A Reconfigurable Computer REOMP

Alessandro Noriaki Ide[1], José Hiroki Saito[2]
[1]Universidade Federal de São Carlos, Programa de Pós-Graduação em Ciência da Computação
Rodovia Washington Luis (SP-310), Km 235, São Carlos, SP, Brasil
{noriaki@dc.ufscar.br}
[2]Universidade Federal de São Carlos, Departamento de Computação
Rodovia Washington Luis (SP-310), Km 235, São Carlos, SP, Brasil
{saito@dc.ufscar.br}

Abstract—
This work describes a proposal of reconfigurable computer, and their application to hardware implementations of neural networks. Although the neural network functions correspond to the brain functions, our computer is based on the current technology, which is completely different from the internal structure of the brain based on the neuronal cells. The proposed Reconfigurable Orthogonal Multiprocessor. REOMP, is based on processing units that are reconfigured to execute the algorithms by demanding driven rule. The performance analysis of the architecture is made with the implementation of an artificial neural network, neocognitron, which involves concurrent operations of a great number of artificial neurons. The analysis of the architecture showed that its speed-up is linear in a wide range, where the implementation of REOMP is appropriate. We conclude that the proposed architecture is able to be used to neural network hardware implementation. To obtain the best performance of the architecture, the neural network model should make use of massively parallel neural processing of the previously processed data that is the case of feedforward neural networks.
Keywords— Reconfigurable Computer, Neural Network, Hardware Implementation, FPGA, MPI

I. INTRODUCTION

This work proposes a reconfigurable hardware to the implementation of neural networks. Although the neural networks corresponds to the functions of the human brain, the proposed architecture is based on the current digital technology, which is completely different from the brain internal structure based on the neuronal cells. This is the same when we think about the airplane construction, based on the flying birds [MAT 98]. The airplane is designed based on the hydro dynamical physics, while the bird's contribution to the airplane construction is the fact that they actually fly over the sky. The effort of creating the brain brings us ingeniously scientific product which eventually leads us to the profound understanding of the brain as well as to the advancement of computer engineering.

The human brain is composed by more than 10 billions of neurons, which are known as processing units, which functions as an analogy component that processes the weighted sum of their great number of input connection values, and produces a response that propagates to other neurons.

Brain functions are spatially distributed. For example, it is known that the temporal cortex region is responsible by the early vision. On the other hand, the Von Neumann sequential computer is temporally distributed, where some general-purpose processing elements are used sequentially to process a sequence of instructions in memory. By this limitation, although very fast to process arithmetic operations, the sequential computer is not suitable to specialized processing as human face recognition. Artificial neural network approaches have been proposed to overcome this shortcoming of the sequential computers.

If we are proposing to implement the brain functions, the exploitation of the spatial distribution of the processing elements seems more suitable, as the neurons are spatially distributed in the brain. This is similar to the most of artificial neural network models, where the threshold elements, or neurons, are spatially distributed. In this case, if our purpose is to implement the whole human brain functions, in a reconfigurable computer, we must have more than 10 billion processing elements spatially distributed, that is obviously impracticable with the current technology.

The construction of a Reconfigurable Orthogonal Multiprocessor Architecture REOMP is justified by the fact of the limitation of sequential computers and the impossibility of the implementation of too many processing elements. Furthermore, the realization of hardware implementation of neural networks, in the REOMP, is scalable, and reconfigurable, so that many neural network processing algorithms may be implemented using a suitable number of processing elements.

II. RECONFIGURABLE COMPUTER

A reconfigurable computer [GOL 00][DEH 00] provides the solution to the hardware implementation of neural networks using the current technology. This approach assumes that not all the neural networks functions are active all the time. Only the active functions are configured in the computer during a snapshot of operation.

The reconfigurable computer has a reconfiguration unit and fixed units. It uses configurable components as Field Programmable Gates Arrays, FPGA's, which are configured after fabrication, to implement a special function in the reconfigurable unit. A FPGA is an array of bit level processing elements whose functions and interconnections can be programmed after fabrication. Most traditional FPGA's use small lookup tables to serve as programmable elements. The lookup tables are wired together with a programmable interconnect, which accounts for most of the area in each FPGA cell. Several commercial devices use four input lookup tables (4-LUT's) for the programmable elements. Commercial architectures have several special purpose features, as carry chains for adders, memory nodes, shared bus lines.

