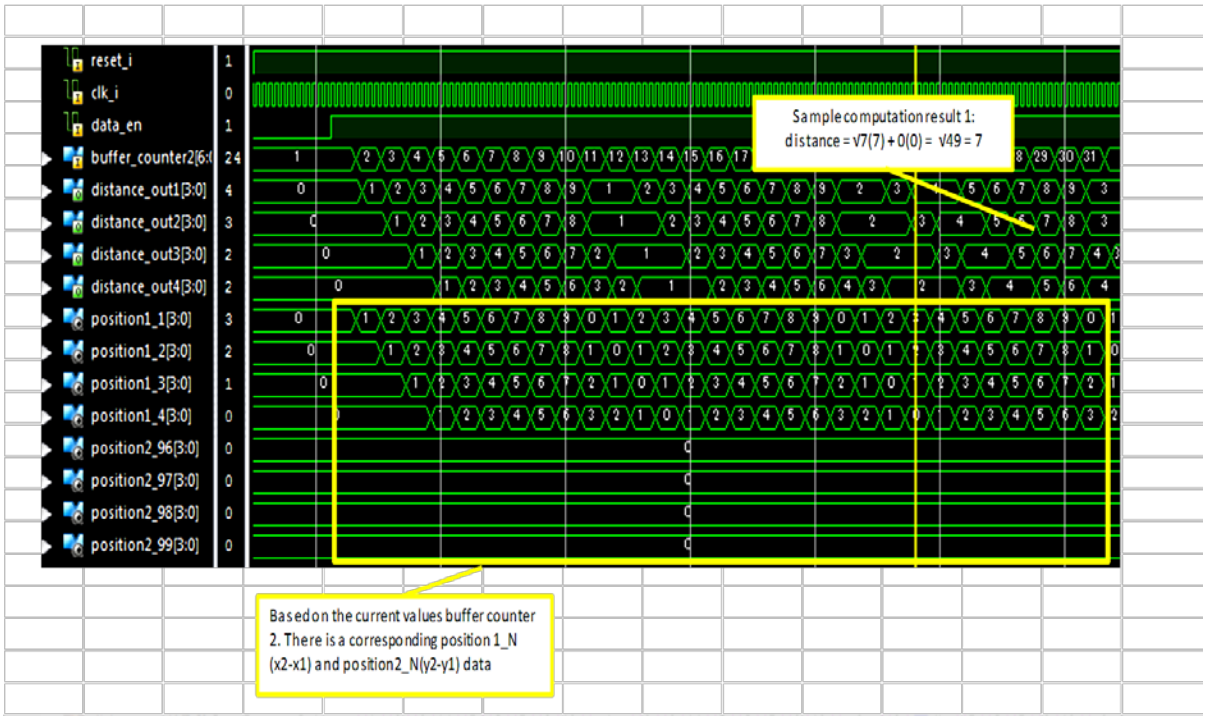


Figure 5.2 (b) Implementation Results of 10 × 10 Image Distance Module (Pipelined)

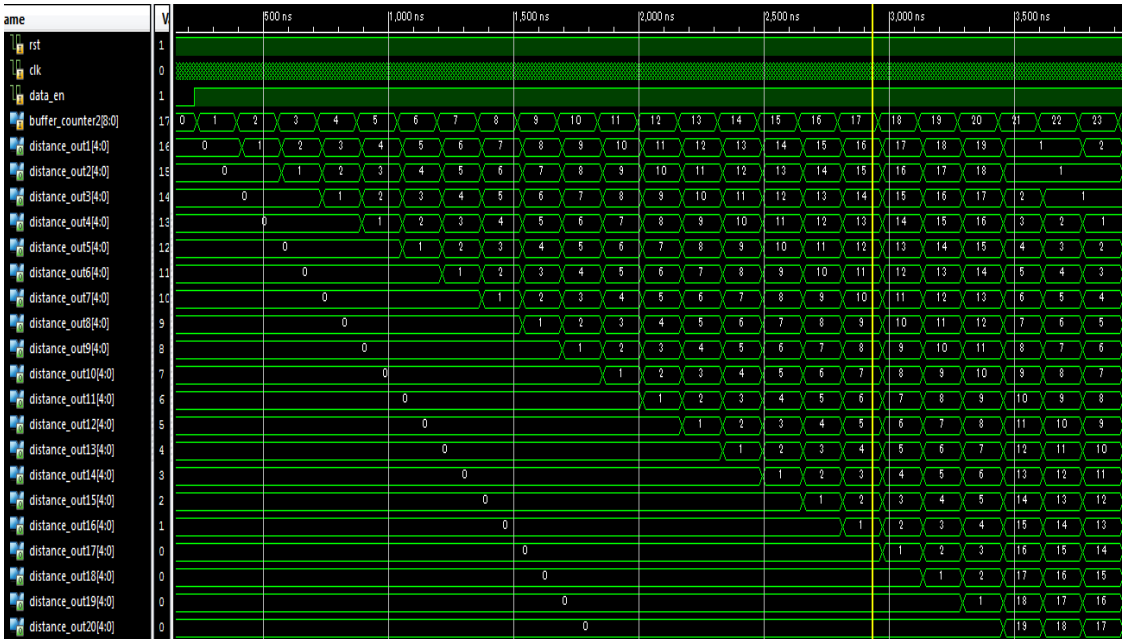
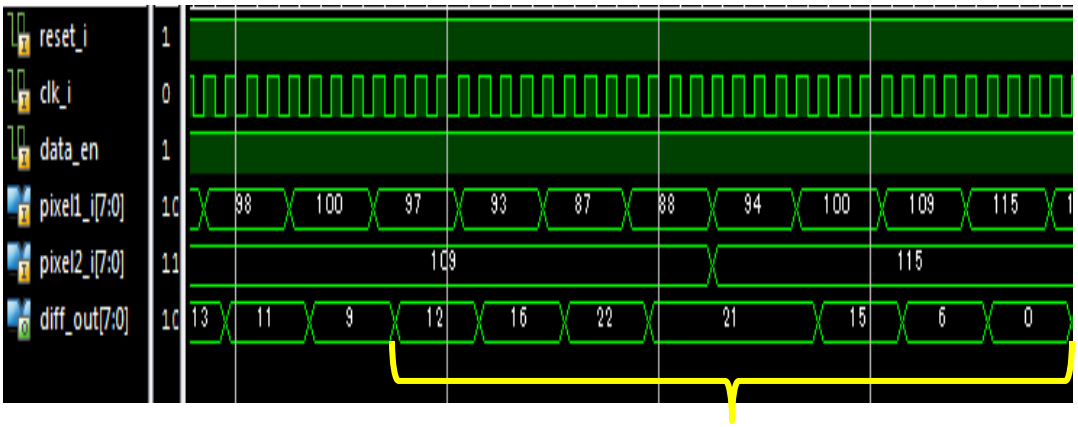


Figure 5.2 (c) Implementation Results of 20 × 20 Image Distance Module (Pipelined)

5.3 Difference Module Implementation Results

The figure below shows the internal signals inside the difference module. It is observed that this module dynamically processes the data with 1 clock cycle delay. The image buffer 2 value is first loaded in the second pixel register. The first pixel register is then loaded sequentially with the first image buffer pixel values. For every pixel 1 register value the absolute difference is calculated as shown in the implementation result. The value in the pixel 2 register is updated when all the values of image buffer are loaded in the pixel 1 register. These difference values are given as the input to the sorting module through the *diff_out* signal, which is continuously updated with the difference values calculated in the difference module.



We can see in the following examples that the circuit correctly computed the difference between 2 input pixels..: **109-97 = 12, 109-93= 16, 109-87= 22 , 109-88= 21 , 115-94=21, 115-100= 15, 115-9 = 6. 115-115= 0**

Figure 5.3 (a) Implementation Results for Difference Module (Non-Pipelined)

The following results shows the simultaneous or parallel computations performed in the difference module since there are replicated sub modules. It can be observed that the image buffer 1 values are replicated in the sub-modules, whereas the image buffer 2 values are loaded to the corresponding modules through the *buffer2_data_in* signal when the data enable is high. Once the sub-modules are loaded with the corresponding second image buffer pixel values the sub-modules start calculating the absolute difference between the pixels.

There are 99 *diff_out* signals for a 10 × 10 image and 399 *diff_out* signals for a 20 × 20

image which are continuously updated with the difference values calculated in the submodules for all the pixel values of image buffer 1 and image buffer 2 as shown in the figure. Figure 5.3(b) and Figure 5.3(c) show that all the data loading, calculation of differences between the pixels in the sub-modules work in a parallel fashion for two different image sizes. Difference and sorting modules are connected to each other though the *diff_out* signals. The difference values are given as the input to the sorting module where these values are sorted and added in the accumulator depending on the distance values.

