International Journal of Computational Intelligence Research.
ISSN 0973-1873 Vol.3, No.1 (2007), pp. 21–25

A Configware Approach for the Implementation of a LVQ Neural Network

Maurcio Kugler[1,2] **and Heitor S. Lopes**[2]
[1]Dept. Computer Science & Engineering, Nagoya Institute of Technology,
Nagoya-shi, Showa-ku, Gokiso-cho, 466-8555, Japan
mauricio@kugler.com

[2]Bioinformatics Lab., Federal University of Technology – Paraná,
Av. 7 de setembro, 3165 80230-901 Curitiba, Brazil
hslopes@pesquisador.cnpq.br

Abstract: **This paper describes a methodology for the implementation of a Learning Vector Quantization (LVQ) neural network in a Field Programmable Gate Array (FPGA) device, especially suited for applications requiring fast throughoutput. A special feature of the implementation is a combinatorial module for distance comparison that allows the execution of this important operation for a LVQ in just one clock cycle. The control code of the LVQ is described by a finite state machine and parametrically programmed in VHDL. The final neural network was implemented with 64 dimensions, 16 subclusters and 2 classes, using an ACEX1k100 reconfigurable device. Using this system with a clock rate of 25MHz, a full classification can be done in 334μs, thus enabling real-time performance for many real-world applications.** ***Keywords***: LVQ, FPGA, Hardware.

I. Introduction

In current embedded systems, it is seldom necessary to process large amount of information in real-time, such as in image and signal processing or in dynamical control systems. Common algorithms for processing and classification of signals may require a reasonable computational power, which cannot be achieved using microprocessors commonly used in embedded systems. This is due to cost or, more frequently, to some technical factor (power consumption, clock frequency, physical dimensions, number of peripheral chips requested, etc). An alternative for the development of embedded systems that require high processing power is the hardware implementation of software algorithms using Programmable Logic Devices (PLDs). These devices can be programmed for a given functionality. Differently from common microprocessors that run software instructions sequentially in a predefined architecture (Von Neumann, Harvard, etc), PLDs have their internal structure defined by the designer, allowing the customization of both the architecture and the functionalities of the system for a given purpose. One of the most outstanding features of such reconfigurable devices is the possibility of designing several hardware blocks that operate in parallel (combinatorial or sequential hardware), increasing the processing power of the system. Nowadays, high-capacity PLDs based on non-volatile memories are known as FPGAs (Field Programmable Gate Arrays) and are becoming very popular due to cost and flexibility. Due to its availability and performance, FPGAs have been used in powerful reconfigurable systems. Therefore, systems based on reconfigurable hardware (or simply, configware) can offer custom-computing machines for specific applications, with orders of magnitude faster than regular software processing in general-purpose processors [1].

The objective of this work is to present a methodology for the implementation of a Learning Vector Quantization (LVQ) Neural Network (NN) using a reconfigurable device. LVQ NNs are frequently used for pattern recognition (see, for instance, [9, 10]), and are particularly interesting for hardware implementation since they are based on the calculation of a geometric distance among samples and reference vectors. This feature eliminates the necessity of multipliers that occupy a large amount of resources in reconfigurable components and request many clock cycles.

II. LVQ Neural Networks

The LVQ NN was created by Kohonen [8], and it is a method for training neural networks for pattern classification in which each output represents a particular class (although several outputs can also be used for a single class). Each class is referred by a vector of weights that, in turn, represents the center of the clusters defining the decision hypersurfaces of the classes. A given class can be defined by a single point or a set of them, for a better representation of irregular decision surfaces. For training this NN it is necessary a set of training patterns with known classes, together with an initial distri-

bution of the reference vectors. During training, the known class T of each input sample x is compared to the class C represented by the cluster center w that is the nearest to the sample. The center of the cluster w is updated according to equation 1, where $= alpha$ is the learning rate of the NN:

$$\begin{cases} If \;\; T = C \quad then \quad w_{new} = w_{prev} + \alpha.[x - w_{prev}] \\ If \;\; T \neq C \quad then \quad w_{new} = w_{prev} - \alpha.[x - w_{prev}] \end{cases} \tag{1}$$

