Real-Time Neural Network Inversion on the SRC-6e Reconfigurable Computer

Russell W. Duren, *Senior Member, IEEE*, Robert J. Marks II, *Fellow, IEEE*, Paul D. Reynolds, *Student Member, IEEE*, and Matthew L. Trumbo, *Student Member, IEEE*

***Abstract*—Implementation of real-time neural network inversion on the SRC-6e, a computer that uses multiple field-programmable gate arrays (FPGAs) as reconfigurable computing elements, is examined using a sonar application as a specific case study. A feedforward multilayer perceptron neural network is used to estimate the performance of the sonar system (Jung *et al.*, 2001). A particle swarm algorithm uses the trained network to perform a search for the control parameters required to optimize the output performance of the sonar system in the presence of imposed environmental constraints (Fox *et al.*, 2002). The particle swarm optimization (PSO) requires repetitive queries of the neural network. Alternatives for implementing neural networks and particle swarm algorithms in reconfigurable hardware are contrasted. The final implementation provides nearly two orders of magnitude of speed increase over a state-of-the-art personal computer (PC), providing a real-time solution.**

***Index Terms*—Field-programmable gate arrays (FPGAs), inverse problems, neural network hardware, particle swarm theory, real-time systems, reconfigurable architectures, sonar.**

I. Introduction

INVERSION of large feedforward neural networks [4] has found application in numerous areas [5], including electromagnetic surface design [6], flight control [7], neural network training [8] assessing the vulnerability of large scale power systems [9], [10], parameter estimation in remote sensing [11], acoustic estimation [12], magnetotelluric data analysis [13], and optimization of sonar performance [9]. For large neural networks, the computational intensity of inversion can prohibit real-time application. The speed of field-programmable gate arrays (FPGAs) can be used to remedy this problem.

Determination of underwater sonar system parameters to provide the best possible ensonification performance at a given location in an environmentally complex water column is a computationally intense inverse problem. Here is the problem we consider. A surface ship dips a sonar unit into the water like a teabag. The depth of the sonar unit is an example of a parameter that can be controlled. The environmental parameters cannot. These include wind speed (surface roughness), bathymetry (the shape of the seafloor), bottom type, and sound velocity as a function of depth. Both the control and environmental parameters determine the effectiveness of the sonar.

The performance of sonar at each point in the water column is determined by the *signal-to-interference ratio* (SIR) defined as the ratio, in decibels, between the wanted signal power and the interference in the channel. We consider the case where ensonification, measured by the SIR, is evaluated on a sagittal plane in the water column.[1] The SIR is determined on pixels in the plane. This simple forward problem, when done using an acoustic emulator such as the Applied Physics Laboratory (APL, the University of Washington, Seattle, WA) sonar simulator [5] is itself computationally intensive. For this reason, data gathered over a long period of time from the emulator was successfully used to train an artificial neural network which, in comparison, generates the SIR profile almost instantaneously. Details of the training of the feedforward neural network are available from [1], [2], [5], and [12]. Our goal is to implement the neural networks described in these works, along with a particle swarm optimization (PSO), on one or more FPGAs. The PSO is used to invert the neural network to search for the set of inputs to the network that achieve a desired output. There are other approaches to perform inversion of a neural network [14], [5]. PSO, however, has been shown to be a highly effective search algorithm for a wide class of problems [14] and has worked well for inversion of neural networks [12].

For an inverse problem, an area in the water column corresponding to a group of joined pixels is chosen. The goal is to ensonify this region with the highest SIR possible. The inverse problem is thus to determine a set or subset of input parameters that will yield a high SIR in the target area. Examples of fitness of the inversion are the sum of the SIR values in the region of interest and the maximization of the minimum SIR in the region. Pixels outside the region of interest are assigned a "don't care" status and are not included in evaluation of the fitness function. Inversion can be performed across any subset of parameters—control or environmental. For example, the neural network can be inverted to find a combination the best sonar parameters *and* the best sound speed profile to ensonify a region of interest. There are other useful variations of the inversion problem in sonar [12]. Further details of the use of the PSO in the inversion of neural networks are given by Thompson *et al.* [12].

The forward sonar problem, when performed using an acoustic emulator, is slow [5], [9]. The neural network emulation increased the speed of the forward problem considerably.

Manuscript received May 2, 2006; revised October 30, 2006; accepted November 4, 2006. A preliminary version of this work was presented at the IEEE Swarm Intelligence Symposium, Pasadena, CA, June 8–10, 2005.

R. W. Duren and R. J. Marks II are with the Department of Electrical and Computer Engineering, Baylor University, Waco, TX 76798 USA (e-mail: Russell_W_Duren@baylor.edu; Rober_Marks@baylor.edu).

P. D. Reynolds is with the Stanford University, Palo Alto, CA 94305 USA (e-mail: paulr2@stanford.edu).

M. L. Trumbo is with Pelco, Fort Collins, CO 80525 USA (e-mail: mtrumbo@pelco.com).

Digital Object Identifier 10.1109/TNN.2007.891679

[1]To assess volume from a neural network trained only on a single plane, a plurality of radially spaced planes can be used.

The inverse problem using a trained neural network can require numerous queries to the neural network. When implemented on a dedicated 1.8-GHz personal computer (PC), the inversion process typically requires two minutes. This is still too slow for real-time implementation. We have mapped the neural network to a reconfigurable computer that uses FPGAs as coprocessors. Using a SRC-6e computer from SRC Computers, Inc., Colorado Springs, CO [16], we were able to decrease the time of the PSO [17], [18] inversion of a trained neural network by two orders of magnitude, rendering possible real-time applications.

There are numerous issues in the implementation of a large neural network interacting with a PSO algorithm on the SRC-6e. Both the neural network sonar emulator and the particle swarm algorithm used to perform inversion must be ported to the FPGA coprocessors. Multiple investigators have implemented neural networks on FPGAs [19]–[24]. To our knowledge, no one has implemented a network inversion by PSO on FPGAs. The imported layered perceptrons are trained offline in software using floating-point arithmetic.[2] In order to achieve maximum execution speed on the FPGAs, a fixed-point implementation is required. Conversion to fixed-point representation and the resulting quantization effects has to be addressed. A forward pass through the sonar emulator neural network requires approximately 92 000 multiply–accumulate operations. The FPGAs used in the SRC-6e can each perform 144 18-b multiplications in parallel [25]. The neural network must be mapped to this architecture.

The outputs of the internal nodes of the network are passed through a nonlinear squashing function. The implementation of the squashing function requires careful selection to keep the number of operations and latency low. Hikawa analyzed the performance of a piecewise linear representation of the squashing function [26]. Tommiska provided an extensive comparison of various representations of the squashing function. These included four piecewise linear representations, a piecewise second-order representation, and a combinational method [27]. Martincigh and Abromo developed a voting circuit that approximates a sigmoid function for pulse-mode neurons [28]. All of these methods are optimized for implementation without multipliers or large memories. As both multipliers and block random access memory (RAM) components are in abundant supply on newer FPGAs, this paper evaluates sigmoid approximations that take advantage of these components. The particle swarm algorithm requires similar care. Classical particle swarm claims better performance when a small random component is added into the update equations for the particle swarm. Several methods of implementing the random component are analyzed. For particle swarm inversion of the sonar neural network, we found no random components are needed.

The neural network and PSO were implemented on an SRC-6e reconfigurable computer. The SRC-6e is a commercial reconfigurable computer developed by SRC Computers, Inc., a company established by S. Cray. It has previously been used by researchers at George Mason University, Fairfax, VA, George Washington University, Washington, DC, and the Naval Postgraduate School, Monterey, CA, to implement various signal processing and cryptographic algorithms [29]–[31].

The final implementation of the sonar neural network partitions the problem into two FPGAs. One FPGA is used to calculate the output of the neural network. This FPGA is pipelined so that one neuron output is computed every clock cycle. The weights and inputs for the network are represented as 16-b fixed-point numbers. A piecewise Taylor series approximation is chosen to implement the nonlinear squashing function. The second FPGA is used to implement the particle swarm algorithm. The resulting architecture solves the sonar inversion problem in less than 2 s.

Implementing inversion of a large neural network trained on sonar data, although dealing with the acceleration of a specific application, also addresses the more general topics of effective implementation of neural networks, a class of nonlinearities, PSO, and random numbers on an FPGA.

II. Background

A. Neural Network

The feedforward neural network used to predict the acoustical performance has a 27-40-50-70-1200 architecture, with 27 inputs corresponding to sonar system and environmental parameters and 1200 outputs corresponding to the SIR, in decibels, of an area of water at points on an 80×15 grid. The outputs of the nodes in the three hidden layers are processed with a sigmoid squashing function.

B. Neural Network Inversion

The inversion of the neural network consists of (1) identification of the set of pixels over which SIR is to be maximized, and (2) identifying each input parameter as "clamped" or "floating" [32]. Clamped input parameters are set to specific values. Typically, the environmental parameters are clamped, although there are important cases where they float [12]. The floating input parameters are those that are adjusted to give the maximum SIR output over a region of interest. Optimization is performed in the space of the floating parameters. Each point in this space is assigned a fitness equal to the sum of errors between the SIR target pixel values and the SIR values achieved by the floating inputs. The smaller this error is, the better the fitness. The optimization space can be viewed as being implicitly parameterized by the clamped input parameters since changing a clamped input will change the optimization space landscape, and therefore, the location of the optimal solution in the search space.