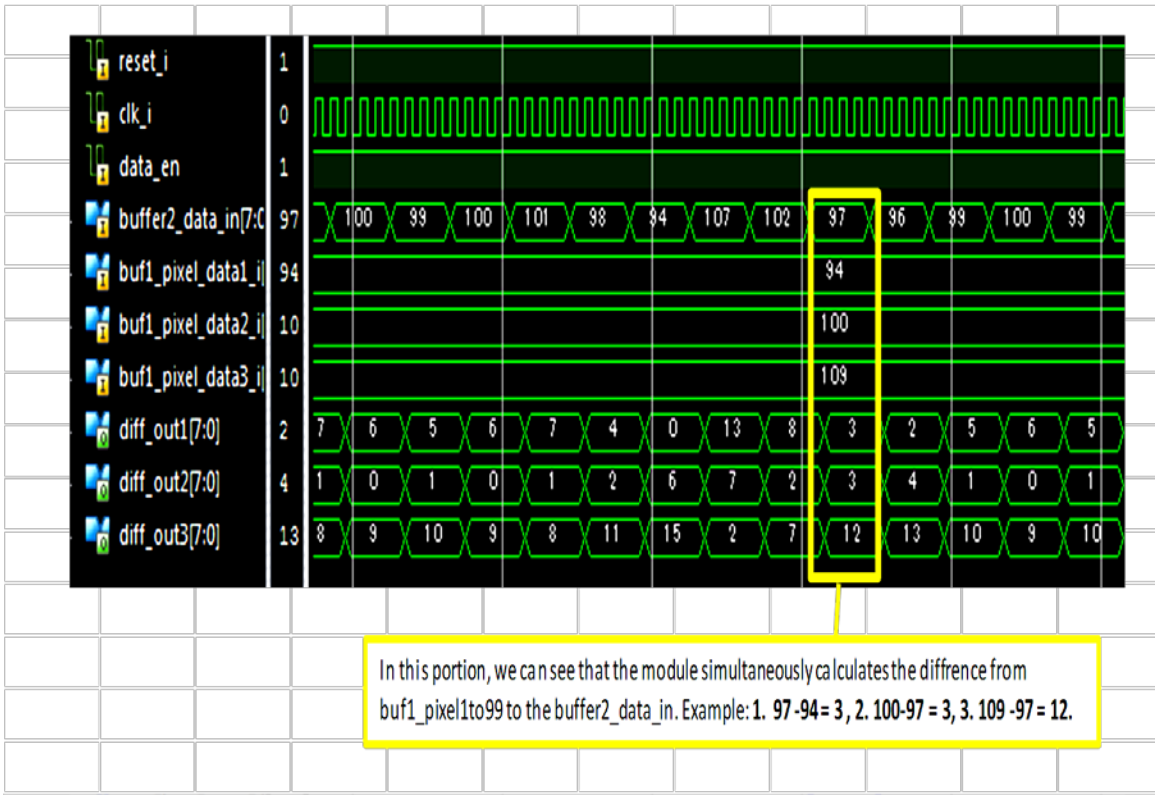


Figure 5.3 (b) Implementation Results of 10 × 10 Image Difference Module (Pipelined)

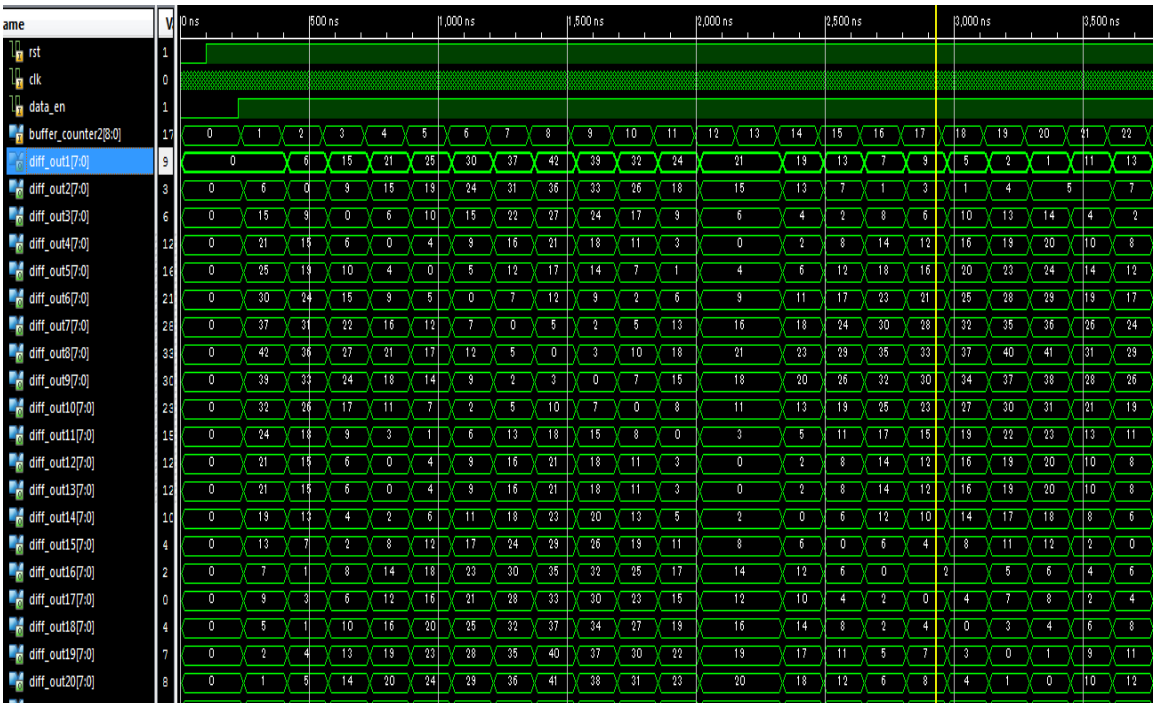


Figure 5.3 (c) Implementation Results of 20 × 20 Image Difference Module (Pipelined)

5.4 Sorting module Implementation results

The data enable is high for the sorting module whenever the first counter increments. The module is inputted with the distance data and difference data through one *diff_in* and *distance_in* signal from the difference and the distance modules. The sorting module uses distance as an offset and increments the corresponding registers calculating the number of pixel values as shown in the Figure 5.4(a). It can be observed from the following figure that the first register *m_out1* is incremented when the module encounters the lag distance to be 1. The difference values are squared, and these values are given as input to the accumulators through *sqrd_of_diff_in* signal. Sum register *sum_out1* is getting accumulated with the square of difference values if the offset is 1 as shown in the implementation result.

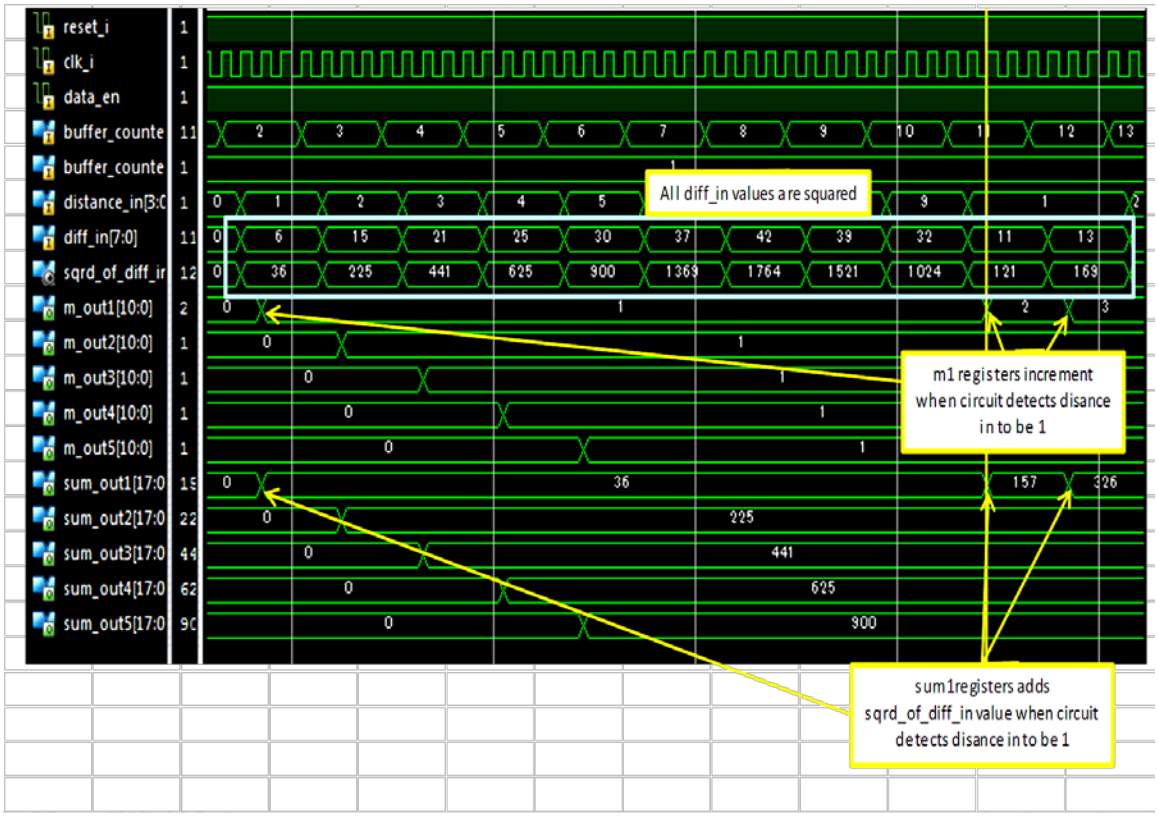


Figure 5.4 (a) Implementation Results for Sorting Module (Non-Pipelined)

In the second architecture, the sorting module contains 33 sub-modules for a 10×10 image and 25 sub-modules for a 20×20 image which are pipelined. Each sub-module of smaller image handles 3 difference and 3 distance values, whereas each sub-module of the bigger image handles 16 difference and 16 distance values. The functionality of all the sub_modules is same as the architecture version 1. The number of pixel pairs is calculated by incrementing the corresponding registers using lag distance as offset. These values are given as the input to the variogram calculating module using *m_out1*, *m_out2* signals. The difference values accumulated in the accumulators are given as input to the final variogram calculation module using *sum_out1*, *sum_out2*, etc., signals.

