Abstract—FPGAs offer a promising platform for the implementation of Artificial Neural Networks (ANNs) and their training, combining the use of custom optimized hardware with low cost and fast development time. However, purely hardware realizations tend to focus on throughput, resorting to restrictions on applicable network topology or low-precision data representation, whereas flexible solutions allowing a wide variation of network parameters and training algorithms are usually restricted to software implementations. This paper proposes a mixed approach, introducing a system-on-chip (SoC) implementation where computations are carried out by a high efficiency neural coprocessor with a large number of parallel processing elements. System flexibility is provided by on-chip software control and the use of floating-point arithmetic, and network parallelism is exploited through replicated logic and application-specific coprocessor architecture, leading to fast training time. Performance results and design limitations and trade-offs are discussed.

I. INTRODUCTION

Artificial neural networks (ANN) are highly parallel computational structures that implement parameterized non-linear functions of several variables. Their most remarkable feature is their generalization capability: when presented with a set of sample input-output pairs roughly describing the desired mapping, they are able to “learn by example” and properly interpolate results from previously unseen input data. This makes neural networks an excellent choice for the modeling of complex systems or classifiers which cannot be described through closed formulae, and real-time adaptive modeling of evolving systems. Sample applications of ANNs include equalization in communications [1], character and speech recognition [2-3], and medical imaging [4].

Typical ANNs are structured as a number of sequential layers of simple, identical computational elements called neurons, which process the outputs from the previous layer in parallel. Implementations of ANN on sequential general-purpose processors are necessarily inefficient because they cannot take advantage of parallelism, which is the main structural property of neural networks. Hence, many research efforts have been expended in the last decades on the development of custom neural hardware in an attempt to overcome the limitations of pure sequential processing. A large number of ASIC architectures have been proposed for the implementation of ANNs, ranging from early analog realizations [5] to modern network-on-chip (NoC) platforms [6]. Recently, since FPGAs have reached a level of density that allows them to store whole networks, they have arguably become a more interesting platform for neural hardware: although ASICs are capable of reaching much higher performance, the evolution of ASIC-based neurocomputers is slow due to increased development time and low demand, whereas improvements in available FPGA technology are much faster thanks to more widespread use [7].

Most of the efforts on optimization of hardware ANN architectures have been concentrated on the implementation of the recall phase, i.e. of already-trained networks, relying on the training phase being performed off-chip using a software algorithm on a different platform. However, network training algorithms present the same parallel and highly interconnected structure as the recall phase, and so are equally well-suited for hardware implementation. ANN training is much more expensive computationally and requires additional functional units, hence FPGA realizations such as [8] tend to rely on heavy optimization and simplification procedures in order to increase throughput and logic efficiency at the expense of flexibility and generality of application. The following restrictions are especially important:

• Data representation: The vast majority of FPGA implementations of ANNs employ fixed-point arithmetic for the internal representation of values; when no other constraints apply, 16 bit values are typically used for network parameters (weights), since this is widely regarded as the minimum precision that guarantees network generalization capability [9]. This format is well-suited for FPGA implementation, as state-of-the-art FPGA devices often include multiple fixed-point multiply-and-accumulate (MAC) cores. However, even though it is argued in [10] that floating-point implementations of neural training on FPGA may be impractical, it is also claimed that floating-point...
precision results in faster and more reliable network convergence, due to its ability to represent numbers with a very small absolute value. Also, the use of a high precision format obviously widens the range of application of the training system.

• Network topology: Efficient implementations of network training that focus on the maximization of throughput normally do so by completely fixing, or at least severely limiting, the network structure at system generation time. Changing the network topology requires modification of the underlying hardware and hence implies system regeneration and device reconfiguration. This is a major drawback due to the extreme difficulty of determining an appropriate topology for a given problem prior to the actual training; there is no established methodology for doing so except for very low-dimensional problems [11]. The selection of an appropriate network topology has a severe impact on the performance of the final neural system: if the network is too small, it may not have enough free parameters to correctly model the underlying target function, whereas a network larger than necessary may lead to overfitting and result in poor generalization on non-training data. There exist evolutionary algorithms for the selection of an optimal network structure for a given problem, especially hybrid genetic approaches [12], but they require the base training algorithm to be executed on different network topologies at some point.

