Proceedings of the 9th International Conference on Neural Information Processing (ICONIP'02), Vol. 2
Lipo Wang, Jagath C. Rajapakse, Kunihiko Fukushima, Soo-Young Lee, and Xin Yao (Editors)

NEURAL NETWORKS IN FPGAS
INVITED

Amos R. Omondi

School of Informatics and Engineering
Flinders University
Bedford Park, SA 5042
AUSTRALIA

Jagath C. Rajapakse

School of Computer Engineering
Nanyang Technological University
N4 Nanyang Avenue
SINGAPORE 639798

ABSTRACT

As FPGAs have increasingly become denser and faster, they are being utilized for many applications, including the implementation of neural networks. Ideally, FPGA implementations, being directly in hardware and having parallelism, will have performance advantages over software on conventional machines. But there is a great deal to be done to make the most of FPGAs and to prove their worth in implementing neural networks, especially in view of past failures in the implementation of neurocomputers. This paper looks at some of the relevant issues.

1. INTRODUCTION

FPGAs have steadily improved in capacity and performance, to their extent where they are now being used for large applications. In this paper we consider some of the issues involved in using FPGAs to implement neural networks. A major motivation in the use of FPGAs is that being hardware, such implementations will have significant performance advantages over software implementations on conventional machines. This, however, is not necessarily so: it is not the case that *any* hardware implementation will be better than a software one, especially given the high performances that current microprocessors have to offer. Section 2 of the paper looks at the hardware versus software issue and, in particular, highlights the need for proper benchmarking. In Section 3, we consider the issue of parallelism, which seems to be a major motivation in many proposed or implemented FPGA-neural networks. The key point made in that section is that the exploitation of parallelism, by itself, should not be a primary goal. Section 4 is a discussion of arithmetic; this is especially important, given that much of the computation in neural networks is arithmetic. However, the general area remains one in which relatively little work has been done. In Section 5 we present a case study of an implementation that we carried out and discuss some of the lessons learned. The last section is a summary.

2. HARDWARE OR SOFTWARE?

Several proposals have been made to implement neural networks on FPGAs on the grounds that neural computation requires high processing rates and that the hardware of FPGAs is better than the software (of a simulation) on a conventional processor. On this basis alone, FPGA implementations would fail for the same reasons that past ASIC implementations of neurocomputers failed. We have discussed this point elsewhere and will not emphasize it here. Nevertheless, a few remarks are in order: Although it is reasonable to claim, in general, that hardware implementations of a given algorithm will be faster that software implementations of of the same algorithm, the validity of such a claim depend on many factors, e.g. exactly what platforms are being compared. Current high-end, conventional off-the-shelf microprocessors have clock rates in excess of 1GHz and employ pipelining and parallelism (limited, though it may be) to execute several instructions in a single cycle. This gives an enormous processing rate. Furthermore, small-scale parallel configurations of such processors are readily available or can be constructed to given even higher performance. Therefore, carefully tuned software implementations of neural networks on such processors are perfectly capable of very high performance. This is even more true for DSPs, which are specialized for the types of arithmetic required for neural networks and are capable of higher performance than conventional processors on their target applications.

FPGAs generally cannot compete with ASICs as far as performance goes, and FPGA neural-networks implementations realized on the basis of performance claims need to be justified with the results of careful comparisons against implementations on processors such as those above. So far there are hardly any such results — at least, not substantial ones that convincingly argue the case — and much work remains to be done in this regard. Of course, one can readily find many results of the type "this neural-network implementation [FPGA or otherwise] performed X times bet-

ter than an implementation on this PC or that workstation". Such results have some limited value, e.g. when narrowly restricted to some particular application and are on hardware platforms of about the same generation, but their general worth is dubious. A close examination of such results will quickly reveal several problems: typically, the basis on which the benchmark applications were selected is never given; the choice of the platform against which the comparison is being made and the parameters of the platform are unspecified; comparisons are made between a neural-network implementation that is essentially running on bare hardware against a processor with many added layers of software; and so forth. What is urgently needed here is a thorough and systematic approach to benchmarking — of the standards that the primary computer-architecture community has now established. We believe that when adequate studies of that type are carried out, it will be hard to justify the use of FPGAs purely on the grounds of performance ("hardware is faster than software"). However, keeping in mind the rationale for FPGAs, it should be possible to justify them on performance *relative* to other factors, such as cost, design time, and flexibility. Designers should therefore pay more attention to these factors, in particular the the last, through reconfiguration of FPGAs for different types of neural network.