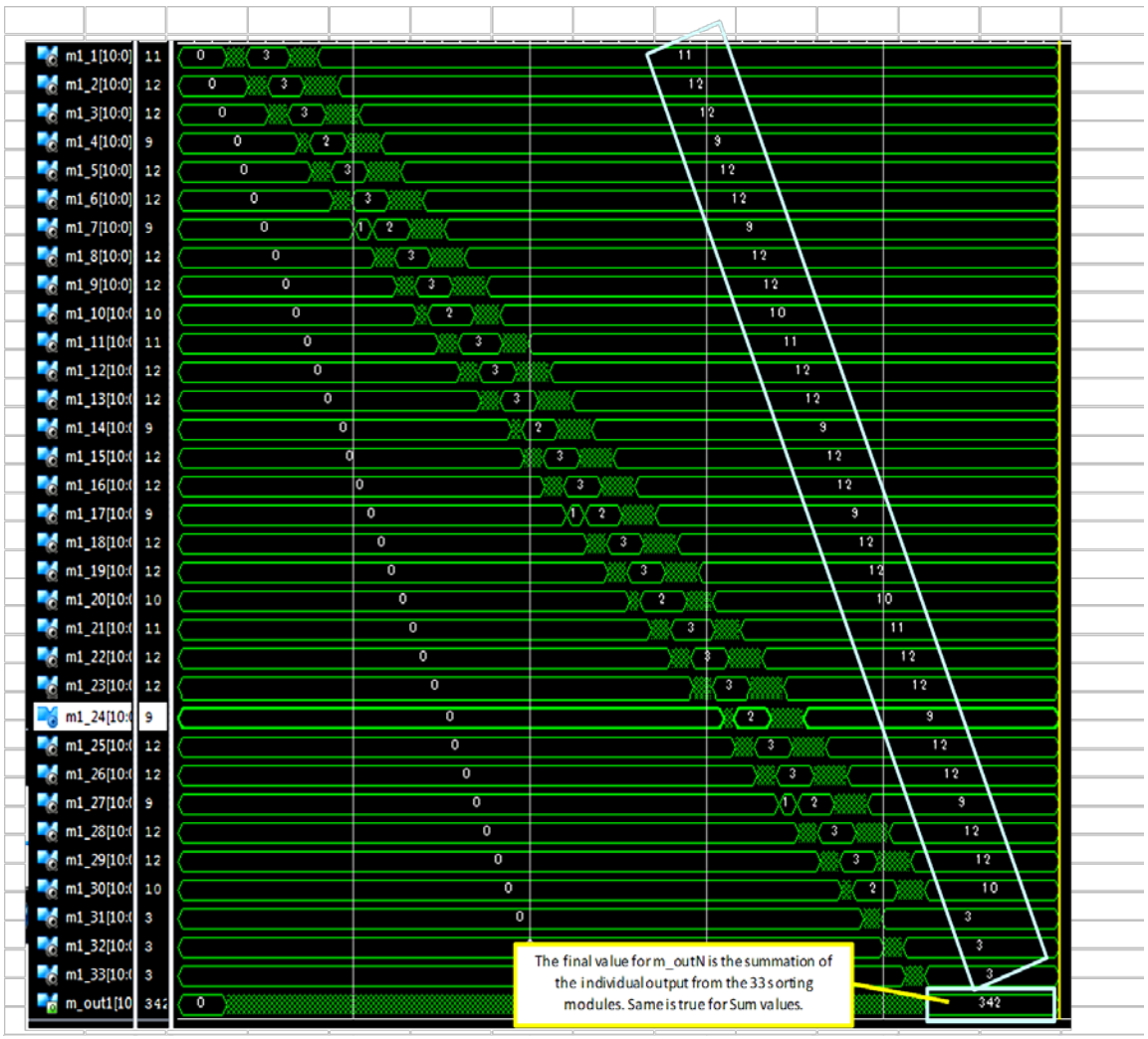


Figure 5.4 (b) Implementation Results for Sorting Module (Pipelined)

5.5 Variogram Calculation Module Implementation Results

5.5.1 Implementation for Architecture Version 1

Figure 5.5.1 shows the Implementation results showing internal signals of the final variogram module. It is observed that number of pixel values, summation values and semivariogram outputs is being updated and finally completed after counter 1 and 2 reach 100. The results show that the architecture version 1 speed is measured to be 400 microseconds for calculating the semivariogram values, as seen in following figure.

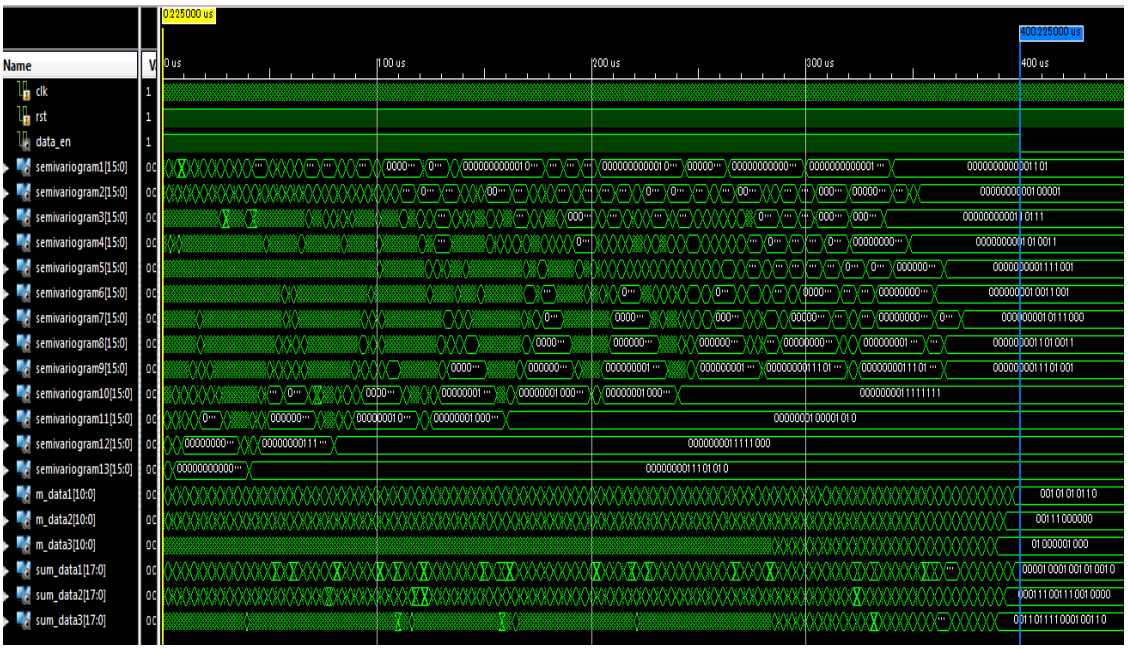


Figure 5.5.1 .Implementation Results for Architecture Version 1

Each counter combination to introduce pixel pairs to the combinational modules lasts for 4 clock cycles, because data inputs for differences and the distance module will be changed every 4 clock cycles. This clock period is used so that the large combinational logic circuits will have stable inputs to perform the calculations. Each clock cycle is 10ns for 100MHz hence a counter increment takes 4 x 10 ns = 40ns. There are two counters, each counting up to 100, when counter 1 is 1, counter 2 goes from 1 to 100, hence there are 100x100 counts. So approximately the total time of computation is 40ns x 100 x 100 = 400,000ns or 400uS. The results are shown in Figure 5.5.1 and are very close to these theoretical model values.

5.5.2 Implementations for Architecture Version 2 for a 10 × 10 Image

The Architecture version 2 is 100 times faster in speed than the first one due to the pipelining. Figure 5.5.2 shows the implementation time from data enable to completion. As discussed before, each count of buffer lasts for 4 clock cycles (40ns), but there is only one counter counting from 1 to 100, as the buffer 1 data is directly connected to inputs from difference module for the parallel calculations so approximately the total time of computation is 100x40ns = 4000ns or 4us, but due to the small logic delay the computational time is 4.27us.