The configuration unit is efficient if it implements the spatially distributed processing elements to exploit the streaming of the data path, as in the neural network. Obviously, some functional systems of the neural networks are always active, that can be implemented at the fixed unit of the computer.

A reconfigurable computer partitions computations between two main groups: (1) reconfigurable units, or fabric; and (2) fixed units. Reconfigurable units exploit the reconfigurable computations, which is efficient when the main computation is executed in a pipelined data path fashion. Fixed units exploit system computations that control the reconfigurable units.

Reconfigurable units are reconfigured to implement a customized circuit for a particular computation, which in a general-purpose computer is cost prohibitive, to be implemented in hardware, because of its reduced use. The compiler embeds computations in a single static configuration rather than an instruction sequence, reducing instruction bandwidth and control overhead. Because the circuit is customized for the computation at hand, function units are sized properly, and the system can realize all statically detectable parallelism.

A reconfigurable unit can outperform a fixed unit processing in cases that: (a) operate on bit widths different from the processor's basic word size, (b) have data dependencies that enable multiple function units operate in parallel, (c) contain a series of operations that can combine into a specialized operation, (d) enable pipelining, (e) enable constant propagation to reduce operation complexity, or (f) reuse the input values several times.

Reconfigurable units give the computational data path more flexibility. However, their utility and applicability depend on the interaction between reconfigurable and fixed computations, the interface between the units, and the way a configuration is loaded.

It is possible to divide reconfigurable computations into two categories: (1) stream-based functions which corresponds to the processing of large, regular data input streams, producing a large data output stream, and having little control interaction with the rest of the computation; and (2) custom instructions which are characterized with a few inputs, producing a few outputs, executed intermittently, and having tight control interactions with the other processes. Stream-based functions are suitable for a system where the reconfigurable unit is not directly coupled to the processor, whereas custom instructions are usually beneficial only when the reconfigurable unit is closely coupled to the processor.

The following session will be concerned to expose some definitions about a reconfigurable card, HOT II PCI, and a parallel processing library, MPI, both used in the proposed architecture.

III. HOT II PCI Development Platform

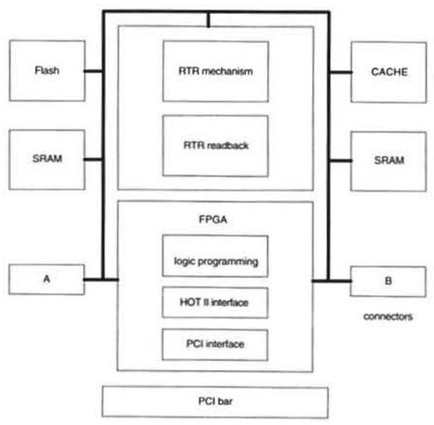

Fig.1 - HOT II PCI development board

The development environment HOT II PCI, Fig.1, produced by VCC (*Virtual Computer Company*), is a standard PCI card, with Xilinx FPGA, XC4013, and FPGA-RTR, Virtex, and a software tool, HotWorks, with a basic set of routines to access and programming of the board FPGA's. The main component (FPGA) is configured as interface PCI32, which allows the user communication with the computational resources of the HOT II PCI card, including two SRAM memory blocks, and a Configuration Cache Manager (CCM). CCM controls the Run Time Reconfiguration (RTR), which changes the system behavior. The FPGA may be configured, with the CCM content. In a flash memory there is an initial configuration of the system, which includes the operation system of the hardware and the PCI interface. Furthermore, it may store three other configurations. The HOT II PCI card has two independent bars, each one with 32 data bits, and 24 address bits.