The goal of this paper is to present a training system on FPGA that can be used to establish the optimal neural network solution for a given problem. The previous discussion implies that such a system must allow the training of arbitrary network topologies and should employ floating-point arithmetic. As far as we know, those characteristics can only be found in purely software (hence less efficient) implementations, or in much more expensive neurochips or processor arrays. We propose a mixed approach, where custom FPGA logic is employed as a hardware framework for efficient computations, and flexibility in the selection of topology is provided by modification of software code, without resorting to hardware reconfiguration.

Our first conception of such a system [13] involved a multiprocessor system-on-chip where a number of separate embedded processors where enhanced with custom neural hardware in an attempt to speed up computations. However, the efficiency in this approach is limited by two principal factors. On one hand, communication operations in software suppose a significant execution overhead. On the other hand, the embedded processors’ interface for custom hardware extensions, based on “new task”/”task finished” handshake signals, is somewhat restrictive and does not lend itself well to instruction pipelining, leading to low logic efficiency. Those considerations suggest the concentration of all computations in a single, independent neural coprocessor with a direct hardware scheme for data distribution and optimized pipelining. The aim of this paper is to investigate this new approach.

II. DESCRIPTION OF ANN TRAINING

A. Multilayer Perceptron

The system architecture proposed in this paper may be applied to a variety of ANN types that allow batch training operation, but the current implementation is constrained to the Multilayer Perceptron (MLP), one of the most widely used neural network paradigms. A MLP comprises several layers (typically two or three) of similar simple processing elements, referred to as neurons. Each neuron uses the previous layer’s neurons as inputs, computes a weighted sum and transforms the result by means of a bounded non-linear activation function $\phi$, whose purpose is to limit the range of the neuron’s output. The transfer function for a neuron $k$ is thus given by

$$o_k = \phi(v_k), \quad v_k = b_k + \sum_j w_{jk} o_j$$

where the sum runs over all neuron inputs, and $o_j$ denotes the output of neuron $j$; if neuron $k$ is in the first layer, then $o_j$ runs over all network inputs. The network’s free parameters are the weights $w_{jk}$ and biases $b_j$ in each neuron, and the MLP topology is completely described by the numbers of layers and of neurons in each layer. We shall use the notation $N_0 / N_1 / \ldots / N_M$ to refer to a MLP with $N_0$ inputs, $N_M$ outputs and $M$ layers of neurons, with the $i$-th layer consisting of $N_i$ neurons.

B. Backpropagation

Network training is the process of adaptation of the free parameters in such a way that the network’s global transfer function approaches some specific target behavior. This target function is defined by means of a set of $V$ training vectors $[x_i, y_i]_{i=1}^V$, representing sample inputs $x_i$ and their associated desired network outputs $y_i$. The set of network weights $W$ is adjusted iteratively with the goal of minimizing the mean square error (MSE)

$$E(W) = \frac{1}{2V} \sum_{i=1}^V \|F(x_i; W) - y_i\|^2$$

where $F$ is the MLP’s transfer function. To do this, computation of the gradient $VE$ is needed. This is most efficiently accomplished using the backpropagation algorithm [14], where neuron errors are propagated through the network layers in reverse order as follows:

Fix a training vector $(x_i, y_i)$. Starting at the output neurons, a local gradient is obtained as

$$\delta_i = \phi'(v_i) \cdot e_i$$
where $e_j$ is the neuron’s error, i.e. the difference between the estimated output $\phi(v_j)$ and the desired output from the training vector. The local gradients in hidden layers are computed iteratively according to the formula

$$
\delta_j = \phi'(v_j) \left[ \sum_k w_{jk} \cdot \delta_k \right]
$$

(4)

where the sum runs over all neurons $k$ in the next layer. Finally, the gradient for weight $w_{jk}$, connecting neuron/input $j$ with neuron $k$ in the next layer, is given by

$$
\frac{\partial E}{\partial w_{jk}} = o_j \cdot \delta_k
$$

(5)

The gradient for the bias $b_k$ is equal to $\delta_k$.

The backpropagation algorithm can thus be structured into three distinct phases that have to be executed sequentially:

- **Forward phase:** The outputs (and derivatives) in each neuron are computed recursively, from the first to the last layer.
- **Backward phase:** The local gradients $\delta_j$ in each neuron are computed recursively, backwards from the last to the first layer.
- **Gradient phase:** Gradients for each free parameter are computed using (5). Computations for this phase may be arranged in any order.

