In Luigi Dadda and Benjamin Wah, editors, *Proceedings of the International Conference on Application-Specific Array Processors*, pages 345–56, Venezia, October 1993.

GENES IV: A Bit-Serial Processing Element for a Multi-Model Neural-Network Accelerator

Paolo Ienne and Marc A. Viredaz
Swiss Federal Institute of Technology
Microcomputing Laboratory & Centre for Neuro-Mimetic Systems
IN-F Ecublens, CH-1015 Lausanne
E-mail: Paolo.Ienne@di.epfl.ch

Abstract

A systolic array of dedicated processing elements (PEs) *is presented as the heart of a multi-model neural-network accelerator. The instruction set of the PEs allows the implementation of several widely-used neural models, including multi-layer Perceptrons with the backpropagation learning rule and Kohonen feature maps. Each PE holds an element of the synaptic weight matrix. An instantaneous swapping mechanism of the weight matrix allows the implementation of neural networks larger than the physical PE array. A systolically-flowing instruction accompanies each input vector propagating in the array. This avoids the need of emptying and refilling the array when the operating mode of the array is changed.*

Both the GENES IV *chip, containing a matrix of* 2×2 *PEs, and an auxiliary arithmetic circuit have been manufactured and successfully tested. The* MANTRA I *machine has been built around these chips. Peak performances of the full system are between 200 and 400 MCPS in the evaluation phase and between 100 and 200 MCUPS during the learning phase (depending on the algorithm being implemented).*

1: Introduction

The field of *neural-networks (NNs)* is now extending above the boundaries of academic research. Applications to real-life problems tend to become competitive with well-established traditional techniques. Recent experiences (as the Santa Fe Institute's competition on *Time Series Prediction and Analysis* [5]) show the maturity of some NN algorithms and the importance of their intrinsic non-linearity contrasted to classical linear approaches. Unfortunately, real applications tend to require large networks and/or large data sets for the learning phase. This makes implementation on conventional hardware platforms extremely inefficient. The four best connectionist entries in [5] took between three hours and three weeks to run, the quickest one being programmed on a CRAY Y-MP. Difficulties in experimenting with large networks also prevent researchers from exploring the field. Often a good result is obtained by repeatedly learning the problem prototypes with different parameters to search for an optimal tuning. This empirical but nevertheless necessary process becomes prohibitive for sufficiently large networks. Hardware accelerators and host-attached computers represent a cost-effective alternative to supercomputers.

In this paper, an NN accelerator, based on a systolic square array of *processing elements (PEs)*, is presented. It is not dedicated to a single algorithm but supports several connectionist models and learning rules. In the next section, a survey of the targeted neural algorithms is proposed. The following section is devoted to the array architecture and deals with the mapping of the NN algorithms on the array. The PE structure and implementation choices are exposed in section 4, followed by details on the system architecture in section 5. Finally, performance evaluation is analyzed in the last section.

2: Neural-networks algorithms

This section briefly introduces the algorithms addressed by the architecture presented in this paper. A more detailed description can be found in a classic introductory book, such as [6]. The typical artificial neuron model represents a device with n inputs and a single output. The output y_i of the i-th neuron of the network is computed as:

$$y_i \quad = \quad \sigma(p_i) \quad = \quad \sigma(\sum_{j=1}^{n} W_{i,j} \cdot x_j) \qquad \text{for } i = 1, 2, \ldots, m; \tag{1}$$

where $W_{i,j}$ represents a coefficient or *synaptic weight* associated with the j-th input x_j and the i-th neuron. The weighted sum p_i is called *potential*. Equation (1) can be rewritten in a matrix form as $\vec{\mathbf{y}} = \sigma(\vec{\mathbf{p}}) = \sigma(\mathbf{W} \cdot \vec{\mathbf{x}})$. Usually, the *activation function* σ represents some saturating non-linear function. Neurons are often organized in layers, all neurons in a layer sharing the same inputs and having their outputs connected to the inputs of the next layer. The weight matrixes are then shown as $\mathbf{W}^{[q]}$, where q is the layer number.

Neural networks usually undergo a learning process. The synaptic-weight matrixes are iteratively updated according to a *learning rule*. One of the simplest one is the *Hebb rule*:

$$\mathbf{W} \quad := \quad \mathbf{W} + \alpha \cdot (\vec{\mathbf{y}} \cdot \vec{\mathbf{x}}^{\mathrm{T}}); \tag{2}$$

where α is a *learning factor*. Though this rule is seldom used as stated, most of the commonly-used learning rules are slight modifications of equation (2).

Multi-layer networks make it possible to implement any arbitrary function $\vec{\mathbf{y}} = \mathbf{\Phi}(\vec{\mathbf{x}})$, $\vec{\mathbf{x}}$ being the input of the first layer and $\vec{\mathbf{y}} = \vec{\mathbf{y}}^{[L]}$ representing the output of the last layer L. Often, the activation function σ is a hyperbolic tangent. The function $\mathbf{\Phi}$ is learned by repeated presentation of input-output pairs $\{\vec{\mathbf{x}}, \vec{\mathbf{d}}\}$, called *prototypes*. The *backpropagation (BP) learning rule* is a gradient-descent algorithm that updates the weights to minimize the square-error on the learning prototypes. For that purpose an error signal is computed for each layer:

