Published in final edited form as:

J Real Time Image Process. 2017 March ; 13(1): 161–180. doi:10.1007/s11554-016-0611-1.

Semivariogram Analysis of Bone Images Implemented on FPGA Architectures

Mukul Shirvaikar¹, Yamuna Lagadapati¹, and Xuanliang Dong²

¹Electrical Engineering Department, The University of Texas at Tyler, Tyler, TX 75799 USA

²Health and Kinesiology Department, The University of Texas at Tyler, Tyler, TX 75799 USA

Abstract

Osteoporotic fractures are a major concern for the healthcare of elderly and female populations. Early diagnosis of patients with a high risk of osteoporotic fractures can be enhanced by introducing second-order statistical analysis of bone image data using techniques such as variogram analysis. Such analysis is computationally intensive thereby creating an impediment for introduction into imaging machines found in common clinical settings. This paper investigates the fast implementation of the semivariogram algorithm, which has been proven to be effective in modeling bone strength, and should be of interest to readers in the areas of computer-aided diagnosis and quantitative image analysis. 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 paper 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 DXA scans are utilized for the experiments. Implementation results show that a significant advantage in computational speed is attained by the architectures with respect to implementation on a personal computer with an Intel i7 multi-core processor.

mshirvaikar@uttyler.edu, Phone: (903) 565-5620, Fax: (903) 565-5877.

Conflict of Interests: The authors declare that there are no conflicts of interest regarding the publication of this paper.

Keywords

semivariogram; FPGA; medical imaging; DXA; Bone

1. Introduction

Osteoporosis is a skeletal disease in which loss of bone mass and deterioration of bone microarchitecture cause a reduction in bone stiffness and strength, thus resulting in an increased risk of fragility fractures. Spine fractures are the most common type of osteoporotic fractures and these are a major concern for healthcare of elderly population. Therefore, early diagnosis of patients with high risk of osteoporotic fractures is essential. Dual energy X-ray absorptiometry (DXA) is currently the clinical tool of first choice for measuring bone mineral density (BMD) and for making clinical decisions of osteoporosis patients. Bone mineral density (BMD) assessed by DXA, is not sufficient to account for all spectrums of fracture risks. Current techniques for quantifying the inhomogeneity consists of descriptive statics such as mean, standard deviation and coefficient of variation. However, these parameters do not describe the spatial variation of bone properties.

Texture is an important pre-attentive cue in human and machine vision and is defined as the characteristic spatial arrangements of features which can distinguish one image from other images. Early attempts at using textural features for image classification were based on second-order statistics derived from images [1]. There are two types of textures 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. Several probability models for textures have gained wide acceptance, because of their modelling power and expressiveness [2]. 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 [3]. In particular, Markov random fields (MRFs) describe a texture in terms of spatial geometry and quantitative strengths of inter-pixel statistical dependency [4]. The semivariogram is a method that can be applied to two dimensional plain-projection images and is based on Markov random fields (MRF) [5]. Texture analysis can also be utilized to investigate bone microarchitecture and recently, applications in the area of medical imaging have been investigated [6]. In this paper, we reported on novel statistical methods for the estimation of bone quality to predict the risk of fracture. This is significant because current methods such as DXA scans do not provide such a measure and other modalities are impractical due to cost and access considerations.

Semivariogram analysis is a computationally intensive statistical algorithm that has typically seen applications in the geosciences and remote sensing areas [7,8]. It can be used as a descriptor of second-order statistics within the image and hence provide a quantitative measure of texture. Stochastic parameters such as correlation length/range, nugget and sill calculated from the semivariogram plot can be used to represent spatial variations in bone images [9]. In this study, a novel stochastic method is proposed to describe the spatial variation of bone properties by quantifying the map of BMD derived from 2D images. For

example, the correlation length describes the degree of smoothness or roughness in the map. A relatively larger correlation length implies a smooth variation or a weak bone, whereas a smaller correlation length corresponds to acute changes over the spatial domain or a stronger bone. However, the algorithm is computationally intensive resulting in the need for efficient real time implementation for user adoption. Once the fast computation of the intensive second-order statistical texture measures is achieved, it can be introduced into DXA machines. This will provide major advantages since, it would be cost effective as most hospitals already have DXA machines and there would be no need for purchasing new equipment. Further, it would be convenient for patients and doctors because of the ready availability of this equipment.

In many hardware implementations, it is desirable to optimize performance in terms of speed, area, and power dissipation. Reconfigurable platforms such as Field Programmable Gate Arrays (FPGAs) have been investigated for implementing many image processing algorithms [10]. An FPGA consists of an array of reconfigurable logic cells and advances in technology and development tools have made it possible to implement complex algorithms on FPGAs [11]. Because FPGA designs can be optimized for a given application, they often have superior performance in terms of speed and power dissipation over generic integrated circuit designs and microprocessor-based implementations. FPGA systems have been demonstrated to have higher throughput compared with DSP and general-purpose microprocessor-based system designs [12]. For this reason, FPGAs are increasingly being used in areas formerly dominated by application specific integrated circuits (ASICs), such as embedded system design and digital image processing applications. In this paper, the potential for using FPGAs to implement the semivariogram is explored.