The complete gradient for one epoch, i.e. presentation of the whole training set, is obtained by averaging the partial contributions from all training vectors:

$$
\frac{\partial E}{\partial w_{jk}} = \frac{1}{V} \sum_{i=1}^{V} \frac{\partial E}{\partial w_{jk}}
$$

(6)

C. Resilient Propagation

One common approach to weight adjustment is MSE minimization by standard gradient descent, where the weights are updated by subtracting a multiple of their respective gradients from them after each epoch:

$$
W^{(e+1)} = W^{(e)} - \mu \cdot \nabla E(W^{(e)})
$$

(7)

However, this procedure may provide very slow global network convergence due to a few neurons becoming saturated, i.e. having outputs close to the bounds given by the activation function and very small derivatives, leading to small weight updates between epochs.

A number of modifications of the weight update mechanism have been proposed in order to address this issue. We have selected the Resilient Propagation (RPROP) algorithm [15], where the magnitude of each weight update is kept independent of the gradient; instead, the last weight update is stored as reference and amplified or reduced depending on whether the gradient maintained or changed its sign. RPROP is reportedly faster than gradient descent by an order of magnitude, and allows a very efficient hardware implementation.

D. Activation Function

The theoretical background justifying the use of MLPs to solve arbitrary modeling problems is given by the Universal Approximation Theorem [16], which states that any given continuous mapping may be approximated, with arbitrary accuracy on a bounded domain, by a suitable MLP with two layers, as long as the activation function $\phi$ is bounded, monotone, and continuously differentiable. Besides, convergence has been shown to be faster if $\phi$ is an odd bipolar function [17]. The most common activation function is the hyperbolic tangent

$$
\phi(t) = \frac{e^t - e^{-t}}{e^t + e^{-t}}
$$

(8)

which satisfies all of the aforementioned hypothesis. However, this function does not lend itself to an efficient digital implementation, requiring large operators to implement exponentials and division.

Most ANN realizations implement the activation function as either a look-up table (LUT) of a low-order approximation of $\phi$, such as a piecewise linear approximation [18], but these approaches are not well-suited for our situation. A LUT with floating-point precision would require too many memory resources, and piecewise linear approximations, while useful for hardware realizations of the recall phase of MLPs (i.e. of pre-trained networks with fixed weights), are inadequate for the implementation of the training phase, since they don’t satisfy the hypothesis of the universal approximation theorem, thus hurting network convergence.

Our solution has been to choose a modified activation function $\tilde{\phi}$, designed as an odd cubic spline approximation of the hyperbolic tangent $\phi$, with fixed exact values at abscissae 0, 0.25, 0.5, 1, 1.5, 2 and 3, saturation at 4, and fixed derivatives at the extreme points. This is a valid activation function since it satisfies all conditions stated previously, so it provides valid ANNs with correct training. Moreover, an exact implementation of the function is feasible in an efficient manner, based on repeated MAC operations. Also, this activation function may be approximated by $\phi$ on other platforms, if needed, with an absolute error below $10^{-3}$. 


III. TRAINING SYSTEM ARCHITECTURE

This paper proposes the system architecture depicted in Fig. 1. A large high-performance neural coprocessor performs all necessary computations, while an embedded software-driven master processor specifies coprocessor operation by controlling its data and instruction input streams. A different set of coprocessor instructions is compiled by the master processor for each training session, depending on the desired network topology. The set of training vectors and the generated coprocessor program are streamed into the coprocessor using DMA devices in order to minimize communication overhead. The neural coprocessor benefits primarily from training-set parallelism, being capable of processing a large number of training vectors in parallel and automatically combining the resulting gradients thanks to replicated logic. Network weights and epoch results (network gradients) are stored in memories external to the coprocessor’s internal logic. The weight update algorithm is executed by the master processor, using an additional memory block to store RPROP variables.

A. Neural Coprocessor Architecture

The contents of the designed neural coprocessor are shown in Fig. 2. The basic elements are the arithmetic processing units (PU), who perform the same coprocessor instructions in parallel and are capable of processing up to 8 different vectors at any given time. The number \( P \) of processing units is a generic parameter of the design, and the coprocessor is able to execute the training algorithm on batches of \( 8P \) vectors at a time.

Processing units contain internal memory blocks employed as large register banks where all intermediate results are stored. These need to be uniquely addressed in order to set up initial values or retrieve individual results. An Active Data Flow Register specifies the active PU for these individual read/write instructions.

