Design, Implementation, and Test of a Multi-Model Systolic Neural-Network Accelerator

THIERRY CORNU[1], PAOLO IENNE[1], DAGMAR NIEBUR[2], PATRICK THIRAN[1], AND MARC A. VIREDAZ[3]

[1]*Swiss Federal Institute of Technology, Centre for Neuro-Mimetic Systems, IN-J Ecublens, CH-1015 Lausanne, Switzerland; e-mail: {cornu,ienne,thiran}@epfl.ch*
[2]*Now with Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA; e-mail: niebur@telerobotics.jpl.nasa.gov*
[3]*Now with NEC Research Institute, 4 Independence Way, Princeton, NJ 08540; e-mail: viredaz@research.nj.nec.com*

ABSTRACT

A multi-model neural-network computer has been designed and built. A compute-intensive application in the field of power-system monitoring, using the Kohonen neural network, has then been ported onto this machine. After a short description of the system, this article focuses on the programming paradigm adopted. The performance of the machine is also evaluated and discussed. © 1996 by John Wiley & Sons, Inc.

1 INTRODUCTION

Neural networks are gaining recognition as a novel technique to solve large classes of problems better than by using traditional algorithms. One of the problems that neural networks encounter in practical applications is the huge computing power required. Conversely, one of the aspects that make neural networks interesting is their high degree of intrinsic parallelism. The union of these two elements is a solid ground for dedicated computers designed for connectionist algorithms [6].

This article presents a special purpose machine on which several popular neural algorithms can be run. The system achieves massive parallelism thanks to a systolic array with up to 40×40 *processing elements* (PEs). This article aims at outlining the many problems that arise in practice for a user to program and use this kind of machine. For this purpose, the hardware structure of the machine is overviewed in Section 2. Section 3 shows how the machine is programmed, from the implementation of low-level routines for the systolic array up to the user-library routines. Each of these software layers raises different problems in terms of performance: The array microcode has to exploit the hardware in the best way, whereas the higher-level routines should hide all the approximations and algorithmic modifications introduced by the dedicated hardware. The performance assessment is discussed in Section 4. Finally, the use of the system for an application of the Kohonen network in power-system security assessment is described in Section 5. Section 6 draws some conclusions on the whole of the project.

Received November 1994
Revised April 1995

Scientific Programming, Vol. 5, pp. 47–61 (1996)
CCC 1058-9244/96/010047-15

2 MANTRA I SYSTEM

The MANTRA I computer is a massively parallel machine dedicated to neural-network algorithms (Fig. 1). It has been designed to provide the basic operations for the following models: (1) single-layer networks (Perceptron and delta rule); (2) multilayer feedforward networks (back-propagation rule); (3) fully connected recurrent networks (Hopfield model); and (4) *self-organizing feature maps* (SOFMS; Kohonen model). A description of these algorithms can be found in any classic introductory book on neural networks (e.g., [4]). The Kohonen feature maps are used in Section 3 to illustrate how these algorithms are mapped on the system.

The MANTRA I accelerator is based on a bidimensional systolic array composed of custom PEs named GENES IV. In the present section, the hardware of the machine is overviewed starting from its system integration in a network of workstations and proceeding down to the internal architecture of the machine and of its computational core.

2.1 MANTRA I System Integration

The MANTRA I machine is controlled by a TMS320C40 *digital signal processor* (DSP) from Texas Instruments. Two of its six eight-bit built-in communication links connect the machine to another TMS320C40 processor inside a SUN SPARCstation (Fig. 2). From a software point of view, the intermediate DSP is transparent. The MANTRA I machine (the systolic array and its control processor) is completely controlled by the front-end workstation but could be easily integrated into any other computer system based on TMS320C40 processors.

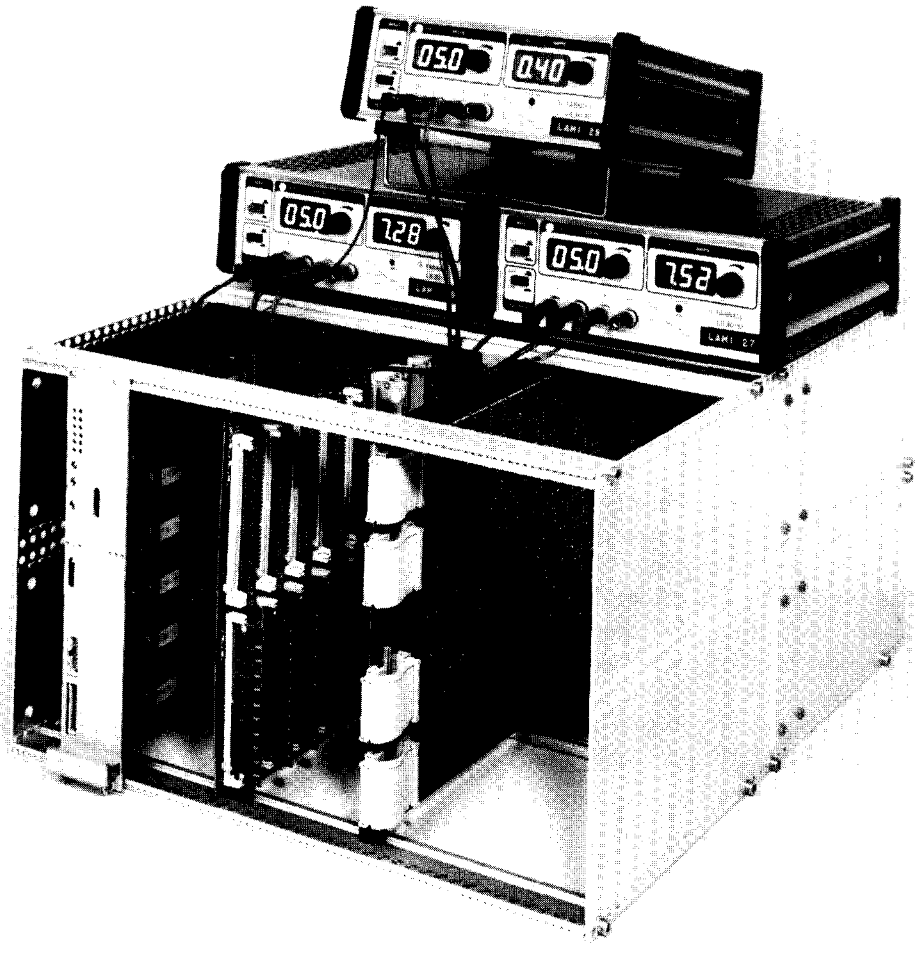

FIGURE 1 The MANTRA I system.