A Field-Programmable Gate Array (FPGA) is a reconfigurable device with the ability of being reprogrammed to satisfy particular requirements [13]. The FPGA board used for this implementation is the Xilinx *XUPV5-LX110T* Development Kit, which utilizes the *Virtex 5* FPGA. Some preliminary results for semivariogram algorithm implementation on the above FPGA target were published limited to smaller image sizes [14]. This paper extends the published architecture to handle larger image sizes and provides some insight into the scalability of the approach. The image size was increased until all resources on the board were consumed. Resource utilization and speedup results are presented for all the architectures and details of the submodule implementations are also provided.

2. Previous Work

This section provides a review of literature related to computational speedup of the semivariogram. The literature survey reveals only one proposed *theoretical architecture* for the fast implementation of semivariogram computations on an FPGA, but *no implementation that have been realized in hardware*.

Wielgosz *et al* have proposed a theoretical FPGA architecture for variogram computation as a part of an overall scheme for Kriging image interpolation [15]. The variogram computation component consists of a set of BRAMs or distributed memory blocks that are loaded with the image data, an approach that can lead to large memory consumption. Image pixel data is

extracted in pairs and presented to a number of identical processing streams that consist of adders and multipliers for distance and difference computations. Each processing stream is theoretically estimated to have a latency of 10 clock cycles not counting the other stages. A set of multiplexers then sorts the values into accumulator bins for further processing. The architecture is not optimal in terms of processing stream utilization due to uneven distribution of the number of pixel pairs for a particular fixed lag distance. Load balancing of the pipelines is suggested as a solution for this problem, including a theoretical solution that involves time-multiplexing the data streams. Data throughput is shown to decrease rapidly as a function of image size and computation time decreases exponentially with a large number of processing streams. Assuming that a sufficient number of processing streams are available the processing time for the raw data computation is theoretically shown to be $0.1 \mu s$, assuming the FPGA is clocked at 100 MHz, for any image size for only the variogram computation part. This projected time does not count the preliminary data presentation stage and post processing stage and the computation of variogram matrix is assumed to be made offline and uploaded to the FPGA.

Marcotte proposed fast computation of variograms based on the Fast Fourier Transform (FFT) algorithm as opposed to the spatial domain [7]. The algorithm is implemented in Matlab to compute direct, cross, and pseudo-cross variograms and is flexible enough to account for any missing data in the matrix. The FFT approach is shown to be faster by roughly an order of magnitude especially for larger image sizes of 256×256 pixels. 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 approach is particularly effective especially when repetitive variogram computations are required. The implementation on FPGA is worth considering in the future, but would require floating point arithmetic capability.

In the absence of other significant literature on the topic, it is worth examining other statistical approaches such as the GLCM that also involve comprehensive pixel-pair data extraction and some of which have been implemented utilizing FPGAs. Most work appears to focus on the fast computation of gray-level co-occurrence matrices (GLCM). The 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 co-occurrence matrices [16]. The implemented architecture is limited to four directions and four distances and symmetrical matrix computations, in view of the computational time required. The architecture calculates GLCMs in parallel using dual-port input block RAMs (BRAM) for the pixel data mapped to BRAMs for each co-occurrence matrix. 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. The processing time for simultaneous computation of sixteen co-occurrence matrices for a 64×64 image with 128 gray levels is in the order of $100 \mu s$ on a *Virtex 5* platform running at 207 MHz. A major disadvantage of the approach is that it is limited by the number and size of BRAM blocks available on the FPGA chip.

Iakovidis *et al* implemented a FPGA architecture for fast parallel computation of co-occurrence matrices [17]. The design utilizes an array of co-occurrence matrix computational units that also accumulate the results. As mentioned earlier, symmetric co-occurrence matrices are considered for practical purposes and the distribution between opposite directions is ignored. The design choices considered were FPGA Block RAM arrays, standard sparse array structures that store pairs of indices and values, and set-associative sparse arrays. BRAM arrays were not used since they lead to a larger area utilization and standard sparse arrays were not used as they result in a low throughput, as the cycles needed to traverse the indices of the array are proportional to its length. Set-associative arrays were used as a flexible alternative with low design complexity. Each computational unit consists of an n -way set-associative array and n comparators, a priority encoder and an adder. Each set-associative array uniquely maps an input pair of grey-level intensities into an address of the data array which are implemented using BRAMs, each of which can hold the co-occurrence matrix elements. The architecture was implemented on a *Xilinx Virtex2-XC2V6000-6* operating at 83MHz with an implementation time of $200\mu\text{s}$ for a 64×64 image with 32 gray levels.