IV. Message Passing Interface

Message Passing Interface [MPI 95] was developed in 1993-1994, by an industrial, governmental, and academic researchers group, called MPI Forum. MPI was one of the first standards accepted to the message passing in parallel processing. It's a library of functions and macros, which may be used to program in C, FORTRAN 77, and C++, which allows the process communication through messages to write parallel application programs, establishing a practical, portable, efficient, and flexible standard for message-passing. It provides a reliable communication interface (i. e., user don't need to deal with communication failure). It is supposed to deliver high performance on high-performance-systems. It should be modular, to accelerate the development of portable parallel libraries. The resource allocation detail is totally left to the implementer, turning the system flexible and efficient. The C, FORTRAN 77, and C++ programs, which must use MPI, may include mpi.h, mpif.h, and mpi++.h, files, respectively. The main advantages of establishing a message-passing communication are its portability and facility of use. We can notice it in multiprocessors systems with distributed memory, where the MPI routines are built using the conceptions of process, communications, and data types. MPI library may be used by a diversity of users, and heterogeneous platforms. MPI provides two ways of communication: point-to-point and collective communication. Point-to-point communication involves a pair of process that establishes the communication among them using the primitives: *send* (source) and *receive* (destination). On the other hand, collective communication involves a group of process that uses a set of MPI functions. The main collective functions are: *barrier* (responsible by the synchronization) and the *broadcast* (responsible for passing the same message for all members of the group). MPI is recommended for parallel machine, tools, environment, and software, developers.

V. Proposed Architecture

The proposed reconfigurable computer REOMP, is composed by: (1) reconfigurable units, responsible by the implementation of neural network functions, partially, not all the functions at the same time; (2) fixed units, responsible by the implementation of the control of reconfigurable units, and the fixed neural network functions; and (3) the memory unit, responsible by data storage.

A PC followed by a HOT II PCI card, connected by a PCI bar, composes a Reconfigurable Processor (RP) of the REOMP. Each RP is connected to other RP's through an interconnection network, to syncronize the reconfigurations, and exchange data and information through the orthogonal memory modules. All the RP's constitute a parallel environment, by the MPI library, responsible to organize the "*threads*" or processes of the spatial computations [DEH 99]. Fig.2 shows the implementation.

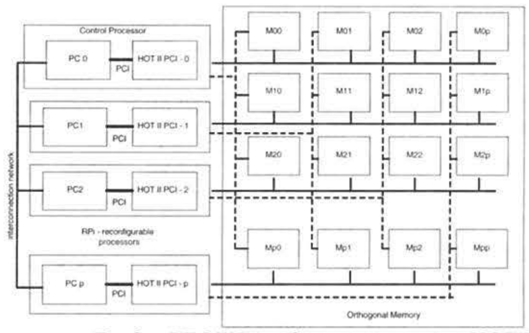

Fig.2 – REOMP implementation using HOT II PCI cards.

The REOMP, implemented through the HOT II PCI card, may emulate the NEOMP (Neural Orthogonal Multiprocessor) architecture [SAI 99], which is composed by several reconfigurable processors, making access of the orthogonal memory [HWA 89][HWA 93], where the data to be processed are stored.

The overall system is an orthogonal multiprocessor, OMP, which is characterized by the parallel processing units, each one accessing their memory modules in two ways: column and row. Each row, and each column, are attributed to an exclusive processor. The row access and column access are performed exclusively, without time-sharing of the buses by the multiple processors, reducing the memory access conflicts.

Fig. 3 shows a typical data spreading operation using the REOMP memory modules. At the left side we can see the diagonal memory modules that correspond to local memory of the processors, with their respective data. Row access allows the processors connected to the exclusive rows spread the data to all row modules, simultaneously. At the right side, we see the result of parallel data spreading operation, with all columns containing the same data, distributed in several memory modules. During the following column access, all processors may access their own column memory modules, which contain the spread data.

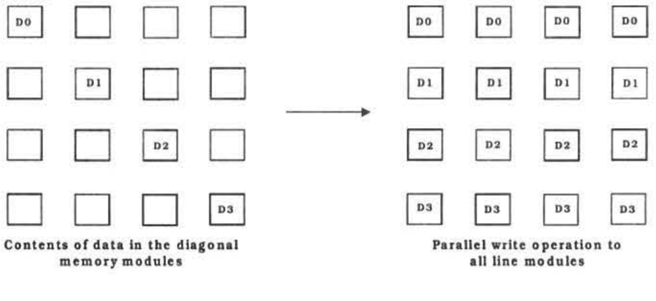
D0
D1
D2
D3
D0
D0
D0
D0
D1
D1
D1
D1
D2
D2
D2
D2
D3
D3
D3
D3
Contents of data in the diagonal memory modules
Parallel write operation to all line modules