To determine the fitness of a set of floating parameters, the trained neural network is provided with the values of the floating inputs to be evaluated. In conjunction with the clamped inputs, the SIR at all pixels is determined by a single forward pass through the trained neural network. The SIR in the region of interest is used to compute the fitness of the floating inputs. Pixels outside the region of interest are ignored.

To achieve the maximum SIR in a specified region, the SIR targets of the pixels are all placed at high unachievable values.

[2]The sonar neural network we use for this emulation was trained at the APL. Discussion on the method of training and the degree of accuracy of the training is discussed elsewhere [1], [2], [5], [12]. The details of the success of the neural network emulator are discussed in these references. Insofar as the neural network, our goal is to reduce it to operational practice on the FPGA.

The search, in attempting to reach these values, will achieve the best fitness allowable by the system.

C. PSO

Searches through the optimization space of floating parameters can be performed by many different search algorithms. PSO has been shown to be a robust and easily implemented search algorithm that works well in problems of the type considered [12], [14]. We will also show that PSO is relatively straightforward to implement on an FPGA.

PSO uses several agents exploring a search space to find the best possible fitness. As the agents traverse the space, they have tendencies to return to their own previous best locations as well as to the overall best global location of the group. The tendency is based on the distance from the best locations and a random component. The update equations used for each agent are

$$v[k+1] = v[k] + c_1^* \mathrm{rand}(\,)^*(p_{\mathrm{best}}[k] - x[k]) + c_2 * \mathrm{rand}(\,)^*(g_{\mathrm{best}}[k] - x[k]) \quad (1)$$

$$x[k+1] = x[k] + a^* v[k]. \quad (2)$$

The next location $x[k+1]$ and next velocity $v[k+1]$ are determined using the following: $x[k]$ as the current location, $v[k]$ as the current velocity, c_1 and c_2 as bias coefficients, $\mathrm{rand}()$ as uniform random variables between 0 and 1, p_{best} as the personal best fitness location, g_{best} as the group best fitness location, and an optional parameter a that has been added to the traditional update equations, providing an additional update constant controlling the resolution of movement.

Frequently used limits are also applied to the particle swarm. Velocity is limited to help keep particle swarm from exploding. The range is also limited to keep particles from using search time to look in impossible areas.

D. SRC-6e Hardware Architecture

The version of the SRC-6e used for this work contains two Pentium 3 microprocessors running at 1 GHz and three Xilinx XC2V6000 FPGAs running at 100 MHz. Two of the three FPGAs are available to the user as reconfigurable computing elements. Each XC2V6000 contains 144 18-b multiplier blocks, 144 18-kb blocks of SelectRAM and approximately six million logic gates [26]. Twenty-four megabytes of static RAM, referred to as onboard memory (OBM), is connected to the FPGAs and partitioned into six individually accessible banks. Data can be transferred between each OBM bank and either of the FPGAs at a rate of 800 MB/s. The two FPGAs are able to communicate with each other through three 64-b ports. If both FPGAs are utilized, they use a master–slave relationship with one controlling the other [33].

III. Implementation of the Feedforward Network

The trained neural network sonar emulation was implemented in one of the two FPGAs available on the SRC-6e. The PSO is implemented in the second FPGA. A master–slave relationship is used between the two FPGAs with the PSO acting as master and the neural network acting as slave. In this relationship, the particle swarm generates the inputs to the neural network and the neural network provides a fitness function for the particle swarm.

If a neural network is originally designed to be implemented and trained on an FPGA, the implementation may be optimized for the FPGA prior to training. Examples of this include using weights that are powers of two and using a lookup table squashing function [30]. The training of the network should compensate for the limited precision of the network. However, the sonar implementation problem involves porting a network that was originally designed and trained offline using floating-point math and a sigmoid squashing function with essentially unlimited precision, to a limited-precision implementation.[3] The impact of finite precision for the weights, the multiplications, and the squashing function must therefore be investigated.

A. Conversion from Floating-Point to Fixed-Point Representations

In order to minimize chip space and computation time, short fixed-point representations of numbers are desired. The FPGAs in the SRC-6e are connected to the onboard memory through six 64-b wide buses. The 64 b can be easily divided into two 32-b numbers or four 16-b numbers. The XC2V6000 FPGAs contain embedded 18-b multipliers. Together, these factors make the use of a 16-b representation desirable. To define the representation, two parameters must be specified, the length of the integer bits and the length of the fractional bits. Computer simulations of the neural network were used to study the impact of converting to fixed-point representation and to select the optimum representations for various parameters.

While all other calculations were performed at maximum accuracy, the bit accuracy of the output of the squashing function was varied. Fig. 1 shows four different gray level maps of the SIR distribution as a function of the accuracy of the squashing function. The vertical direction depicts water depth with the water surface at the top. The horizontal direction depicts range. Each representation corresponds to a maximum depth of 180 m and a range of 6 km.

Fig. 2 shows the SIR distribution resulting from changing the bit accuracy of weights while performing all other calculations at maximum accuracy. Fig. 3 shows the combined effect of limiting the accuracy of the weights and the squashing function. The results in Fig. 3 represent the averaging 100 test cases. Additional simulations reveal the values presented to the input of the squashing function range from -50 to 85. This range requires a minimum of eight bits: one sign bit and seven magnitude bits. The inputs and outputs are a few orders of magnitude greater than the network calculations. However, the inputs and outputs have consistent orders of magnitudes among themselves and can also be stored in a fixed-point representation. The corresponding input and output weights can be scaled to account for the difference, making all layer calculations appear to be of the same order of magnitude.

[3]To alleviate this problem, the neural network trained offline could be constrained to have weights of limited precision. Details of doing so, including training algorithms (generic error backpropagation requires floating-point precision and cannot be used) and even the ability to train such a network with the sonar data is not considered here.

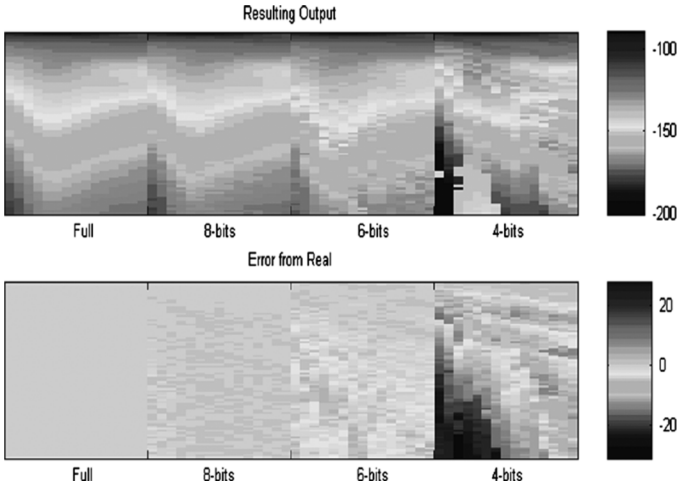


Fig. 1. Example accuracy for squashing function of different precisions. Using one of the sonar problem's inputs, the image map output was calculated using a neural network with a squashing function rounded to various levels of accuracy. The input and weights were kept at complete accuracy. The precision is shown under each image. The four outputs use, from left to right, full accuracy, eight, six, and four fractional bits for the squash output. The grayscale range is in decibels.

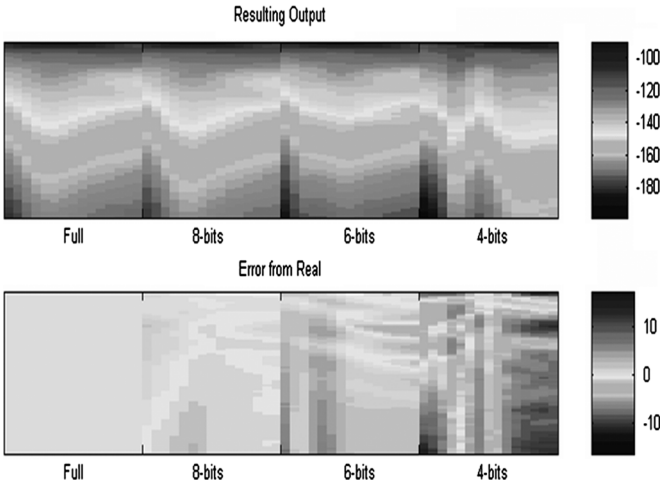


Fig. 2. Accuracy sweep of weights of different precisions. Using one set of sonar inputs, the image map output is calculated using weights rounded to various levels of accuracy. The input and squashing function maintained complete accuracy. The precision is shown under each image. The four outputs use, from left to right, full accuracy, eight, six, and four fractional bits for weights. The grayscale range is in decibels.

The result of simulations confirm that 16 b provide sufficient accuracy, allowing representation with one sign bit, seven integer bits, and eight fractional bits. Computer simulations confirm that this representation results in an average error of 0.866 dB per pixel. Since typical pixel values are on the order of magnitude of 100 dB, the error is less than one percent.

B. Implementation of the Squashing Function

The squashing function is used 160 times[4] per neural network evaluation. A small, quick, and accurate implementation is desired. The familiar sigmoid, or logistic function, is used as a

[4]These are the number of hidden neurons; $160 = 40 + 50 + 70$. The inputs are not subjected to any nonlinearity. Neither are the output neurons.