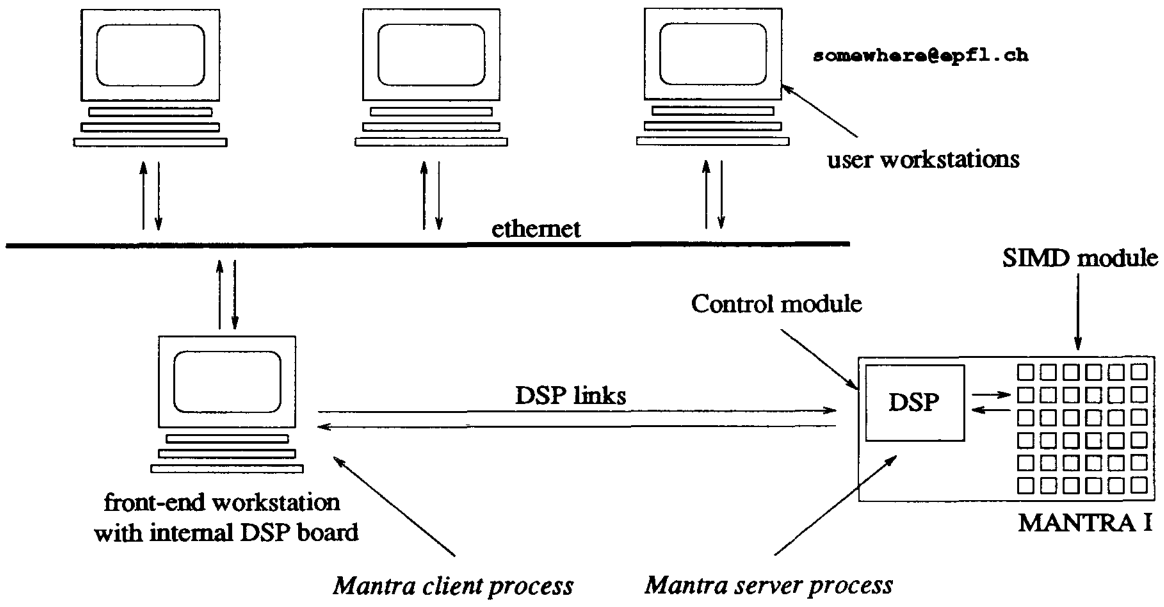

FIGURE 2 The MANTRA I system integration.

2.2 MANTRA I System Architecture

The structure of the MANTRA I system [18] is shown in Figure 3. The *control module* is the SISD system based on the DSP. It controls the *parallel* or *SIMD module* by dispatching horizontally coded instructions through an FIFO. The SIMD module is frozen when no instruction is pending.

Three FIFOs are used to feed data to the SIMD module and two to retrieve results. Temporary results can be held in four static RAM banks connected to the systolic array. The large DSP dynamic RAM can be used when the capacity of the static RAM is insufficient to contain the application. Two units based on look-up tables, noted $\sigma(\nu)$ and $\sigma'(\nu)$, are inserted on the data path and are typically used to compute the nonlinear function of neuron outputs. The latter unit is coupled with a linear array of auxiliary arithmetic units called GACD1 required in some phases of supervised algorithms.

2.3 GENES IV PE

The systolic array at the heart of the SIMD part of the machine is a square mesh of GENES IV PEs

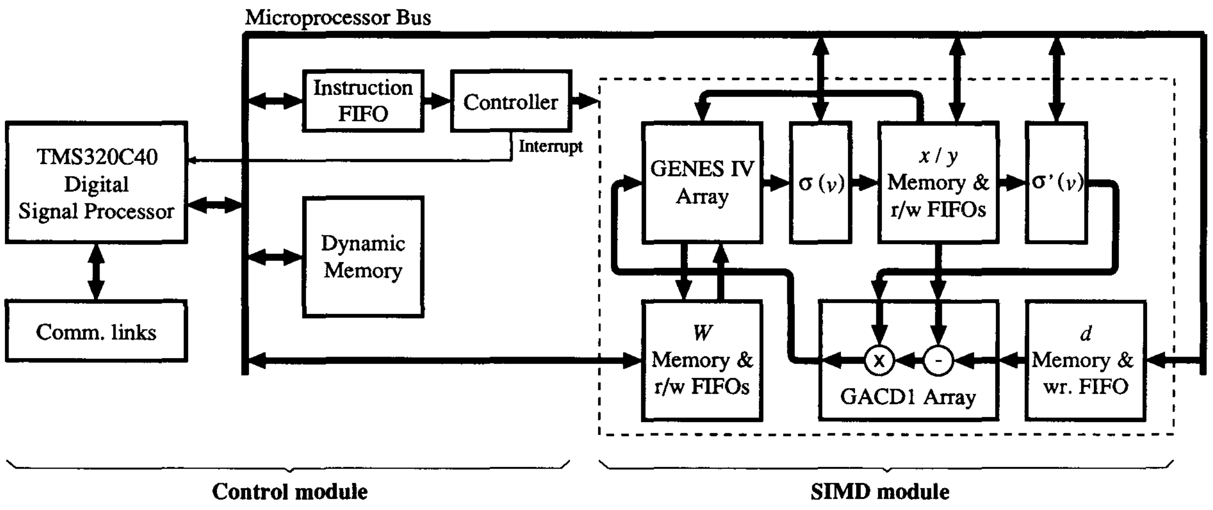

FIGURE 3 Architecture of the MANTRA I machine.

[9], each connected by serial lines to its four neighbours, as shown in Figure 4. All input and output operations are performed by the PEs located on the north-west to south-east diagonal.

Each PE, whose structure is shown in Figure 4, contains one element of a matrix $\mathbf{W}$ (weight unit). The `WGTin-WGTout` path (shown in Fig. 5 but omitted in Fig. 4) is used to load and retrieve matrices. Two vectors $\vec{\mathbf{I}}^h$ and $\vec{\mathbf{I}}^v$ are presented as input at each cycle. Table 1 shows the operations that can be performed $\mathbf{W}_i$ represents the i-th row of the stored matrix, usually containing the weights of a neuron. These operations have been chosen to implement most popular neural-network algorithms including those mentioned at the beginning of Section 2.

All of the operations may also be performed on the transposed matrix $\mathbf{W}^T$ (with $\vec{\mathbf{I}}^h$ and $\vec{\mathbf{I}}^v$ as well as $\vec{\mathbf{O}}^h$ and $\vec{\mathbf{O}}^v$ exchanged). This is shown in Table 1 only for the operation `mprod`^{T}.

For problems involving matrices and vectors larger than the physical array size, the task can be divided in small submatrices and subvectors treated sequentially. The partial sums of several consecutive `mprod`, `mprod`^{T}, and `euclidean` operations can be accumulated thanks to the additive term $\vec{\mathbf{I}}^h$ or $\vec{\mathbf{I}}^v$. The weight unit consists of two registers: one is used for the current computation, whereas the other is connected to the `WGTin-WGTout` path. This makes it possible to load a matrix in the background, without any overhead.

Because an instruction is associated with each pair of input vectors, a new operation can be started on each cycle and processed in a pipelined fashion. The result is available $2N$ cycles later.

The computation is performed on signed fixed-point values. The inputs and the weights are coded on 16 bits. The weights have 16 additional bits, but these are used only during learning

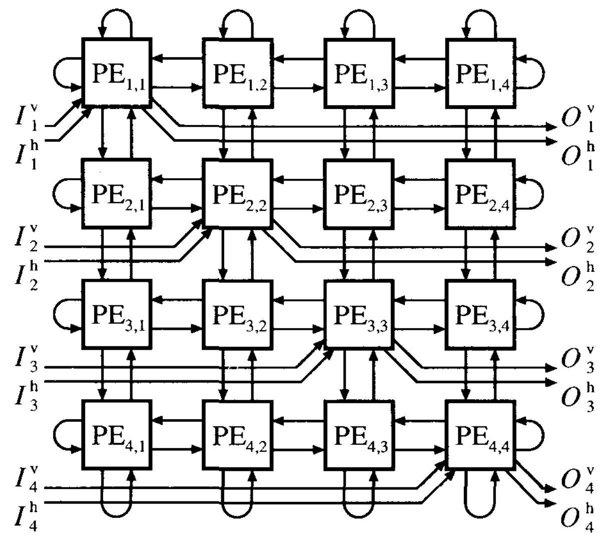

FIGURE 4 Architecture of the GENES IV systolic array. Sample 4×4 array.