Haralick [18] texture feature extraction algorithms can be implemented in two parts: computing the co-occurrence matrices followed by calculation of texture features using them. Akoushideh *et al* proposed a parallel FPGA architecture for this purpose [19]. In the proposed architecture, first, the co-occurrence matrices are computed then thirteen texture features are calculated in parallel. The architecture was implemented on a *Virtex 5 fx130T-3* FPGA device operating at 250MHz with an implementation time of $48\mu\text{s}$ for a 64×64 image with 256 gray levels to compute only the GLCM matrices.

Girisha *et al*, presented an FPGA architecture for Gray Level Co-occurrence Matrix (GLCM) using Verilog [20]. The GLCM is computed for four different angles 0° , 45° , 90° and 135° for a given intensity of an image using finite state machines. The method has been extended for texture feature extraction from video frames for 8×8 sub-images with eight gray levels [21].

In summary, there are a number of approaches that have been implemented to compute second-order statistics from images but they are mostly limited to GLCMs due to their popularity. Only one of the approaches directly addresses semivariogram computation for Kriging applications with a suggested theoretical architecture. In this paper, some ideas from this approach have been utilized to realize a working architecture on a FPGA platform and is described in detail.

3. The Semivariogram Algorithm

This section provides an overview of semivariogram based analysis. The semivariogram displays the average change of a property and relation between a pair of pixels with changing lag (the distance between a pair of pixels). The semi-variance typically increases with lag distance and converges to the sum of the nugget variance and the sill variance when the separation distance (h) approaches infinity. The 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 physical distance between the two locations $(x_i, x_i + h)$ in an image 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 and characteristics are specific to the application domain under consideration. The experimental variogram is calculated by averaging one half the differences squared of the pixel values $z(x_i)$ 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:

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

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 of h 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 few constraints for a semivariogram model to be able to represent typical measured data. 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.

Figure 1 shows the plot of experimental semivariogram with the stochastic parameters marked. Range, sill and nugget variance are the stochastic parameters of the semivariogram. 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 of the bone mineral density over the spatial domain.

The semivariogram plots obtained from real image data usually differ slightly from the theoretical typical plot shown at the top of the figure. Curve fitting is usually attempted using several types of models based on imaging domain and the exponential and hole-effect models are popular. The value that the semivariogram model attains at the range (the value on the y-axis) is called the sill. It can be used to refer to the “amplitude” of certain component of the semivariogram. The sill of the variogram represents the variance of the BMD map.

According to theory, the semivariogram value at the zero separation distance ($\text{lag} = 0$) should be zero. However, at an infinitesimally small separation distance, the semivariogram often exhibits a nugget effect, which is some value greater than 0. The nugget represents variability at distances smaller than the typical sample spacing. Variation at micro scales smaller than the sampling distances will appear as part of the nugget effect. It represents the sum of noise in a image and measurement errors of calculations.

Figure 2 shows DXA images for hip and spine from patients. The images are obtained using a Hologic QDR Discovery Dual energy X-ray absorptiometry machine shown, operating in a fan beam mode [22]. Typical spatial resolution per pixel for DXA images is in the $1 - 10\text{mm}$ range, while the typical resolution for microCT images is in the $50 - 300\mu\text{m}$ range [23]. The data for the experiments were sub-images cropped from the hip image. For algorithmic implementation purposes however, the actual image data used does not matter as long as the results are verified with respect to a baseline algorithm. A semivariogram value can be calculated for a lag distance and angle combination in the image under consideration. As long as the displacement vector formed by the lag distance and angle fits within the image window, with the pixel pairs forming a dipole at the ends of the vector, a semivariogram value can be computed. If the angle is fixed and the semivariogram is plotted for different possible distances, then it provides some directional information. Such a semivariogram is considered to be directional or anisotropic. More commonly the variogram values are computed for each distance value by averaging the values obtained for different angles from $0^\circ - 360^\circ$ for the same distance value. Such a variogram is considered to be isotropic in nature and is used for bone image analysis.

For practical considerations from a real-time algorithmic point-of-view, it is easier to have a single algorithm consider all possible pixel pairs within a window. Each difference value will have a distance value associated with it equal to the Euclidian distance between the pixels (lag distance). Based on the lag distance information, the difference value can then be assigned for computation of a particular variogram for that lag distance. In practice this can be implemented as an accumulator to achieve the summation of equation (1). A count of the number of pixel pairs for each lag distance corresponding to m in the equation, would have to be maintained and would vary based on the lag distance. This implementation of the isotropic algorithm has the advantage of going through the pixel pair data only once to compute the entire graph. The computational complexity is still $O(n^2)$ where n is the number of pixels in the image region, but multiple passes through the image for each lag distance are avoided, and parallel implementation is now possible.