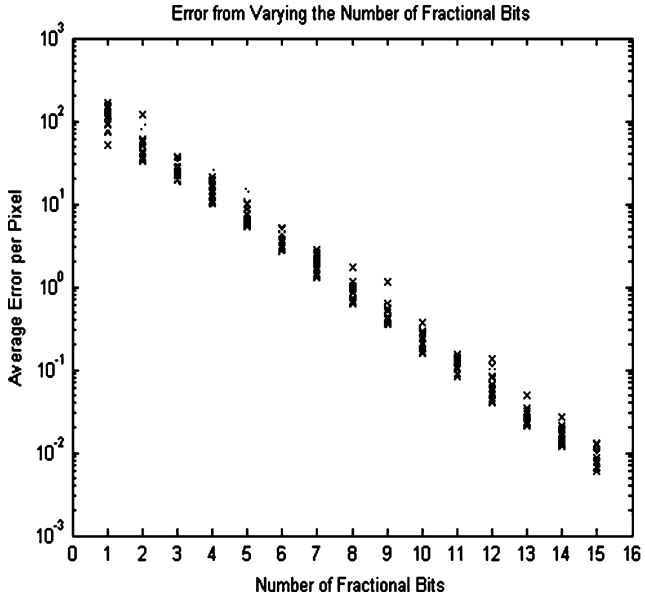


Fig. 3. Accuracy sweep of fractional bits. Using one hundred sets of inputs, the average error per pixel in decibels is calculated using a neural network with all numbers rounded to various levels of bit accuracy. The error decreases logarithmically as the number of fractional bits increases.

squashing function by the nodes in the three hidden layers of the network [4]. The equation defining the sigmoid function is

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{3}$$

The sigmoid can be found using high-precision methods, such as a lookup table or a coordinate rotation digital computer (CORDIC) function [35]. Another common method is to use a simple piecewise linear approximation implemented with a shift–add approach [36]. However, each of these methods has undesirable aspects. In order to keep the entire network internal to one chip, a lookup table is undesirable. A CORDIC function gains accuracy at the cost of latency, where latency is defined as the number of clock cycles required from the start of the calculation until the resulting data is ready. Each additional stage in the CORDIC calculation increases the accuracy, but it also increases the time required to complete the calculation by one or more clock cycles. Within a particular layer all calculations are pipelined, so the latency penalty is incurred only once per layer. The piecewise linear approximation, while small in area and quick in execution, is not smooth. A smooth squashing function approximation that can approximate a sigmoid to arbitrary accuracy is desired. A piecewise Taylor series approximation proved best. Details follow.

1) Lookup Table Implementation: The simplest sigmoid implementation is use of a lookup table. In order to make a lookup table, a limited operating range must be determined. The sigmoid squashing function has a nearly odd property

$$f(-x) = 1 - f(x). \tag{4}$$

The size of the lookup table can be, therefore, decreased to half the desired range. Since the sigmoid is nearly 1 for $x > 8$ and 0 for $x < -8$, the nonsaturation range is between -8 and 8 and the lookup table only needs to operate between 0 and 8. This

requires three integer bits and all eight fractional bits to be used as address bits. Any numbers not in that range are considered to be in saturation and are assigned an output value of 1. The resulting table has 11 address bits selecting the eight bit fractional portion, using 2 kB of memory. This fits nicely into one 18-kb block RAM in the FPGA. The lookup table implementation of a sigmoid has a latency of three clock cycles.

Calculations show that the maximum error of the lookup table is 0.005 out of 1. This results in an average pixel error of 0.4015 dB per pixel in simulations. The lookup table is the best choice if sufficient block RAMs are available. For this application, all of the block RAMs are used for storage of weights and variables. Therefore, a different method is required.

2) CORDIC Implementation: A second method to calculate the sigmoid uses the CORDIC algorithm to calculate the hyperbolic sine and cosine followed by division to get the hyperbolic tangent [35]. The tangent can then be used in the sigmoid equivalent

$$y(x) = \frac{1}{2}\tanh\frac{x}{2} + \frac{1}{2}. \tag{5}$$

The CORDIC algorithm works by rotating a vector by known angles until the sum of the angles is equivalent to the desired angle. For this application, the CORDIC uses the properties

$$\begin{pmatrix}\cosh(a\pm b)\\ \sinh(a\pm b)\end{pmatrix} = \begin{pmatrix}\cosh(b) & \pm\sinh(b)\\ \pm\sinh(b) & \cosh(b)\end{pmatrix}\begin{pmatrix}\cosh(a)\\ \sinh(a)\end{pmatrix}. \tag{6}$$

With a small amount of algebra, this becomes

$$\begin{pmatrix}\cosh(a\pm b)\\ \sinh(a\pm b)\end{pmatrix} = \cosh(b)\begin{pmatrix}1 & \pm\tanh(b)\\ \pm\tanh(b) & 1\end{pmatrix}\begin{pmatrix}\cosh(a)\\ \sinh(a)\end{pmatrix} \tag{7}$$

$$\cosh(a\pm b) = \cosh(b)(\cosh(a)\pm\tanh(b)\sinh(a)) \tag{8}$$

$$\sinh(a\pm b) = \cosh(b)(\sinh(a)\pm\tanh(b)\cosh(a)). \tag{9}$$

By starting with the hyperbolic sine and cosine of known angle a, and rotating the angle forward or backward by known angles b, a desired hyperbolic sine and cosine can be calculated by applying (7) and (8). If the known angle is greater than the desired one, the next rotation is backward; if it is less than the desired one, the next rotation is forward. The equations can be applied repeatedly with other known bs until the proper sum is reached. By choosing $\tanh(b)$ to be negative powers of 2, such as $1/2, 1/4, 1/8$, etc., all the multiplications can be executed as shifts.

The commonly used initial argument is zero, with the starting vector as $(x, y) = (1.20750)$. However, the range of the CORDIC algorithm starting at this vector is limited to the sums of the known bs. When using $\tanh(b)$ as only powers of 2, the radius of convergence is slightly greater than 1.13. This creates a problem with the sigmoid implementation. Using the almost odd property, the desired sigmoid range is from 0 to 8. Since the argument is divided by two, the necessary range of the hyperbolic tangent is 0–4, which is out of the convergence range. In order to get the necessary range to converge, the desired range is divided into segments the same size as the standard range. In this case, two segments were used, 0–2 and 2–4. Then, when a tangent needs to be found, the initial vector is chosen based on the argument. If bigger range is necessary, more segments can be used. If a more accurate result is desired, more CORDIC rotations can be used.

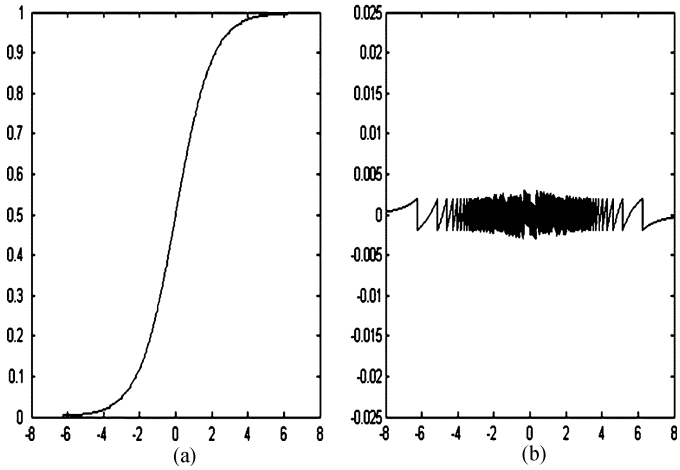


Fig. 4. VHDL approximation of CORDIC sigmoid function. (a) Output of the VHDL implementation of the CORDIC squash. (b) CORDIC approximation error.

Once the hyperbolic cosine and sine are found, the tangent is found by division. A standard Xilinx core is used for division. A shift of 1 b is used to divide the tangent by two. Then, one half is added to the result. The 11-stage CORDIC algorithm and divide implementation fits into a pipeline that has a latency of 50. The performance of the CORDIC algorithm is shown in Fig. 4. The approximation of the sigmoid remains within 0.005 for the entire range. Fig. 5 shows the result of using the CORDIC algorithm in the neural network. The average error resulting from the CORDIC implementation in hardware is 0.4279 dB per pixel.

3) Shift–Add Implementation: Another common implementation of the sigmoid function is a piecewise linear approximation with many segments of the form

$$y = mx + b. \tag{10}$$

If the segments are chosen wisely, the sigmoid can be calculated using only bit shifts and additions [36]. However, the bit shift method has a limited accuracy, with no possibility for improvement. At its worst, the approximation is nearly 0.025 off the actual value of the sigmoid.

Another problem is that the piecewise linear approximation is not very smooth. In computer simulations, even when a network is trained using the piecewise approximation of the sigmoid, the output demonstrates a piecewise character. The error performance of the shift–add implementation are shown in Figs. 6 and 7. The shift–add implementation of the sigmoid has a latency of five.