(weight update operations). Outputs are computed on 40 bits.

A VLSI chip with a subarray of 2×2 PEs has been designed in CMOS 1 μm standard-cell technology. It contains 71,690 transistors (3,179 standard cells) on a die measuring 6.3×6.1 mm^2.

3 MANTRA I SOFTWARE

Several problems arise when putting to work a specialized computer like MANTRA I. Some of them hardly come to light at early stages of prototype testing and only manifest themselves when running a real application. Users are not supposed to program MANTRA I directly but have a

Table 1. GENES IV Array Basic Operating Modes

Operation	$\vec{\mathbf{O}}^h$	$\vec{\mathbf{O}}^v$	$\mathbf{W}$
`mprod`	$\mathbf{W} \cdot \vec{\mathbf{I}}^v + \vec{\mathbf{I}}^h$	$\vec{\mathbf{I}}^v$	$\mathbf{W}$
`mprod`^{T}	$\vec{\mathbf{I}}^h$	$\mathbf{W}^T \cdot \vec{\mathbf{I}}^h + \vec{\mathbf{I}}^v$	$\mathbf{W}$
`euclidean`	$\lvert\vec{\mathbf{I}}^v - \mathbf{W}_i^T\rvert^2 + I_i^h$	$\vec{\mathbf{I}}^v$	$\mathbf{W}$
`min`	I_i^h if $I_i^h \le \min_j (I_j^v)$; $+\infty$ otherwise	$\vec{\mathbf{I}}^v$	$\mathbf{W}$
`max`	I_i^h if $I_i^h \ge \max_j (I_j^v)$; $-\infty$ otherwise	$\vec{\mathbf{I}}^v$	$\mathbf{W}$
`hebbian`	$\vec{\mathbf{I}}^h$	$\vec{\mathbf{I}}^v$	$\mathbf{W} + \vec{\mathbf{I}}^h \cdot \vec{\mathbf{I}}^{vT}$
`kohonen`	$\vec{\mathbf{I}}^h$	$\vec{\mathbf{I}}^v$	$\mathbf{W}_i + I_i^h \cdot (\vec{\mathbf{I}}^{vT} - \mathbf{W}_i)$

set of libraries available on the front-end workstation.

The first neural-network algorithm implemented on MANTRA I is the Kohonen's SOFM, because this model is required by the target application described in Section 5.

Section 3.1 is devoted to the description of the Kohonen algorithm. Section 3.2 describes the mapping of the Kohonen routine on the systolic array to yield a basic Kohonen program in fixed-point arithmetics. Section 3.3 outlines the problems in the actual production of microcode for the array and describes the approach taken to handle the task. Finally, Section 3.4 contains the details on the software interface between the lower programming level and the user level, and explains how this interface hides from the user some constraints specific to the systolic hardware.

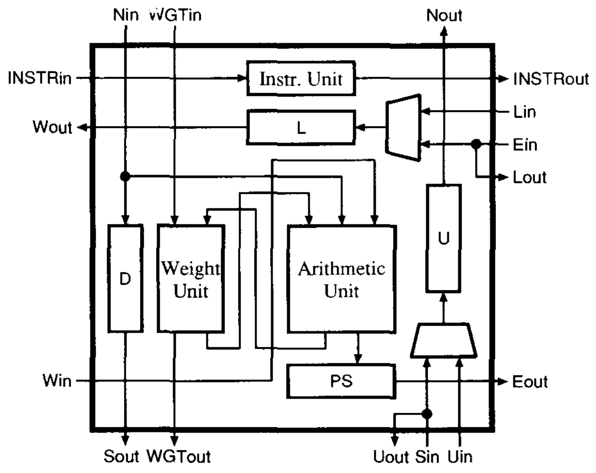

FIGURE 5 Basic structure of a GENES IV PE.

3.1 Kohonen SOFMs

Kohonen's SOFMs are among the most widely used unsupervised artificial neural-network models. Their learning algorithm performs a nonlinear mapping from a high-dimensional input space onto a set of neurons [10]. These neurons are organized as regular maps and a topological relation between them is defined. Two-dimensional grids or meshes, as shown in Figure 6, are typical topologies, but hexagonal grids or more exotic topologies are possible as well.

All neurons share the same inputs. Training is an iterative process: For every input vector $\vec{\mathbf{x}}$, its similarity with the weights of each neuron i is measured in the n-dimensional input space. The most frequently used similarity measure is the Euclidean distance:

$$y_i = \Delta_{\text{in}}(\vec{\mathbf{x}}, \mathbf{W}_i) = \sqrt{\sum_{j=1}^{n} (x_j - W_{i,j})^2}, \quad \text{for } i = 1, 2, \ldots, m; \qquad (1)$$

Other common similarity measures include the Manhattan distance and the scalar product. Usually, the input space has a much higher dimensionality than the topological space of the map. The *winner* neuron $I \in \{1, 2, \ldots, m\}$ is defined as the neuron whose weight vector is the closest to the input vector:

$$\Delta_{\text{in}}(\vec{\mathbf{x}}, \mathbf{W}_I) \le \Delta_{\text{in}}(\vec{\mathbf{x}}, \mathbf{W}_i), \quad \forall i \in \{1, 2, \ldots, m\}. \qquad (2)$$

During the learning phase, the weights are updated as:

$$\mathbf{W}_i := \mathbf{W}_i + \alpha \cdot \lambda(\Delta_{\text{map}}(i, I)) \cdot (\vec{\mathbf{x}}^{\mathrm{T}} - \mathbf{W}_i), \quad \text{for } i = 1, 2, \ldots, m; \qquad (3)$$

where $\Delta_{\text{map}}(i, I)$ is the distance between neuron i and the winner I on the topological map. The *neighborhood function* λ restricts the update to neurons close to the winner. The basic idea of the update rule is to bring the winner and its neighbors closer to the input vector. The *adaptation gain* α should be decreasing during the training process to ensure its convergence.

Apart from the differences arising from the choice of the similarity measure Δ_{in} and the distance on the map Δ_{map}, variations exist in the way the winner is detected and the weights updated [12].

3.2 Mapping Kohonen Networks on the Systolic Array

On MANTRA I, the first step of the computation consists of evaluating the Euclidean distances between the input vector and the synaptic weights of each neuron, with the `euclidean` operation (Table 1). The winner is then implicitly identified by processing the vector containing the m distances (m being the number of neurons) with the `min` operation. The sigma unit is used to convert $+\infty$ to 0 and any other value to 1. The result is a binary vector of m elements, all equal to 0 except for the neuron(s) closest to the input.

The vector containing the m elements $\alpha \cdot \lambda(\Delta_{\mathrm{map}}(i, I))$ is computed by multiplying the above binary vector by an $m \times m$ matrix $\mathbf{\Lambda}$ (mprod operation). The elements of this symmetric matrix are $\Lambda_{i,j} = \alpha \cdot \lambda(\Delta_{\mathrm{map}}(i, j))$ and therefore contain all the information about the topology of the map. The advantage of this formulation is that there are no constraints on the dimensionality of the map, on the arrangement of the neurons (orthogonal, hexagonal, or other grids), nor on the shape of the neighborhood (e.g., rectangular or triangular).