3. PARALLELISM

In the past, a great deal of work in the hardware implementation of neural networks (including neurocomputers) has centered around the exploitation of parallelism. The argument in such work is usually that neural networks are inherently parallel, and, therefore, there are advantages in exploiting this. Since FPGAs are also inherently parallel, in so far as a typical FPGA consists of identical cell-blocks that can operate in parallel, there appears to be a ready match between FPGAs and neural networks. Not surprisingly, therefore, many proposals have been made in this direction, including a few for neural supercomputers based on FPGAs. This may not necessarily be the best way to go: many of the arguments given for the exploitation of parallelism in neural networks in FPGAs are exactly the same ones given about a decade ago for doing the same thing in ASICs, and they will not yield fruitful results for much the same reasons that earlier efforts failed [2]. Parallelism is not an end to itself; performance is the main issue.

Whether or not a neurocomputer is realized through software simulations or in hardware does not matter, as long as the performance goals are satisfied. Similarly, whether or not an implementation or simulation is sequential or parallel does not matter as long as performance is adequate. There are three main points that need to be taken into account as far as the parallelism goes, but which tend to be glossed over: the first is that algorithm-to-hardware mapping is not always easy, and, therefore, sequential computation is to be preferred, provided the choice does not affect performance; the second is that algorithms inevitably have sequential parts, which means that Amdahl's Law applies and there are, therefore, limits to the usefulness of parallelism; and the third (for implementation in FPGAs) the performance gap between ASIC and FPGA technologies. The obvious implication of Amdahl's Law, it is more critical to expend effort on the those aspects of computations that have low parallelism, rather than emphasize, as currently tends to be the case, the highly parallel parts that can readily be mapped onto the parallel structures of FPGAs. In this regard the the latest FPGA chips offer an unexplored opportunities: several such chips contain both a typical reconfigurable component (which can be used for high parallelism) and an embedded core processor (which can be used for control and sequential computations). On such a platform, one cannot merely argue for the exploitation of parallelism, since the sequential processor may well be capable of doing a better job (on a given task) than the reconfigurable part. The challenge therefore is to carefully consider the tradeoffs and to find useful ways of exploiting both aspects of the hardware; most likely, the winning aspect of the FPGA will be its reconfigurability (for different tasks), rather than the parallelism that ot offers.

4. ARITHMETIC

Much of the computation that has to be carried out for neural networks consists, essentially, of arithmetic operations, such as matrix arithmetic (addition, subtraction, multiplication, and transposition), which are essentially just data movement and multiply-accumulate type of vector operations, operations for squares, absolute values, search min/max, rounding, normalization of weights, weight saturation, and some elementary functions (e.g. exponential, tanh, and sigmoid) . In contrast with other aspects of neural-network hardware implementations, whether in ASIC or FPGA, arithmetic is one area where much more work is required.

There are roughly three main ways in which arithmetic facilities may be provided for neural-network processing. One is to use combinations of basic neural networks (i.e. those with just weighted sums, threshold activations, and weight assignments) to realize the arithmetic operations. The second approach is to replicate a simple, special-purpose processor that can carry out all of the desired operations; this is the basis of many hardware neurocomputers. And the third is to use conventional (i.e. off-the-shelf) hardware. We have argued elsewhere that only the last approach is currently reasonable [2]; and in what follows it will be the main concern, particularly with respect to FPGAs.