Fig. 3- A typical parallel data spreading at the REOMP memory modules

VI. REOMP RECONFIGURATION

The orthogonal memory is considered as containing several cell planes, each one corresponding to neural network array of cells, or their corresponding temporary data (such as input values, intermediate values and output values). Fig. 4 shows the data flow diagram of the REOMP processing, where the cell planes, which constitutes the orthogonal memory contents are read and processed by an ALU (Arithmetic and Logic Unit) and the result is written back. Each word of the orthogonal memory is been manipulated as a register, so that in a binary operation two different registers may be addressed to provide operands of the ALU. The result can be addressed to be written on another register. At the Fig. 4, it is shown several cell planes (1, 2, ..., n), each one corresponding to an orthogonal memory module.

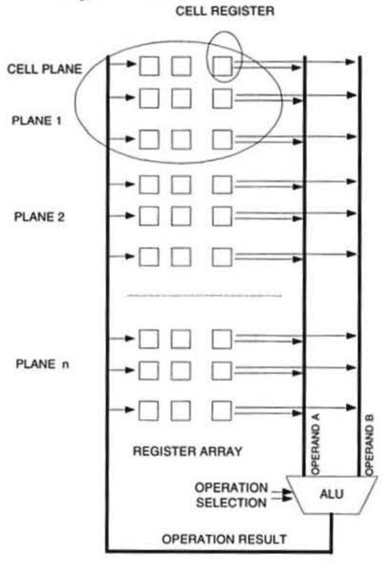

Fig. 4 - Data flow of the REOMP processing.

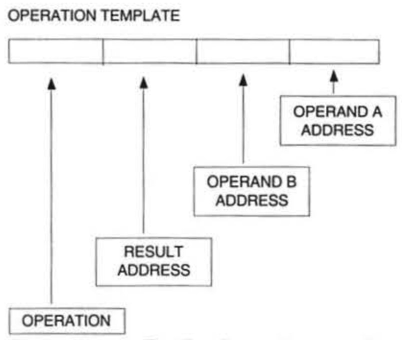

Fig 5 - Operation template

REOMP processing is based on operation templates, which indicates the two operands (A, and B) of the ALU, the ALU operation, and the result storage address, Fig.5. These templates are queued in the reconfigurable unit, Fig.6, and executed in pipeline. Fig.7 shows the template operation during its execution.

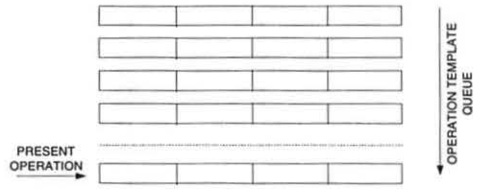

Fig 6 - Operation template queue.

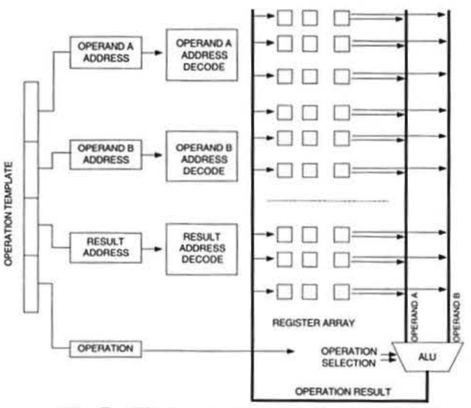

Fig 7 - The operation template and their execution.

Fig.8 shows the data flow to the spread operation. After data processing, the resulting data are spread out to the suitable cell planes to the next stage operation. This operation is performed with a buffer register, connecting all the cell planes of the register array. This data spread operation is possible changing the access mode of the orthogonal memory, from column to row, or vice-versa, depending on what access mode is been used. A special operation template, indicating the spread element address, and a mask containing the information about the cell planes to be written, may execute a data spread operation. Fig.9 shows the spread operation template, composed by the spread element address and a set of mask planes, responsible to enable memory module access.