Finally the weights can be modified by injecting this update vector as $\vec{\mathbf{I}}^{\mathrm{h}}$ and the original input vector as $\vec{\mathbf{I}}^{\mathrm{v}}$, and by performing the kohonen operation. Weight matrices larger than the systolic array can be decomposed into submatrices, which are then time-multiplexed on the array.

As it is the case with most dedicated hardware systems for neural networks, the described mapping produces an algorithm that is, in several respects, slightly different from the basic algorithm. The main differences are:

1. *Fixed-point arithmetic.* GENES IV PEs are designed for fixed-point number representation. In the absence of general analytical results on the required precision (see also Section 4.3), simulations of the application described in Section 5 have been used to determine the precision to be implemented.
2. *Batch update.* A characteristic of the systolic architecture is that the time required to produce a result (latency) is much longer than the delay between two successive inputs (inverse of the throughput). Therefore, it is natural to process batches of input vectors to hide the latency. For this purpose, the same weights are used to compute the distances for all these vectors. Hence, the last ones of the batch do not take advantage of the weight modifications that would have resulted from the first ones.
3. *Learning parameter discretization.* In the described implementation, all the information on the topology of the map is contained in a relatively large matrix. This matrix depends on the learning and neighborhood coefficients α and λ, both evolving with time during the learning process. Whereas for some shapes of the λ function an update of the topology matrix inside the array is possible, the current implementation recomputes a new matrix in the SISD module. This implies that, for efficiency purposes, the learning factors α and λ, instead of continuously evolving during the learning process, should be changed as seldom as possible. However, this does not appear as a major obstacle to the algorithm convergence.
4. *Multiple winners.* In traditional implementations, when multiple neurons have the same minimal distance from the input, one is arbitrarily selected (e.g., the one with the smallest index); on the contrary, in MANTRA I all these neurons get updated. This multiple update is similar to the sequential presentation of some input vectors, each slightly closer to one of the winners. Hence, it represents a small distortion of the probability distribution and should not hinder the convergence of the network. A more severe consequence is that, if two or more neurons have the same weights and the neighborhood is null, the neurons can no longer be separated, therefore reducing the mapping capabilities of the network. The weight resolution of MANTRA I is rather high and this problem should seldom occur. Additional techniques could also be applied to minimize the problem in critical cases.

3.3 Scheduling the Systolic Array

Hand-coding programs for MANTRA I is an extremely complex task. As shown in Figure 3, a single instruction controls all pipeline stages of the SIMD module in parallel. This implies that an instruction may depend on several activities started at different times in the past and sometimes conceptually independent. The complexity and reconfigurability of the pipeline make it difficult to mimic its delays in the instruction decoder. However, it would be desirable to code the whole processing of a data set in the same instruction.

The basic idea that has been implemented is to describe complete operations on groups of data independently from other concurrent activities [5]. Operations include, for instance, the complete matrix downloading process or the computation of a set of distances.

Each of these algorithm building blocks, called *microtasks* (MT), is similar to a microoperation in a microcoded processor, and alleviates the programmer from dealing with low-level machine details. In contrast to traditional microoperations, MTs extend over many cycles, often in the order of twice the number of PEs per side. If no special action is taken, executing MTs in sequence causes the parallelism of the machine to be lost. MTs should start as early as the required resources are available and, in general, before all phases of the previous MT have been completed. This process is very similar to issuing instructions in a pipelined processor: It requires verifying the availability of resources and data and generating stall cycles if these conditions are not met.

In contrast to what happens in pipelined processors, here the sections of the pipeline are heavily interdependent and data are synchronously transmitted between stages. Therefore, if one stage is stalled, the others will be halted as well. Hence, hazards must be forecasted and an MT can only be started if it can be completely performed. The blocks of microcode are therefore treated as rigid entities and the program is compacted by overlapping each MT with the neighboring ones, provided that no conflict arises. For each cycle, the horizontally coded MANTRA I instruction is then determined by the MTs taking place in the different parts of the machine.

As a result of this kind of program optimization, the programmer can describe all activities as if completely serialized, including typically concurrent tasks such as background weight matrix exchanges. Compared to a hypothetical hand-coded implementation, the loss in performance observed in the implementation of the Kohonen algorithm appears negligible [5]. The compaction algorithm is also very quick, making it possible to prepare the horizontal code just before run-time when the problem size is completely defined.

3.4 High-level Implementation

Requirements

The low-level Kohonen procedure does not hide all the machine-dependent aspects that should be made transparent to the user. A further software layer is required with two primary purposes: (1) automatic conversion of user's floating-point data (weights, input vectors) to and from fixed-point representation and (2) discrete variation of all parameters that should evolve in time during the learning process, namely the learning coefficient α and the radius of the neighborhood function λ.

Constraints Imposed by the Hardware

The hardware architecture imposes a few constraints in order to get the best performance from the MANTRA I machine. First, the optimal epoch should be at least $2N$, where N is the number of PEs per side in the bidimensional systolic array. This is because $2N$ is also approximately the depth of the pipeline realized by the systolic machine. In the current configuration, $N = 20$ so that the optimal epoch is $T = 40$ (larger values do not improve the parallelization but make the algorithm even more different from the sequential version). The number of input vectors passed to one call of the low-level procedure should be a multiple of the chosen epoch, also for performance considerations.

Another set of constraints concerns the number of input vectors and the epoch. Each time that one of these is changed, the microcode for the systolic unit is prepared and recompacted internally in the low-level Kohonen procedure (Section 3.3). This process causes an overhead in the computation that should be avoided as far as possible. Therefore, the value of the epoch and of the number of input vectors should be kept constant in the multiple calls to the low-level procedure that will occur during a learning process, if at all possible.

Implementation

The MANTRA I machine communicates with user processes, running on the Unix front-end computer, in a client/server fashion. No more than one user process at a time may be connected to the machine. A user process issues remote procedure calls executed by the server program running on the MANTRA I control processor. The potential parallelism between the front-end workstation and the MANTRA I SISD component has not been exploited so far: For the sake of software simplicity, processes on the Unix system wait idle until the end of the remote procedure call.

The software upper layer is implemented in the server program on the MANTRA I DSP. The procedure provided to the end user implements a complete Kohonen algorithm. It is included in a library that users can link to their own programs running on the workstation. Thus, the MANTRA I machine operates as an accelerator for the workstation.

The user has to provide the high-level procedure with the neural-network size, floating-point input data, a function $\alpha_u(t)$ describing the evolution of the adaptation gain during learning, and a two-dimensional function $\lambda_u(d, t)$ describing the evolution of the neighborhood function $\lambda(d)$. Finally, the desired number of learning iterations should be provided. The procedure returns the weight matrix after training. If desired, the weights may also be extracted at intermediate points during learning, in order to monitor the training process.

The software upper layer first searches the training set for the maximum and minimum values for each parameter of the input vectors. Each input component x_i in the interval $[x_{\min}, x_{\max}]$ is then mapped to the interval $[-2^{14}, 2^{14} - 1]$; the weights are initialized with small random values. The Kohonen update rule then ensures that no overflow in the fixed-point computation will ever occur during learning.