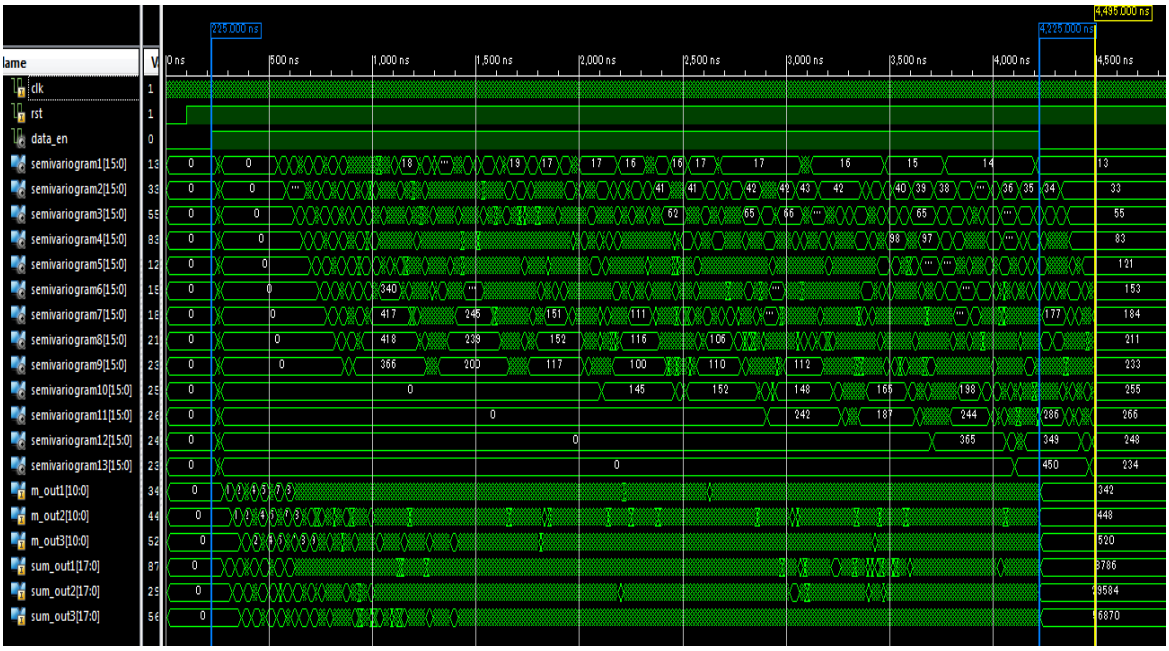


Figure 5.5.2 Implementation Results for Architecture Version 2 for a 10 × 10 Image

5.5.3 Implementations for Architecture Version 2 for a 20 × 20 Image

Architecture version 2 for a 20 × 20 image has 399 distance and 399 difference modules and 16 sorting modules for 400 pixel values. Due to the increase in the modules and complexity in the circuit by 4 times the implementation time has increased 16 times the implementation time of 10 × 10 image as shown in the following figure. Sorting modules have to be reduced because of the increase in the distance and difference sub-modules which is also one of the reasons for the increase in the computational time. Figure 5.5.3 shows the implementation time from data enable to completion. Here, each count of buffer lasts for 16 clock cycles. As discussed, each clock cycle is 10ns for 100MHz hence a counter increment takes 16 x 10 ns = 160ns, but there is only one counter counting from 1 to 400, as the buffer 1 data is directly connected to inputs from difference module for the parallel calculations so approximately the total time of computation is 400x160ns = 64000ns or 64us, but due to the small logic delay the computational time is 64.9us.

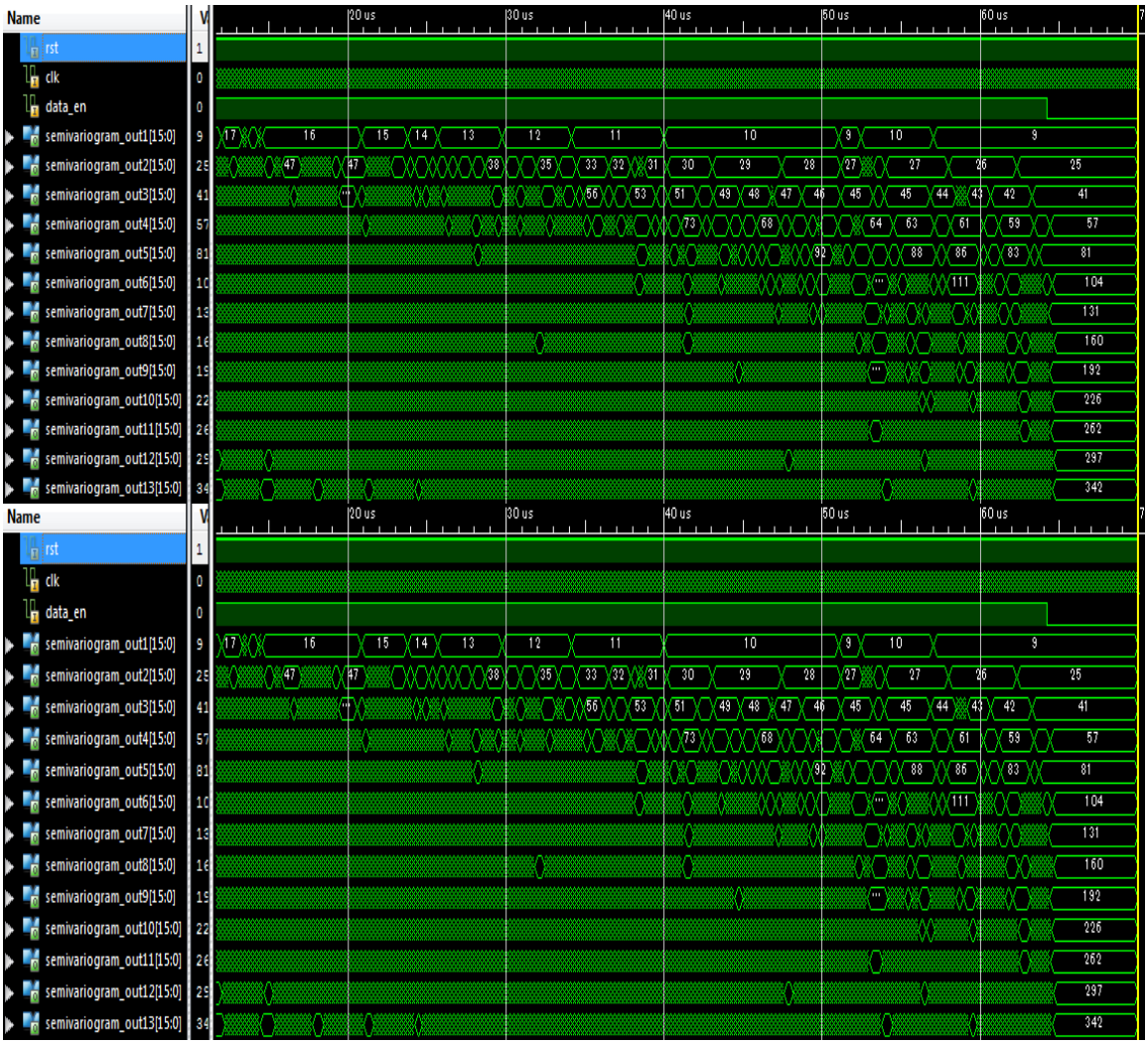


Figure 5.5.3 Implementation Results for Architecture Version 2 for a 20 × 20 Image

Table 5.1 below shows the computation times for the Matlab implementation, architecture version 1 (non-pipelined) and architecture version 2 (pipelined) for a 10 × 10 image and 20 × 20 image. The head-to-head results indicate significant performance improvement using the non-pipelined version, which is greatly improved by pipelining.

Table 5.1 Computational Time Comparison for Matlab Code, Non-Pipelined and Pipelined Architecture Versions of Implementations.

Implementation	Computational Time
Matlab code for 10 × 10 image	2 seconds
Matlab code for 20 × 20 image	7.13 seconds
Architecture version 1 (non-pipelined) for 10 × 10 image	400 microseconds
Architecture version 2 (pipelined) for 10 × 10 image	4.27 microseconds
Architecture version 2 (pipelined) 20 × 20 image	64.9 microseconds

The maximum Euclidian distance between pixels in a 10×10 image is 12.72 and for a 20×20 image is 26.43. Hence there will be 13 output semivariogram values for the smaller image and for the bigger one there will be 26 output semivariogram values for the algorithm being tested, corresponding to each value of '*h*'. Table 5.2 and Table 5.3 compares the semivariogram values obtained from the Matlab code to the implementation results from the FPGA for the two image sizes.

Table 5.2 Verification of the Semivariogram Values Computed Using the Specified Algorithm for a 10 × 10 Image.

Lag Distances	Matlab Computational Values	Observed Values
1	12.845	13
2	33.0179	33
3	54.6827	55
4	82.8918	83
5	120.7689	121
6	152.5643	153
7	184.4761	184
8	211.0484	211
9	233.216	233
10	254.9362	255
11	266.0833	266
12	248.4375	248
13	234	234

Table 5.3 Verification of the Semivariogram Values Computed Using the Specified Algorithm for a 20 × 20 Image.

Lag Distances	Matlab Computational Values	Observed Values
1	11.09	9
2	23.46	25
3	42.24	41
4	58.45	57
5	73.62	81
6	105.93	104
7	134.52	131
8	159.03	160
9	194.02	192
10	227.95	226
11	267.79	262
12	303.84	297
13	345.67	342
14	386.22	388
15	434.69	432
16	477.09	477
17	507.25	502
18	524.72	533
19	531	535
20	504.35	505
21	472.55	482
22	422.67	423
23	382.68	384
24	306.77	312
25	218	218
26	92.5	136

A summary of the synthesis report, device requirement and utilization shown in Table 5.4. As indicated by a study of the architectures, a significant price in extra hardware is required to improve the speed using pipelining. This can be attributed to the replicated modules and data supply arrangements to these extra replicated modules in the pipelined architecture. The maximum frequency and power consumption are also shown in the

table and indicate that both versions are utilizing the FPGA board at almost the maximum frequency allowed by the timing constraints for the designs. Pixel pair data presentation appears to be a significant factor in the speed of the algorithm.