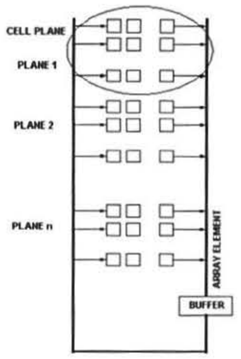

Fig 8. Data flow of REOMP during memory data spread operation.

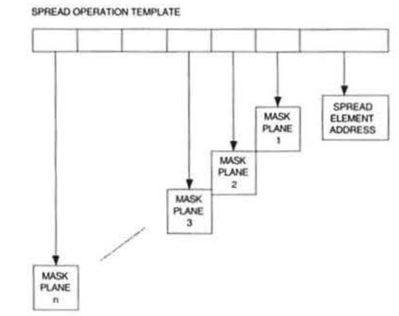

Fig 9. Operation template for the data spread operation.

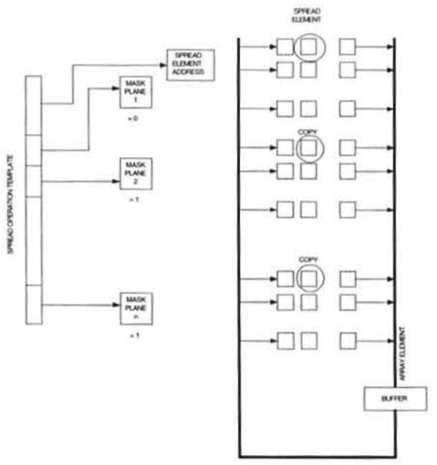

Fig 10. The data spread operation in the REOMP.

Fig.10 shows one example of the data spread operation during execution. The spread element address marks the cell plane position where the spread element is located. The data is spread to the enabled cell planes, according to the template, using an intermediate register buffer. The spread element is copied at the same position of the corresponding cell planes.

The spread operation provides an interesting solution to the neuronal data exchange after one layer processing, in a feedforward network implemented in parallel architecture.

To the validation of the proposal, it is used the parallel processing of a feedforward neural network, known as neocognitron, and its performance analysis, as follows.

VII. REOMP Reconfiguration: A Case Study

A. Neocognitron, a Feedforward Neural Network

As an example it is used the configuration of REOMP to the implementation of a neural network model simulating the human vision function, known as neocognitron.

Neocognitron is a massively parallel neural network, composed by several layers of neuron cells, proposed by Fukushima [FUK 79] [FUK 82] [FUK 92] [FUK 96] [SAI 98] inspired by Hubel and Wiesel's model of biological vision. As other neural network models, neocognitron has its self-organized training phase and the recognition phase, when used in pattern recognition.

The lowest stage of the network is the input layer U_0. Each of the succeeding i-th stages has a layer U_{Si} consisting of S-cells followed by a layer U_{Ci} of C-cells. Each layer of S-cells or C-cells is composed by a number of two-dimensional arrays of cells, called *cell-planes*. Each cell-plane is associated to a single feature extracted from the training patterns, during the learning phase. The S-cells are responsible by the feature extraction, and the input connection weights of the S-cells are adjusted during the training phase. C-cells are responsible by the distorted features correction. Each cell inside a cell-plane receives input connections from the preceding layer cell-planes.

The recognition phase computation of neocognitron, follows the sequence showed at the following algorithm, Fig.11, from layer 1 to L, where L is the number of stages.

```
Program neocognitron ();
begin
   For $l = 1$ to $L$ do compute_stage ($l$);
end;
```

Fig 11. Neocognitron computation sequence.

```
Procedure compute_stage ($l$) ;
Begin
  for $k = 1$ to $K_l$ do begin
      for $n = 1$ to $N$ do begin
         for $\kappa = 1$ to $K_{l-1}$ do
            for all $\nu \varepsilon S_\nu$ do begin
               $e(n,k):=e(n,k)+a(\nu,\kappa,k).u_{Cl-1}(n+\nu,\kappa)$ ;
               $h(n,k):= h(n,k)+c(\nu).\{u_{Cl-1}(\kappa,n+\nu)\}^2$ ;
            end;
            $u_{Sl}(n,k):=(\theta/(1-\theta))$ .
            $\varphi((1+e(n,k))/(1+\theta.b(k).sqrt(h(n,k)))-1)$ ;
```

```
        for all $v \varepsilon S_v$ do
            $u_{Cl}(n,k):=u_{Cl}(n,k+ d(v).u_{Sl}( n+v, k)$;
        $u_{Cl}(n,k):= \Psi\ ( u_{Cl}(n,k))$;
    end
  end;
end;
```

Fig 12. Algorithm to compute the S-cells and C-cells of the l-th stage.