The number of iterations required by the user is divided into a number of intervals of equal length l. This length is, if possible, a multiple of the required epoch length. For each of these intervals, the low-level Kohonen procedure will be called with l input vectors, randomly chosen in the training set, and with constant values for α and for the function $\lambda(d)$, computed by discretizing the original $\alpha_u(t)$ and $\lambda_u(d, t)$ functions provided by the user.

4 PERFORMANCE ASSESSMENT

The performance of the machine is conditioned by the efficiency of the different levels that build up the user library. The first component is the efficiency of the low-level routines used to access the systolic hardware. To that, one should add the effects of the algorithmic modifications outlined in Sections 3.2 and 3.4 ("Constraints Imposed by the Hardware"). This section presents performance measurements of the low-level routines and discusses the impact of the key modifications on the connectionist performance. For the later purpose, a general framework to judge the performance of hardware dedicated to neural networks is introduced.

4.1 Performance of the Low-level Subroutine

The peak performance in *connection updates per second* (CUPS) for single-layer networks can be roughly computed as:

$$P = \frac{N^2 \cdot f}{n_{op} \cdot N_{PS}} \cdot U \tag{4}$$

where N^2 is the total number of PEs, f is the clock frequency, $N_{PS} = 40$ is the number of clock cycles per operation (bit-serial communication), and U is the utilization rate. The constant n_{op} evaluates the number of operations required to update a connection; for instance, in a Kohonen network with the same number of inputs and neurons, $n_{op} = 4$ (`euclidean`, `min`, `mprod`, and `kohonen`). Considering the largest possible configuration of the MANTRA I system (40×40 PEs) running at a clock frequency of $f = 8$ MHz, the peak performance ($U = 100\%$) of the system is 80 MCUPS for the Kohonen network. Equation 4 gives the performance for single layers; otherwise the global performance is given by a weighted harmonic mean of the individual layer performances. For instance, a back-propagation network with one hidden layer and the same number of inputs and neurons on both layers would have a peak performance of 128 MCUPS.

In practice, the performance degrades from the ideal value because of several components. Namely, the utilization rate U is the product of three independent elements:

1. A *spatial utilization rate*, expressing the fact that, depending on the size of the map, some PEs may be left idle during some phases of the computation.
2. A *temporal utilization rate*, indicating the ratio of active instructions (array instructions other than no operation) over the complete microprogram.
3. A *dynamic utilization rate*, which is the ratio of clock cycles when the SIMD module is active over the total. It models situations when the control DSP is delaying the parallel module because of unavailability of data or instructions.

The performance P has been measured on the current configuration of the MANTRA I prototype (8 MHz, 20×20 PEs). The results are shown in Figure 7. In the most favorable conditions, a fraction slightly below 70% of the ideal performance has been measured.

4.2 Neural-Network Performance on Dedicated Hardware

In addition to the MANTRA I machine, other programmable machines, based on custom digital chips and aimed at running neural algorithms, have been described in the literature (see [6] for a review). Among the most interesting, in terms of advertised performance and versatility, one can mention the *CNAPS* machine from Adaptive Solu-

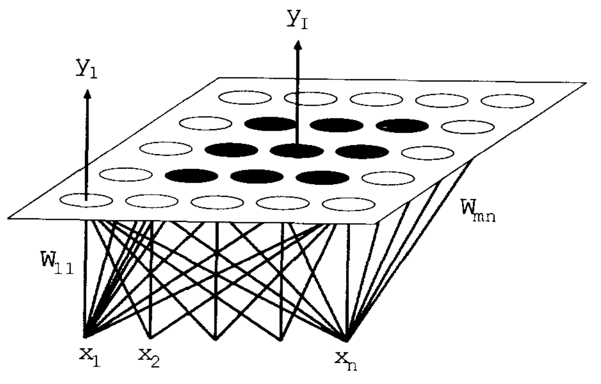

FIGURE 6 Kohonen feature map arranged on a two-dimensional grid.

tion [3], the *MY-NEUPOWER* machine from Hitachi [15] and the *SYNAPSE* machine from Siemens [14]. All these machines are essentially based on SIMD parallel architectures and somehow modify the alrogithms to make them more suitable to the hardware [1].

The kind of modifications listed in Section 3.2 is far from being peculiar to MANTRA I: For instance, almost all of the machines dedicated to neural networks implement fixed-point arithmetic because it is one of the fundamental sources of simplification of the hardware. Similarly, batch processing is a typical requirement of pipelined machines or of systems that associate a heavy cost to partitioning large networks on smaller hardware. In the latter case, it is sensible to average this cost over a batch of input vectors.

At the same time, these problems are not specific to Kohonen maps but, in different ways, affect most algorithms. Concerning batch update, although some connectionist models have an intrinsically batch nature (such as conjugate-gradient optimization techniques), others are originally designed to perform a weight update after each vector is presented (e.g., stochastic back-propagation or Kohonen self-organizing maps) and may suffer from an injudicious conversion to batch update. The problem of incorporating these effects in a fair assessment of the achieved speed-up is addressed in the next section.

Speed-Up Revisited

Performance of neurocomputers is traditionally measured in CUPS, from which the time necessary for one training iteration can be derived. A fair performance metric should rather depend on the time taken during training to reach a predetermined output error, and not on the time taken to execute a predetermined number of learning iterations. Let E be the training error of the neural-network model M implemented. For supervised neural networks, E can be the output error while for unsupervised models such as the Kohonen model, one may choose the quantization error. (The quantization error is the sum of the Euclidean distances between each vector and the weight vector of the corresponding winner. It essentially measures the quadratic error incurred by representing the input data by the closest neurons in the map.) The speed-up achieved by a neurocomputer compared to a reference machine, should be defined by

$$S_{\mathrm{M}}(E_0) = \frac{t_{\mathrm{cc}}(E_0)}{t_{\mathrm{hw}}(E_0)}, \tag{5}$$

where E_0 is some predetermined value of the convergence metric, and $t_{\mathrm{hw}}(E_0)$ and $t_{\mathrm{cc}}(E_0)$ are, respectively, the times necessary for the neurocomputer and for a reference computer (for instance, a conventional workstation using double-precision floating-point variables) to reach the desired convergence value E_0 (Fig. 8).

To link this new definition of speed-up to the traditional CUPS ratings, a measure of the quality of convergence of the algorithms may be introduced: The *algorithmic efficiency* of a neurocomputer implementation of model M can be defined as follows:

$$A_{\mathrm{M}}(E_0) = \frac{k_{\mathrm{cc}}(E_0)}{k_{\mathrm{hw}}(E_0)}, \tag{6}$$

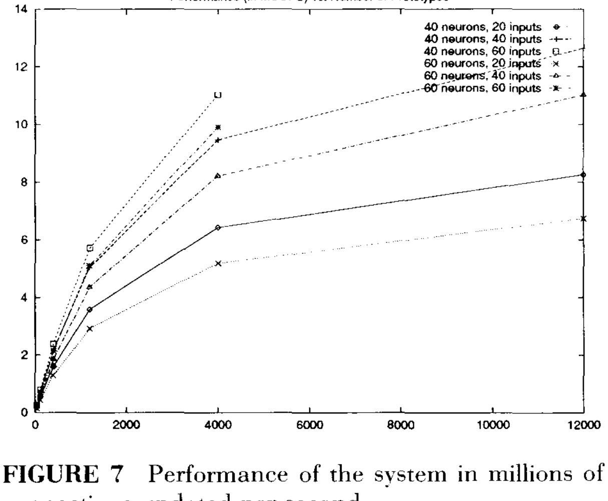

FIGURE 7 Performance of the system in millions of connections updated per second.