4) Piecewise Taylor Series Approximation: The fourth approximation examined uses a Taylor series around 0. When one approximation is used for the entire range of 0 to 8, many terms are needed for a suitable approximation. To avoid this problem, several second-order segments of Taylor series about different points are used, with a general formula of

$$y(x) = -y_0''*(x - x_0)^2 + y_0'^*(x - x_0) + y_0. \tag{11}$$

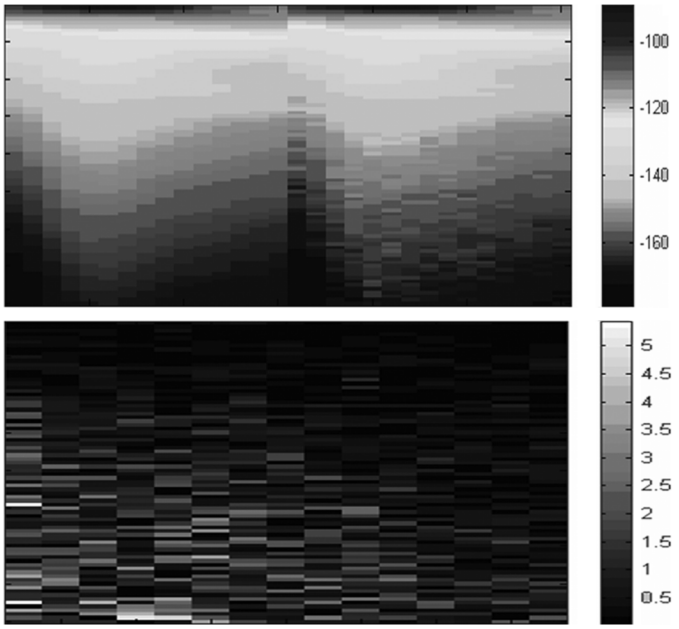


Fig. 5. Comparison of CORDIC FPGA output and the full-precision output. The map on the left shows a comparison of the full-precision image and that produced by the FPGA when a CORDEC implementation of the sigmoid is used. The absolute difference of the two images is shown in the map on the right.

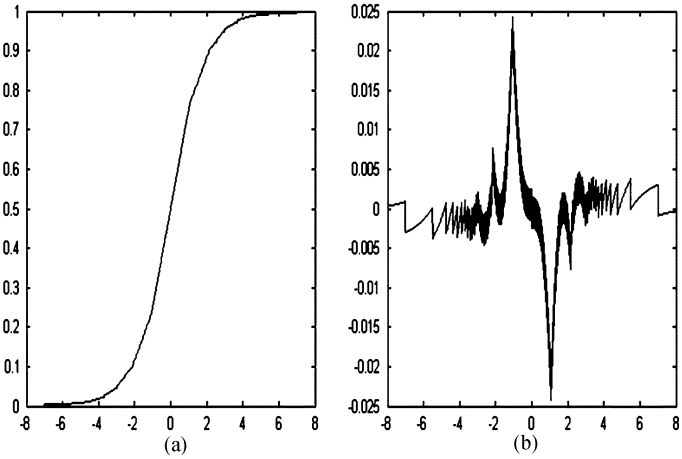


Fig. 6. VHDL approximation of shift–add sigmoid function. (a) Output of the VHDL implementation of the shift–add squash. (b) Shift–add approximation error. The shift–add approximation of the sigmoid is nearly 3% off from the actual at its worst.

Then, given the argument, the proper offset and coefficients are chosen. The accuracy of the approximation can be improved by increasing the number of segments used in the approximation. This implementation uses three multipliers, three adders, three multiplexers, and a number of comparators equivalent to the number of segments. Five segments are used for the final implementation. The input bounds for these segments and the resulting Taylor series coefficients are shown in Table I.

The approximation is pipelined to obtain maximum throughput. A block diagram of the pipeline is shown in Fig. 8. One might expect this pipeline would have a latency of eight. However, the multipliers on the XC2V6000 require registering to operate at 100 MHz, resulting in a latency of two for

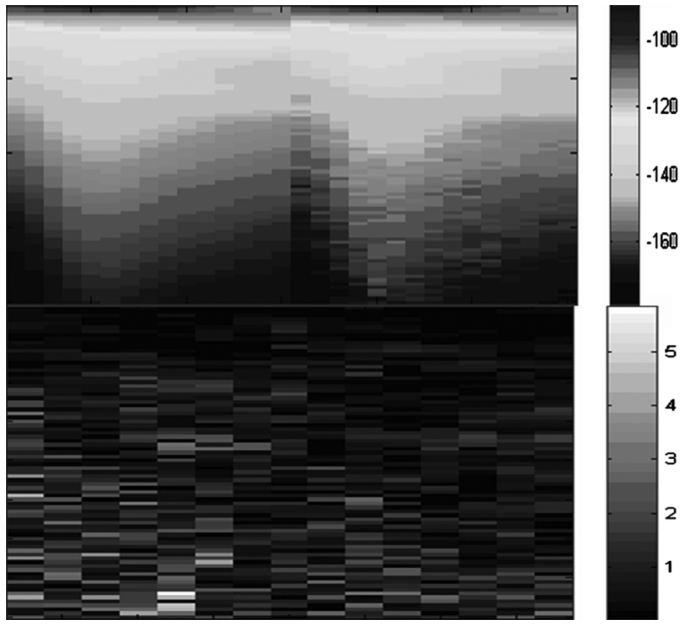


Fig. 7. Comparison of shift–add FPGA output with full-precision output. The map on the left shows a comparison of the full-precision image and that produced by the FPGA in comparison to the result using a shift–add approximation of the sigmoids. The absolute difference of the two images in SIR decibels is shown in the map on the right.

TABLE I
TAYLOR SERIES COEFFICIENTS

Lower Bound	x_0	y_0''	y_0'	y_0
7.293	--	--	--	1.00000000000
4.771	6	0.001220703125	002441406250	0.99755859375
3.317	4	0.008544921875	0.017578125000	0.98205566406
2.482	2.75	0.024780273438	0.056396484375	0.93994140625
.425	1	0.045288085938	0.196533203125	0.73107910156
0	0	--	0.250000000000	0.50000000000

each multiplier stage. Since there are two stages of multipliers, the total latency is, therefore, ten. The error performance of the Taylor series implementation is shown in Figs. 9 and 10.

5) Comparison of Squashing Function Implementations: Table II shows a comparison of the different squashing function implementations. The table lists the FPGA resources used, the latency, and the average pixel error for each approximation. The average pixel error was found using computer simulations while holding all other calculations at maximum accuracy.

The lookup table approximation uses the fewest logic slices, has the lowest latency, and the lowest average error. Under most circumstances, it would be the best solution. Unfortunately, all 144 of the block RAM memories are required for storage of the weights in the neural network implementation. This eliminates the lookup table approach.

The shift–add implementation is small with a low latency and uses no block RAMs or multipliers for its implementation. However, it has the worst error, three times that of any other implementation. It also has no method for error improvement.

The CORDIC version has the second lowest error, though is significantly larger in chip area than the other four versions.

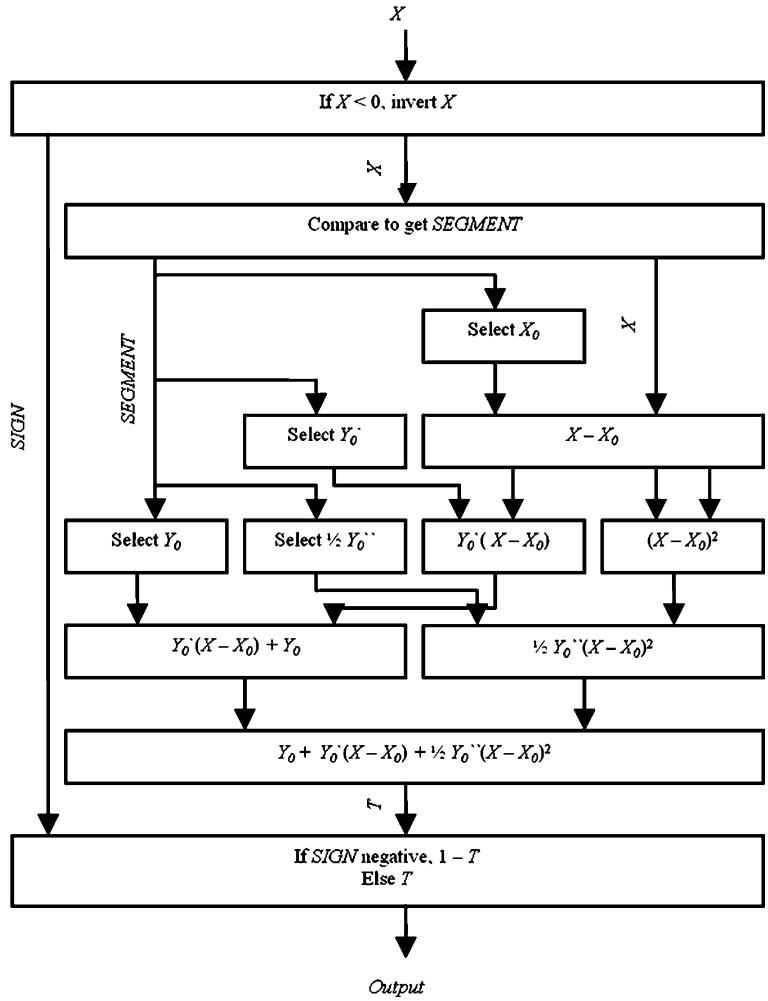


Fig. 8. Block diagram for a Taylor series implementation of the sigmoid.