$$\delta_i^{[L]} \quad = \quad \left(d_i - y_i^{[L]}\right) \cdot \sigma'(p_i^{[L]}) \tag{3}$$

$$\delta_i^{[q]} \quad = \quad \left(\sum_{k=1}^{m_{q+1}} W_{k,i}^{[q+1]} \cdot \delta_k^{[q+1]}\right) \cdot \sigma'(p_i^{[q]}) \qquad \text{for } q = 1, 2, \ldots, L-1; \tag{4}$$

where $\sigma'(v) = \mathrm{d}\sigma(v)/\mathrm{d}v$. Equations (3) and (4) are valid for all neurons $i = 1, 2, \ldots, m_q$ of layer q. Once the errors have been back-propagated, the weights are updated as:

$$\mathbf{W}^{[q]} \quad := \quad \mathbf{W}^{[q]} + \alpha \cdot \vec{\boldsymbol{\delta}}^{[q]} \cdot \vec{\mathbf{y}}^{[q-1]\mathrm{T}} \qquad \text{for } q = 1, 2, \ldots, L; \tag{5}$$

where $\vec{\mathbf{y}}^{[0]} = \vec{\mathbf{x}}$.

Fully-connected recurrent networks, as the *Hopfield model*, can memorize and detect input patterns. Given an initial input, the output is computed and fed back until a stable pattern is reached. A possible learning rule for this model is the Hebb rule of equation (2) with $\vec{\mathbf{y}} = \vec{\mathbf{x}}$.

A different neural network model is the *Kohonen self-organizing map*. Several variations of the algorithm exist. Neurons are organized as regular meshes where some topological relations are defined (e.g., distance). They all share the same inputs. For every input vector $\vec{\mathbf{x}}$, its distance from each neuron i is measured in the n-dimensional input space. Common distances are the scalar product of equation (1) or the Euclidean distance:

$$y_i \quad = \quad \Delta_{\text{in}}(\vec{\mathbf{x}}, \mathbf{W}_i) \quad = \quad \sqrt{\sum_{j=1}^{n} (x_j - W_{i,j})^2} \qquad \text{for } i = 1, 2, \ldots, m; \tag{6}$$

where $\mathbf{W}_i$ indicates the i-th row of matrix $\mathbf{W}$. Often, the input space has a much higher dimensionality than the topological space of the map. The winner neuron $I \in \{1, 2, \ldots, m\}$, whose weight vector is the closest to the input vector, is identified by:

$$\Delta_{\text{in}}(\vec{\mathbf{x}}, \mathbf{W}_I) \quad \leq \quad \Delta_{\text{in}}(\vec{\mathbf{x}}, \mathbf{W}_i) \qquad \forall i \in \{1, 2, \ldots, m\}. \tag{7}$$

The weights are then modified as:

$$\mathbf{W}_i \quad := \quad \mathbf{W}_i + \alpha \cdot \lambda(\Delta_{\text{topo}}(i, I)) \cdot (\vec{\mathbf{x}}^{\mathrm{T}} - \mathbf{W}_i) \qquad \text{for } i = 1, 2, \ldots, m; \tag{8}$$

where $\Delta_{\text{topo}}(i, I)$ is the distance of neuron i from the winner neuron I in the topological space. The *neighbourhood function* λ restricts the update to neurons close to the winner. The result, after repeated presentation of the input vectors, is a population of neurons whose selection through equation (7) is equiprobable. In another version of the algorithm, instead of detecting the winning neuron I and updating the weights with $\lambda(\Delta_{\text{topo}}(i, I))$, these operations are replaced with the output after convergence of a fixed recurrent network. The latter network is described by equation (1) with the input $\vec{\mathbf{x}}$ of one iteration being set to the output $\vec{\mathbf{y}}$ of the previous one.

3: Array architecture

It has been shown by Blayo [3] that some common connectionist algorithms can be decomposed in simple operations that can be conveniently implemented on a 2-D mesh of PEs. This has been demonstrated with the implementation of a recurrent network in VLSI [9]. The *GENES IV* PE extends the basic scheme to include on-chip learning and to address more connectionist models. Figure 1 (a) shows the basic architecture, together with all the operands: (1) the matrix $\mathbf{W}$, resident in the $N \times N$ PEs; (2) the sequence of input vectors $\vec{\mathbf{I}}^{\mathrm{v}}$ on the top side; (3) the sequence of input vectors $\vec{\mathbf{I}}^{\mathrm{h}}$ on the left side. The result is produced in the vectors $\vec{\mathbf{O}}^{\mathrm{h}}$ or $\vec{\mathbf{O}}^{\mathrm{v}}$, or by modifying the resident matrix $\mathbf{W}$.

The evaluation of the function σ in equation (1) is performed with a look-up table outside the mesh. Similarly, the subtraction and the multiplication of equation (3) are performed externally by a linear array of auxiliary arithmetic units.