For many of the arithmetic functions that need to be im-

plemented for neural networks, it is hard to find good comparative results for ASIC implementations. For example, it is doubtful that one can readily identify the best implementation of the sigmoid function for a given ASIC technology, even though this is an important function whose implementation has been studied for some time. One can find (or derive) some comparative results, but almost all of these are relatively crude, in so far as they are estimates based on numbers of gates (for cost) and gate delays (for performance), and so forth; the results might provide some rough guidance, but that is about all. In the case of FPGAs, even such results are practically non-existent and are urgently required. In the case of the sigmoid function, for example, there are there are no results readily available that compare (or on which one can compare) the possible implementations in FPGA. An additional problem that needs to be addressed is that many of the designs that have been developed over the years for implementing arithmetic functions have been optimized for FPGAs but have mostly been carried over unchanged into FPGAs. Moreover, most researchers in the area do not appear to keep track of continuing developments in computer arithmetic. Consequently, many implementations of arithmetic function in FPGA-neural networks are far from ideal.

Another critical area that needs much work is in the choice of various functions of neural networks. In many cases, functions have been chosen mainly for their value from a neural-network point of view and not with hardware implementation in mind; this is especially so for the more complex functions. Given that FPGAs offer many opportunities to consider functions that previously one would not have considered for hardware implementation, the imbalance now needs to be addressed.

5. A CASE STUDY

As a demonstration vehicle for discussion some issues that we think neural-network need to consider, with respect to hardware implementations, we will take an application, independent component analysis, whose FPGA-implementation we have studied [6]. Independent component analysis (ICA) transforms a multivariate random signal into a signal with components that are mutually independent, and in doing so eliminates higher-order statistical dependencies of the signal and provides components that are not correlated in the sense of higher-order statistics [1]. A common use of ICA is in blind signal separation, in which the input is taken to be a set of linear mixtures of independent sources, and the aim is to extract the independent sources with minimal assumptions about the original sources. Because it is used extensively, there is growing interest in implementing it efficiently. In the study, we looked at the implementation of *independent component neural networks* (ICNNs) for carrying out ICA.

5.1. The computation

The input signal $\mathbf{x} = (x_1, x_2, \ldots, x_n)^{\mathrm{T}}$ to the neural network is assumed to be a set of mixtures of independent sources or components (ICs) $\mathbf{c} = (c_1, c_2, \ldots, c_m)^{\mathrm{T}}$ where n and m are the numbers of input signals and of independent components, respectively. For simplicity, we consider the case of *complete ICA* in which $m = n$. We then have

$$\mathbf{x} = \mathbf{A}\mathbf{c} \tag{1}$$

where $\mathbf{A} = \{a_{ij}\}_{\mathrm{n\times n}}$ denotes the linear mixing matrix. The ICNN learns to produce the *demixing matrix* $\mathbf{W}$ such that $\mathbf{W} = \mathbf{A}^{-1}$ with with a minimal knowledge of $\mathbf{A}$.

Consider a single-layer ICNN with n nodes at the input and n neurons at the output layer. Let the weight matrix of the network be $\mathbf{W} = \{w_{ij}\}_{\mathrm{n\times n}}$ and the activation function of neuron i be $f_i(\cdot)$. Then the network output $\mathbf{y} = (y_1, y_2, \ldots, y_n)^{\mathrm{T}}$ is given by

$$\mathbf{y} = \mathbf{f}(\mathbf{u}) \tag{2}$$

where $\mathbf{f}(\mathbf{u}) = (f_1(u_1), f_2(u_2), \ldots, f_n(u_n))^{\mathrm{T}}$. The synaptic input to the neurons $\mathbf{u} = (u_1, u_2, \ldots, u_n)^{\mathrm{T}}$ is given by

$$\mathbf{u} = \mathbf{W}\mathbf{x} \tag{3}$$

After learning, the output components of the ICNNs will be independent. That is

$$p_{\mathbf{y}}(y_1, y_2, \ldots, y_n) = \prod_{i=1}^{n} p_{y_i}(y_i) \tag{4}$$

where $p_{y_i}(y_i)$ indicates the marginal density of the component y_i.

We consider three ICNNs optimizing three different contrast functions: maximizing mutual information (MI) between input and output signals of the network, minimizing divergence of the output (DO) of the network, and maximizing likelihood of the inputs (LI). It has been shown that the learning equation of networks can be reduced to

$$\Delta\mathbf{W} = \mu[\mathbf{W}^{-\mathrm{T}} + \mathbf{\Phi}\,(\mathbf{u})\mathbf{x}^{\mathrm{T}}] \tag{5}$$

where $\mathbf{\Phi}\,(\mathbf{u}) = (\Phi_1(u_1), \Phi_2(u_2), \ldots, \Phi_n(u_n))^{\mathrm{T}}$ and $\Phi_i(\cdot)$ is a nonlinear function. μ is the learning factor [3].