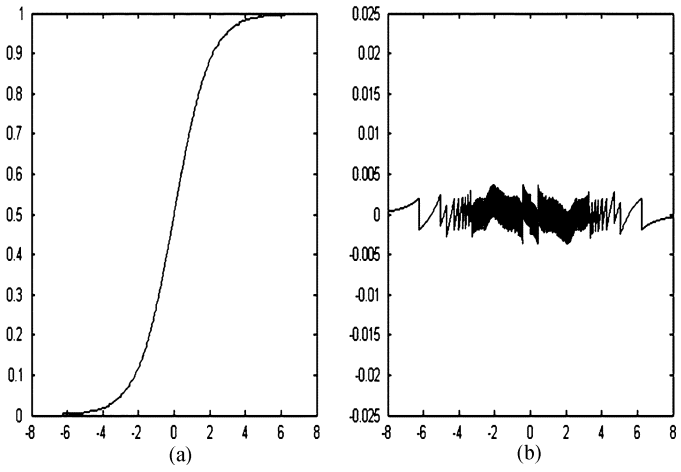


Fig. 9. VHDL approximation of the Taylor series sigmoid. (a) Output of VHDL implementation of the Taylor series squash. (b) Taylor series approximation error. The approximation of the sigmoid remains within an error of 0.005 for the entire range.

This version also has the longest latency, which, in a four-layer network, would add 200 clock cycles versus the next slowest 40. However, error improvement is easily achieved by adding more stages as long as chip area is available.

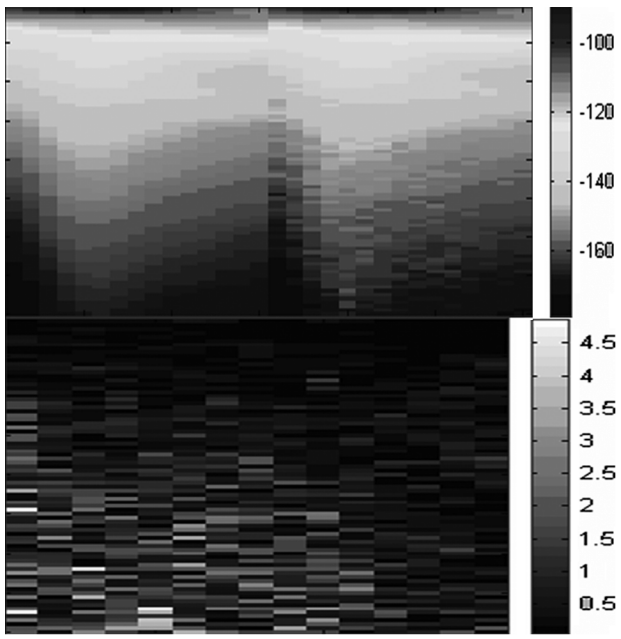


Fig. 10. Example of the comparison of FPGA output with full-precision output. The image map on the left was produced by the FPGA using a Taylor series approximation of the sigmoid function and the image map on the right was produced by the original neural network. The map on the far right is the corresponding absolute error between the maps.

TABLE II
COMPARISON OF SIGMOID APPROXIMATIONS

Implementation	Slices	Memory	Multi-pliers	Latency	Average Pixel Error
Lookup Table	1951	2kbytes	0	3	.4015
Shift-add	2026	0	0	5	1.3281
CORDIC	3475	0	0	50	.4279
Taylor Series	2085	0	3	10	.4306

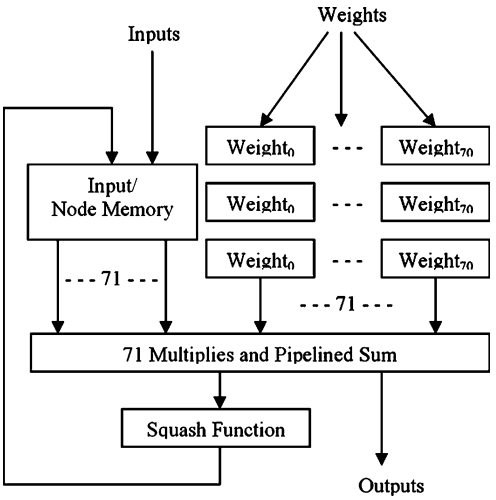


Fig. 11. Simplified block diagram of the node implementation. The node implementation uses 71 multipliers in parallel and one squashing function to output the results of one node every clock.

The Taylor series approximation has the third lowest error of four implementations, though it is not much worse than the smallest error. The small improvement in latency gained by using the shift–add implementation is outweighed by the increase in error of the shift–add approximation. The desire for speed and smaller circuit area provided by the Taylor series approximation also outweighs the small error improvement that would be gained by switching to a CORDIC implementation. The Taylor series approximation was selected for use in the FPGA neural network implementation.

C. Network Architecture Implementation

The main objective of the neural network implementation is, for an acceptable accuracy level, minimization of the time required to calculate a forward pass through the network. A forward pass through the sonar neural network requires 91 940 multiply–accumulate operations. Each FPGA contains 144 18-b multipliers. This does not support calculating an entire layer at a time. The output neurons required the largest number of multiplications for any individual neuron: 71 multiplications corresponding to the outputs of the previous layer and one additional accumulation for the bias weight. A pipelined network that allowed the calculation of one neuron per clock cycle was, therefore, chosen.

The neural network implementation performs all multiplications for the calculation of one node during one clock cycle. A block diagram of the node parallel calculation is shown in Fig. 11. The weights are stored in the FPGA in block RAM. The block RAMs are configured such that 70 weights and one bias term can be accessed simultaneously. Due to restrictions on partitioning the block RAM components in the Virtex FPGAs and limitations of the SRC-6e development environment, storage of the weights requires all of the available 144 block RAM components within the FPGA. The inputs and outputs are held in registers. This structure allows multiple weights and the entire layer of inputs to be accessed concurrently. When a layer is complete, the outputs write over the inputs for calculation of the next layer. The previous layer's outputs are multiplied by the corresponding weights for the current node. While the products are being summed, the next node's weights are multiplied by the same set of outputs, creating an efficient pipeline. Since all the node outputs are required for calculations in the next layer, the pipeline must wait several clock cycles for the previous layer to finish before continuing with the next. In order to simplify the weight storage of the network, all layers are considered to be the same size as the largest, in this case, 70 nodes. Weights not needed by the smaller layers are set to zero. However, calculation of all 70 nodes for each layer is not required, so the number of nodes calculated per layer is controlled in order to save clock cycles. The pseudocode shown later describes the calculation of the output for one node.

```
Multiply all inputs by all current weights
Sum all the products
If not in the output layer
     Squash the sum
Save the squashed sum in output memory
Increment weight counter
Increment output counter
If output counter equal number of next
layer nodes
     Reset the output counter
     Write output memory over input memory
     Increment layer counter
```

This design takes 1465 clocks to complete one network evaluation. Given the 100-MHz clock on the SRC-6e, this translates to 14.65 μs per forward calculation. This allows the network to be evaluated more than 60 000 times per second. A Pentium 4 running at 1.8 GHz can theoretically perform the forward calculation in 0.116 ms if it performs one calculation per clock. However, due to memory access time and a nondedicated processor, the actual forward calculation time is 0.28 ms. This means the FPGA implementation provides a gain of 19 over the Pentium 4 for the forward pass through the network.

IV. IMPLEMENTATION OF THE PARTICLE SWARM INVERSION

The PSO update equations consist of simple multiplications and additions, easily implemented on an XC2V6000. Setting the bias coefficients to powers of two and using shifts in place of multiplications further simplifies the implementation. For this implementation, the value of $c1$ was set to $1/8$ and the value of $c2$ was set to $1/16$. These values represent negative powers

of two, corresponding to right shifts of 3- and 4-b positions, respectively. These values were found to work well in repeated experiments.

The PSO algorithm is implemented with ten particles. The default search space is over all 27 possible inputs to the neural network. The search space is constrained by providing minimum and maximum values for each input dimension. The maximum particle velocity in each dimension is also constrained to be less than a predetermined maximum value. The starting positions and velocities of all ten particles are set to pseudorandom values within the input space.

For most practical applications of the system, some of the 27 inputs would be set to constant values and the system would optimize the remaining inputs. With this implementation, constant inputs can be implemented by setting the minimum and maximum values to the same number. Alternatively, if there is a small uncertainty in the some of the constant inputs, the uncertainty can be bounded by the minimum and maximum values.

The large neural network serves as the fitness function for each particle. The output values of the neural network are calculated in one of the two FPGAs available for user logic in the SRC-6e computer. The remaining particle swarm calculations are implemented in the second FPGA. This allows the position and velocity of one particle to be updated while the fitness of another particle is being calculated.

The generic PSO algorithm requires generation of random numbers. We examined three different implementations. The first implementation did not add any random component to the updates. The other two implementations used two different methods to generate random variables.