Table 5.4 Device Utilization and Synthesis Report

Parameters	Architecture Version 1 for 10 × 10 image (Not Pipelined)	Architecture Version 2 for 10 × 10 image (Pipelined)	Architecture Version 2 for 20 × 20 image (Pipelined)
Slice Registers	10,220	24,739	72,156
Slice LUT's	5,653	35,755	2,36,497
LUT Flip Flop pairs	11,843	40,572	35,227
Unique control sets	23	439	640
Bonded IOB's	21	21	19
LOCed IOB's	21	21	21
BUFG/BUFGCTRLs	1	2	2
Number of DSP48EIs	3	63	64
Percentage of hardware utilized	11.22	35.8	78.2
Calculation time at 100Mhz	400 micro seconds	4.27 micro seconds	64.9 micro seconds
Calculation time at 50Mhz	800 micro seconds	8.54 micro seconds	129.8 micro seconds
Number of Clock Cycles	40000 cycles	427 cycles	6490 cycles
Maximum Frequency	100.392 MHz	100.251 MHz	100.251 MHz
Clock Frequency	100 MHz	100 MHz	100 MHz
Power Consumption	91 MW	331 MW	558 MW
Lines of Code	14,207	90,568	16,8000

CHAPTER SIX

DISCUSSION AND CONCLUSION

In this study we have demonstrated that the computational time of the semivariogram can be improved through hardware implementation when compared to the software. This chapter includes the conclusions from this thesis and the summary of the work. Future works based on this thesis are also proposed in the following sections.

6.1 Conclusion

Digital images have several features, such as, texture, color, shape etc. Texture is one of the important features and texture analysis has an important role in image processing, computer vision and pattern recognition. Texture feature extraction is the first step of texture analysis. There are many methods to extract texture features, and the Semivariogram method has application to medical image processing. In this thesis, two architectures Architecture version 1 (non-pipelined) and Architecture version 2 (pipelined) are proposed for the computation of the semivariogram using a custom FPGA architecture to reduce the computational time required.

Experiments were performed on a 10×10 and 20×20 image. Our results showed that hardware implementation with the implemented pipelined architecture can improve the computation time of variogram. It can be observed in the first architecture that the hardware implementations of semivariogram calculation reduce the time of computation when compared to the software implementation. The second architecture implemented massive parallelism in the form of replicated distance, difference and sorting modules that are further pipelined. The basic difference between the two architectures is the massively pipelined sub-modules in the second version with 99 sub modules for the 10×10 image and 399 sub-modules for 20×20 which run in parallel fashion. The output of these modules is connected to the sub-modules of the sorting module. Hence, for the massively

parallel pipelined architecture implemented results in a speedup of 100 over the non-pipelined architecture for a 10×10 image. It is envisaged that the speed is proportional to the number of parallel sub-modules and further experiments for bigger images are planned in the future.

To summarize, the following has been accomplished in this thesis:

- Implementation of semivariogram texture feature extraction with hardware descriptive language with verification of values.
- Design of two FPGA-based hardware architectures for texture extraction and implementation on Virtex 5 FPGA board.
- Comparing the performance of the designs with the software implementation in Matlab.

The circuit description language used in this thesis is VHSIC Hardware Description Language (VHDL) which appears almost the same as a programming language. But the difference still exists, the software being the sequential processes, whereas in the hardware descriptive language everything runs concurrently.

6.2 Future Work

The pipelined architecture can be utilized for small matrices, but the implementation for larger matrices imposes a challenge due to the large design size and FPGA limitations. The current design rounds off data to the nearest integer. Future work can involve a design that can process floating point numbers and can provide precision up to two decimal values. Due to the memory size and resources constraints operators generally do not work on a bigger image size. Hence other challenges to be addressed in future work also include increasing the image size, and further parallelization and pipelining to mitigate the effects of massive ordered pixel pair data access. The loop architecture proposed can also be improvised to meet the timing and memory constraints for the bigger images in the future.

REFERENCES

[1] R.M. Haralick, M. Robert, K. Shanmugam, I.H. Dinstein, "Textural Features for Image Classification," IEEE Transactions on Systems, Man and Cybernetics, vol. 3, no. 6, pp. 610-621, Nov 1973.

[2] J. R. Carr and F.P. De Miranda, "The Semivariogram in Comparison to the Co-Occurrence Matrix for Classification of Image Texture," IEEE Transactions on Geosciences and Remote Sensing, vol. 36, no. 6, pp. 1945–1952, Nov. 1998.

[3] X. Dong, M. Shirvaikar, X. Wang, "Biomechanical Properties and Microarchitecture Parameters of Trabecular Bone are Correlated with Stochastic Measures of 2D Projection Images," Bone 56, vol. 2, pp. 327-336, October, 2013.

[4] D. Marcotte, "Fast Variogram Computation with FFT," Computers and Geosciences 22, vol. 22, issue 10, pp. 1175–1186, Dec. 1996.

[5] M. Roumi, "Implementing Texture Feature Extraction Algorithms on FPGA," M.Sc. Thesis, Delft University of Technology, 2009.

[6] M. Harshavardhan, S. Visweswara Rao, "GLCM architecture of image extraction," International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE), vol. 3, issue 1, pp. 75-82, January 2014.

[7] A.R. Akoushideh, A. Shahbahrami, and B.M.N. Maybodi, "High performance implementation of texture features extraction algorithms using FPGA architecture,"Journal of Real-Time Image Processing, vol 9, Issue 1, pp 141-157, 2014.

[8] A.B. Girisha, M.C. Chandrashekhar, M.Z. Kurian, "FPGA implementation of GLCM", International Journal of Advanced Research in Electrical, Electronics, and Instrumentation Engineering (IJARECE), vol. 2, issue 6, pp. 2618-2621, June 2013.

[9] A.B. Girisha, M.C. Chandrashekhar, M.Z. Kurian, "Texture Feature Extraction of Video Frames Using GLCM", International Journal of Engineering Trends and Technology (IJETT), vol. 4 Issue 6, pp. 2718-2721, June 2013.

[10] M. Wielgosz, M. Panaggabean, L.A. Ronningen, “FPGA architecture for Kriging Image Interpolation,” International Journal of Advanced Computer Science and Applications (IJACSA). vol. 4, no. 12, pp. 193-201, 2013.

[11] K. Wada, (2013), Ring Buffer Basics, [Online]. Available: http://www.embedded.com/electronics-blogs/embedded-roundtable/4419407/The-ring-buffer.

[12] I. Daemon (2012), Crossbar Switch Architecture [Online], Available: http://www.inetdaemon.com/tutorials/networking/lan/switching/architectures/crossbar.shtml.

[13] G. Akila, D. Peterson, G. Lee Warren, R.J. Hindle, and R.J. Harrison, “A Pipelined and Parallel Architecture for Quantum Monte Carlo Implementations on FPGA’s”, VLSI Design, vol. 2010, Article ID 946486, pp. 283–286, 2010.

[14] A. E. Nelson, “Implementation of Image processing algorithms on FPGA hardware,” M.Sc. Thesis, Vanderbilt University, 2000.

[15] S. Rose, “Development of parallel Image Processing architecture in VHDL,” M.Sc. Thesis, University of West Australia, 2012.

[16] A. Baraldi and F. Panniggiani, "An Investigation of the Textural Characteristics Associated with Gray Level Cooccurrence Matrix Statistical Parameters", IEEE transactions on geosciences and remote sensing, vol. 33, no. 2, pp- 293 - 304, March 1995.

[17] G. Bohling (2005), Introduction to Geostatic and Variogram analysis [Online], Available:http://people.ku.edu/~gbohling/cpe940/Variograms.pdf.

[18] R. Pinninti, “Stochastic Assessment of Bone Fragility in Human Lumbar Spine,” M.Sc. Thesis, University Of Texas, Tyler, 2015.

[19] K.C. Chang, Digital Design and Modeling with VHDL and Synthesis, IEEE Computer Society Press - Wiley, 1997

[20] K. Heikkinen and P. Vuorimaa, Computation of Two Texture Features in Hardware, Proceedings of the 10th International Conference on Image Analysis and Processing, Venice, Italy, pages 125-129, pp. 27- 29, September1999.