Input and output data streams are implemented as FIFO queues and the integrated, directly externally accessible memory blocks are implemented as dual-port memories. This allows the internal coprocessor logic and the rest of the FPGA system to run on independent clocks. Additionally, a dedicated control port allows access to several control registers. Most importantly, a Valid Data Level Register specifies how many training vectors are currently being processed (up to \( 8P \)), so that only results coming from valid vectors are considered when combining gradients. Additional control signals allow clearing the contents of gradient memory and automatic endianness conversion for incoming and outgoing data streams, which can be useful when considering communication with a remote host using a different byte order in 32-bit words.

B. Processing Units

The description of the contents of each PU is presented in Fig. 3. Its basic blocks are the arithmetic units (adder and multiplier), a local memory block, and a reduced number of local registers. Every block implements an 8-cycle pipeline and accepts new data on every clock cycle. Thus, the PU operates on 8-cycle stages and acts on 8 independent time-multiplexed data streams at each stage.

The adder and multiplier units are instantiated as 32-bit floating-point IP cores. Inputs may be chosen from local registers, current arithmetic outputs, and hardwired constants (0 for the adder and 1 for the multiplier). If an arithmetic operator is not activated by an instruction, it does not get disabled; instead, it is set up as an 8-cycle shift register by feeding its current output back and using the hardwired constants. This is necessary to ensure that other functional units still read a stream of time-multiplexed data from the arithmetic operator’s output. A number of bypass signals are externally accessible, allowing the rearrangement of adder subsystems from different PUs.

Each PU includes three general-purpose local registers, named L0, L1 and L2, which hold values from either local or global weight memory that are to be used as inputs to the arithmetic operators. Two additional special-purpose registers C0 and C1 are used to store polynomial coefficients for the computation of the spline function \( \phi \). The abscissa value is taken from L0 and the coefficients are read from a small 512-bit LUT. Register C0 is used for multiplicative coefficients and C1 for additive ones. All registers are implemented as 8-cycle shift registers, except for L2, whose value is not a stream of time-multiplexed data but a constant value loaded from global memory. A dual-port memory block (one read and one write port) is included to store intermediate results. A set of 8 time-multiplexed memory blocks is simulated by coding the correct time slot into three bits from each port’s address input.

C. Coprocessor Programming Model

The neural coprocessor accepts two types of instruction words: short 32-bit and long 64-bit instructions. Short instructions are used to access individual positions in one of the processing units’ internal memory blocks, and are consumed at a rate of one per clock cycle, following a three stage instruction pipeline executing any number of the following sequential tasks: modifying the Active Data Flow Register, reading a value from the active data flow’s internal memory into the output queue, and loading the first word in the vector queue into an internal
memory address. The coprocessor is automatically stalled if the queues are full or empty, respectively, and the corresponding operation is enforced. From a coprocessor programming point of view, a homogeneous set of $8P$ data flows are available, with no external distinction whether data flows are allocated in different PUs or time slots.

Long instructions are divided into four instruction pipeline stages that are executed in sequential order: loading register values from memory, multiplication, addition, and storing results into local memory. These are 8-cycle stages, and new instructions are accepted at a rate of one every 8 clock cycles. The same instruction is executed on all PUs and all time-multiplexed data flows in each PU, turning the coprocessor into a Single-Instruction, Multiple-Data (SIMD) machine. These instructions follow a VLIW (Very Long Instruction Word)-like format, in that a sequence of operations is specified, all of which may or may not be executed. The addition stage comes after the multiplication stage, in order to support MAC operations. Control signals for the PU datapath are directly read from the instruction word with almost no need for additional control logic.

Data hazards are possible between memory read and write operations on consecutive instructions with data dependence, but no additional logic is included to handle these cases. Instead, the prevention of these issues is left to the software routine that compiles coprocessor code, which is responsible for generating an optimal hazard-free coprocessor program, inserting NOP instructions or ad-hoc data forwarding as needed.