A. *Deterministic Particle Swarm*

The first method is to simply ignore the random component of the PSO. The random component was previously removed successfully to prove the stability of the algorithm [36]. Removing the random component simplifies the implementation of the particle swarm update equations, but can also degrade PSO performance. In order to estimate the effectiveness of such an implementation, the nonrandom or deterministic particle swarm inversion was simulated on a conventional computer. The bias coefficients were decreased so that the average bias would be the same. In the inverse accuracy test, input i is used to compute output o. The network is inverted using o with a result of $\hat{i}$. For 100 such trials of the deterministic particle swarm, the average error $\Sigma_n|o_n - \hat{o}_n|/n$ was 2.3587 dB per pixel. For comparison, a standard particle swarm inversion incorporating uniform random variables was also run on a conventional computer. Using the same inverse accuracy test for 100 trials, the average error for the standard particle swarm was 1.9385 dB per pixel. Next, both the random and the deterministic particle swarms were run for 10 000 iterations for 30 searches. The global best fitness was plotted for each run as well as the average of all swarms. This plot is shown in Fig. 12.

For our problem, including the random component enhances swarm performance by, on average, approximately 1 dB. The deterministic particle swarm was implemented in the FPGA.

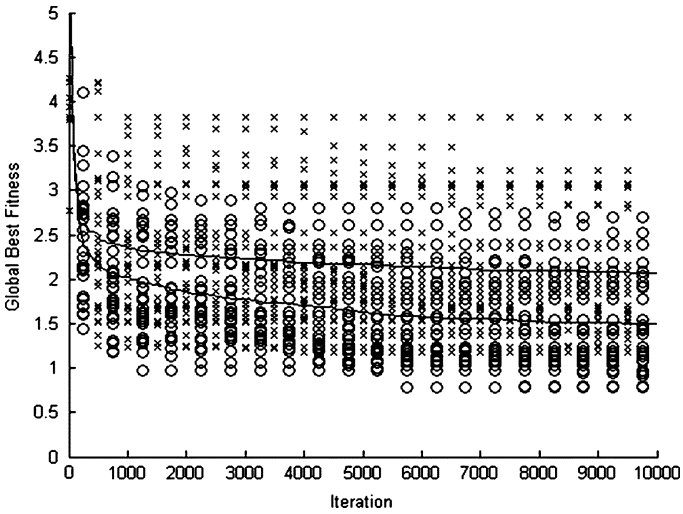


Fig. 12. PSO with and without random noise. Random and deterministic PSO were run for 10 000 iterations 30 times. All the results are shown here. The crosses are the global best results from the deterministic particle swarm and the top line is the average. The circles are the global best results from the particle swarm with randomness and the bottom line the average. The lower stochastic PSO line performs approximately 1 dB better than the deterministic PSO. The deterministic PSO, however, is more straightforwardly implemented on the FPGA. In practice, the tradeoff between the simplicity and speed of implementation must be weighed against the lower accuracy.

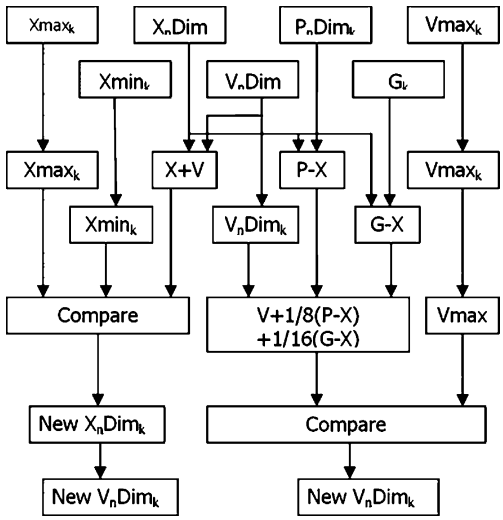


Fig. 13. Deterministic particle swarm block diagram. The deterministic particle swarm implementation performs both the velocity and position updates in parallel and has a latency of three clock cycles.

The deterministic particle swarm update equations lend themselves to a parallel hardware implementation since velocity and position can be calculated at the same time. The update equations are implemented in a pipeline and one dimension can be updated on every clock cycle.[5] The pipeline has a latency of

[5]The stochastic nature of PSO, and indeed, of many optimization algorithms, improves performance. For the specific case of the neural network inversion, however, the stochastic component of PSO can be sacrificed at the cost of degraded performance. All optimization is faced with tradeoffs between implementation constraints and accuracy. For the inversion problem, we could, in principle, perform an exhaustive search and find a solution better than that found using a stochastic PSO, but the time constraint prohibits us from doing so. The choice of a deterministic PSO buys faster implementation speed. As with any optimization, if the resulting accuracy is not acceptable, alternate methods must be investigated with a probable sacrifice in implementation properties.

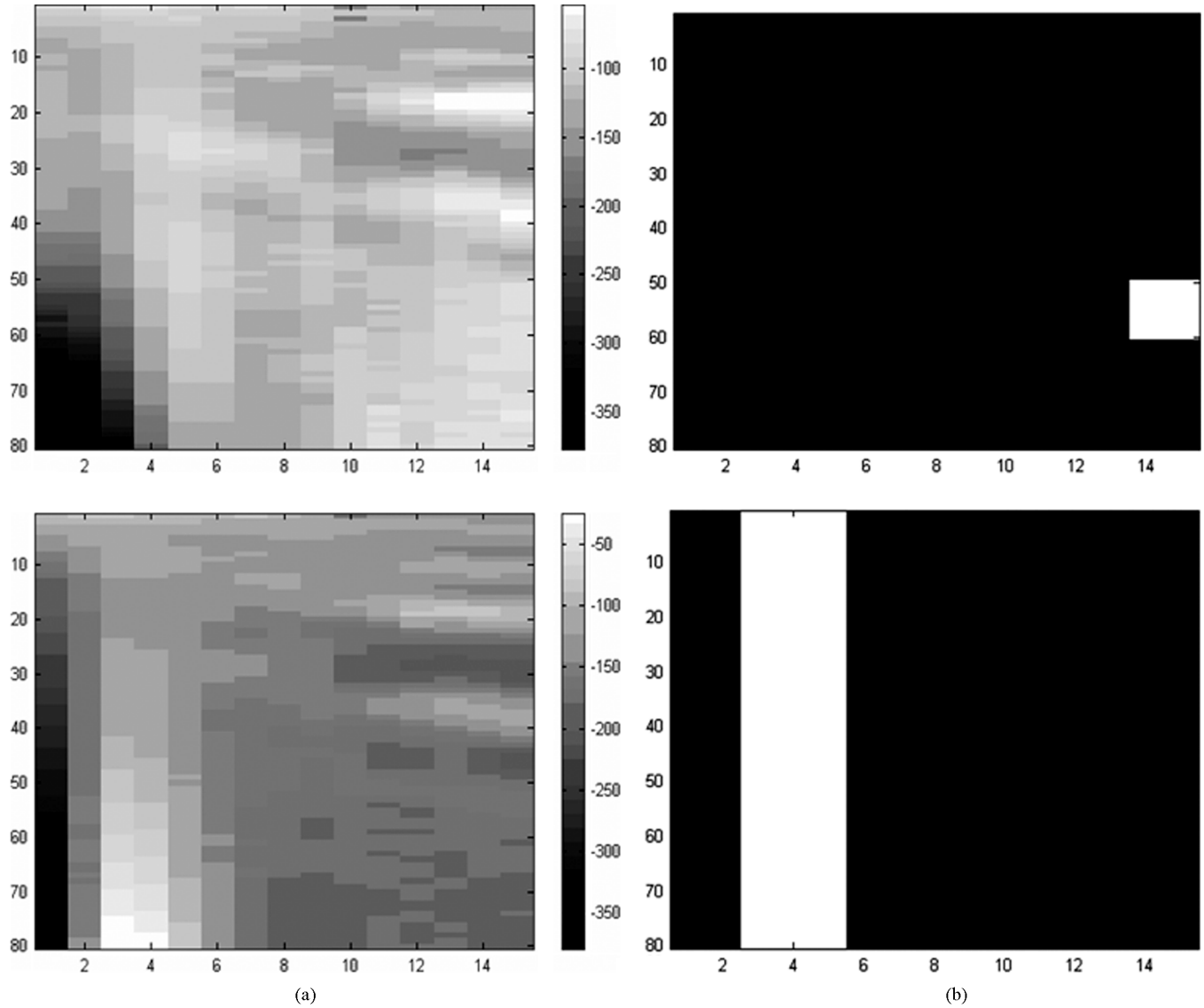


Fig. 14. PSO inversion for SIR maximization in a specified area. (a) Outputs from the solution found by the particle swarm, where lighter areas represent higher SIRs. (b) White areas show the desired maximization areas.

three clock cycles, so all 27 dimensions can be updated in a total of 29 clock cycles, three clock cycles for the first dimension and one clock cycle for each of the remaining 26 dimensions. This results in a particle update time of 290 ns. The block diagram for the pipelined hardware implementation is shown in Fig. 13.

B. Particle Swarm With Randomization

In order to implement random numbers for the PSO, a function was implemented that generated two pseudorandom numbers per clock. Two stages were added to the update pipeline to multiply the personal bias and global bias by the generated random numbers.

1) Linear Feedback Shift Register: The first method for generating pseudorandom numbers uses a linear feedback shift register (LFSR). This method is typically used in testing digital logic designs. The LFSR uses a shift register where the next bit shifted in is determined by a logical combination of the bits in the previous number [37]. For a 16-b random number, the last 16 b were taken from a 20-b LFSR. Using the inverse accuracy test over 100 trials, the average error for the hardware PSO with LFSR randomness was 2.3522 dB per pixel.