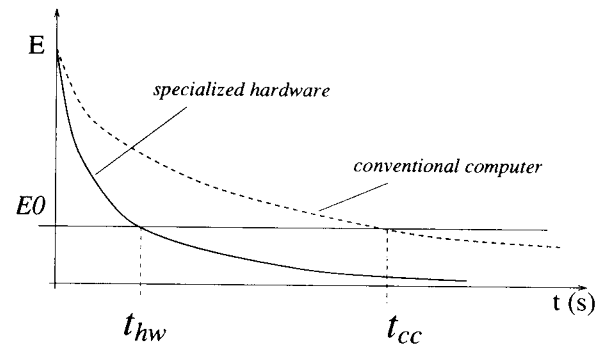

FIGURE 8 Convergence speed of a neurocomputer compared to a conventional computer as a function of time.

where $k_{hw}(E_0)$ and $k_{cc}(E_0)$ are, respectively, the number of iterations necessary for the neurocomputer and for a reference computer to reach the same E_0 (Fig. 9). The efficiency, always positive, will be typically below unity, showing the losses introduced by the hardware constraints.

The definition (Equation 5) of speed-up and (Equation 6) of algorithmic efficiency then yields

$$S_M(E_0) = \frac{\tau_{cc}}{\tau_{hw}} \cdot A_M(E_0), \qquad (7)$$

where τ_{hw} and τ_{cc} are the times necessary to process one learning iteration on the neurocomputer and on the reference computer, respectively. The ratio τ_{cc}/τ_{hw} expresses the traditional notion of hardware performance measured as the ratio of the CUPS on the special-purpose hardware and on the reference system.

Equation 7 weighs the hardware speed-up with the algorithmic efficiency of the implementation. A good implementation should have an efficiency as close as possible to unity. The more the algorithm has to be tuned to fit the hardware constraints, the smaller the resulting efficiency.

This suggests that there may be a trade off between improving the parallelization efficiency in a neurocomputer implementation and preserving an acceptable algorithmic efficiency. A compromise might lead to the optimum global speed-up.

4.3 Effects of the Hardware Constraints

The implementation constraints on MANTRA I induce important modifications of the Kohonen algorithm as described in Section 3, "Constraints Imposed by the Hardware." These may lead to a poor algorithmic efficiency or even prevent the convergence. The two most important ones are the quantization on a finite number of bits of the input signals and synaptic weights and the batch updating of the weights.

Quantization Effects

Contrary to other neural networks, the Kohonen algorithm with quantized weights and inputs has received little attention so far. Three factors influencing its correct convergence can be put in evidence [17]. Clearly, there is a minimal *number of bits* required to encode the weights, depending on the input distribution and dimension, as well as the number of neurons. Second, the adaptation gain α must decrease slowly enough, or have an initially large value, because otherwise the weight updates get rounded to zero before the algorithm has converged. Finally, the neighborhood function should decrease with the distance from the winner neuron, especially if the input dimension is low. These qualitative results were confirmed by a mathematical analysis based on the Markovian formulation of the algorithm [16], giving the necessary and sufficient conditions for the self-organization of the map in the case where the input and weight spaces are one dimensional. Roughly speaking, the results proven for the continuous case [2] also apply in the quantized case if the number of bits is large enough.

Batch Updating

As explained in Section 3.2, the Kohonen algorithm is implemented in a modified batch version on MANTRA I. The batch mode is fundamental to exploit the parallelism of the systolic array [8]. Up to now, the differences between the *batch* and the classical *online* versions have not been studied in the literature for this particular model. The MANTRA I version is not purely batch, because the winner neuron is computed with the value of the weights at the beginning of the epoch but the weight update is an online operation. This implementation proved to be the most economic in terms of hardware complexity. In fact, although more counterintuitive, the convergence of the implemented algorithm has been proven in the case of scalar inputs and time invariant parameters, whereas the pure batch algorithm convergence can only be proven with more restrictive assumptions on the parameters [7]. In the one-dimensional case, it has been proved, using Markov chains' properties, that when the neighborhood function is rectangularly shaped and time invariant, and the adaptation gain is also time invariant, the weights self-organize with probability 1. When the adaptation gain decreases to zero, it has then also to be proved, using the *ordinary differential equation* (ODE) method [11], that the weights converge with probability 1 to the same asymptotic values as the ones reached by the original, unmodified algorithm.

Comparative simulations for a wide range of learning parameters have been performed to support the theoretical results. On a real-size benchmark (speech codebook classification, 10 × 10 neurons, 12 inputs), the quadratic quantization error has been measured as a performance indication of the self-organizing map (E). Figure 10 shows the evolution of the convergence metric for the three versions of the algorithm. Already after the 8th epoch, the original algorithm is within 20% of the best result, assessed after 200 epochs. The fully batch version is distinctly slower during the whole convergence and needs 28 epochs to fall below the same threshold. Thus, the algorithmic efficiency of the batch version compared to the reference algorithm is only $A_M = 8/28 = 0.29$. For lower values of E_0, the efficiency will become even lower and eventually reach 0, because the asymptotic minimum error reached is larger in the neurocomputer than in the reference algorithm. However, the best final error achieved by the MANTRA I algorithm with batches of 50 vectors is within a few *ppm* from the result of the standard version. Moreover, the algorithmic efficiency at the same E_0 is 0.57 and for lower values of E_0 it becomes close to 1. These results indicate that the batch nature of the MANTRA I algorithm does not severely hinder the convergence speed of the model.

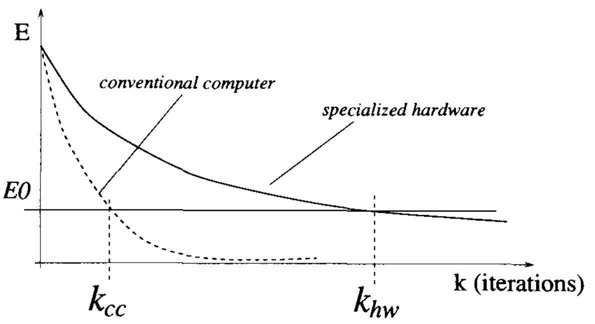

FIGURE 9 Convergence speed of a neurocomputer compared to a conventional computer as a function of algorithm iterations.

5 POWER-SYSTEM APPLICATION

Putting the MANTRA I machine to work on a real application was one of the main objectives of the project. A target application in the field of power-system security assessment has been chosen for its heavy computational requirements. Section 5.1 briefly describes the application. Section 5.2 gives an estimate of the computational power required. Finally, Section 5.3 gives an overview of the convergence quality and of the performance currently reached on the prototype for this application.

5.1 Application to Power-System Security Assessment

The application of Kohonen SOFMs to power-system security was first developed and implemented on a conventional workstation [13]. The purpose of this application is to predict whether the power flows in the branches (lines and transformers) and the bus voltages of a system will, after an unforeseen outage, exceed the supported limit of the corresponding components or not.