Training is done for all input variables several times, always taken them in a random order. Usually, training is concluded when clusters get stable, or either a previously specified number of iterations is reached. Basically, after being trained, a LVQ NN becomes a vector comparator. Every new input will be assigned to a class which cluster center is the most similar to it. The similarity (or dissimilarity) measure of two generic points x and y can be implemented as the geometric distance between them. A general distance norm is given by equation 2, where n is the dimensionality of the space and w_i a weighting coefficient.

$$d_p(x, y) = \left\{ \sum_{i=1}^{n} w_i.(|x_i - y_i|)^p \right\}^{\frac{1}{p}} \tag{2}$$

The two most usual cases of equation 2 are the Euclidian distance ($p = 2$) and the Manhattan distance ($p = 1$), both using $w_i = 1$ (non-weighted). For applications that require a fast computation of distance, the Manhattan distance is clearly the most suitable, reducing equation 2 to:

$$d_1(x, y) = \sum_{i=1}^{n} w_i.|x_i - y_i| \tag{3}$$

III. Clustering and Subclustering

Clustering with NNs are more usual in unsupervised learning, where a previous knowledge about the class to which belongs every input vector is not available. In this case, clustering techniques are useful for revealing similarities among vectors, therefore creating groups for classification.

In the particular case when the class of input vectors are previously known, it is not adequate the direct application of clustering techniques. However, sometimes it is not interesting to represent each class using only one cluster center, since this could lead to a very bad accuracy rate of the classifier. Then, a problem emerges on how to initialize every cluster center such that the training time can be reduced without losing the initial classification set. A possible solution proposed in this paper is to separate each class into independent spaces, therefore creating several independent clustering problems. This technique we call ”subclustering”. In these isolated spaces, the centers of the obtained clusters are, in fact, subclasses centers. The difference between class and subclass should be stressed, since these concepts are important in both training and testing of the NNs. Considering each isolated class a new space, one can cluster the input vectors using any usual clustering technique. In this work we used the well-known k-means clustering algorithm that is usual for LVQ applications [6].

IV. Architecture of a LVQ NN in FPGA

Some works [2, 3, 5] have proposed generic implementations of NNs in hardware. They focused on generality instead of performance, in the sense that particular characteristics of each NN type are not taken into account. However when exploring specific features of a given NN architecture, better results can be achieved [7]. In this work, we explore the key point of a LVQ NN: the classification by means of a geometric distance comparison. This feature allows a significant reduction of the number of logical elements and processing time in reconfigurable hardware implementations.

The basic block that computes a geometric distance is shown in figure 1. Each element of the vector that represents each cluster center (components of the reference vectors of the NN) is sequentially applied to the subtractor input, together with the dimensions of the unknown sample to be classified. The module of the difference between these values is taken. It represents the distance between the points, in the current dimension of the space. The result is summed to the values of the accumulated distance of the other dimensions (see equation 3), in the first register. It is important to emphasize that all these operations are done in a single clock cycle. The output of this register is also applied to the input of a comparator, which has in its second input the value of the smallest distance to the center of a given cluster up to now. If the new distance is smaller than that previously stored, it is loaded in the second register. This operation also takes nothing more than a single clock cycle. When these operations are repeated sequentially for all clusters, the cluster closest to the sample can be found and, therefore, the class of the unknown sample.

A block diagram of the system, shown in figure 2, has three main blocks:

- $rom\_cluster$: this is a ROM (Read-Only Memory) that is used for storing the set of centers of the clusters, which were previously defined by means of training sessions;
- $fifo\_lvq$: this is a FIFO (First-In, First-Out memory) that is used for receiving new samples while the previous samples are being classified;
- lvq_ctrl: this is the control module of the system that implements the finite state machine (FSM) detailed in table 1.