Bell and Sejnowski [4] proposed an ICNN that adapts to maximize information transfer between the input and output of the network (infomax criteria). In this case the learning equation is given by Eq. (5) with

$$\Phi_i(u_i) = \frac{f_i''(u_i)}{f_i'(u_i)} \tag{6}$$

where $f_i(\cdot)$ is the i th neuron's activation function and $f_i'(\cdot)$ is the first derivative of $f_i(\cdot)$.

Amari et al. proposed an ICNN that uses the Kullback-Leibler (K-L) divergence as the contrast function [5]. Here, the learning equation is given by Eq. (5) with

$$\Phi_i(u_i) = -\frac{3}{4}u_i^{11} - \frac{25}{4}u_i^9 + \frac{14}{3}u_i^7 + \frac{47}{4}u_i^5 - \frac{29}{4}u_i^3 \quad (7)$$

where u_i is the synaptic input to the i th neuron.

Lastly, Lee et al. have proposed an ICNN that uses the likelihood of inputs as the contrast function. If the synaptic inputs to the neurons are mutually independent, the likelihood of inputs is maximized [8]. The gradient-learning equation n this case is given by Eq. (5) with

$$\Phi_i(u_i) = \frac{p'_{u_i}(u_i)}{p_{u_i}(u_i)} \quad (8)$$

where $p_{u_i}(u_i)$ is the probability density function (pdf) of total synaptic input to the i th neuron.

5.2. Implementation

The particular device on we used in the study is is the Xilinx XCV812E, which consists of over 0.25 million logic gates (in about 20K slices) and around 1Mbits of RAM. We looked at two types of implementation: one based on combinational logic and one based on lookup tables.

Lookup-table

In the LUT approach, the implementations of the nonlinear functions $\phi_i(u_i)$, given the synaptic inputs u_is, are done by using a lookup table. The size of the lookup table depends on the number of bits in the synaptic input u_i. The synaptic input value is used as the reference in selecting the output of the nonlinear function, and the corresponding values of the nonlinear functions given by Eqs. (6) and (7) are stored in the RAMs in the design of two neural networks.

The storage does not map all possible input patterns to individual entries; if that were done, the table-size would be extremely large. Instead, advantage is taken of the fact that in many cases several input patterns will lead to the same output pattern and that many of the subsequent operations greatly truncate the results, in such a way that some distinct patterns end up producing the same results. Taking all the various numerical factors into account, a highly compressed table can be used.

Combinational logic (CL)

If the activation function of neurons is given by the sigmoidal $f_i(u) = a(1-e^{-bu})/(1+e^{-bu})$, then the nonlinear function $\phi_i(u)$ is given by

$$\begin{aligned}\phi_i(u_i) &= \frac{f''(u_i)}{f'(u_i)} \\ &= -b(1-e^{-bu}) \times \frac{1}{(1+e^{-bu})} \end{aligned} \quad (9)$$