Power flows in a power system may be computed by solving a set of nonlinear equations using an iterative method, for instance Newton–Raphson's. Given a current operating point, classical static security analysis considers every possible combination of outages and iteratively computes a power flow for each. This heavy computational burden prevents real-time computation on sequential hardware. Moreover, conventional simulation provides only quantitative results, leaving the interpretation of the current system state and its potential stochastic evolution to the power-system operators. Because decisions in power transmission system control centers often have to be taken under time pressure and stress, a fast, concise, and synthetic representation of security assessment results is essential.

In this approach, the operational points of the multidimensional power-system state space are mapped onto a two-dimensional Kohonen network by dividing the security space into categories. The centers of each class are the neurons located at the coordinates defined by the weight vectors. The two-dimensional picture of the network gives a quite accurate interpretation of the situation.

5.2 Computational Requirements

In the proposed application, a new security analysis and a new learning process are to be performed every day to account for the daily changing operating conditions. A utility, even if it controls only a small part of the system, should take a major part of the network into account to yield accurate results. The Swiss high-voltage transmission network, for instance, consists of roughly 150 busses and 250 lines. The learning phase requires the processing of input vectors composed of 300 elements. The number of neurons in the feature map may not realistically be much more than 1,000, because for each of them an expensive power-flow computation has to be performed *after* the neural-network training to interpret the results. Considering the 10^5 iterations is an ordinary training length for a large Kohonen network, the number of connection updates necessary for a real-world power system may be roughly evaluated to a minimum of $300 \times 1000 \times 10^5 = 3 \cdot 10^{10}$ connection updates for a daily training. Such an amount of computation takes more than 8 h on a conventional workstation (approximately 1 million of connection updates performed per second), which is too slow to be used in daily operation. An increase in computational power of one to two orders of magnitude is required. Accelerators such as the one proposed in this article could drastically reduce the learning time for larger power systems down to acceptable levels.

5.3 MANTRA I Performance on the Power-System Application

As described in Section 4, the performance of the system has been addressed by attempting to separate the hardware speed-up from the algorithmic effects. The test experiments are described in the next sections.

Quality of Convergence

Figure 11 shows the evolution of the quantization error for one run of the power system application on the MANTRA I machine compared with a run of the original algorithm. The training set has been generated using the IEEE 24-bus 38-line power system, one of the smaller standard test systems developed for benchmark studies of power-system software. With a dimension of 76 for the input vectors, this power system was well suited for the test of the MANTRA I and could fit into the DSP dynamic RAM of the prototype. The same set of

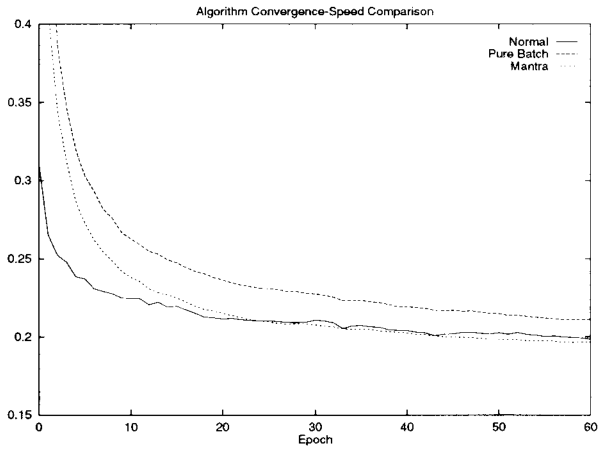

FIGURE 10 Comparison, on floating-point simulations, of the convergence speed of the Kohonen MANTRA I batch implementation with that of the pure batch algorithm and with the original online algorithm. The graph measures the quantization error of the network after every learning step composed of $T = 50$ vectors.

learning parameters has been used for the original algorithm and for the MANTRA I. Figure 11 allows a rough evaluation of the algorithmic efficiency of MANTRA I in this application. Each unit on the X-axis corresponds to 120 training iterations.

It should be noticed that because of a combined effect of integer arithmetic, batch implementation, and multiple winners, MANTRA I converges with a slightly higher final error than the sequentially implemented floating-point version of the Kohonen algorithm. On a target error rate of 50% more than the minimum error of the original algorithm, the latter and the MANTRA I implementation need, respectively, 12×120 and 16×120 iterations to reach the desired error rates. This yields an algorithmic efficiency of $12/16 = 75\%$.

To confirm that the algorithmic efficiency on MANTRA I is high and not very far from unity, it was tested with additional data from other applications. Test runs confirmed that the discrepancies between the MANTRA I version and the original version of the Kohonen algorithm are smaller than the standard deviation of the original algorithm itself. For instance, averaging ten runs for each of several sets of learning parameters in the speech codebook classification problem, MANTRA I actually performed better in approximately 50% of the cases.

Like most neural-network algorithms, the performance of the Kohonen algorithm, including the

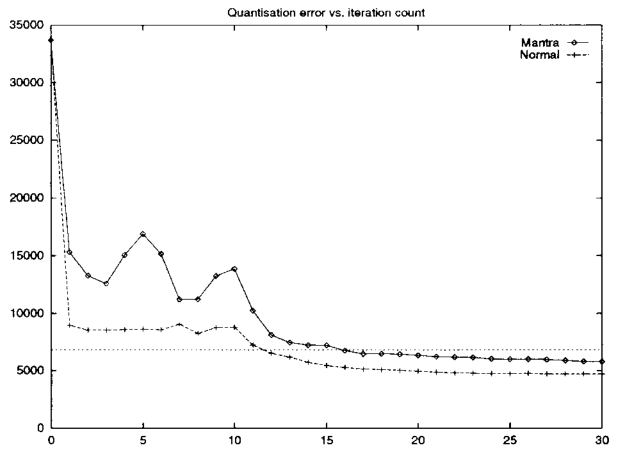

FIGURE 11 Measurement on the power-system application of the convergence speed of the Kohonen MANTRA I implementation compared to a floating-point implementation.

version running on MANTRA I, is quite sensitive to an inappropriate choice of the training parameters. Because the Kohonen algorithm converges stochastically toward an equilibrium point, different random initializations of the weight vectors coupled with a nonideal choice of training parameters can result in different error rates. Experiments with comparatively small sets of training data (for instance, 1,562 vectors in the case of the power system data, each presented several times chosen at random) indicated that for several runs differences in the final quantization error were higher than expected and the reasons for this behavior have to be investigated further. Whether the ideal learning parameters are identical for the original and the MANTRA I version of the Kohonen algorithm is also an open question at present time. Even though no evidence of a strong change in robustness has been found, more extensive experiments should be conducted to produce statistically significant observations.

In conclusion, the final rate reached by MANTRA I appears acceptable for the power-system application and the number of iterations required is approximately equivalent to that needed by a traditional floating-point version of the Kohonen algorithm.

Hardware Speed-Up

Table 2 shows the actual MCUPS performance reached by the current machine with the Kohonen algorithm on the power-system application, for different problem sizes. For efficiency reasons, the number of neurons is chosen as a multiple of the number of PEs per side of the array (20 in the current machine), so that as few processors as possible remain idle. The actual performance depends heavily on the problem size; the machine performs badly on small problems, where conventional workstations could be sufficient. However, for larger problems, up to 14 MCUPS have been reached for the Kohonen algorithm, whereas a conventional platform performs around 1 MCUPS on the same algorithm [1].