3.1: Feed-forward data flow

The first class of operations produces its results in $\vec{\mathbf{O}}^{\mathrm{h}}$ or $\vec{\mathbf{O}}^{\mathrm{v}}$ but does not modify the weight matrix. The operations performed by the PE of indexes i, j are:

$$h_{i,j} \quad := \quad \psi(W_{i,j}, v_{i-1,j}, h_{i,j-1}) \tag{9}$$

$$v_{i,j} \quad := \quad v_{i-1,j} \qquad \text{for } i, j = 1, 2, \ldots, N. \tag{10}$$

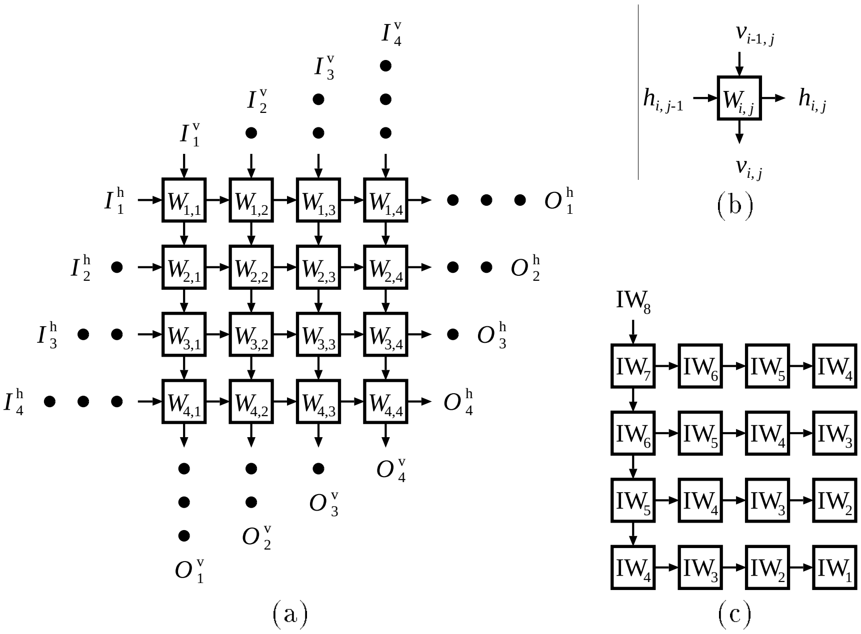

Figure 1. (a) GENES IV systolic array architecture. (b) Definition of the input/output path of a PE. (c) Systolic flow of instruction words (IWs).

Boundary conditions at the edges of the array are:

$$h_{i,0} := I_i^{\mathrm{h}} \tag{11}$$
$$v_{0,j} := I_j^{\mathrm{v}} \tag{12}$$
$$O_i^{\mathrm{h}} := h_{i,N} \tag{13}$$
$$O_j^{\mathrm{v}} := v_{N,j} \qquad \text{for } i, j = 1, 2, \ldots, N. \tag{14}$$

Table 1 shows the functions ψ required for each operation that uses this data flow. The symbols $+\infty$ and $-\infty$ indicate respectively the maximum and minimum representable numbers. The mprod operation implements a classic matrix-vector multiplication scheme. The euclidean operation is very similar, the product being replaced by a squared difference. It computes the squared Euclidean distance between a vector and each row of the matrix $\mathbf{W}$. As the resulting vector of distances is used only in equation (7) to identify the minimum, there is no need to perform a square root. The presence of the additive term $\vec{\mathbf{I}}^{\mathrm{h}}$ in both

Operation	$\psi(a, b, c)$	Result in normal mode	Reference
mprod	$a \cdot b + c$	$\vec{\mathbf{O}}^{\mathrm{h}} = \mathbf{W} \cdot \vec{\mathbf{I}}^{\mathrm{v}} + \vec{\mathbf{I}}^{\mathrm{h}}$	Eq. (1)
euclidean	$(b-a)^2 + c$	$O_i^{\mathrm{h}} = \sum_{j=1}^{N} (I_j^{\mathrm{v}} - W_{i,j})^2 + I_i^{\mathrm{h}}$	Eq. (6)
min	c if $c \leq b$ $+\infty$ if $c > b$	if $I_I^{\mathrm{h}} = \min_i (I_i^{\mathrm{h}})$ then $O_I^{\mathrm{h}} = I_I^{\mathrm{h}}$, all other elements in $\vec{\mathbf{O}}^{\mathrm{h}}$ being $+\infty$	Eq. (7)
max	c if $c \geq b$ $-\infty$ if $c < b$	if $I_I^{\mathrm{h}} = \max_i (I_i^{\mathrm{h}})$ then $O_I^{\mathrm{h}} = I_I^{\mathrm{h}}$, all other elements in $\vec{\mathbf{O}}^{\mathrm{h}}$ being $-\infty$	Eq. (7)

Table 1. Feed-forward operations in normal mode.

the mprod and euclidean operations is fundamental for cumulating the result over different matrixes as required when the physical array is smaller than the implemented network.

The searches for the minimum and the maximum element of a vector are provided. The algorithm requires the input vector to be entered at both $\vec{\mathbf{I}}^{\mathrm{v}}$ and $\vec{\mathbf{I}}^{\mathrm{h}}$. The basic idea is—e.g., in the case of the min operation—to let propagate horizontally an element only if it is smaller than the one entering the PE vertically. If it is not, it is saturated to the maximum representable value. As each element meets all the others before reaching the right side of the array, only the smallest element is output unsaturated. This operation (min) should be used for Euclidean distance while the max operation is appropriate for normalized weights and the scalar product (the minimum angle between the input vector and the neuron weight producing the maximum scalar product).