Table 1 shows the computational complexity of the semivariogram algorithm for the isotropic computation case. The computation of the semivariogram value for a particular distance and angle is non-isotropic. The algorithm implemented in this paper calculates values for each distance value by averaging the values for the angles and hence referred to as isotropic in nature. From a practical perspective each pixel pair has a distance and angle vector separating them and within an image there will exist only $\sqrt{(M-1)^2 + (N-1)^2}$ possible integer values of lag distance. An accumulator bin is created for each integer lag distance value. The semivariogram value for each pixel pair is accumulated into the lag

distance bin that is closest to the distance between the pixels in the pair under consideration. Consequently, values for all possible angles corresponding to a particular lag distance are accumulated and averaged by the isotropic algorithm. Thus, for $P = M \times N$ pixels in the image there are $P(P-1)/2$ unique pixel pairs. Calculation of the variogram value includes the computation of the coordinate distances and gray level differences between all the pixel pairs, and summation of the squares of differences. Each Euclidian distance calculation has $3 \times (P(P-1)/2)$ additions and $3 \times (P(P-1)/2)$ multiplications (squares and square roots are counted as one multiply operation). The calculation of the differences between the pixel values in each pair involves $P(P-1)/2$ subtractions and $P(P-1)/2$ multiplications. Calculation of the final variogram involves $P(P-1)/2$ additions. Hence, the variogram value can be computed with $5 \times (P(P-1)/2)$ sums and $2 \times (P(P-1)/2)$ multiplications each. The maximum value of the lag distance h can be computed to be $\sqrt{(M-1)^2 + (N-1)^2}$. The computational complexity of the variogram calculation can be shown to be $O(P^2)$, where $P = M \times N$ is the number of pixels in the image region.

For example, for $P = M \times N$ ($M = 10$, $N = 10$), the total number of pixel pairs is 4,950 assuming distinct pixels, and the total number of summations required would be 24,750 and the total number of multiplications would be 19,800. Similarly for $P = M \times N$ ($M = 20$, $N = 20$), the total number of pixel pairs is 79,800 and the total number of summations 399,000 and multiplications are 319,200. As observed, the values are particularly large even for 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, especially in DXA images, involve regions of interest with around 50×50 pixels, resulting in exponentially larger number of computations: approximately 15.6 million sums and 12 million multiplications.

4. FPGA Implementation

This section describes the architectures that were designed for the fast computation of semivariograms on the FPGA. 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 distinct 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.

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 [24]. 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 [25]. 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 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.

4.1 Architecture Version 1 (Not Pipelined)

The block diagram and description of the basic architecture is shown in Figure 4. Initially the image pixel values are loaded into the image buffers as data vectors. If the image is of size $N \times N$ then the pixel vector is of size N^2 (for the experiments $N = 10$ and 20 have been used). Then the counters for the two image buffers start incrementing when the data enable is high, similar to nested “for loops”, and the image data is loaded into the difference module. Counter 1 starts from 1 and increments every time the second counter counts up to the index N^2 of the last pixel in the data vector and rolls back over to 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, respectively. 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.

4.2 Architecture Version 2 (Pipelined)

Since the pixel data loading process can be time-consuming, the second architecture addresses this process in a brute force fashion. Figure 6 shows the block diagram for the architecture version 2 which is pipelined heavily 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). The pixel values and location index data are replicated directly in each sub-module. Each of the replicated distance sub-modules computes the Euclidean distances and each difference sub-module computes the absolute difference between pixel pairs. The image buffer for the pipelined architecture has just one multiplexer which presents the data to the difference sub-modules sequentially. 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. Figures 5 and 6 show the block diagram of image buffers for the two architectures. The distance and difference sub-modules perform

calculations for one fixed pixel with reference to all the other loaded pixel values. For example, the first distance and the difference modules calculate distances and differences between pixel 1 and pixels 1 to N^2 , the second distance and the difference modules calculate distances and differences between pixel 2 and pixels 2 to N^2 and so on. The first distance and the difference modules start computations as soon as the pixel values are replicated and pixel 1 is presented to it and the second sub-modules does the same without waiting for the first module to complete.

The sorting module has the same functionality as the first architecture version, with the exception that it has 33 replicated sub-modules for the 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 distance and difference values are handled in each of the 25 sorting modules in the 20×20 image version. The outputs of each sorting module consists of the number of pixel pairs (m) and the sum of the semivariances for each lag distance, which is then passed to the variogram calculation module for final calculations.

A detailed description of each block is provided in the following sections.

4.3 Distance Module

The distance module calculates the Euclidian distance between two pixels in a pixel pair. The pixel index values or positions in the image vector are stored in the internal registers of the distance module. Based on image dimension information, and 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 internal to the distance module or sub-module. The distance between pixels in the pair can then be computed using the simple Euclidian equation. 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.