The usual long instruction pipeline can be modified in order to implement an adder tree, making use of the external bypass signals shown in Fig. 3. In that case, the adder blocks in each PU are rearranged into a pipelined adder tree structure responsible of adding the multiplier outputs from all PUs together; this rearrangement of arithmetic connections is illustrated in Fig. 4 for $P = 8$. The resulting configuration is divided in two distinct operating phases, as outlined in Fig. 5. The first phase consists of $P - 1$ adder blocks in a logarithmic adder tree configuration where spatial accumulation is carried out, i.e. results from different PUs are combined, resulting in a stream of 8 time-multiplexed partial sums corresponding to addition of all results from each time slot. If $P$ is not a power of two, missing adders are replaced by shift registers. This phase needs $\log_2 P$ 8-cycle pipeline stages to complete. The second phase is responsible of temporal accumulation, combining partial sums from different time slots using the last available adder and shift registers. From
the eight available time slot resources, seven are used to implement a 3-stage logarithmic tree structure similar to phase one, finally obtaining the desired sum. The last time slot accumulates the result with the previous value in gradient memory. This second phase is independent of \( P \) and needs five 8-cycle stages to complete.

IV. SYSTEM SOFTWARE

This implementation of MLP training exploits parallelism in the neural network training problem mainly through job partitioning, i.e., using a single set of software instructions to simultaneously process a large number of different data flows. This restricts the system to batch-mode training, using training-set parallelism, and possibly the implementation of the recall phase on batches of independent input samples. System software is structured into two different software layers: the coprocessor program, written in coprocessor code and implementing the backpropagation algorithm, and the master processor program, written in C for the Nios II embedded processor core acting as system controller.

A. Coprocessor Program

Two different sections of coprocessor code are generated for MLP training. The first part of the code consists entirely of short instructions, responsible of loading values from the vector queue into the appropriate local memory position for each data flow. This code section provides an initialization of the backpropagation routine on all 8P data flows. These instructions are consumed at a rate of one per clock cycle, so it is important to ensure fast code transfer. Hence, these short instructions are stored in on-chip memory. The second part of the code, consisting of long instructions, provides an implementation of the backpropagation algorithm and gradient and square error accumulation. This section of the program has the highest impact on overall system performance, hence the training algorithm has been carefully analyzed with the goal of achieving an optimal translation into coprocessor instructions. This code section is larger and its instructions are consumed at a lower rate (two 32-bit words every 8 cycles), so it may be stored in off-chip memory with no performance degradation.

The coprocessor program takes advantage of the internal structure of the processing units to perform MAC operations and all associated data transfers in a single instruction. The activation function and its derivative are evaluated as a sequence of \( n \) MACs, where \( n \) is the polynomial degree:

\[
\tilde{\phi}(x) = a_n x^3 + a_2 x^2 + a_1 x + a_0 = (a_3 \cdot x + a_2) \cdot x + a_1) \cdot x + a_0
\]

\[
\tilde{\phi}'(x) = b_3 x^2 + b_1 x + b_0 = (b_2 \cdot x + b_1) \cdot x + b_0
\]

(9)

The designed PU architecture allows the simultaneous computation of both polynomials (the activation function and its derivative) by alternating the use of the adder and the multiplier, in such a way that in a given 8-cycle stage, the adder is evaluating \( \tilde{\phi} \) while the multiplier evaluates \( \tilde{\phi}' \), or vice versa. This allows the computation of both functions with \( 2n - 1 \) instructions (5 in our case).

B. Master Controller Program

Each training session begins with the initial weights being properly initialized and stored in the coprocessor’s weight memory. After that, both sections of coprocessor code are generated and stored. The coprocessor code is different for each network topology, hence the need for runtime compilation. Any MLP topology is supported, the only restriction being that all temporary variables for one vector must fit into its assigned local memory. Network training is executed based on an epoch loop, stopping after either a maximum epoch number is reached or the network’s MSE falls below a certain threshold. Each epoch begins by issuing a DMA transfer of the whole training set into the vector queue. After that, the regenerated coprocessor code is sent to the coprocessor via DMA. Since the code only loads and processes a batch of 8P training vectors, this operation must be iterated as many times as needed until the training set is exhausted. On epoch start, the control port is used to tell the coprocessor to overwrite gradient memory. On the last iteration it is necessary to update the Valid Data Level Register to reflect the correct number of vectors left, so that results from inactive data flows are ignored by the gradient accumulation process. In the end, the gradient memory contains the complete network gradient given by (6), except for the constant multiplicative factor \( 1/V \). This gradient is then used by the master processor to compute the new network gradients using RPROP.