2) Modulus Implementation: A second method of generating pseudorandom numbers is based on a common software implementation [38]. In this implementation, the next number in a sequence of random numbers is found by taking the previous number multiplied by a constant a added to offset c modulus m. The modulus implementation used is very similar, choosing the next number in the sequence by using the fractional portion of the square of the previous number added to a constant c. Using the fractional portion is equivalent to modulus one. The squaring operation is similar to the multiplier and constant a. In the hardware implementation, 18-b fixed-point numbers were used.

C. Comparison of Particle Swarm Implementations

In the hardware implementation, the average pixel error for the deterministic swarm over one hundred trials is 2.36 dB per

pixel. The particle swarm with an LFSR generating random numbers has an average pixel error of 2.35 dB. The particle swarm using the modulus implementation had an average pixel error of 2.37 dB. When searching for known achievable sets, all three fixed-point implementations produce approximately the same level of output error. Due to its simplicity, this makes the deterministic method most desirable for the problem at hand. Note, interestingly, that the deterministic method introduces a small amount of randomness due to truncation caused by the fixed-point calculations. None of the hardware implementations are as accurate as the conventional computer average error of 1.94 dB per pixel. In order to account for this increase, note that the hardware implementation uses fixed-point math, while the conventional computer uses floating-point math. The final conclusion is, on the average, that the deterministic FPGA implementation introduces an additional error of about 0.4 dB per pixel.

V. Performance of the Complete Implementation

The output from the hardware particle swarm inversion has an average per pixel difference of 2.54 dB from a known achievable desired output or an average difference of 1.53%. This low error implies that the particle swarm inversion will be able to find a set of inputs that produces outputs closest or near-closest to a desired output set. This error is 0.42 dB per pixel greater than the error obtained using a conventional Pentium processor with floating-point math.

Fig. 14 shows two sets of outputs from inputs found with the goal of maximizing a specific area. The images on the right show the desired areas for maximization. All other areas were ignored for calculation of fitness. Localized maximization is equivalent to attempting to find infinite signal to interference ratio, which, of course, is outside the achievable set.

It was determined that 100 000 particle updates provide a satisfactory solution, so the circuit is set to report results after 100 000 updates. The fitness function requires 14.65 μs to calculate, or 1.465 s for 100 000 updates. Particle updates require 290 ns to calculate, but this time is hidden in the fitness function update time as the particle updates are performed in parallel on a second FPGA. The total computation time for the hardware PSO requires less than 1.8 s. The additional 0.335 s is attributed to communication overhead between the two FPGAs and between the FPGAs and the Pentium processor on the SRC-6e that provides the user interface to the program.

The time to complete the same 100 000 iteration PSO on a conventional PC using only a 1.8-MHz Pentium 4 processor is nearly 2 min. At 100 MHz, the two-chip hardware implementation takes under 1.8 s to complete, approximately 65 times faster. We developed several additional implementations of the neural network. Details are in [39].

VI. Conclusion

We have described a real-time implementation of a particle swarm neural network inversion for calculation of sonar operating parameters. The neural network was implemented on the SRC-6e reconfigurable computer. A speedup of a factor of 65 was obtained because of the careful design and the use of two Virtex 2 FPGAs.

Several interesting conclusions can be developed concerning the details of implementing such an algorithm in FPGAs. For the problem considered, these include the following.

1) A simple lookup table provides the best implementation of a sigmoid squashing function when sufficient block RAM components are available. When these components are not available, a piecewise Taylor series approximation works best. Both techniques offer the combined benefits of the use of minimal hardware, low latency, and high accuracy when compared to the other methods that were considered.
2) The addition of a random component to the swarm update equations resulted in better performance for a floating-point solution on a conventional computer, but not for the fixed-point implementation on the reconfigurable computer. It is thought that the noise added by conversion to fixed-point math, coupled with the relatively smooth fitness function, effectively eliminated the need for the intentional addition of random noise.
3) Conversion from a network originally trained using a convention computer with floating-point math to a reconfigurable computer using fixed-point math resulted in a significant speedup without a significant change in accuracy. In should be noted that these points may be specific to the problem considered.

The reconfigurable computer implementation of the neural network inversion effectively reduced computation time to near real-time levels. The 100 000 evaluations in a conventional computer particle swarm take nearly 2 min to complete. The same inversion can be performed in the current SRC-6e-based implementation in about 1.8 s. Such a calculation rate is sufficient for most real-time applications. The current particle swarm implementation uses two identical Virtex 2 FPGAs operating at 100 MHz and containing 144 multipliers. The latest generation of Xilinx Virtex 4 FPGAs operates at 500 MHz and contains 512 multipliers. Utilizing these chips, the clock speed increase alone would allow the inversion time to decrease from 1.8 to 0.36 s. The additional multipliers could be used to perform fitness evaluations of several agents at the same time or to improve the speed of a single fitness evaluation. Predicted speedup based on the increase in multipliers is about seven. This combined with the faster chip speed would allow nearly 20 network inversions to be performed every second.

Acknowledgment

The authors would like to thank Prof. D. Fouts and the U.S. Naval Postgraduate School for providing access to the SRC-6e computer located at the school. They would also like to thank Dr. W. J. Fox at the Applied Physics Laboratory for providing the weights of the trained neural network sonar emulator.

References

[1] J.-B. Jung, M. A. El-Sharkawi, R. J. Marks II, R. T. Miyamoto, W. L. J. Fox, G. M. Anderson, and C. J. Eggen, "Neural network training for varying output node dimension," in *Proc. Int. Joint Conf. Neural Netw.*, Washington, D.C., 2001, pp. 1733–1738.

[2] W. L. J. Fox, R. J. Marks, M. U. Hazen, C. J. Eggen, and M. A. El-Sharkawi, "Environmentally adaptive sonar control in a tactical setting," *Impact Environ. Variability Acoustic Predictions Sonar Performance* pp. 595–602, 2002 [Online]. Available: http://www.ecs.baylor.edu/faculty/marks/REPRINTS/2002_EnvironmentallyAdaptiveSonar.pdf

[3] P. D. Reynolds, R. W. Duren, M. L. Trumbo, and R. J. Marks II, "FPGA implementation of particle swarm optimization for inversion of large neural networks," in *Proc. 2005 IEEE Swarm Intell. Symp.*, Pasadena, CA, Jun. 8–10, 2005, pp. 389–392.

[4] R. D. Reed and R. J. Marks II, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. Cambridge, MA: MIT Press, 1999.

[5] C. A. Jensen, R. D. Reed, R. J. Marks, M. A. El-Sharkawi, J. Jung, R. T. Miyamoto, G. M. Anderson, and C. J. Eggen, "Inversion of feedforward neural networks: Algorithms and applications," *Proc. IEEE*, vol. 87, no. 9, pp. 1536–1549, Sep. 1999.

[6] J. N. Hwang, C. H. Chan, and R. J. Marks II, "Frequency selective surface design based on iterative inversion of neural networks," in *Proc. Int. Joint Conf. Neural Netw.*, San Diego, CA, Jun. 17–21, 1990, vol. I, pp. I39–I44.

[7] B. S. Kim and A. J. Calise, "Nonlinear flight control using neural networks," *J. Guid. Control Dyn.*, vol. 20, no. 1, pp. 26–33, 1997.

[8] R. D. Reed and R. J. Marks II, "An evolutionary algorithm for function inversion and boundary marking," in *Proc. IEEE Int. Conf. Evol. Comput.*, Nov. 26–30, 1995, pp. 794–797.

[9] C. A. Jensen, M. A. El-Sharkawi, and R. J. Marks II, "Power security boundary enhancement using evolutionary-based query learning," *Eng. Intell. Syst.*, vol. 7, no. 9, pp. 215–218, Dec. 1999.

[10] I. N. Kassabalidis, M. El Sharkawi, and R. J. Marks II, "Border identification for power system security assessment using neural network inversion: An overview," in *Congr. Evol. Comput./IEEE World Congr. Comput. Intell.*, Honolulu, HI, May 12–17, 2002, pp. 1075–1079.

[11] L. Tsang, Z. Chen, S. Oh, R. J. Marks II, and A. T. C. Chang, "Inversion of snow parameters from passive microwave remote sensing measurements by a neural network trained with a multiple scattering model," *IEEE Trans. Geosci. Remote Sens.*, vol. 30, no. 5, pp. 1015–1024, Sep. 1992.

[12] B. B. Thompson, R. J. Marks, M. A. El-Sharkawi, W. J. Fox, and R. T. Miyamoto, "Inversion of neural network underwater acoustic model for estimation of bottom parameters using modified particle swarm optimizers," in *Proc. Int. Joint Conf. Neural Netw.*, 2003, pp. 1301–1306 [Online]. Available: http://www.ecs.baylor.edu/faculty/marks/REPRINTS/2003-07_InversionOfNeuralNetworkUnderwater.pdf

[13] V. Spichak and I. Popova, "Artificial neural network inversion of magnetotelluric data in terms of three-dimensional earth macroparameters," *Int. Geophys. J.*, vol. 142, no. 1, pp. 15–26, Jul. 2000.

[14] J. N. Hwang, J. J. Choi, S. Oh, and R. J. Marks II, "Query based learning applied to partially trained multilayer perceptrons," *IEEE Trans. Neural Netw.*, vol. 2, no. 1, pp. 131–136, Jan. 1991.