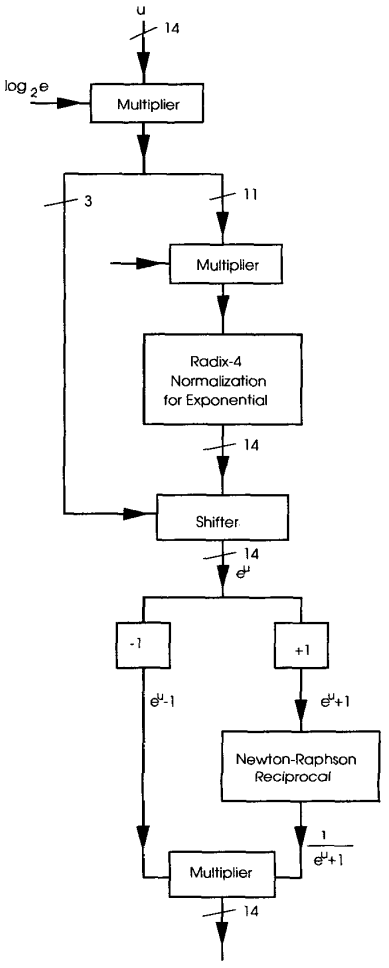

Figure 1: $\phi_i(u_i) = \frac{f''(u_i)}{f'(u_i)}$

For the implementation of the nonlinear function corresponding to the ICNN maximizing MI, the exponentials are computed using the radix-4 normalization (R4N) technique [7]. The division in Eq. (9) was carried out by using Newton-Raphson Reciprocation (NRR) and multiplication [7]. A block diagram of the implementation of the above function with $a = b = 1.0$ is given in Figure 1, in which

the synaptic input is normalized to lie between -1 and +1 before computation of the exponential and the final value is obtained by appropriate shifting of bits.

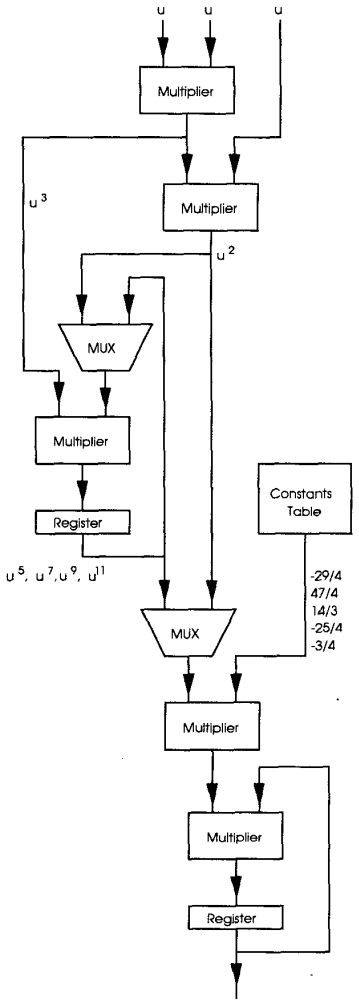

Figure 2: $\phi_i(u_i) = -\frac{3}{4}u_i^{11} - \frac{25}{4}u_i^9 + \frac{14}{3}u_i^7 + \frac{47}{4}u_i^5 - \frac{29}{4}u_i^3$

Small prototypes of the ICNNs, each consisting of several neurons, In order to implement the ICNN minimizing the DO, as seen in Eq. (7), a number of odd-order powers of u_i need to be evaluated. Since computation up to the 11th power is involved, the input values u_is were restricted between -2 and +2. As a number of powers of u is evaluated, the function value, given a synaptic value, was computed in 5 iterations using the design shown in Figure 2.

were implemented and compared in terms of cost and performance. Figure 4 is a block diagram of the implementation of a single neuron. As shown, each neuron is implemented in a pipeline of four processing stages: a RAM stage for the weight values, a stage that computes the synaptic inputs, a stage that computes (by table-lookup or arithmetic units) the nonlinear function ϕ in the learning equation, and a stage for the computation of the change of weights. Input signals stored in the RAM cycle through the loops until the weight-updating converges. Once the weights stabilize, the system produces the separated sources.

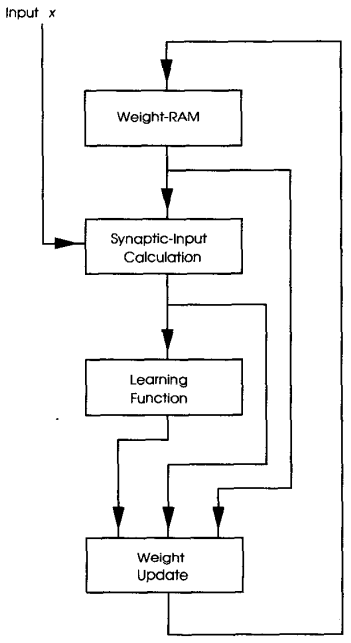

Figure 3: Implementation of one ICNN-neuron

The results (cost and performance figures) for a single neuron are given in Tables 1 and 2. Table 2 shows that the LUT approach is both faster and cheaper to implement than the other two approaches; however, in FPGA devices with limited memory, there may still be a case for not using a total-LUT approach. Table 2 also shows that the DO learning function is cheaper to implement than the MI function, which indicates the importance of the choice of the learning function: whereas at a theoretical level several learning functions may be taken to be equal, in so far as they yield similar results, for hardware implementation a further level of differentiation is required.