The FSM is detailed in table 1, where CS, SD, EV, ED, NS stand respectively for: current state, state description, event, event description and next state. In the FSM, states $S1$ and $S2$ correspond to the transfer of the input sample from the

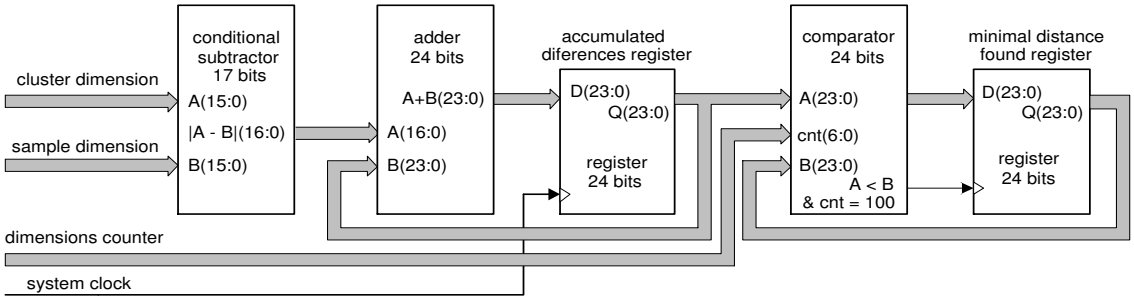

Figure. 1: Block diagram for minimal distance computation and register.

CS	SD	EV	ED	NS
S0	Initialization:	T0	LVQ_Start=0	S0
	cnt_dim=0; $cnt\_cluster$=0; $dist\_final$=7FFFh	T1	LVQ_Start=1	S1
S1	Enable FIFO memory reading; Signal FIFO memory as busy; Increment cnt_dim	T2		S2
S2	Write to data RAM value read from FIFO	T3	cnt_dim <100	S1
		T4	cnt_dim=100	S3
S3	Signal FIFO memory as free; cnt_dim=0	T5		S4
S4	Compute addresses of data and clusters memories for further reading, using cnt_dim and $cnt\_cluster$	T6		S5
S5	Load the value read from cluster memory; keep the address and the enable signal to ensure a correct reading	T7		S6
S6	Wait for valid data in the data RAM output	T8		S7
S7	Store the partial distance computed	T9	(cnt_dim=100 and (distance>the last one) and ($cnt\_cluster$ <40)	S3
		T10	cnt_dim <100	S4
		T11	(cnt_dim=100 and $cnt\_cluster$ =40)	S9
		T12	(cnt_dim=100 and (distance<the last one)	S8
S8	Store the new smallest distance	T13		S3
S9	Output the class of the selected cluster ($class\_out$); Signal LVQ_finish =1; Stop processing	T15		S0

Table 1: Tabular description of the FSM of the LVQ NN.

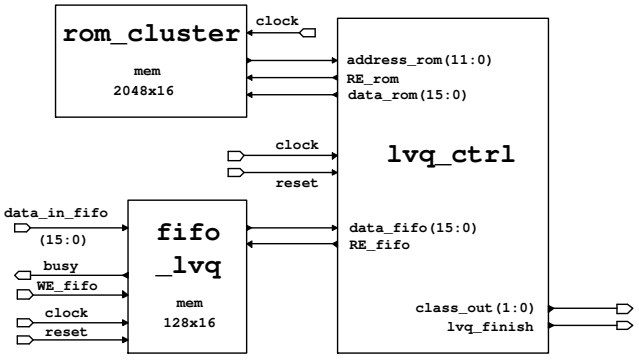

Figure. 2: Main blocks of the implemented LVQ NN.

FIFO to the internal RAM. States from $S3$ to $S8$ do the successive comparisons of each dimension of the input sample with the reference vectors, and store the reference vector with the smallest distance to the input sample. The final result of the classification is given in state $S9$. In principle, the implemented LVQ NN was limited to a maximum of 100 dimensions, 20 subclusters per class and 2 classes.