Figure 9 and Figure 10 illustrate the distance module for architecture 1 and 2 respectively. In the architecture version 2, the distance module is replicated 99 times for the 10×10 image and 399 times for the 20×20 image, with each module performing 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 pixels till the last one N^2 . In the meantime, the second distance module starts calculating the distance from pixel 2 to pixels 2, ..., N^2 . Hence all the sub-modules calculate the distance between the two pixels in a 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. The module is designed such that duplicated pixel values pairs are skipped to reduce computational time.

4.4 Difference Module

The difference module handles computation of the absolute difference between two pixel gray levels. In architecture 1, the pixel value from image buffer multiplexer 1 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 image buffer multiplexer 2 into the pixel 2 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 11 and Figure 12 illustrate the difference module for architecture 1 and 2 respectively.

In the architecture version 2, 99 modules for the 10×10 image and 399 modules for the 20×20 image are replicated. Each pixel value of the image buffer is loaded into its particular module whereas the first image buffer is directly replicated in all the modules. The first module calculates the difference between the first pixel value and the values of all pixels up to the last one N^2 . At the same time, module 2 calculates differences of values for pixels 2 to N^2 once it gets the second pixel value from the image buffer multiplexer, and so on. All the modules hence start calculating differences in a parallel fashion. Although the previous modules are calculating differences between the pixel pairs, the pixel 2 values are loaded and the corresponding submodules start calculating the differences. As soon as the single pixel difference value has been computed in the sub-modules, it is sent as the input to the sorting module without waiting for all the calculations to be completed. Similar to the distance module, the pixel values with duplicated computations are skipped to reduce computational time.

4.5 Sorting Module

After calculating the lag distances and the differences between the pixel pairs, the outputs from both modules are connected to the sorting module. This module sorts the difference values by keying off the lag distances to compute the summation of the square of differences for each lag distance. The range starts from 1 to the maximum distance between the pixel pairs. The sorting module also calculates the number of pixel pairs for each distance. Figure 13 and Figure 14 illustrate the sorting module for architecture 1 and 2 respectively.

The internal registers which act as accumulators, check the input distance values 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 add 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 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.

Implementation 2 performs the same sorting and summation processes to find total sum and number of pixel values for each lag distance. However in this implementation, 33 sub-modules (for the 10×10 image) and 25 (for the 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 and output from three difference sub-modules. Similarly 25 sub-modules in the 20×20 image handle 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 399 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. The number of pixel values (m) and summation values acts as the input to the final module for calculation.

4.6 Variogram Calculation Module

The final stage is the same for both 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 15 illustrates the variogram calculation module.

5. Performance Evaluation

This section evaluates the semivariogram implementation in terms of the speedup, hardware resources utilized, throughput, and test results. Initially a MATLAB program was used to implement the algorithm and find the baseline variogram values. The baseline values were utilized to ensure that the FPGA algorithms are operating correctly and to evaluate their accuracy. The algorithm was also implemented in C on a computer with *Intel Core i7 CPU 920 @ 2.67GHz X 8* in order to obtain speed comparisons. The operating system was *Ubuntu Linux running 14.04 LTS* and the compiler used was *gcc (Ubuntu 4.8.4-2ubuntu1~14.04.1) 4.8.4* with *-O3*, the highest level of optimization available. Figure 16 shows a plot of the computational time required for different values of N . It can be seen that the implementation time increases exponentially with respect to N .

The design was implemented on the FPGA board for a 10×10 image and a 20×20 image. It should be noted from the hardware resource utilization figures in Table 6 that the percentage of hardware consumed by the architecture increased to the point where a 25×25 implementation did not fit on the board. This is because for the architecture the size increases based on image size. This has been noted as an area of investigation for future

refinement of the design. Both the architectures and the results obtained are presented and discussed in this section, to highlight the evolution of the design and the thought process involved.

5.1 Implementation results for Architecture Version 1 for a 10×10 image

The first architecture was implemented without any pipelining or other enhancements, in order to benchmark the feasibility of implementation. The results show that the speed is measured to be 400 microseconds for calculating the semivariogram values using this preliminary architecture. 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 $4 \times 10\text{ns} = 40\text{ns}$. There are two counters, each counting up to 100, when counter 1 is 1, counter 2 goes from 1 to 100, resulting in 100×100 counts. So approximately the total time of computation is $40\text{ns} \times 100 \times 100 = 400,000\text{ns}$ or $400\mu\text{s}$. The results obtained are very close to these theoretical values. This architecture was a straight linear implementation of the sequential algorithm in software and it is observed that the FPGA implementation is slower than the C code. This is not surprising since the FPGA can operate only at the maximum frequency of 100 MHz.