With combinational logic, it should be noted that it is possible to obtain faster and smaller implementations than those indicated by Tables 1 and 2. For example, in the DO

Implementation	Cost (slices)	Time (cycles)
Lookup-table	93 (DO)	1
	230 (MI)	1
Comb. logic (DO)	585	30
Comb. logic (MI)	1044	56

Table 1: Implementations of learning functions.

Unit	Cost (slices)	Time (cycles)
Weight-RAM	23	1
Input derivation	334	8
Δw-computation	352	5

Table 2: Implementation of other units of neuron.

implementation, all the multipliers were pre-synthesized multipliers, each consisting of carry-save adders (CSAs) and a carry-propagate adder (CPA). But given the structure of the implementation, it is possible to remove all but one of the CPAs, thus greatly speeding up the operation. Recoding would also further improve the speed of the multipliers, by trading off latency for throughput.

Perhaps just as important as the purely-hardware techniques, careful consideration ought to be paid to the choice of the learning function; for example, it would be helpful to a learning function with fewer terms in the polynomial but without a loss of accuracy. Techniques such as economization-of-power-series are useful in this regard, but once again, the indication is that a more hardware-directed approach to devising the learning function should be exercised. Overall, further investigation should be carried out on the implementation of such functions.

Techniques similar to those just described could also be used to realize a more effective implementation of the MI function. Furthermore, since division can be carried out normalization, the cost could be greatly reduced by using the same hardware for both exponentiation and division. What is more significant, however, is again the indication that careful decisions at the theoretical level can greatly assist in the hardware implementation. In the MI implementation, a great deal of the hardware is used just for range reduction (to pre-process the input and correspondingly post-process the output). Obviously it would be much better if it arranged for the input was in the proper.

6. SUMMARY

We have discussed several issues that we think are important for future implementations of neural networks in FPGAs. Four main points have been made. The first point is that careful benchmarking is required to determine the worth of hardware implementations of neural networks and, therefore, when best to use them. The second point is that implementation should look beyond just the exploitation of parallelism, in particular they address situations where high parallelism is not always available, and also take advantage of developments in technology. With both of these points, the key areas are, perhaps, to emphasize other primary aspects of FPGAs, such as reconfigurability (which implies evaluation with suites of applications, rather than one or a few applications), development time, and rapid-prototyping. The third point made is that the entire field of computer arithmetic, in the context of neural networks, needs to be thoroughly explored, especially for FPGA implementations. And the last point is that neural-network functions ought to be selected with hardware implementation in mind.

7. REFERENCES

[1] P. Comon, "Independent component analysis - a new concept?", *Signal Processing*, vol. 36, no. 3, pp. 287-314, 1994.

[2] A. R. Omondi, "Neurocomputers: a dead end?," *International Journal of Neural Systems*, vol. 10, no. 6, pp. 475-481, 2000.

[3] J. C. Rajapakse and W. Lu, "Unified approach to independent component neural networks", *Neural Computation*, 2000.

[4] A. J. Bell and T. J. Sejnowski, "An information-maximization approach to blind separation and blind deconvolution," *Proceedings of 1995 International Symposium on Nonlinear Theory and Application (NOLTA-95)*, pp. 43-47, 1995.

[5] S. Amari, A. Cichocki and H. Yang, "A new learning algorithm for blind signal separation," *Advances in Neural Information Processing Systems 8*, 1996.

[6] A. B. Lim, R. C. Rajapakse, and A. R. Omondi, "Comparative study of implementing ICNNs on FPGAs,", *Proceedings, International Joint Conference on Neural Networks*, pp. 177-182, 2001.

[7] A. R. Omondi, *Computer Arithmetic Systems*, Prentice-Hall, UK, 1994

[8] T-W. Lee, M. Girolami, and T. J. Sejnowski. "Independent component analysis using an extended Informax algorithm for mixed sub-gaussian and super-gaussian sources," *Neural Computation*, vol. 11, no. 2, pp. 409-433, 1999.