APPENDIX A

VHDL CODE

--Semivariogram Top Module

```
 Library IEEE;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_unsigned.all;


  Library work;
  Entity semivariogram_top is
   port(
   clk : in std_logic;
   rst : in std_logic;
   SDATA : in std_logic;
   SCLK : in std_logic;
   R_BTN : in std_logic;
   L_BTN : in std_logic;
   START_CALC : in std_logic;
   SHOW_EXP : in std_logic;
   LED : out std_logic_vector(12 downto 0)
      );
   end entity;
  architecture rtl of semivariogram_top is
  component counter is
    port (
      rst                 : in  std_logic;
    clk                 : in  std_logic;
      start_calc            : in  std_logic;
      data_ready            : in  std_logic;
```

```
--   buffer_counter1_o : out std_logic_vector (6 downto 0);
     buffer_counter2_o : out std_logic_vector (6 downto 0);
     data_en                 : out std_logic
 );
 end component counter;
 component image_buffer is
   port (
   rst                 : in  std_logic;
   clk                 : in  std_logic;
     serial_data_in          : in  std_logic;
     serial_clock_in         : in  std_logic;
     data_en                       : in  std_logic;
     buffer_counter2         : in std_logic_vector (6 downto 0);
     buffer2_data_out    : out std_logic_vector (7 downto 0);
     data_ready              : out std_logic;
   buf1_pixel_data1_o   : out std_logic_vector(7 downto 0);
   buf1_pixel_data2_o   : out std_logic_vector(7 downto 0);
   buf1_pixel_data3_o   : out std_logic_vector(7 downto 0);
   buf1_pixel_data4_o   : out std_logic_vector(7 downto 0);
   buf1_pixel_data5_o   : out std_logic_vector(7 downto 0);
   buf1_pixel_data6_o   : out std_logic_vector(7 downto 0);
   buf1_pixel_data7_o   : out std_logic_vector(7 downto 0);
   buf1_pixel_data8_o   : out std_logic_vector(7 downto 0);
   buf1_pixel_data9_o   : out std_logic_vector(7 downto 0);
   buf1_pixel_data10_o  : out std_logic_vector(7 downto 0);
   buf1_pixel_data11_o  : out std_logic_vector(7 downto 0);
   buf1_pixel_data12_o  : out std_logic_vector(7 downto 0);
   buf1_pixel_data13_o  : out std_logic_vector(7 downto 0);
```

```
    buf1_pixel_data14_o  : out std_logic_vector(7 downto 0);
    buf1_pixel_data15_o  : out std_logic_vector(7 downto 0);
    buf1_pixel_data16_o  : out std_logic_vector(7 downto 0);

  );
end component image_buffer;

component distance_top is
   port(
    reset_i               : in  std_logic;
    clk_i                 : in  std_logic;
    data_en               : in  std_logic;
      buffer_counter2           : in  std_logic_vector (6 downto 0);
    distance_out1         : out  std_logic_vector (3 downto 0);
    distance_out2         : out  std_logic_vector (3 downto 0);
    distance_out3         : out  std_logic_vector (3 downto 0);
    distance_out4         : out  std_logic_vector (3 downto 0);
    distance_out5         : out  std_logic_vector (3 downto 0);
    distance_out6         : out  std_logic_vector (3 downto 0);
    distance_out7         : out  std_logic_vector (3 downto 0);
    distance_out8         : out  std_logic_vector (3 downto 0);
    distance_out9         : out  std_logic_vector (3 downto 0);
    distance_out10        : out  std_logic_vector (3 downto 0);
    distance_out11        : out  std_logic_vector (3 downto 0);
    distance_out12        : out  std_logic_vector (3 downto 0);
    distance_out13        : out  std_logic_vector (3 downto 0);
    distance_out14        : out  std_logic_vector (3 downto 0);
    distance_out15        : out  std_logic_vector (3 downto 0));
```

```
);

end component distance_top;


component difference_top is
  port (
   reset_i              : in  std_logic;
   clk_i                : in  std_logic;
   data_en              : in  std_logic;
     buffer2_data_in      : in std_logic_vector(7 downto 0);
   buf1_pixel_data1_i   : in std_logic_vector(7 downto 0);
   buf1_pixel_data2_i   : in std_logic_vector(7 downto 0);
   buf1_pixel_data3_i   : in std_logic_vector(7 downto 0);
   buf1_pixel_data4_i   : in std_logic_vector(7 downto 0);
   buf1_pixel_data5_i   : in std_logic_vector(7 downto 0);
   buf1_pixel_data6_i   : in std_logic_vector(7 downto 0);
   buf1_pixel_data7_i   : in std_logic_vector(7 downto 0);
   buf1_pixel_data8_i   : in std_logic_vector(7 downto 0);
   buf1_pixel_data9_i   : in std_logic_vector(7 downto 0);
   buf1_pixel_data10_i  : in std_logic_vector(7 downto 0);
   buf1_pixel_data11_i  : in std_logic_vector(7 downto 0);
   buf1_pixel_data12_i  : in std_logic_vector(7 downto 0);
   buf1_pixel_data13_i  : in std_logic_vector(7 downto 0);
   buf1_pixel_data14_i  : in std_logic_vector(7 downto 0);
   buf1_pixel_data15_i  : in std_logic_vector(7 downto 0);
   diff_out1          : out std_logic_vector (7 downto 0);
   diff_out2          : out std_logic_vector (7 downto 0);
```

```
    diff_out3          : out std_logic_vector (7 downto 0);
    diff_out4          : out std_logic_vector (7 downto 0);
    diff_out5          : out std_logic_vector (7 downto 0);
    diff_out6          : out std_logic_vector (7 downto 0);
    diff_out7          : out std_logic_vector (7 downto 0);
    diff_out8          : out std_logic_vector (7 downto 0);
    diff_out9          : out std_logic_vector (7 downto 0);
    diff_out10         : out std_logic_vector (7 downto 0);
    diff_out11         : out std_logic_vector (7 downto 0);
    diff_out12         : out std_logic_vector (7 downto 0);
    diff_out13         : out std_logic_vector (7 downto 0);
    diff_out14         : out std_logic_vector (7 downto 0);
    diff_out15         : out std_logic_vector (7 downto 0);

 );
 end component difference_top;



component sorting_top is
  port(
   reset_i              : in  std_logic;
   clk_i                : in  std_logic;
   data_en              : in  std_logic;
  -- buffer_counter1      : in  std_logic_vector (6 downto 0);

   buffer_counter2      : in  std_logic_vector (6 downto 0);
   distance_in1         : in  std_logic_vector (3 downto 0);
   distance_in2         : in  std_logic_vector (3 downto 0);
   distance_in3         : in  std_logic_vector (3 downto 0);
```

```
distance_in4          : in  std_logic_vector (3 downto 0);
distance_in5          : in  std_logic_vector (3 downto 0);
distance_in6          : in  std_logic_vector (3 downto 0);
distance_in7          : in  std_logic_vector (3 downto 0);
distance_in8          : in  std_logic_vector (3 downto 0);
distance_in9          : in  std_logic_vector (3 downto 0);
distance_in10         : in  std_logic_vector (3 downto 0);
distance_in11         : in  std_logic_vector (3 downto 0);
distance_in12         : in  std_logic_vector (3 downto 0);
distance_in13         : in  std_logic_vector (3 downto 0);
distance_in14         : in  std_logic_vector (3 downto 0);
distance_in15         : in  std_logic_vector (3 downto 0);
diff_in1              : in std_logic_vector (7 downto 0);
diff_in2              : in std_logic_vector (7 downto 0);
diff_in3              : in std_logic_vector (7 downto 0);
diff_in4              : in std_logic_vector (7 downto 0);
diff_in5              : in std_logic_vector (7 downto 0);
diff_in6              : in std_logic_vector (7 downto 0);
diff_in7              : in std_logic_vector (7 downto 0);
diff_in8              : in std_logic_vector (7 downto 0);
diff_in9              : in std_logic_vector (7 downto 0);
diff_in10             : in std_logic_vector (7 downto 0);
diff_in11             : in std_logic_vector (7 downto 0);
diff_in12             : in std_logic_vector (7 downto 0);
diff_in13             : in std_logic_vector (7 downto 0);
diff_in14             : in std_logic_vector (7 downto 0);
diff_in15             : in std_logic_vector (7 downto 0);
diff_in16             : in std_logic_vector (7 downto 0);
```

```
 m_out1                : out std_logic_vector (10 downto 0);
  m_out2               : out std_logic_vector (10 downto 0);
  m_out3               : out std_logic_vector (10 downto 0);
  m_out4               : out std_logic_vector (10 downto 0);
  m_out5               : out std_logic_vector (10 downto 0);
  m_out6               : out std_logic_vector (10 downto 0);
  m_out7               : out std_logic_vector (10 downto 0);
  m_out8               : out std_logic_vector (10 downto 0);
  m_out9               : out std_logic_vector (10 downto 0);
  m_out10              : out std_logic_vector (10 downto 0);
  m_out11              : out std_logic_vector (10 downto 0);
  m_out12              : out std_logic_vector (10 downto 0);
  m_out13              : out std_logic_vector (10 downto 0);
  sum_out1       : out std_logic_vector (17 downto 0);
  sum_out2       : out std_logic_vector (17 downto 0);
  sum_out3       : out std_logic_vector (17 downto 0);
  sum_out4       : out std_logic_vector (17 downto 0);
  sum_out5       : out std_logic_vector (17 downto 0);
  sum_out6       : out std_logic_vector (17 downto 0);
  sum_out7       : out std_logic_vector (17 downto 0);
  sum_out8       : out std_logic_vector (17 downto 0);
  sum_out9       : out std_logic_vector (17 downto 0);
  sum_out10      : out std_logic_vector (17 downto 0);
  sum_out11      : out std_logic_vector (17 downto 0);
  sum_out12      : out std_logic_vector (17 downto 0);
  sum_out13      : out std_logic_vector (17 downto 0)
);
```

```
end component sorting_top;

component calc_variogram is
     port(