5.2 Implementation results for Architecture Version 2 for a 10×10 image

Architecture Version 2 is 100 times faster in speed than the first one due to pipelining. As discussed before, each count 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 $100 \times 40\text{ns} = 4000\text{ns}$ or $4\mu\text{s}$, but due to the small logic delay the computational time is $4.27\mu\text{s}$. The results appear to match well with the theoretical pipelined architecture presented in the literature, especially since most of the time appears to be due to the pixel pair data presentation time and post processing rather than for the multiple parallel processing streams implemented [15]. The number of processing streams in our implementation is limited by the FPGA resources available on the chip, since other parts of the system such as data presentation and post-processing were also realized. Table 2 shows the results for the FPGA implemented architecture. The FPGA board does not support unlimited increase in the number of modules in the initial stage. As a result, it is envisaged that for larger image sizes ($N > 10$) the data is processed in sequence through the distance and difference modules whereas the subsequent sorting and variogram modules are modified to handle the larger number of distances possible. The FPGA implementation speeds for larger image sizes are projected based on the number of times ($N^2/100$) sets of 100 pixel pairs can be processed through the system sequentially ($N^2/100$) times, with each set consuming $4.27\mu\text{s}$. The time required for loading image pixel data is assumed to be minimal due to fast data transfer from image memory. The speedup obtained appears to be consistent as seen in Table 3.

5.3 Implementation results for Architecture Version 2 for a 20×20 Image

In an attempt to speed up the system even further, architecture version 2 for a 20×20 image was designed with 399 distance and 399 difference modules and 16 sorting modules for 400

pixel values. Due to the increase in the number of modules and complexity in the circuit (by a multiple of 4 times), the implementation time has increased 16 times. Sorting modules have to be reduced because of the increased resource utilization by the distance and difference sub-modules, which is also one of the reasons for the increase in the computational time. Here, each buffer count lasts for 16 clock cycles. As discussed, each clock cycle is $10ns$ for $100MHz$, hence a counter increment takes $16 \times 10 ns = 160 ns$. But there is only one counter counting from 1 to 400, as the sorting module is directly connected to inputs from the difference module for parallel calculation. The total time of computation is approximately $400 \times 160ns = 64000ns$ or $64\mu s$. Due to the small logic delay, the actual computational time is measured to be $64.9\mu s$.

Table 2 shows the computation times for the C 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 pipelined FPGA version. Again, the FPGA board does not support unlimited increase in the number of modules in the initial stage. As a result, it is envisaged that for other image sizes ($N > 10$) the data is processed in sequence through the distance and difference modules whereas the subsequent sorting and variogram modules are modified to handle the larger number of distances possible. The FPGA implementation speeds for larger image sizes are projected based on the number of times ($N^2/400$) sets of 400 pixel pairs can be processed through the system sequentially ($N^2/400$) times, with each set consuming $64.9\mu s$. The time required for loading image pixel data is assumed to be minimal due to fast data transfer from image memory. The speedup obtained appears to be consistent as seen in Table 3.

Table 3 gives the summary of speed up obtained in the FPGA hardware implementation when compared to the software. Figure 16 provides a plot of the computational times for different image sizes using C code and the FPGA. It can be observed that the projected FPGA implementation times and corresponding speedup values are consistent for both the architectures, with only a marginal speedup obtained by changing the number of modules in the FPGA 20×20 architecture.

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 lag value of ' h '. Table 4 and Table 5 compare the semivariogram values obtained from the baseline algorithm to the implementation results from the FPGA for the two image sizes.

A summary of the synthesis report, device requirement and utilization is shown in Table 6. 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.

6. Conclusions

Osteoporotic fractures are a major concern for the healthcare, especially for elderly and female populations and early diagnosis of patients with a high risk of osteoporotic fractures is desirable. It has been shown that such diagnosis can be enhanced by introducing second-order statistical analysis of bone image data using techniques such as variogram analysis. Such analysis is computationally intensive thereby creating an impediment for introduction into imaging machines found in common clinical settings. This paper investigates the fast implementation of the semivariogram algorithm, which has been proven to be effective in modeling bone strength, and should be of interest to readers in the areas of computer-aided diagnosis and quantitative image analysis. In this paper, custom FPGA architectures are implemented to reduce the computational time required. Architecture version 1 (non-pipelined) and two embodiments of the Architecture version 2 (pipelined) are implemented. The results show that the second FPGA implementation with the pipelined architecture can improve the computation times significantly, as compared to the software implementation. The architecture implements massive parallelism in the form of replicated distance, difference and sorting modules that are further pipelined. The architecture described in this paper can be utilized for small matrices, but scalability of the design to larger images just by replication of modules imposes a challenge due to FPGA resource limitations. This challenge can be overcome by dividing the image into parts and sequencing it through the design. The current design also rounds off data results to the nearest integer. Future work can involve increasing the efficiency of the pipelined architecture to handle larger image sizes and floating point numbers to provide precision up to two decimal values. These issues will be addressed in ongoing and future work on the project.