It can be noted that the error backpropagation requires a matrix-vector multiplication in equation (4), but, compared to equation (1), the weight matrix is transposed. The computation on the transpose matrix can be performed simply by exchanging the role of the horizontal and vertical data flows:

$$h_{i,j} := h_{i,j-1} \tag{15}$$

$$v_{i,j} := \psi\left(W_{i,j}, h_{i,j-1}, v_{i-1,j}\right) \qquad \text{for } i,j = 1,2,\ldots,N. \tag{16}$$

The role of the inputs is also exchanged, and the output is now:

$$\vec{\mathbf{O}}^{\mathrm{v}} = \mathbf{W} \cdot \vec{\mathbf{I}}^{\mathrm{h}} + \vec{\mathbf{I}}^{\mathrm{v}}. \tag{17}$$

Though all other operations can be executed in the *transpose mode*, this has hardly any practical use.

3.2: Weight update data flow

The second class of operations modifies the weight matrix $\mathbf{W}$ without producing any result in $\vec{\mathbf{O}}^{\mathrm{h}}$ or $\vec{\mathbf{O}}^{\mathrm{v}}$. The operations performed by the PE of indexes i,j are:

$$h_{i,j} := h_{i,j-1} \tag{18}$$

$$v_{i,j} := v_{i-1,j} \tag{19}$$

$$W_{i,j} := \psi\left(W_{i,j}, v_{i-1,j}, h_{i,j-1}\right) \qquad \text{for } i,j = 1,2,\ldots,N. \tag{20}$$

Boundary conditions at the edges of the array are given by equations (11) and (12). Table 2 shows the weight updating operations. In both cases, the input vector $\vec{\mathbf{I}}^{\mathrm{h}}$ has already been multiplied, in auxiliary arithmetic units, by the coefficient α. The term $\lambda\left(\Delta_{\mathrm{topo}}\left(i, I\right)\right)$ in equation (8) is computed as follows:

1. The vector containing the m distances (m being the number of neurons) is processed by the min or max operation. An external look-up table is used to convert all the $+\infty$ or $-\infty$, respectively, to 0's and any other value (i.e., the usually single minimum or maximum) to 1.

Operation	$\psi\left(a,b,c\right)$	Result in normal mode	Reference
hebbian	$a + c \cdot b$	$\mathbf{W} = \mathbf{W} + \vec{\mathbf{I}}^{\mathrm{h}} \cdot \vec{\mathbf{I}}^{\mathrm{v}\,\mathrm{T}}$	Eq. (2), (5)
kohonen	$a + c \cdot (b - a)$	$\mathbf{W}_i = \mathbf{W}_i + I_i^{\mathrm{h}} \cdot (\vec{\mathbf{I}}^{\mathrm{v}\,\mathrm{T}} - \mathbf{W}_i)$	Eq. (8)

Table 2. Learning operations.

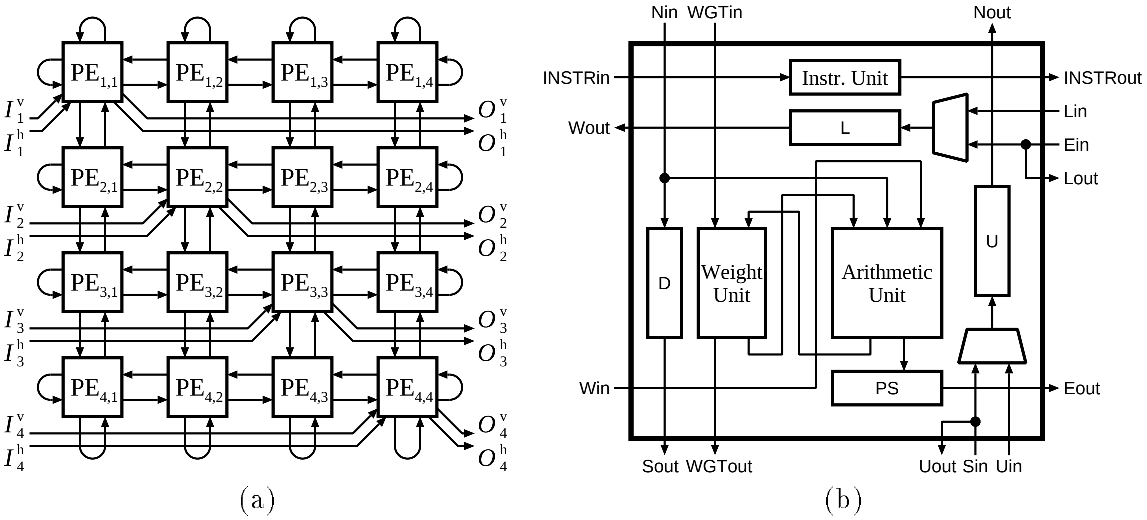

Figure 2. (a) 4 $\times$ 4 array architecture. (b) Processing element basic structure.

2. This vector is then multiplied (mprod operation) by an $m \times m$ matrix $\mathbf{\Lambda}$ expressing at the same time the topological distance of the neurons and the neighbourhood function. The elements of this symmetric matrix are $\mathbf{\Lambda}_{i,j} = \lambda\left(\Delta_{\text{topo}}\left(i,j\right)\right)$. This matrix formulation of the topological distance has the advantage of allowing to compute any type of network (e.g., multi-dimensional orthogonal arrays or 2-D hexagonal arrays).