    reset_i                : in    std_logic;
    clk_i                  : in    std_logic;
    data_en                : in    std_logic;
      m_out1                   : in   std_logic_vector (10 downto 0);
      m_out2                   : in   std_logic_vector (10 downto 0);
      m_out3                   : in   std_logic_vector (10 downto 0);
      m_out4                   : in   std_logic_vector (10 downto 0);
      m_out5                   : in   std_logic_vector (10 downto 0);
      m_out6                   : in   std_logic_vector (10 downto 0);
      m_out7                   : in   std_logic_vector (10 downto 0);
      m_out8                   : in   std_logic_vector (10 downto 0);
      m_out9                   : in   std_logic_vector (10 downto 0);
      m_out10                  : in   std_logic_vector (10 downto 0);
      m_out11                  : in   std_logic_vector (10 downto 0);
      m_out12                  : in   std_logic_vector (10 downto 0);
      m_out13                  : in   std_logic_vector (10 downto 0);
      sum_out1               : in   std_logic_vector (17 downto 0);
      sum_out2               : in   std_logic_vector (17 downto 0);
      sum_out3               : in   std_logic_vector (17 downto 0);
      sum_out4               : in   std_logic_vector (17 downto 0);
      sum_out5               : in   std_logic_vector (17 downto 0);
      sum_out6               : in   std_logic_vector (17 downto 0);
      sum_out7               : in   std_logic_vector (17 downto 0);
      sum_out8               : in   std_logic_vector (17 downto 0);
```

```
      sum_out9              : in   std_logic_vector (17 downto 0);
      sum_out10               : in   std_logic_vector (17 downto 0);
      sum_out11               : in   std_logic_vector (17 downto 0);
      sum_out12               : in   std_logic_vector (17 downto 0);
      sum_out13               : in   std_logic_vector (17 downto 0);
      semivariogram_out1    : out std_logic_vector (15 downto 0);
      semivariogram_out2    : out std_logic_vector (15 downto 0);
      semivariogram_out3    : out std_logic_vector (15 downto 0);
      semivariogram_out4    : out std_logic_vector (15 downto 0);
      semivariogram_out5    : out std_logic_vector (15 downto 0);
      semivariogram_out6    : out std_logic_vector (15 downto 0);
      semivariogram_out7    : out std_logic_vector (15 downto 0);
      semivariogram_out8    : out std_logic_vector (15 downto 0);
      semivariogram_out9    : out std_logic_vector (15 downto 0);
      semivariogram_out10   : out std_logic_vector (15 downto 0);
      semivariogram_out11   : out std_logic_vector (15 downto 0);
      semivariogram_out12   : out std_logic_vector (15 downto 0);
      semivariogram_out13   : out std_logic_vector (15 downto 0)
     );
  end component calc_variogram;



component output_mux is
  port (
    rst                 : in  std_logic;
    clk                 : in  std_logic;
      show_expected           : in  std_logic;
      right_button            : in  std_logic;
```

```
      left_button             : in  std_logic;
      semivariogram_1     : in std_logic_vector (15 downto 0);
      semivariogram_2     : in std_logic_vector (15 downto 0);
      semivariogram_3     : in std_logic_vector (15 downto 0);
      semivariogram_4     : in std_logic_vector (15 downto 0);
      semivariogram_5     : in std_logic_vector (15 downto 0);
      semivariogram_6     : in std_logic_vector (15 downto 0);
      semivariogram_7     : in std_logic_vector (15 downto 0);
      semivariogram_8     : in std_logic_vector (15 downto 0);
      semivariogram_9     : in std_logic_vector (15 downto 0);
      semivariogram_10    : in std_logic_vector (15 downto 0);
      semivariogram_11    : in std_logic_vector (15 downto 0);
      semivariogram_12    : in std_logic_vector (15 downto 0);
      semivariogram_13    : in std_logic_vector (15 downto 0);
      semivariogram_out   : out std_logic_vector (15 downto 0);
      current_data_num    : out std_logic_vector (3 downto 0)
);
end component output_mux;

  -- Signal Declaration
  signal  buffer_counter1 :  std_logic_vector (6 downto 0);
    signal  buffer_counter2 :  std_logic_vector (6 downto 0);
    signal  data_en         :  std_logic;
  signal  buffer1_data      :  std_logic_vector (7 downto 0);
  signal  buffer2_data      :  std_logic_vector (7 downto 0);
  signal    diff      :    std_logic_vector (7 downto 0);
  signal   distance   :    std_logic_vector (3 downto 0);
```

```
signal  m_data1    :  std_logic_vector (10 downto 0);
signal  m_data2    :  std_logic_vector (10 downto 0);
signal  m_data3    :  std_logic_vector (10 downto 0);
signal  m_data4    :  std_logic_vector (10 downto 0);
signal  m_data5    :  std_logic_vector (10 downto 0);
signal  m_data6    :  std_logic_vector (10 downto 0);
signal  m_data7    :  std_logic_vector (10 downto 0);
signal  m_data8    :  std_logic_vector (10 downto 0);
signal  m_data9    :  std_logic_vector (10 downto 0);
signal  m_data10    :  std_logic_vector (10 downto 0);
signal  m_data11    :  std_logic_vector (10 downto 0);
signal  m_data12    :  std_logic_vector (10 downto 0);
signal  m_data13    :  std_logic_vector (10 downto 0);
signal  sum_data1    :  std_logic_vector (17 downto 0);
signal  sum_data2    :  std_logic_vector (17 downto 0);
signal  sum_data3    :  std_logic_vector (17 downto 0);
signal  sum_data4    :  std_logic_vector (17 downto 0);
signal  sum_data5    :  std_logic_vector (17 downto 0);
signal  sum_data6    :  std_logic_vector (17 downto 0);
signal  sum_data7    :  std_logic_vector (17 downto 0);
signal  sum_data8    :  std_logic_vector (17 downto 0);
signal  sum_data9    :  std_logic_vector (17 downto 0);
signal  sum_data10    :  std_logic_vector (17 downto 0);
signal  sum_data11    :  std_logic_vector (17 downto 0);
signal  sum_data12    :  std_logic_vector (17 downto 0);
signal  sum_data13    :  std_logic_vector (17 downto 0);
signal  semivariogram1    :  std_logic_vector (15 downto 0);
signal  semivariogram2    :  std_logic_vector (15 downto 0);
```

```
 signal  semivariogram3    :  std_logic_vector (15 downto 0);
signal  semivariogram4    :  std_logic_vector (15 downto 0);
signal  semivariogram5    :  std_logic_vector (15 downto 0);
signal  semivariogram6    :  std_logic_vector (15 downto 0);
signal  semivariogram7    :  std_logic_vector (15 downto 0);
signal  semivariogram8    :  std_logic_vector (15 downto 0);
signal  semivariogram9    :  std_logic_vector (15 downto 0);
signal  semivariogram10    :  std_logic_vector (15 downto 0);
signal  semivariogram11    :  std_logic_vector (15 downto 0);
signal  semivariogram12    :  std_logic_vector (15 downto 0);
signal  semivariogram13    :  std_logic_vector (15 downto 0);
signal  current_data_num    :  std_logic_vector (3 downto 0);
signal  data_ready   :  std_logic;
signal semivariogram_out   :  std_logic_vector (15 downto 0);
signal  buf1_pixel_data1   :  std_logic_vector(7 downto 0);
signal  buf1_pixel_data2   :  std_logic_vector(7 downto 0);
signal  buf1_pixel_data3   :  std_logic_vector(7 downto 0);
signal  buf1_pixel_data4   :  std_logic_vector(7 downto 0);
signal  buf1_pixel_data5   :  std_logic_vector(7 downto 0);
signal  buf1_pixel_data6   :  std_logic_vector(7 downto 0);
signal  buf1_pixel_data7   :  std_logic_vector(7 downto 0);
signal  buf1_pixel_data8   :  std_logic_vector(7 downto 0);
signal  buf1_pixel_data9   :  std_logic_vector(7 downto 0);
signal  buf1_pixel_data10  :  std_logic_vector(7 downto 0);
signal  buf1_pixel_data11  :  std_logic_vector(7 downto 0);
signal  buf1_pixel_data12  :  std_logic_vector(7 downto 0);
signal  buf1_pixel_data13  :  std_logic_vector(7 downto 0);
signal  buf1_pixel_data14  :  std_logic_vector(7 downto 0);
```

```
signal  buf1_pixel_data15  :  std_logic_vector(7 downto 0);
signal  buf1_pixel_data99  :  std_logic_vector(7 downto 0);
signal  buf1_pixel_data100 :  std_logic_vector(7 downto 0);
signal    distance_out1         :  std_logic_vector (3 downto 0);
signal    distance_out2         :  std_logic_vector (3 downto 0);
signal    distance_out3         :  std_logic_vector (3 downto 0);
signal    distance_out4         :  std_logic_vector (3 downto 0);
signal    distance_out5         :  std_logic_vector (3 downto 0);
signal    distance_out6         :  std_logic_vector (3 downto 0);
signal    distance_out7         :  std_logic_vector (3 downto 0);
signal    distance_out8         :  std_logic_vector (3 downto 0);
signal    distance_out9         :  std_logic_vector (3 downto 0);
signal    distance_out10        :  std_logic_vector (3 downto 0);
signal    distance_out11        :  std_logic_vector (3 downto 0);
signal    distance_out12        :  std_logic_vector (3 downto 0);
signal    distance_out13        :  std_logic_vector (3 downto 0);
signal    distance_out14        :  std_logic_vector (3 downto 0);
signal    distance_out15        :  std_logic_vector (3 downto 0);
signal    diff_out1         :  std_logic_vector (7 downto 0);
signal    diff_out2         :  std_logic_vector (7 downto 0);
signal    diff_out3         :  std_logic_vector (7 downto 0);
signal    diff_out4         :  std_logic_vector (7 downto 0);
signal    diff_out5         :  std_logic_vector (7 downto 0);
signal    diff_out6         :  std_logic_vector (7 downto 0);
signal    diff_out7         :  std_logic_vector (7 downto 0);