Fig.12 shows the algorithm to compute the output value $u_{Sl}(n, k)$ of a *S*-cell, and the output value $u_{Cl}(n, k)$ of a *C*-cell, from the stage *l*. It is computed the output values to all K_l cell-planes and all *N* cell-positions inside the cell-plane.

To obtain the $u_{Sl}(n, k)$ value, it is computed the weighted sum, *e(n,k)* and *h(n,k)*, of all inputs coming from all K_{l-1} cell-planes of the preceding layer, in a given connection area *S.*, which surrounds the position *n*, of the preceding layer *C*-cell, or input layer, by the following commands (1) and (2)

$$e(n,k):=e(n,k)+a(v,\kappa, k).u_{Cl-1}(n+v,\kappa) , \qquad (1)$$

and

$$h(n,k):=h(n,k)+c(v).\{u_{Cl-1}(\kappa,n+v)\}^2 \qquad (2)$$

Then the $u_{Sl}(n, k)$ value is obtained by the assignment:

$$u_{Sl}(n,k):=(\theta/(1-\theta)).\varphi((1+ e(n,k))/ (1+\theta.b(k).sqrt(h(n,k))-1), \qquad (3)$$

where

$\varphi(x) = x$, when $x > 0$, and $\varphi(x) = 0$, elsewhere. The variable θ represents the threshold of the function, whose value is behind *0*, and *1*, and *b(k)* represents the inhibition coefficient.

To obtain the $u_{Cl}(n, k)$ value, it is at first computed the weighted sum of all inputs corresponding to the previously obtained $u_{Sl}(n, k)$, in a given connection area *S.*, which surrounds the position *n*, of the preceding *S*-cell layer, by the following assignment:

$$u_{Cl}(n,k):=u_{Cl}(n,k)+d(v).\ u_{Sl}(n+v,k) \qquad (4)$$

followed by calculation of transfer function $\Psi(x)= \varphi(x)/(1+\varphi(x))$ which limits *C*-cell output to the range *[0,1]*

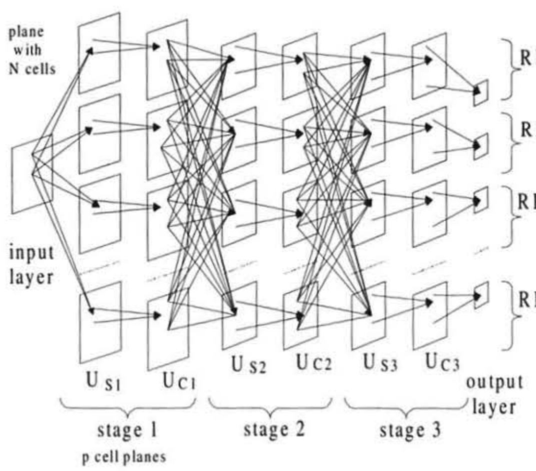

Fig 13. Neocognitron structure with 3 stages of S-cells (U_{Si}) and C-cells (U_{Ci}).

Fig. 13 shows a neocognitron structure with *3* stages of *S*-cells and *C*-cells. All *k* cell planes from stage*1* takes the input values from the input layer. After the parallel processing of the *k* cell planes in layer *l* by the corresponding Reconfigurable Processor (RP_i), all results are used by all cell planes of the layer *l+1*, as shown by the interconnections between layers. These interconnections refer to the orthogonal memory data spreading operation, which occurs after the layer processing. After the data spreading operation, each processor can access all the preceding layer data, directly in the orthogonal memory without conflict. The last stage is connected to the output layer, which corresponds to network result layer.

B. Concurrent Computing Modules

By the analysis of the neocognitron model described at previous section it was extracted a set of concurrent computation modules, which corresponds to compound vector functions [HWA 93]. Four modules *E, H, S,* and *C*, correspond to the neocognitron computing algorithm.