3.3: Systolic instruction flow

The number of stages T of the 2-D mesh together with the look-up table and the auxiliary arithmetic units impose to process a minimum of T input vectors at a time to optimize the utilization rate. If an *epoch* (batch) of T vectors is used, the first result becomes available the cycle after the last input vector has been injected in the mesh. The latter is thus still processing the old vectors while a new operating mode would be required to process the results. A conventional *single instruction flow, multiple data flow (SIMD)* approach would require the array to be emptied before the new computation phase can be started. To improve the throughput, an *instruction word (IW)* can be associated with each input vector and systolically propagated with it, as shown in figure 1 (c). This allows the operation mode to be changed progressively and avoids intermediate inactive cycles.

4: Processing Element (PE) structure and implementation

The architecture of the implemented array and the structure of the PEs are illustrated in figure 2. Unlike the basic architecture shown in figure 1 (a), the access to the processing array is done through input and output ports located on the top-left to bottom-right diagonal. These inputs (Uin and Lin) and outputs (Uout and Lout) are present on all PEs for modularity but are used only on diagonal cells. They are always bypassed or left unconnected on non-diagonal cells. As figure 2 (b) shows, two further registers (U and L) are required for this path. The advantage is that no external hardware is required to *diagonalize* the inputs or *un-diagonalize* the outputs to and from the array (see figure 1 (a)). All the components of a vector are entered or read at the same time.

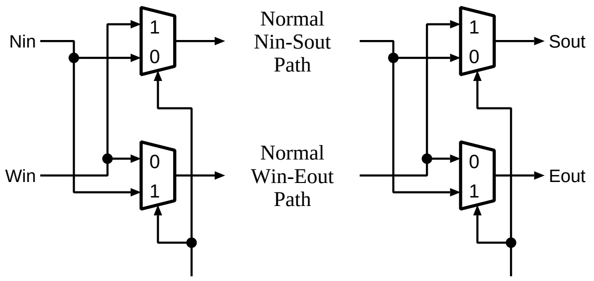

Figure 3. Circuit to operate on the transpose matrix.

The transposition mechanism, described by equations (15) and (16), is added to the PE by providing two control signals from the systolic instruction unit. A first signal exchanges the roles of Nin and Win, while the other exchanges Sout and Eout, as shown in figure 3. When a computation on the transpose matrix begins, the inputs are exchanged. The quantities in the PE registers, belonging to the previous computation are output on the regular path. On the next computation step, the outputs are also exchanged. The same applies when the operation mode reverts to the direct matrix.

The inter-PE communication is bit-serial LSB-first. The bit-serial protocol is imposed by the silicon area of a parallel multiplier (preventing multiple PEs on a single die) and by the number of pads required for the parallel communication (about 400). Of course, the performance is reduced by a factor equal to the number of bits of the longest data word in the PE.

4.1: Double weight-register unit

Figure 2(b) also shows two inter-PE connections not seen in figure 2(a): the weight path and the instruction path. The latter one has already been discussed in connection with figure 1(c). The *weight units* are loaded through a dedicated vertical data path independent from the two standard data inputs. A key feature of a general-purpose NN accelerator is the ability to support problems larger than the physical array available. The weight unit contains two weight registers, one connected on the weight data path, and the other used for the computation. The multiplexers shown in figure 4 allow the two registers to be exchanged in between two instructions. This way, a new weight matrix can be loaded in the background while an epoch of input vectors is being processed with another one. Figure 5 shows the timing of the exchange, supposing that every epoch a new weight

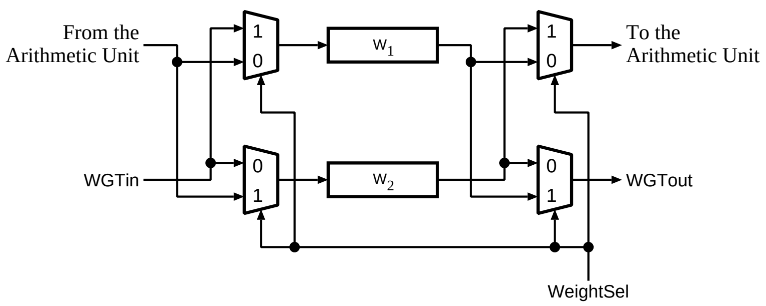

Figure 4. Double weight-register unit.