[15] R. C. Eberhart, Y. Shi, and J. Kennedy, *Swarm Intelligence*. San Mateo, CA: Morgan Kaufmann, 2001.

[16] SRC Computers, Inc., Colorado Springs, CO [Online]. Available: http://www.srccomputers.com

[17] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Proc. 6th Int. Symp. Micro Machine Human Sci.*, Oct. 1995, pp. 39–43.

[18] R. Eberhart and J. Kennedy, "Particle swarm optimization," in *Proc. IEEE Int. Conf. Neural Netw.*, Nov. 27–Dec. 1 1995, vol. 4, pp. 1942–1948.

[19] E. Ros, E. M. Ortigosa, R. Agis, R. Carrillo, and M. Arnold, "Real-time computing platform for spiking neurons (RT-spike)," *IEEE Trans. Neural Netw.*, vol. 17, no. 4, pp. 1050–1063, Jul. 2006.

[20] Y. Maeda and M. Wakamura, "Simultaneous perturbation learning rule for recurrent neural networks and its FPGA implementation," *IEEE Trans. Neural Netw.*, vol. 16, no. 6, pp. 1664–1672, Nov. 2005.

[21] N. Mtetwa and L. S. Smith, "Precision constrained stochastic resonance in a feedforward neural network," *IEEE Trans. Neural Netw.*, vol. 16, no. 1, pp. 250–262, Jan. 2005.

[22] D. Anguita, A. Boni, and S. Ridella, "A digital architecture for support vector machines: theory, algorithm, and FPGA implementation," *IEEE Trans. Neural Netw.*, vol. 14, no. 5, pp. 993–1009, Sep. 2003.

[23] M. Bracco, S. Ridella, and R. Zunino, "Digital implementation of hierarchical vector quantization," *IEEE Trans. Neural Netw.*, vol. 14, no. 5, pp. 1072–1084, Sep. 2003.

[24] Z. Nagy and P. Szolgay, "Configurable multilayer CNN-UM emulator on FPGA," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 50, no. 6, pp. 774–778, Jun. 2003.

[25] Xilinx, Inc., "Virtex-II platform FPGAs: Complete data sheet," San Jose, CA, 2005 [Online]. Available: http://www.xilinx.com/bvdocs/publications/ds031.pdf

[26] H. Hikawa, "A digital hardware pulse-mode neuron with piecewise linear activation function," *IEEE Trans. Neural Netw.*, vol. 14, no. 5, pp. 1028–1037, Sep. 2003.

[27] M. T. Tommiska, "Efficient digital implementation of the sigmoid function for reprogrammable logic," in *Inst. Electr. Eng. Proc. Comput. Digit. Tech.*, Nov. 2003, vol. 150, no. 6, pp. 403–411.

[28] M. Martincigh and A. Abramo, "A new architecture for digital stochastic pulse-mode neurons based on the voting circuit," *IEEE Trans. Neural Netw.*, vol. 16, no. 6, pp. 1685–1693, Nov. 2005.

[29] R. Duren, D. Fouts, and D. Zulaica, "Performance comparison CORDIC implementations on the SRC-6E reconfigurable computer," presented at the 2003 MAPLD Int. Conf., Washington, D.C., Sep. 9–11, 2003 [Online]. Available: http://www.klabs.org/richcontent/MAPLDCon03/MAPLDCon03.html, unpublished

[30] R. Duren, D. Fouts, and D. Zulaica, "Algorithm and programming considerations for embedded reconfigurable computers," presented at the 2003 7th Annu. Workshop High Performance Embedded Comput., Lexington, MA, Sep. 23–25, 2003 [Online]. Available: http://www.ll.mit.edu/HPEC/pdfs/cfp03.pdf, unpublished

[31] H. Diab, M. Huang, K. Gaj, T. El-Ghazawi, and N. Alexandridis, "An automated pipeline balancing in the SRC reconfigurable computer and its application to the RC5 cipher breaking," presented at the 2004 MAPLD Int. Conf., Washington, D.C., Sep. 8–10, 2004 [Online]. Available: http://www.klabs.org/mapld04/index.html, unpublished

[32] R. J. Marks II, S. Oh, and L. E. Atlas, "Alternating projection neural networks," *IEEE Trans. Circuits Syst.*, vol. 36, no. 6, pp. 846–857, Jun. 1989.

[33] SRC-6 C Programming Environment ver. v1.7 Guide, SRC Computers, Inc., Colorado Springs, CO, 2004.

[34] H. Hahn, D. Timmermann, B. J. Hosticka, and B. Rix, "A unified and division-free CORDIC argument reduction method with unlimited convergence domain including inverse hyperbolic functions," *IEEE Trans. Comput.*, vol. 43, no. 11, pp. 1339–1344, Nov. 1994.

[35] J. Zhu and P. Sutton, "FPGA implementation of neural networks—A survey of a decade of progress," in *Proc. 13th Int. Conf. Field-Programmable Logic Appl.*, 2003, pp. 1062–1066 [Online]. Available: http://eprint.uq.edu.au/archive/00000827/

[36] M. Clerc and J. Kennedy, "The particle swarm—Explosion, stability and convergence in a multidimensional complex space," *IEEE Trans. Evol. Comput.*, vol. 6, no. 1, pp. 58–73, Feb. 2002.

[37] J. F. Wakerly, *Digital Design Principles and Practices*, 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 2000, pp. 730–733.

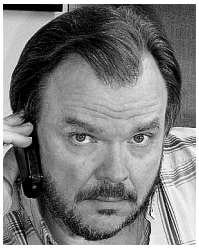
[38] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*. Cambridge, U.K.: Cambridge Univ. Press, 1992, pp. 274–300.

[39] P. D. Reynolds, "Algorithm implementation in FPGAs demonstrated through neural network inversion on the SRC-6e," M.S. Thesis, Dept. Eng., Baylor University, Waco, TX, May 2005.

Russell W. Duren (S'76–M'78–SM'96) received the B.S. degree in electrical engineering from the University of Oklahoma, Norman, in 1978 and the M.S. and Ph.D. degrees in electrical engineering from Southern Methodist University, Dallas, TX, in 1985 and 1991, respectively.

He spent 17 years in industry. The majority of this time was spent designing avionics at the Lockheed Martin Aeronautics Company, Fort Worth, TX. After that, he spent seven years teaching and performing research in the fields of avionics and reconfigurable computing at the Naval Postgraduate School, Monterey, CA. Currently, he is an Associate Professor in the Department of Electrical and Computer Engineering, Baylor University, Waco, TX. He is the author of over 30 publications. His research interests include avionics, embedded systems, FPGA digital design, and reconfigurable computing.

Dr. Duren is the recipient of the 1991 Frederick E. Terman Award for Outstanding Electrical Engineering Graduate Student from Southern Methodist University, the 1991 Myril B. Reed Outstanding Paper Award from the 34th IEEE Midwest Symposium on Circuits and Systems, and the 2002 Naval Postgraduate School Award for Outstanding Instructional Performance.



Robert J. Marks II (S'71–M'72–SM'83–F'94) is the Distinguished Professor of Engineering at the Department of Engineering, Baylor University, Waco, TX. He is a founding Member of the University of Washington's Christian Faculty Network. He is an Associate Member of Christian Leadership Ministries and served as the faculty advisor to the University of Washington's chapter of Campus Crusade for Christ. He has over 300 publications. Some of them are very good. Seven of his papers have been reproduced in volumes of collections of outstanding papers. He has three U.S. patents in the field of artificial neural networks and signal processing. He's also written some books.

Dr. Marks is Fellow of The Optical Society of America. He was awarded the Outstanding Branch Councilor award by IEEE and was presented with the IEEE Centennial Medal. He was named a Distinguished Young Alumnus of Rose-Hulman Institute of Technology and is an inductee into the Texas Tech Electrical Engineering Academy. In 2000, he was awarded the Golden Jubilee Award by the IEEE Circuits and Systems Society. He is also the first recipient of the IEEE Neural Networks Society Meritorious Service Award and the first honorary member of the Puget Sound Section of the Optical Society of America. He was also corecipient of a NASA Tech Brief Award for the paper "Minimum Power Broadcast Trees for Wireless Networks," and the Judith Stitt Award for best paper at the American Brachytherapy Society 23rd Annual Meeting. He served as a Distinguished Lecturer for the IEEE Computational Intelligence Society. He served a six-year stint of the Editor-in-Chief of the IEEE TRANSACTIONS ON NEURAL NETWORKS.

Paul D. Reynolds (S'00) received the B.S. and M.S. degrees in electrical and computer engineering from Baylor University, Waco, TX, in 2004 and 2005, respectively. Currently, he is working towards the Ph.D. degree in electrical engineering at Stanford University, Palo Alto, CA.

He worked as a Co-Op Engineer for L-3 Communications, Waco, TX, in 2003 and 2005, as a Research Assistant to Dr. R. W. Duren from 2004 to 2005, and, most recently, as a Design Engineer for Rosedale Medical, Inc., Cupertino, CA, designing a glucose meter.

Matthew L. Trumbo (S'02) received the B.S. and M.S. degrees in electrical and computer engineering from Baylor University, Waco, TX , in 2004 and 2006, respectively.

Currently, he is an Image Analytics Specialist within the Image Architecture Team, Pelco, Inc., Fort Collins, CO. He continues to be enthralled by topics within the computational intelligence field and image processing advances.