The ultimate goal is for user adoption by introduction into machines such as the DXA which are found commonly. One possible way is to provide the semivariogram based bone quality indicator module as an optional hardware add-on that can read image data from the machine and output diagnostic data to help the medical examiner. It can be envisaged that such a clinical setup would involve the medical personnel panning over the image or selecting sub images of interest with the bone statistics being displayed in a real-time fashion to aid diagnosis and focus on areas of concern. This would be cost effective as most hospitals already have DXA machines and there would be no need for purchasing new equipment. Further, it would be convenient for patients and doctors because of the economy of DXA scans compared to higher resolution modalities such as the microCT scan. It can also be extrapolated that the technique can be extended to other modalities to assist medical personnel in increasing their diagnostic efficacy.

Acknowledgments

This study was financially supported by the NIH/NIAMS under award number R15AR061740.

References

1. Haralick RM, Shanmugam K, Dinstein IH. Textural Features for Image Classification. Systems, Man and Cybernetics, IEEE Transactions on. 1973; SMC-3(6):610–621. DOI: 10.1109/TSMC.1973.4309314

2. Cootes T, Taylor C. Anatomical statistical models and their role in feature extraction. 2014
3. Baraldi A, Parmiggiani F. An investigation of the textural characteristics associated with gray level cooccurrence matrix statistical parameters. *Geoscience and Remote Sensing, IEEE Transactions on*. 1995; 33(2):293–304.
4. Kervrann C, Heitz F. A Markov random field model-based approach to unsupervised texture segmentation using local and global spatial statistics. *Image Processing, IEEE Transactions on*. 1995; 4(6):856–862.
5. Carr JR, de Miranda FP. The semivariogram in comparison to the co-occurrence matrix for classification of image texture. *Geoscience and Remote Sensing, IEEE Transactions on*. 1998; 36(6):1945–1952. DOI: 10.1109/36.729366
6. Dong XN, Shirvaikar M, Wang X. Biomechanical properties and microarchitecture parameters of trabecular bone are correlated with stochastic measures of 2D projection images. *Bone*. 2013; 56(2): 327–336. DOI: 10.1016/j.bone.2013.05.023 [PubMed: 23756232]
7. Marcotte D. Fast variogram computation with FFT. *Computers & Geosciences*. 1996; 22(10):1175–1186. doi:[http://dx.doi.org/10.1016/S0098-3004\(96\)00026-X](http://dx.doi.org/10.1016/S0098-3004(96)00026-X).
8. Bohling G. Introduction to Geostatic and Variogram Analysis. 2005
9. Dong XN, et al. Random field assessment of inhomogeneous bone mineral density from DXA scans can enhance the differentiation between postmenopausal women with and without hip fractures. *Journal of biomechanics*. 2015; 48(6):1043–1051. [PubMed: 25683520]
10. Nelson, AE. Implementation of image processing algorithms on FPGA hardware. Vanderbilt University; 2000.
11. Akila G, Gregory DP, Lee W G. A Pipelined and Parallel Architecture for Quantum Monte Carlo Simulations on FPGAs. *VLSI Design*. 2010; 2010
12. Rose, S. Development of parallel Image Processing architecture in VHDL. University of West Australia; 2012.
13. Pedroni, VA. Circuit design and simulation with VHDL. The MIT Press; 2010.
14. Lagadapati, Y., Shirvaikar, M., Dong, X. IS&T/SPIE Electronic Imaging. International Society for Optics and Photonics; 2015. Fast semivariogram computation using FPGA architectures; p. 940005-940005-940010.
15. Wielgosz M, Panggabean M, Rønningen LA. FPGA Architecture for Kriging Image Interpolation. *International Journal of Advanced Computer Science and Applications (IJACSA)*. 2013; 4(12)
16. Roumi, M. MSc Thesis. Faculty of Electrical Engineering, Mathematics and Computer Science; 2009. Implementing Texture Feature Extraction Algorithms on FPGA, The Netherlands.
17. Iakovidis DK, Maroulis DE, Bariamis DG. Fpga architecture for fast parallel computation of cooccurrence matrices. *Microprocessors and Microsystems*. 2007; 31(2):160–165.
18. Haralick RM. Statistical and structural approaches to texture. *Proceedings of the IEEE*. 1979; 67(5):786–804.
19. Akoushideh A, Shahbahrami A, Maybodi BN. High performance implementation of texture features extraction algorithms using FPGA architecture. *J Real-Time Image Proc*. 2014; 9(1):141–157. DOI: 10.1007/s11554-012-0283-4
20. Girisha A, Chandrashekhar M, Kurian DM. FPGA implementation of GLCM. *International Journal of Advanced Research in Electrical, Electronics, and Instrumentation Engineering*. 2013; 2(6)
21. Girisha A, Chandrashekhar M, Kurian M. Texture Feature Extraction of Video Frames Using GLCM. *International Journal of Engineering Trends and Technology*. Jun; 2013 4(6):2718–2721.
22. Pinninti, R. Stochastic Assessment of Bone Fragility in Human Lumbar Spine. University of Texas at Tyler; 2015.
23. Dougherty, G. Medical image processing: techniques and applications. Springer Science & Business Media; 2011.
24. Daemon, I. Crossbar Switch Architecture. 2012. <http://www.inetdaemon.com/tutorials/networking/lan/switching/architectures/crossbar.shtml>
25. Wada, K. Ring Buffer Basics. 2013. <http://www.embedded.com/electronics-blogs/embedded-roundtable/4419407/The-ring-buffer>