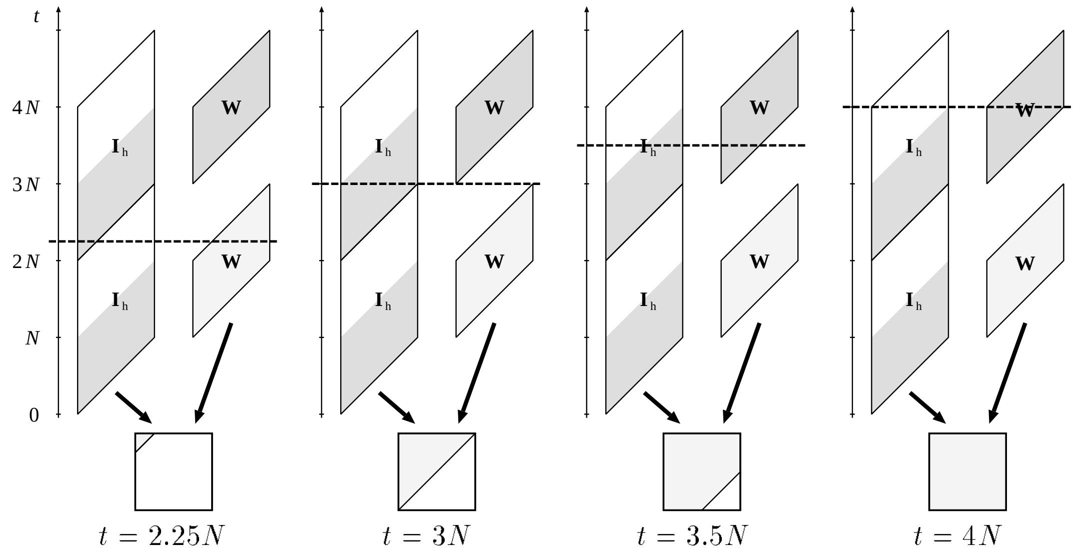

Figure 5. Background weight exchange mechanism.

matrix has to be used. The square at the bottom represents the $N \times N$ array of PEs and the shaded areas indicate which of the weight matrixes is currently in use by the PEs in that zone. The groups of $2N$ input vectors, each composed of N elements, are depicted as parallelograms injected from the top side even if the inputs are actually on the diagonal. At $t = 2.25N$, most of the N PEs in the first column are still using an old matrix. By $t = 3N$, thus after N vectors have been entered, until just before a new epoch is started (N cycles later, i.e., at $t = 4N$ in the figure), all the PEs in the first column use the first matrix shown in figure 5. During this time, the column of weights can be shifted in the background without affecting the current computation. The whole process requires an epoch length of at least $2N$ vectors, which happens to be compatible with the latency of the whole array. It can be noted that this process applies to the other columns but one cycle later for each column, thus requiring the weight matrix to be injected diagonalized. Three registers could have allowed a non-diagonalized download at the expenses of a slightly larger PE structure.

4.2: Arithmetic precision

The choice of the arithmetic precision of the hardware is quite delicate. A first, typical choice, motivated by hardware simplicity, is to represent all variables in two's complement fixed point. As for the required number of bits for each element, theoretical analysis usually leads to absolute minimum boundaries that, on themselves, cannot guarantee convergence [13]. In the absence of strong analytical grounds, simulation usually provides the designer with the required information. In [2] and [7] simulations are performed to determine the needs of typical backpropagation applications. These have been completed by simulations of the Kohonen model in the application that prompted the development of the present system. This resulted in the following choices:

- The weights for the forward operations (mprod and euclidean) are stored on $N_{\mathsf{W}} = 16$ bits. Simulations show that more than 9–10 bits are required for real-world applications of the Kohonen network. The closest larger multiple of 8 was therefore chosen. Similarly, input data are stored on $N_{\mathsf{D}} = 16$ bits. $N_{\mathsf{W}} = N_{\mathsf{D}}$ because in the

Kohonen network the weights and the inputs belong to the same space.

- The minimum number of bits to avoid overflow in the accumulation register PS for the mprod operation (in two's complement) is:

$$N_{\mathsf{PS}} \quad = \quad N_{\mathsf{W}} + N_{\mathsf{D}} + \lfloor \log_2(n) \rfloor \tag{21}$$

  where n is the number of inputs of the network being simulated (independently from the size of the physical array on which the computation is performed). 39 bits (plus one overflow bit) have been chosen, allowing networks with up to 255 inputs to be computed with no overflow, in mprod mode (somewhat less in euclidean mode). Much larger networks can be simulated if occasional overflows can be tolerated.

- The results of weight-update computations hold on about $2N_{\mathsf{W}}$ bits. To avoid truncation (whose negative effects are shown in [2]) and to keep the maximum information, a hidden fractional part, consisting of 16 bits more, is added to the weight registers. It is hidden in the sense that this part is not used in the forward mode but it accumulates the fractional part of successive updates. This has the important consequence of allowing convergence to continue also when α becomes small enough that single updates no longer reach unity.

In practice, due to the requirements for the min and max operands, and because of the transposition mechanism, all the data registers are composed of 40 bits and the weight registers have a total of 33 bits each (all registers include an overflow bit).

4.3: Arithmetic unit

The arithmetic unit (computing the functions $\psi(a, b, c)$ listed in tables 1 and 2) requires three building blocks: a multiplier, an adder, and a subtracter. The choice of the multiplier is a delicate one. The mprod operation is not critical, a compact serial-parallel multiplier could be used since the weight comes from an internal register and can easily be read in parallel while the other factor comes from the vertical data channel in serial form. Unfortunately this scheme could not be used for the weight update (hebbian or kohonen) and for the square in the euclidean operation, where both factors serially arrive from external inputs. A solution could be to wait for the arrival of one of the elements, and then multiply by the other, delayed as required. Unfortunately, this is only possible with a loss in performance (additional cycles for operand transmission) or an increase in the pipelined nature of the array (requiring an even more complex control logic). Typical serial-serial multipliers have some cycles of latency between the first input bit and the first output one [4]. This would again lead to additional delay cycles in operations like euclidean, where the accumulation is performed on the longest data word of the system. An improved multiplication scheme was therefore developed for two's complement numbers [8]. A 17-bit multiplier has been implemented in the PE. Apart from presenting only a combinational delay, this design has the additional advantage of indefinitely sign-extending the output as long as the inputs remain correctly sign extended. This is useful in many situations, e.g., when adding a 34-bit result to a 39-bit word.