V. PERFORMANCE EVALUATION

There is no value that is universally accepted as the single best performance metric in the ANN field. The most common metric for neural training systems is CUPS (Connection Updates Per Second), defined as the rate at which network parameters are updated. For batch training algorithms, updates from each training vector must be considered separately. Hence, this value may be computed as

\[
CUPS = \frac{W}{T_{\text{vector}}} = \frac{WV}{T_{\text{epoch}}} = \frac{WVN_{\text{epochs}}}{T_{\text{training}}}
\]

(10)

where \( W \) is the number of network weights. Here, \( N_{\text{epochs}} \) is the length of the training in epochs, and \( T_{\text{vector}}, T_{\text{epoch}} \) and \( T_{\text{training}} \) denote the time needed to process a single vector, an epoch, and the whole training session, respectively. CUPS is considered a somewhat objective performance metric because it is more or less independent of network size.

A theoretical value for system performance may be obtained by noticing that any implementation of the backpropagation algorithm can be divided into operations per weight and operations per neuron, such as computation of the activation
function. As the network size grows, the number of weights increases at a superlinear rate with the number of neurons (if the size ratios between layers remain constant, then $W$ actually increases quadratically with the number of neurons). Hence, for asymptotically large networks, only operations per weight need to be considered. Each weight is involved in either two or three multiplications or MACs: one in the forward stage, computing the associated neuron’s output (1); one in the backward stage, computing the sum involved in evaluation of local gradient (4), except for weights connecting the inputs to the first hidden layer, which are not involved in this phase; and finally, one in the gradient stage, obtaining that weight’s gradient. It follows that training time for large networks is basically $m$ operations per weight, with $m$ between 2 and 3 depending on network topology. Thus a theoretical bound on performance can be estimated by

$$\text{CUPS}_{\text{max}} = Pf/m$$

where $P$ is the number of simultaneous multiplications or MACs available and $f$ is the rate at which they can be carried out. In our system architecture, these values coincide with the quantity of PUs and the coprocessor’s internal clock frequency. There are two factors that prevent us from reaching that bound: first, neural networks are not infinitely large and so there is influence from the operations per neuron; second, the communication overhead needs to be considered too (in our case, the loading of training vectors, requiring $N_0 + N_M$ clock cycles per vector).

VI. IMPLEMENTATION AND RESULTS

The system has been implemented on an Altera Stratix II DSP development board, with a Stratix II EP2S60F1020C3 FPGA and 32 MB external memory for the training vector repository. A 32 KB internal memory for coprocessor instructions is necessary to contain the vector loading section of the program for all applicable MLP topologies. The system makes use of the on-board Ethernet controller to allow the master processor to be connected to an IP network, so that a remote computer can use the FPGA system to perform any network training by providing MLP topology, the training set, and training stop conditions. The main limitation on the maximum size of the coprocessor that fits into the FPGA comes from the amount of available on-chip memory blocks for the realization of local memories. Reducing the amount of local memory addressable by each data flow increases the maximal number of PUs, but restricts the system to the training of smaller networks. This establishes a trade-off between training speed and system generality that needs to be resolved at system generation time.

The system was first generated with small local memories in order to obtain maximal performance. It was established that 128 words (4 kbit) were enough for each data flow to allow the training of small networks with up to approximately 40 neurons. Under this condition, it is possible to fit up to $P = 20$ processing units in the coprocessor for this FPGA. Table I summarizes the resource occupation for the whole coprocessor, each PU and its individual components. Weight and gradient memory are allocated in the two available 64 KB M-RAM blocks. The coprocessor uses an internal 125 MHz clock while the rest of the system runs on a 100 MHz clock. Hence, the coprocessor has a peak performance of 2.5 GMACs (MAC operations per second), consistently reached during most of the execution of the backpropagation algorithm and gradient accumulation, and (11) gives a theoretical upper bound between 833 and 1250 MCUPS.

One of the largest topologies supported by this system is the 4/18/18/3 MLP. Training time has been measured for this network and three smaller topologies. Performance results are plotted in Fig. 6. Training performance increases approaching an asymptotical value as the training set grows larger, as expected from a system where parallelization speedup stems primarily from training set parallelism. Vector loading time is constant for a given problem, hence the fraction of execution time spent on computations is higher for more complex topologies; thus, system performance increases as the network size grows, up to a 691 MCUPS peak value for the largest topology.

Table II shows performance comparisons with other FPGA implementations of MLP training, as well as training on a standard PC using Matlab (although optimized coding could possibly increase speed by a factor of up to 10 on the PC). Only sources with explicit available CUPS values have been included. We remark that only our implementations ([13] and this one) and PC training feature floating-point precision and allow multiple network topologies without hardware reconfiguration.