V. Hardware Implementation

The main drawback in implementing a LVQ NN in hardware is the large amount of memory necessary for storing the reference vectors. In our experiments, a $64/16/2$[1] network was implemented so as to fit in the device chosen. The synthesis of the proposed system was done with a low-cost ACEX1k100 device (Altera Corporation, San Jose, USA), with a 25 MHz clock rate. This device has 49152 bits of internal RAM and 4992 logic cells. The circuitry described in VHDL (VERILOG Hardware Description Language) was compiled, simulated and synthesized using Quartus II environment from Altera.

The LVQ NN is started by the lvq_start signal, when data in the FIFO are transferred to the internal RAM. This operation takes 5.12 μs and during this transfer, the $fifo\_busy$ signal indicates that the FIFO is blocked for writing.

In the comparison of the input vector with the first reference vector stored in the ROM there are three special coun-

[1]In this work, the convention used is "$d/s/c$" for representing, respectively, dimensions, subclusters per class and classes.

ters: cnt_dim (dimension counter), $cnt\_cluster$ (subcluster counter) and cnt_class (class counter). These counters are used to address memories when dimensions are compared sequentially. It takes 170 ns for the comparison of one dimension. Differences of each dimension are computed in dif_temp and later accumulated in dif_acc. In the comparison of the input vector with a center of a cluster that is identical to it, the accumulated differences will be null, but the temporary class indicator $class\_tmp$ changes, since the input vector is of the opposite class. Finishing the classification procedure, lvq_finish signal indicates the end of the LVQ processing and the final result is made available when $class\_out$ is activated. The LVQ total processing time (from lvq_start to lvq_finish, for a $64/16/2$ NN) is around 334μs, using a 25 MHz clock. As the number of points to be compared increases, processing time also increases proportionally. The number of points to be compared is represented by the product between the number of dimensions, the number of subclusters and the number of classes.

The implementation of a $64/16/2$ LVQ NN in an ACEX1k100 device required 284 logic cells and 34816 memory bits, corresponding, respectively, to 5.7% and 70.8% of the available resources. The parametric implementation of the code allows the easy setting of the three parameters (dimensions, subcluster per class and classes) for a given purpose, making it flexible and adaptable for several applications. Increasing the number of dimensions and/or subclusters per class also increases the use of the internal memory of the device, but does not increases significantly the number of logic cells used. Using 16-bit fixed-point notation, the number of memory bits necessary for the implementation is given by the sum of bits used by FIFO, internal RAM and ROM, and can be obtained using the expression: $M_T = 16.d.(2 + c.s)$, where: M_T is the number of total bits necessary in the device; d is the number of dimensions; c is the number of classes and s is the number of subclusters per class.

Figure 3 shows the limits allowed for the parameters of the implemented LVQ NN using the ACEX1k100 device. Both axes have dimensionless logarithmic scales that represent a large range of possible values for the number of dimensions (d) and number of subclusters per class (s) of a LVQ NN. Diagonal lines represent the number of classes, and indicate the allowable limit for the other parameters (dimensions and subclusters per class) that use around the total amount of memory bits of the specific device. In fact, a large number of classes (say, above 128) is not usual for real-world implementations. In the same way, small values for both axes are useless, and are shown only for illustration. Although figure 3 was constructed for a specific device, for another devices the same approach can be used, ensuring that a given combination of $d/c/s$ LVQ NN is feasible in that device.

The processing speed of the LVQ NN implemented in FPGA is a function of the total number of clock cycles necessary for the full classification of an input vector. This processing speed also depends on the number of dimensions, the number of subclusters per class, and the number of classes, and its growth has the same behavior as that presented in figure 3 for the memory resources.