It can be noted in figure 2(b) that the arithmetic unit gets the external operands directly from the input channels and that the registers are placed after it. This simplifies the implementation of the min and max search and allows, on the last clock cycle, when the comparison has been performed, to saturate to $+\infty$ or $-\infty$ the value propagating in the PS register. Similarly, this allows to saturate the result of other operations, when an overflow is detected. The saturation should allow the convergence of algorithms even in presence of

Operation	Condition	Sign
mprod	$\psi(a,b,c) \notin [-2^{38}, 2^{38}-1]$	$\text{sgn}(\psi(a,b,c))$
euclidean	$\psi(a,b,c) \notin [0, 2^{38}-1]$	$+$
hebbian	$\psi(a,b,c) \notin [-2^{31}, 2^{31}-1]$	$\text{sgn}(\psi(a,b,c))$
	$c \notin [-2^{16}, 2^{16}-1]$	$\text{sgn}(c \cdot b)$
kohonen	$\psi(a,b,c) \notin [-2^{31}, 2^{31}-1]$	$\text{sgn}(\psi(a,b,c))$
	$c \notin [-2^{16}, 2^{16}-1]$	$\text{sgn}(c \cdot (b-a))$

Table 3. Overflow conditions.

occasional overflows. A sticky-bit is nevertheless added to the PS register (and consequently to all the other data registers) and to the weight registers. This flag informs the user that an anomalous situation occurred and the network may require attention. Table 3 list the saturation sign under the possible overflow conditions. In the table, a, b and c are the inputs to the arithmetic unit and $\psi(a,b,c)$ is its full-precision result. Operand b is never checked to hold on 16 bits, because this is already imposed by the external hardware. When $c \notin [-2^{16}, 2^{16}-1]$ in weight update operations, the product cannot be determined with the 17-bit multiplier and it is therefore assumed that it always dominates over the additive term a.

5: System architecture

The *MANTRA I* system [14] is shown in figure 6. The computational heart is a GENES IV array of 40×40 PEs. The sequencing of the systolic array is performed by a TMS320C40 *digital signal processor (DSP)* from Texas Instruments. It also handles the communications with a host workstation and between different MANTRA I machines connected through the dedicated communication links of the DSP.

Two look-up tables (indicated as σ and σ' in figure 6) are used to compute the evaluation function and its derivative. A double look-up table scheme allows a coarse-grain table to map the whole input space with reduced precision and a fine-grain one to map a small window with high precision (up to one entry per input value). It is possible to zoom the fine-grain window in or out by trading window width for precision. As already mentioned, a linear array of auxiliary arithmetic units (*GACD1*) computes the error for the

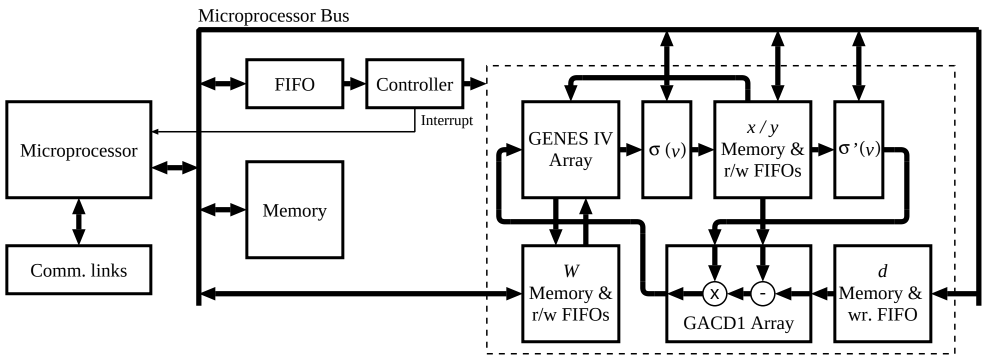

Figure 6. General architecture of the MANTRA I machine.

Model	Evaluation phase Operations	MCPS	Learning phase Operations	MCUPS
Perceptron, ADALINE, Delta rule, and BP rule (output-layer)	mprod	400	mprod + hebbian	200
Backpropagation (BP) rule (all layers but the last)	mprod	400	$2 \times$ mprod + hebbian	133
Kohonen with Euclidean distance and minimum	euclidean + min	200	euclidean + min + mprod + kohonen	100

Table 4. GENES IV peak performance.

backpropagation rule. Some memory banks locally store inputs, outputs, desired outputs, and weights, to minimize transfers from and to the DSP memory.

6: Performance

Two VLSI chips were designed in CMOS 1μm technology using VLSI Technology standard cells: the first one integrating three GACD1 cells and the second one housing an array of 2$\times$2 GENES IV PEs. The latter chip contains 71,690 transistors (3,179 standard cells) on a die measuring $6.3 \times 6.1\,\text{mm}^2$. Both chips have been successfully tested. The MANTRA I system has been built and debugged. It is composed of four different printed-circuit boards and will host about 900 integrated circuits in the largest configuration.