***E*-module** computes to all *N* positions of the cell-plane, the weighted sum of the input $u_{Cl-1}(n+v,\kappa)$, with the weight $a(v,\kappa,k)$, within the connection region *Sv*, which results in a partial value of *e(n,k)* , corresponding to the plane κ. Each *E*-module issue repeats the flow graph *N* times to compute all positions of the *E* matrix. Its result is added to the *E* matrix that will accumulate the weighted sum of all κ cell-planes, after κ issues of the *E*-module function.

***H*-module** is similar to the *E*-module but the weight values are *c(v)* and the input values are squared before the computation of the weighted sum. It results in the *H* matrix, which will contain the weighted sum of all preceding layer cell-planes, after κ issues of the *H*-module function.

***S*-module** computes $u_{Sl}(n,k)$, to all N positions of the cell-plane, using the results of the previously described *E*-module and *H*-module functions.

***C*-module** corresponds to the computation of the $u_{Cl}(n,k)$, to all N cells. It computes the weighted sum of $u_{Sl}(n+\nu, k)$, by $d(\nu)$, and then the function Ψ.

TABLE I

Processing Modules

Module	Operations Per issue	Complexity O( )	Typical Number of Operations	Execution Time (ms)
E	$2.N.S_v + N$	$N.S_v$	20,400	2.04
H	$N.S_v.3 + N$	$N.S_v$	30,400	3.04
S	N.4	N	1,600	0.16
C	$N.S_v.2$	$N.S_v$	20,000	2.00

We classified the module size as the TABLE I, which shows at the first column the identified modules; at the second column, the number of arithmetic operations per issue of the processing module; followed by the complexity of the processing, in *O* function; at next column it is showed the typical number of arithmetic operations, using $N = 400$, $S_v = 25$, and $K = 50$; and finally, the processing time, using a processor cycle of $\tau = 100$ ns.

It is considered that the arithmetic operations are executed in sequence within the module, disregarding the instruction and operands memory access overheads. We are not considering here the parallel execution, or vector processing, which may improve the processing time of the majority of the modules.

VIII. Performance Analysis

As an example, the mapping of the neocognitron to the proposed architecture may be resumed, as follows. The modules E, H, S, and C, are processed at the vector processors, and the other functions at the scalar processor, or at the host processors.

In a general version of REOMP we propose a special use of the orthogonal memory, as follows. When the processors are writing their results in the orthogonal memory, to the next phase of orthogonal access, they write simultaneously on all accessible memory modules, in congruent addresses. After that, the processors wait for the next orthogonal access, which occurs automatically, after the last processor finishes the processing. At the next orthogonal access, the access mode is changed, to row or column, depending on the last access. When the mode is changed, all the processors have the preceding processing results of all the processors, because of the simultaneous writing on all accessible modules, in the preceding phase. In this way, no times are lost to memory spreading operation, so that the speed-up is linear, as the ideal case.

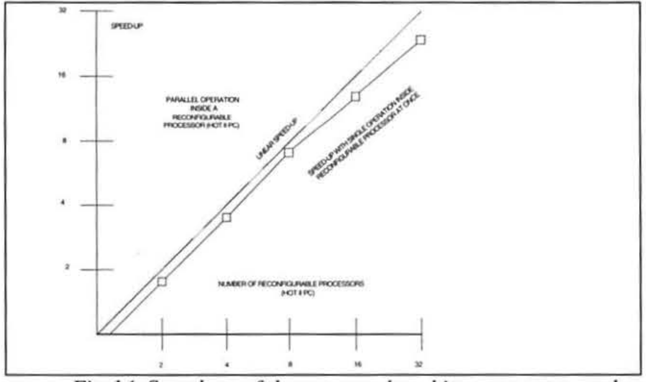

Fig 14. Speed-up of the proposed architecture compared to the ideal speed-up.