Biographies

Mukul Shirvaikar is currently Professor of Electrical Engineering at the University of Texas at Tyler. He received his doctorate from the University of Tennessee, M.S. degree from the University of Maine, and a B.Tech. in Electrical Engineering from Banaras Hindu University, India. His current research interests include medical image processing and real-time systems.

Yamuna Lagadapati received the M.S. degree in Electrical Engineering from the University of Texas at Tyler in 2015. Shee obtained her B.S. degree in Electronics Engineering from Jawaharlal Nehru Technological University, India. She is currently specializing in the information technology industry.

Xuanliang Dong is currently Assistant Professor of Health and Kinesiology at the University of Texas at Tyler. He received his doctorate from Columbia University, M.S. degree from the Tsinghua University, China and a B.S. in Mechanical Engineering from Huazhong University, China. His current research interests include biomechanics and biological systems engineering.

Figure 1. Characteristics of a typical semivariogram (top) and theoretical exponential model (left) and hole-effect model (right) fitted over the experimental variogram of the bone mineral density map

Figure 2. Medical DXA images (left) hip bone scan used for experiments (top right) DXA machine (bottom right) lumbar vertebra scan

Figure 3. Block Diagram of Counter Module (Not Pipelined)

Figure 4. Block diagram of Architecture Version 1 (Non Pipelined)

Figure 5. Block Diagram of Counter Module (Pipelined)

Figure 6. Block diagram of Architecture Version 2 (Pipelined)

Figure 7. Block Diagram of Image Buffer Module (Not Pipelined)

Figure 8. Block Diagram of Image Buffer Module (Pipelined)

Figure 9. Block Diagram of Distance Module (Not Pipelined)

Figure 10. Block Diagram of Distance Module (Pipelined)

Figure 11. Block Diagram of Difference Module (Not Pipelined)

Figure 12. Block Diagram of Difference Module (Pipelined)

Figure 13. Block Diagram of Sorting Module (Not Pipelined)

Figure 14. Block Diagram of Sorting Module (Pipelined)

Figure 15. Block Diagram of Variogram Module

Figure 16. Computational times plotted for different image sizes

Table 1
Number of Computations for Variogram Calculation for $P = M \times N$ Image

Operation	Number of Computations for $P = M \times N$ image
Unique pixel pairs	$P(P-1)/2$
Distance coordinate additions/subtractions	$3 \times (P(P-1)/2)$
Euclidian distance squares/roots	$3 \times (P(P-1)/2)$
Differences between pixel values	$P(P-1)/2$
Squares of differences	$P(P-1)/2$
Sum of squared differences	$P(P-1)/2$
Total number of additions	$5 \times (P(P-1)/2)$
Total number of multiplications	$2 \times P(P-1)$

Table 2

Computational Time Comparison for C Code, Non-Pipelined and Pipelined Architecture Versions of Implementations.

Implementation	Computational Time
C code for 10×10 image	112.9 μs
C code for 20×20 image	571.2 μs
Architecture version 1 (non-pipelined) for 10×10 image	400 μs
Architecture version 2 (pipelined) for 10×10 image	4.27 μs
Architecture version 2 (pipelined) 20×20 image	64.9 μs

Table 3
Summary of results for the pipelined architecture with comparison to C code

N	C Code Time (us)	FPGA 1 10 × 10 Time (us)	Speedup	FPGA 2 20 × 20 Time (us)	Speedup
10	188.4	4.3	44	4.1	46
20	3122.1	68.3	46	64.9	48
30	15902.6	345.9	46	328.6	48
40	50239.7	1093.1	46	1038.4	48
50	123428.6	2668.8	46	2535.2	49
60	256256.4	5533.9	46	5256.9	49
70	475256.3	10252.3	46	9739.1	49
80	810700.8	17489.9	46	16614.4	49
90	1299570.9	28015.5	46	26613.1	49
100	1979765.8	42700.0	46	40562.5	49

Table 4

Verification of the Semivariogram Values Computed for a 10×10 Image.

Table 5

Verification of the Semivariogram Values Computed for a 20×20 Image.

Lag Distances	Baseline Algorithm	FPGA Implementation
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	531.00	535
20	504.35	505
21	472.55	482
22	422.67	423
23	382.68	384
24	306.77	312
25	218.00	218
26	92.50	136

Table 6
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