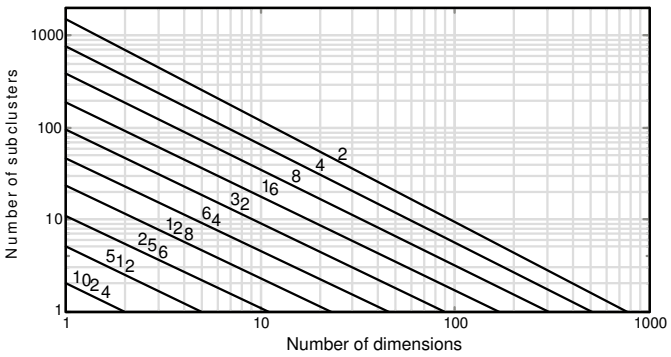

Figure. 3: Limits for the configuration parameters of a LVQ NN implemented in an ACEX1k100 FPGA device.

VI. Conclusions

In this work, we exploited a specific feature of LVQ NNs: the classification by means of a geometric distance comparison. Basically, this kind of NN is a vector comparator, and the use of the Manhattan distance yielded a significant reduction of the number of both logical elements allocated in the FPGA and the processing time. In the same way, for another types of NNs, a systematic analysis of their peculiarities may give inspiration to particular implementations using FPGA potentialities.

The implementation of a LVQ NN using FPGA possibly can achieve better results regarding speed, when compared with other conventional approaches with similar clock frequency and equivalent bus size. This clear advantage comes from the fact that in this implementation we used specific combinatorial circuits in parallel and most operations are done in few clock cycles. On the other hand, conventional processors spend precious time for fetching instructions, pipelining, testing branch conditions and memory addressing/storing/retrieving.

The limiting factor of this implementation is the amount of internal memory used for the reference vectors ($rom\_cluster$). However, for larger implementations, this problem can be easily circumvented using a device with more memory, or using an external memory, together with a circuitry for parallelizing the task of reading the next vectors while using the current one for classification.

This work has shown the feasibility to implement a LVQ NN in FPGA. The performance achieved in both size and speed encourages the application of this methodology for real-time embedded applications that require portable computational power. Further work will include the implementation of this system in other commercial devices of higher performance, and its application to pattern recognition problems, such as

those involved in real-time signal processing [9].

References

[1] Becker, J., Hartenstein, R.: Configware and morphware going mainstream. Journal of Systems Architecture **49** (2003) 127–142

[2] Blake, J.J., Maguire, L.P., McGinnity, T.M., Roche, B.,McDaid, L.J.: The implementation of fuzzy systems, neural networks and fuzzy neural networks using FPGAs. Information Sciences **112** (1998) 151–168

[3] Botros, N.M. and Abdul-Aziz, M.: Hardware implementation of an artificial neural network using field programmable gate arrays (FPGA's). IEEE Transactions on Industrial Electronics **41** (1994) 665–667

[4] Cox, C.E., Blanz, W.E.: Ganglion – a fast field-programmable gate array implementation of a connectionist classifier. IEEE Journal of Solid-State Circuits **27** (1992) 288–299

[5] Gorgoń, M., Wrzesiński, M.: Neural network implementation in reprogrammable FPGA devices – an example for MLP. Lecture Notes in Artificial Intelligence **4029** (2006) 19–28

[6] Huang, Y.S., Chiang, C.C., Shieh, J.W., Grimson, E.: Prototype optimization for nearest-neighbor classification. Pattern Recognition **35** (2002) 1237–1245

[7] Izeboudjen, N., Farah, A., Titri, S., Boumeridja, H.: Digital implementation of artificial neural networks: from VHDL description to FPGA implementation. Lecture Notes in Computer Science **1507** (1999) 139–148

[8] Kohonen, T., Huang, T.S., Schroeder, M.R. (Eds.): Self-Organizing Maps. Springer-Verlag, Heidelberg (2000)

[9] Kugler, M., Lopes, H.S.: Using a chain of LVQ neural networks for pattern recognition of EEG signals related to intermittent photic-stimulation. In: Proc. VII Brazilian Symposium on Neural Networks, IEEE Computer Society, Los Alamitos (2002) 173–177

[10] Ölmez, T., Dokur, Z.: Classification of heart sounds using an artificial neural network. Pattern Recognition Letters **24** (2003) 617–629