Fig.14 shows the speed-up diagram of a typical implementation of REOMP. It can be noted that the speed-up is close to linear along the range of 1 to 16 reconfigurable processors, and after that, degrades slowly, until 32 processors. A good implementation of the proposed REOMP may vary from 2 to 8 processors, where the number of memory modules varies, not excessively, from 4 to 64. Note that in this range we can use the zero spreading time implementation, to reach the ideal speed-up. We note that if it is used the parallel operation inside a reconfigurable processor unit (HOT II PC), implementing several template queues, we have the speed-up improved at the area above the one operation speed-up line.

IX. Conclusions

This work proposes a reconfigurable computer to the hardware implementation of neural networks. Although the neural networks functions corresponds to the brain functions, the proposed architecture is completely different from the brain internal structure based on the neuronal cells. The proposed reconfigurable processors (RP's) are based on commercially available FPGA's (Field Programmable Gate Arrays) and a development environment card, HOT II PCI. The algorithms are processed in the RP´s that are connected by a parallel environment, MPI library. It was chosen due to its flexibility, facility of use and portability. If the processing is finished all the RP's of the REOMP architecture is reconfigured to another algorithm. As the brain, some algorithms or functions are constantly in operation, and others may be alternated. The proposed architecture comports hardware implementation of algorithms that simulates the brain functions, occupying physically restricted space. This is possible by the almost sequential behavior of the brain functioning. The proposed architecture is able to be used as a brainway computer, as the dynamical configuration is executed adequately. The future works are concerned to the development of the other algorithms to the proposed computer, which may result in other reconfiguration design of REOMP. Another work is concerned to the development of software tool to the generation of operation templates to the algorithm.

REFERENCES

[DEH 99] DEHON, A. & WAWRZYNEK J. Reconfigurable Computing: What Why, and Design Automation Requirements ? , Proceedings of the 1999 Design Automation Conference, pp 610 – 615 , June 1999.

[DEH 00] DEHON, A. - The Density Advantage of Configurable Computing, IEEE Computer, Vol. 33, N. 4, 2000.

[FUK 79] FUKUSHIMA, K. - Neural-network model for a mechanism of pattern recognition unaffected by shift in position - neocognitron , *Trans.IECE Japan*, vol.62-A, no.10, 1979.

[FUK 82] FUKUSHIMA, K.& Miyake, S. - Neocognitron: A New Algorithm for Pattern Recognition Tolerant of Deformations and Shift in Position, *Pattern Recognition*, vol. 15, no.6, 1982.

[FUK 92] FUKUSHIMA,.K. & WAKE, N. - Improved Neocognitron with Bend-Detecting Cells, *IEEE – International Joint Conference on Neural Networks*, Baltimore, Maryland, 1992.

[FUK 96] FUKUSHIMA,K.& TANIGAWA, M. - Use of Different Thresholds in Learning and Recognition, *Neurocomputing*, 11, 1996.

[GOL 00] GOLDSTEIN, S.C.; SCHMIT, H.; BUDIU, M.; CADAMBI, S. Moe, M. & TAYLOR, R.R. - PipeRench: A Reconfigurable Architecture and Compiler, IEEE Computer, Vol. 33, n. 4, 2000.

[HWA 89] HWANG, K.; TSENG, P & KIM, D.- An Orthogonal Multiprocessor for Parallel Scientific Computations, *IEEE Trans. On Computers,* Vol.38, N.1, 1989.

[HWA 93] HWANG, K. - *Advanced Computer Architecture – Parallelism, Scalability, Programmability*. McGrawHill, 1993.

[MAT 98] MATSUMOTO, G. - The Brain and Brainway Computer – *Proceedings of The Fifth International Conference on Neural Information Processing*, Kitakysushu, Japan, 1998.

[MPI 95] MPI FORUM - MPI: *A Message-Passing Interface Standard* – University of Tennessee, Knoxville, Tennessee, June 1995.

[SAI 98] SAITO, J.H. & FUKUSHIMA, K. - Modular Structure of Neocognitron to Pattern Recognition, *Proc. ICONIP'98, Fifth Int. Conf. On Neural Information Processing*, Kitakyushu, Japan, 1998.

[SAI 99] SAITO, J.H. - A Vector Orthogonal Multiprocessor NEOMP and its Use in Neural Network Mapping, *Proceedings of the SBAC-PAD'99 – 11th Symposium on Computer Architecture and High Performance Computing*, Natal, RN, Brazil, 1999.