signal    diff_out8         :  std_logic_vector (7 downto 0);
signal    diff_out9         :  std_logic_vector (7 downto 0);
signal    diff_out10        :  std_logic_vector (7 downto 0);
```

```
  signal    diff_out11        :  std_logic_vector (7 downto 0);
  signal    diff_out12        :  std_logic_vector (7 downto 0);
  signal    diff_out13        :  std_logic_vector (7 downto 0);
  signal    diff_out14        :  std_logic_vector (7 downto 0);
  signal    diff_out15        :  std_logic_vector (7 downto 0);
  );
begin

   u_counter : counter
    port map (
     rst                => rst,
   clk                => clk,
     start_calc           => START_CALC,
     buffer_counter2_o      => buffer_counter2,
     data_en                => data_en,
     data_ready             => data_ready
    );
   u_image_buffer : image_buffer
    port map (
    rst                  => rst ,
    clk                  => clk ,
    serial_data_in       => SDATA ,
    serial_clock_in      => SCLK ,
    data_en                    => data_en ,
    buffer_counter2      => buffer_counter2 ,
    buffer2_data_out     => buffer2_data ,
    data_ready           => data_ready ,
    buf1_pixel_data1_o   => buf1_pixel_data1   ,
```

```
  buf1_pixel_data2_o   => buf1_pixel_data2   ,
  buf1_pixel_data3_o   => buf1_pixel_data3   ,
  buf1_pixel_data4_o   => buf1_pixel_data4   ,
  buf1_pixel_data5_o   => buf1_pixel_data5   ,
  buf1_pixel_data6_o   => buf1_pixel_data6   ,
  buf1_pixel_data7_o   => buf1_pixel_data7   ,
  buf1_pixel_data8_o   => buf1_pixel_data8   ,
  buf1_pixel_data9_o   => buf1_pixel_data9   ,
  buf1_pixel_data10_o  => buf1_pixel_data10  ,
  buf1_pixel_data11_o  => buf1_pixel_data11  ,
  buf1_pixel_data12_o  => buf1_pixel_data12  ,
  buf1_pixel_data13_o  => buf1_pixel_data13  ,
  buf1_pixel_data14_o  => buf1_pixel_data14  ,
  buf1_pixel_data15_o  => buf1_pixel_data15  ,
);

 u_distance_top : distance_top
  port map (
  reset_i             => rst,
  clk_i               => clk,
  data_en             => data_en,
  buffer_counter2         => buffer_counter2,
  distance_out1   => distance_out1  ,
  distance_out2   => distance_out2  ,
  distance_out3   => distance_out3  ,
  distance_out4   => distance_out4  ,
  distance_out5   => distance_out5  ,
  distance_out6   => distance_out6  ,
```

```
      distance_out7   => distance_out7  ,
      distance_out8   => distance_out8  ,
      distance_out9   => distance_out9  ,
      distance_out10  => distance_out10 ,
      distance_out11  => distance_out11 ,
      distance_out12  => distance_out12 ,
      distance_out13  => distance_out13 ,
      distance_out14  => distance_out14 ,
      distance_out15  => distance_out15 ,
      );

u_difference_top : difference_top
   port map(
     reset_i         => rst,
     clk_i           => clk,
     data_en         => data_en,
       buffer2_data_in => buffer2_data,
      buf1_pixel_data1_i   => buf1_pixel_data1   ,
      buf1_pixel_data2_i   => buf1_pixel_data2  ,
      buf1_pixel_data3_i   => buf1_pixel_data3  ,
      buf1_pixel_data4_i   => buf1_pixel_data4  ,
      buf1_pixel_data5_i   => buf1_pixel_data5  ,
      buf1_pixel_data6_i   => buf1_pixel_data6  ,
      buf1_pixel_data7_i   => buf1_pixel_data7  ,
      buf1_pixel_data8_i   => buf1_pixel_data8  ,
      buf1_pixel_data9_i   => buf1_pixel_data9  ,
      buf1_pixel_data10_i  => buf1_pixel_data10 ,
      buf1_pixel_data11_i  => buf1_pixel_data11 ,
```

```
    buf1_pixel_data12_i  => buf1_pixel_data12 ,
    buf1_pixel_data13_i  => buf1_pixel_data13 ,
    buf1_pixel_data14_i  => buf1_pixel_data14 ,
    buf1_pixel_data15_i  => buf1_pixel_data15 ,
    diff_out1          =>  diff_out1,
    diff_out2          => diff_out2 ,
    diff_out3          => diff_out3 ,
    diff_out4          => diff_out4 ,
    diff_out5          => diff_out5 ,
    diff_out6          => diff_out6 ,
    diff_out7          => diff_out7 ,
    diff_out8          => diff_out8 ,
    diff_out9          => diff_out9 ,
    diff_out10         => diff_out10,
    diff_out11         => diff_out11,
    diff_out12         => diff_out12,
    diff_out13         => diff_out13,
    diff_out14         => diff_out14,
    diff_out15         => diff_out15,);


);
  u_sorting_top : sorting_top
    port map (
    reset_i             => rst,
    clk_i               => clk,
    data_en             => data_en,
    buffer_counter2     => buffer_counter2,
```

```
  diff_in1    => diff_out1 ,
  diff_in2    => diff_out2 ,
  diff_in3    => diff_out3 ,
  diff_in4    => diff_out4 ,
  diff_in5    => diff_out5 ,
  diff_in6    => diff_out6 ,
  diff_in7    => diff_out7 ,
  diff_in8    => diff_out8 ,
  diff_in9    => diff_out9 ,
  diff_in10   => diff_out10,
  diff_in11   => diff_out11,
  diff_in12   => diff_out12,
  diff_in13   => diff_out13,
  diff_in14   => diff_out14,
  diff_in15   => diff_out15,
distance_in1    => distance_out1  ,
distance_in2    => distance_out2  ,
distance_in3    => distance_out3  ,
distance_in4    => distance_out4  ,
distance_in5    => distance_out5  ,
distance_in6    => distance_out6  ,
distance_in7    => distance_out7  ,
distance_in8    => distance_out8  ,
distance_in9    => distance_out9  ,
distance_in10   => distance_out10 ,
distance_in11   => distance_out11 ,
distance_in12   => distance_out12 ,
distance_in13   => distance_out13 ,
```

```
distance_in14   => distance_out14 ,
distance_in15   => distance_out15 ,
  m_out1                =>    m_data1            ,
  m_out2                =>    m_data2            ,
  m_out3                =>    m_data3            ,
  m_out4                =>    m_data4            ,
  m_out5                =>    m_data5            ,
  m_out6                =>    m_data6            ,
  m_out7                =>    m_data7            ,
  m_out8                =>    m_data8            ,
  m_out9                =>    m_data9            ,
  m_out10               =>    m_data10           ,
  m_out11               =>    m_data11           ,
  m_out12               =>    m_data12           ,
  m_out13               =>    m_data13           ,
  sum_out1        =>    sum_data1          ,
  sum_out2        =>    sum_data2          ,
  sum_out3        =>    sum_data3          ,
  sum_out4        =>    sum_data4         ,
  sum_out5        =>    sum_data5         ,
  sum_out6        =>    sum_data6         ,
  sum_out7        =>    sum_data7         ,
  sum_out8        =>    sum_data8         ,
  sum_out9        =>    sum_data9         ,
  sum_out10       =>    sum_data10        ,
  sum_out11       =>    sum_data11        ,
  sum_out12       =>    sum_data12        ,
  sum_out13       =>    sum_data13
```

```
);
u_calc_variogram : calc_variogram
port map (
reset_i             => rst,
clk_i               => clk,
data_en             => data_en,
m_out1                        => m_data1  ,
m_out2                        => m_data2  ,
m_out3                        => m_data3  ,
m_out4                        => m_data4  ,
m_out5                        => m_data5  ,
m_out6                        => m_data6  ,
m_out7                        => m_data7  ,
m_out8                        => m_data8  ,
m_out9                        => m_data9  ,
m_out10                 => m_data10 ,
m_out11                 => m_data11 ,
m_out12                 => m_data12 ,
m_out13                 => m_data13 ,
sum_out1                => sum_data1  ,
sum_out2                => sum_data2  ,
sum_out3                => sum_data3  ,
sum_out4                => sum_data4  ,
sum_out5                => sum_data5  ,
sum_out6                => sum_data6  ,
sum_out7                => sum_data7  ,
sum_out8                => sum_data8  ,
sum_out9                => sum_data9  ,
```

```
sum_out10                => sum_data10 ,
sum_out11                => sum_data11 ,
sum_out12                => sum_data12 ,
sum_out13                => sum_data13 ,
semivariogram_out1       => semivariogram1  ,
semivariogram_out2       => semivariogram2  ,
semivariogram_out3       => semivariogram3  ,
semivariogram_out4       => semivariogram4  ,
semivariogram_out5       => semivariogram5  ,
semivariogram_out6       => semivariogram6  ,
semivariogram_out7       => semivariogram7  ,
semivariogram_out8       => semivariogram8  ,
semivariogram_out9       => semivariogram9  ,
semivariogram_out10      => semivariogram10 ,
semivariogram_out11      => semivariogram11 ,
semivariogram_out12      => semivariogram12 ,
semivariogram_out13      => semivariogram13
);

u_output_mux : output_mux
port map (
rst                => rst,
clk                => clk,
show_expected      => SHOW_EXP,
right_button         => R_BTN,
left_button          => L_BTN,
semivariogram_1     => semivariogram1  ,
semivariogram_2     => semivariogram2  ,
```

```
    semivariogram_3    => semivariogram3  ,
    semivariogram_4    => semivariogram4  ,
    semivariogram_5    => semivariogram5  ,
    semivariogram_6    => semivariogram6  ,
    semivariogram_7    => semivariogram7  ,
    semivariogram_8    => semivariogram8  ,
    semivariogram_9    => semivariogram9  ,
    semivariogram_10   => semivariogram10 ,
    semivariogram_11   => semivariogram11 ,
    semivariogram_12   => semivariogram12 ,
    semivariogram_13   => semivariogram13 ,
    semivariogram_out => semivariogram_out,
    current_data_num => current_data_num
    );
  LED <= current_data_num & semivariogram_out(8 downto 0);

end rtl;
```