6 CONCLUSION

The MANTRA I machine has been designed, realized, and tested. This article concentrated on the phases beyond hardware development, toward practical applications and real use. Topics such as programming and practical performance seem unfortunately seldom described in literature [6] and appear to be often disregarded as secondary to the hardware design itself. The MANTRA I experience shows that a number of key problems arise only when one tries to put the hardware to work and abandon toy problems to tackle real ones.

First, a crucial issue is the complexity of low-level programming of this type of dedicated machines, which may lead either to an overwhelming programming complexity or to poor hardware utilization and performance. The difficulties arise because of the many independent execution units (e.g., systolic arrays, look-up tables) exposed to the programmer view and because of long pipelines. Techniques to preserve the hardware efficiency and at the same time to structure the code have been presented.

On the other hand, neural networks are not so insensitive to algorithmic modifications and to reduced precision as often supposed, especially by hardware designers. For instance, the MANTRA I system has to use a counterintuitive version of the

Table 2. Implementation of the Kohonen Algorithm: Measured Sustained Performance for Different Problem Sizes

Neurons	Inputs	Iterations	MCUPS
6 × 10	76	1,200	0.8
		12,000	5.4
		60,000	12.2
		100,000	13.9

Kohonen algorithm for which convergence properties similar to those of the original algorithm have been proved. Simulations are also presented to confirm the theoretical results.

Finally, other problems had to be taken into account when interfacing the dedicated hardware itself with a conventional computational server and designing an efficient programming environment.

Despite the many difficulties, the MANTRA I prototype, with one fourth of the supported PEs, displays a performance about one order of magnitude above that of a conventional workstation. Still, to provide users with dedicated machines with performances on neural networks close to those of supercomputers but at desktop prices, larger machines should be built with more sophisticated technologies (e.g., custom layout instead of standard cells, higher integration, and clock rate). Also, more advanced packaging technology would also be required to solve severe reliability problems that have been experienced on the current prototype.

On the grounds of the gained experience, future research should address the following critical directions: (1) Find more flexible and powerful programming models to facilitate the low-level programming of novel neural-network models by trained users and thus offer the flexibility that users require. (2) Find techniques to perform the microcode compaction online in hardware to avoid an overhead that may become impractical for algorithms whose control flow is heavily data dependent. These include emerging connectionist models such as evolutive networks. (3) Improve the generality of the basic PE architecture to support this broadening of the algorithmic target.

Only addressing the above problems as a whole and not restricting oneself to the latter, dedicated systems for neural networks may become attractive for potential users and competitive with large and expensive computational servers.

ACKNOWLEDGMENTS

The MANTRA research project is supported by the Swiss National Fund for Scientific Research through the SPP-IF grant 5003-34353 and by the Swiss Federal Institute of Technology.

Many people contributed to this project and deserve the authors' gratitude: François Blayo and Christian Lehmann for laying down the foundations; Peter Brühlmeier, André Guignard, Christophe Marguerat, and Georges Vaucher for helping in designing and building the hardware: and Giorgio Caset for undertaking the arduous task of programming the machine. This work would not have been possible without the constant support of Professors Jean-Daniel Nicoud, Alain Germond, and Martin Hasler.

REFERENCES

[1] T. Cornu and P. Ienne, "Performance of digital neuro-computers," in *Proc. Fourth International Conference on Microelectronics for Neural Networks and Fuzzy Systems*, 1994, p. 87.

[2] M. Cottrell and J.-C. Fort, "Étude d'un algorithme d'auto-organisation," *Ann. Institut Henri Poincaré*, vol. 23, pp. 1–20, 1987.

[3] D. Hammerstrom, A highly parallel digital architecture for neural network emulation, in J. G. Delgado-Frias and W. R. Moore, Eds. *VLSI for Artificial Intelligence and Neural Networks*. New York: Plenum, 1991, pp. 357–366.

[4] J. Hertz, A. Krogh, and R. G. Palmer, *Introduction to the Theory of Neural Computation*. Santa Fe Institute Studies in Sciences of Complexity. Redwood City, CA: Addison-Wesley, 1991.

[5] P. Ienne, Horizontal microcode compaction for programmable systolic accelerators, in *Proc. International Conference on Application Specific Array Processors*, 1995, p. 85.

[6] P. Ienne and G. Kuhn, "Digital systems for neural networks," in *Digital Signal Processing Technology*, vol. CR57 of *Critical Reviews Series*, P. Papamichalis and R. Kerwin, Eds. Orlando, FL: SPIE Optical Engineering, 1995, pp. 314–345.

[7] P. Ienne, P. Thiran, and N. Vassilas, "Modified self-organising feature map algorithms for efficient digital hardware implementation," *IEEE Trans. Neural Networks*, 1995 (Submitted).

[8] P. Ienne and M. A. Viredaz, "Implementation of Kohonen's self-organizing maps on MANTRA I," in *Proc. Fourth International Conference on Microelectronics for Neural Networks and Fuzzy Systems*, 1994, p. 273.

[9] P. Ienne and M. A. Viredaz, "GENES IV: A bit-serial processing element for a multi-model neural-network accelerator," *J. VLSI Signal Proc.*, vol. 9, pp. 257–273, 1995.

[10] T. Kohonen, *Self-Organization and Associative Memory*, vol. 8 of *Springer Series in Information Sciences*. Berlin: Springer-Verlag, 3rd ed., 1989.

[11] H. J. Kushner and D. S. Clark, *Stochastic Approximation for Constrained and Unconstrained Systems*, vol. 26 of *Applied Mathematical Sciences*. Berlin: Springer-Verlag, 1978.

[12] C. Lehmann, "Réseaux de neurones compétitifs de grandes dimensions pour l'auto-organisation:

analyse, synthèse et implantation sur circuits systoliques," PhD Thesis no. 1129, École Polytechnique Fédérale de Lausanne, Lausanne, 1993.

[13] D. Niebur and A. J. Germond, "Unsupervised neural network classification of power system static security states, *Int'l J. Electrical Power Energy Systems*, vol. 14, pp. 233–242, 1992.

[14] U. Ramacher, "SYNAPSE—A neurocomputer that synthesizes neural algorithms on a parallel systolic engine," *J. Parallel Distrib. Comput.*, vol. 14, pp. 306–318, 1992.

[15] Y. Sato, K. Shibata, M. Asai, M. Ohki, M. Sugie, T. Sakaguchi, M. Hashimoto, and Y. Kuwabara, "Development of a high-performance general purpose neuro-computer composed of 512 digital neurons, in *Proc. of the International Joint Conference on Neural Networks*, vol. II, 1993, p. 1967.

[16] P. Thiran and M. Hasler, "Self-organisation of a one-dimensional Kohonen network with quantized weights and inputs," *Neural Networks*, vol. 7, pp. 1427–1439, 1994.

[17] P. Thiran, V. Peiris, P. Heim, and B. Hochet, "Quantization effects in digitally behaving circuit implementations of Kohonen networks," *IEEE Trans. Neural Networks*, vol. XX-5, pp. 450–458, 1994.

[18] M. Viredaz, "Design and analysis of a systolic array for neural computation," PhD Thesis no. 1264, École Polytechnique Fédérale de Lausanne, Lausanne, 1994.