No comprehensive benchmark has been run so far, but a peak performance can be calculated using the two classic metrics for neuro-computers: (1) the number of *connections per second* or *CPS* (evaluation phase) and (2) the number of *connection updates per second* or *CUPS* (learning phase). The expected performance of GENES is given by:

$$P = \frac{N^2 \cdot f}{n_{\text{op}} \cdot N_{\mathsf{PS}}} \cdot U \tag{22}$$

where N^2 is the total number of PEs, f is the clock frequency, n_{op} is the number of operations required to compute a connection (evaluation phase) or to update it (learning phase), $N_{\mathsf{PS}} = 40$ is the number of clock cycles per operation, and U is the utilization rate. Using the maximum supported array in the MANTRA I system (40×40) at a clock frequency of $f = 10\,\text{MHz}$, the peak performance ($U = 100\,\%$) of GENES IV is shown in table 4. Delta rule algorithm tests on the prototype show that a utilization rate $U \approx 95\,\%$ can easily be sustained if the problem data are already in the DSP memory.

These values compare well to those reported for supercomputers (e.g., 130 MCUPS on the NEC SX-3 [12]). The results show that, all differences taken into account (such as the fixed or floating point), neuro-computers can efficiently and cost-effectively compete with larger general-purpose systems. The performance is also expected to be comparable or superior to most other NN-dedicated systems (e.g., the RAP, 45 MCUPS on a backpropagation problem [11]) or VLSI accelerators (Lneuro 1.0, 32 MCUPS [10]). The CNAPS Server/512 [1] has a higher peak performance of 1,460 MCUPS (at a lower precision) thanks to a higher clock rate, parallel PE communication, and a more advanced VLSI technology. Its PE on-chip memory may however severely limit the size of the networks that can be simulated.

7: Conclusions

Neural-network accelerators connected to scientific workstations are appealing alternatives to large computers, when very large computational power is required and reduced cost is at a premium. The described architecture represents an acceptable compromise between simplicity and regularity of design (thus implying a reduced cost) and generality of use. Primitives are implemented to provide most of the typical neural-network algorithm building blocks. Future evolution may attempt to improve generality even more by slightly regularizing the processor structure (general purpose registers, wider instruction word, etc.) and obtain a reduced system complexity by simplifying the array interface. Finally, the small PE size, the reduced inter-PE connectivity and the high regularity make this architecture a good candidate for a wafer scale integration implementation.

Acknowledgements

This work is supported from the Swiss National Fund for Scientific Research through the SPP-IF program and from the Swiss Federal Institute of Technology through the MANTRA project. The authors wish to thank Prof. Jean-Daniel Nicoud for his constant support. The original works of François Blayo and Christian Lehmann provided the bases of the present project.

References

[1] Adaptive Solutions, Inc., Beaverton, Oreg. *CNAPS Server*. Preliminary datasheet.

[2] Krste Asanović and Nelson Morgan. Experimental determination of precision requirements for back-propagation training of artificial neural networks. In *1st International Conference on Microelectronics for Neural Networks*, pages 9–15, Dortmund, 1990.

[3] François Blayo. *Une implantation systolique des algorithmes connexionnistes*. PhD Thesis N° 904, École Polytechnique Fédérale de Lausanne, Lausanne, 1990.

[4] Luigi Dadda. Fast multipliers for two's-complement numbers in serial form. In *IEEE 7th Symposium on Computer Arithmetic*, pages 57–63, 1985.

[5] Neil A. Gershenfeld and Andreas S. Weigend. The future of time series: Learning and understanding. In A. S. Weigend and N. A. Gershenfeld, editors, *Time Series Prediction: Forecasting the Future and Understanding the Past*, pages 1–70. Addison-Wesley, Reading, Mass., 1993.

[6] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the Theory of Neural Computation*. Santa Fe Institute Studies in Sciences of Complexity. Addison-Wesley, Redwood City, Calif., 1991.

[7] Jordan L. Holt and Thomas E. Baker. Back propagation simulations using limited precision calculation. In *International Joint Conference on Neural Networks*, Seattle, Wash., July 1991.

[8] Paolo Ienne and Marc A. Viredaz. Bit-serial multipliers and squarers. *IEEE Transactions on Computers*, To appear.

[9] Christian Lehmann. *Réseaux de neurones compétitifs de grandes dimensions pour l'auto-organisation: analyse, syntèse et implantation sur circuits systoliques*. PhD Thesis N° 1129, École Polytechnique Fédérale de Lausanne, Lausanne, 1993.

[10] Nicolas Mauduit, Marc Duranton, Jean Gobert, and Jacques-Ariel Sirat. Lneuro 1.0: A piece of hardware LEGO for building neural network systems. *IEEE Transactions on Neural Networks*, 3(3):414–22, May 1992.

[11] Nelson Morgan, James Beck, Phil Kohn, Jeff Bilmes, Eric Allman, and Joachim Beer. The Ring Array Processor: A multiprocessing peripheral for connectionist applications. *Journal of Parallel and Distributed Computing*, 14:248–59, 1992.

[12] Urs A. Müller, Bernhard Baümle, Peter Kohler, Anton Gunzinger, and Walter Guggenbül. Achieving supercomputer performance for neural net simulation with an array of digital signal processors. *IEEE Micro*, pages 55–65, October 1992.

[13] Patrick Thiran, Vincent Peiris, Pascal Heim, and Bertrand Hochet. Quantization effects in digitally behaving circuit implementations of Kohonen networks. *IEEE Transactions on Neural Networks*, To appear.

[14] Marc A. Viredaz. MANTRA I: An SIMD processor array for neural computation. In *Proceedings of the Euro-ARCH'93 Conference*, München, October 1993. In press.