In order to test our architecture’s performance on real applications with larger networks, the system was generated with different local memory sizes and number of PUs, and training performance was measured on several small- to medium-sized ANNs suggested for license plate recognition in [22]. Results are summarized in Table III, including number of PUs, coprocessor clock frequency and training speed for each network topology.

VII. CONCLUSIONS

An implementation of the backpropagation algorithm for ANNs is proposed, using a hardware coprocessor with replicated pipelined arithmetic operators to support the simultaneous processing of a large number of training vectors, controlled by
software to admit arbitrary network topologies. Training performance exceeds 690 MCUPS, a competitive value compared to other state-of-the-art FPGA implementations for fixed network topologies and less precise arithmetic representation. The main drawback is the large usage of on-chip memory, forcing a trade-off between training speed and size of allowed training topologies; maximal training performance can only be achieved by small networks with up to 40 neurons. It should be noted that our implementation is restricted to a particular FPGA development board – FPGAs with more memory resources and boards with different external memory structure should yield better performance.

TABLE I. RESOURCE OCCUPATION

<table>
<thead>
<tr>
<th>Module</th>
<th>ALMs</th>
<th>Regs.</th>
<th>Memory blocks</th>
<th>36x36 MCUPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coprocessor</td>
<td>20295</td>
<td>29393</td>
<td>164</td>
<td>2</td>
</tr>
<tr>
<td>PU</td>
<td>1280</td>
<td>1410</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Adder</td>
<td>540</td>
<td>438</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Multiplier</td>
<td>281</td>
<td>360</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Memory</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Registers</td>
<td>493</td>
<td>516</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

1 ALM is equivalent to 2 logic cells

TABLE II. COMPARISON WITH OTHER IMPLEMENTATIONS

<table>
<thead>
<tr>
<th>Source Platform</th>
<th>Clock</th>
<th>Arithmetic</th>
<th>MCUPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC Pentium IV</td>
<td>3 GHz</td>
<td>64-FLP</td>
<td>15</td>
</tr>
<tr>
<td>Previous [13]</td>
<td>50 MHz</td>
<td>32-FLP</td>
<td>20</td>
</tr>
<tr>
<td>This EP2C70</td>
<td>125 MHz</td>
<td>32-FLP</td>
<td>691</td>
</tr>
<tr>
<td>[8] X2V500</td>
<td>60 MHz</td>
<td>16-FLP</td>
<td>326</td>
</tr>
<tr>
<td>[20] EP1S20 + PC</td>
<td>25 MHz</td>
<td>16-FLP</td>
<td>254</td>
</tr>
<tr>
<td>[21] XC2VP20</td>
<td>100 MHz</td>
<td>16-FLP</td>
<td>432</td>
</tr>
</tbody>
</table>

FXP = fixed point, FLP = floating point arithmetic

TABLE III. MAXIMUM PERFORMANCE FOR DIFFERENT ANNS

<table>
<thead>
<tr>
<th>Topology</th>
<th>Required local memory size</th>
<th>PUs</th>
<th>Clock</th>
<th>MCUPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>50/60/100/36</td>
<td>710 words</td>
<td>2</td>
<td>166 MHz</td>
<td>88</td>
</tr>
<tr>
<td>420/26/38</td>
<td>688 words</td>
<td>2</td>
<td>166 MHz</td>
<td>116</td>
</tr>
<tr>
<td>108/50/35</td>
<td>433 words</td>
<td>5</td>
<td>137 MHz</td>
<td>261</td>
</tr>
<tr>
<td>24/15/36</td>
<td>249 words</td>
<td>10</td>
<td>133 MHz</td>
<td>340</td>
</tr>
<tr>
<td>15/10/8</td>
<td>85 words</td>
<td>20</td>
<td>125 MHz</td>
<td>491</td>
</tr>
<tr>
<td>16/20/9</td>
<td>121 words</td>
<td>20</td>
<td>125 MHz</td>
<td>641</td>
</tr>
</tbody>
</table>

ACKNOWLEDGMENTS

This work was supported by the Spanish Ministry of Science and Innovation under FPU Grant AP2006-04275 and project FPA2007-65013-C02-02 and the Education Department of the Valencian Government under project GVPRE/2008/082.

REFERENCES


