Proceedings of SPIE 1997, Orlando, Florida

Application of Highly Parallel Computing Hardware to Pattern Recognition Problems

Kevin Priddy, Karl Mathia, Tim Robinson and Robert Pap
Accurate Automation Corporation
7001 Shallowford Road
Chattanooga, TN 37421

email address: marketing@accurate-automation.com

Abstract

Neural networks are well known for their ability to perform pattern recognition tasks. This paper discusses the use of parallel neural network hardware for performing pattern recognition tasks. We address the need for neural network hardware and how it can dramatically improve system performance both in training and in actual application. The use of specialized parallel processing hardware is discussed as well as alternative hardware and software approaches. Finally we give some comparisons between multiprocessor computer architecture, Pentium class microcomputers and custom hardware.

Key Words: Neural Networks, Parallel Computing, Pattern Recognition, MIMD, SIMD

1. INTRODUCTION

Neural networks have been shown to be able to solve a wide variety of tasks[2,4,5] which require high speed and accurate solutions. This paper provides insight into using neural network hardware[3] to solve pattern recognition problems. Accurate Automation Corporation has developed a Multiple Input Multiple Data (MIMD)[6,7] parallel computing architecture which is specifically designed to execute neural network structures. The Neural Network Processor (NNP®) is depicted in Figure 1. A single NNP® provides up to 16K neurons and 32K connections. The NNP® is designed to operate at approximately 135 million connections per second (CPS) for a single processor with up to ten parallel processors, for a total of 1.35 billion CPS with 16K neurons and 320K connections. The flexibility of the architecture allows the neural network designer to implement a variety of neural network paradigms as well as increase throughput by adding additional processors. The choice of which neural network hardware to use for a particular problem is highly dependent upon the problem which is being solved. A fairly complete listing of available neural network hardware is given by Lindsey[1]. In this paper we look at using the NNP® for a signal processing task and at ways of improving performance through analysis of the problem at hand.

2. NNP® DESCRIPTION

The underlying philosophy of the NNP® design has been to achieve maximum computational efficiency in both a single processor and multiprocessor environment by optimizing the design to compute neuron values - and nothing but neuron values. Indeed, this is ideally suited to a neural network applications and stands in stark contrast to previously proposed processors which are typically based on classical single

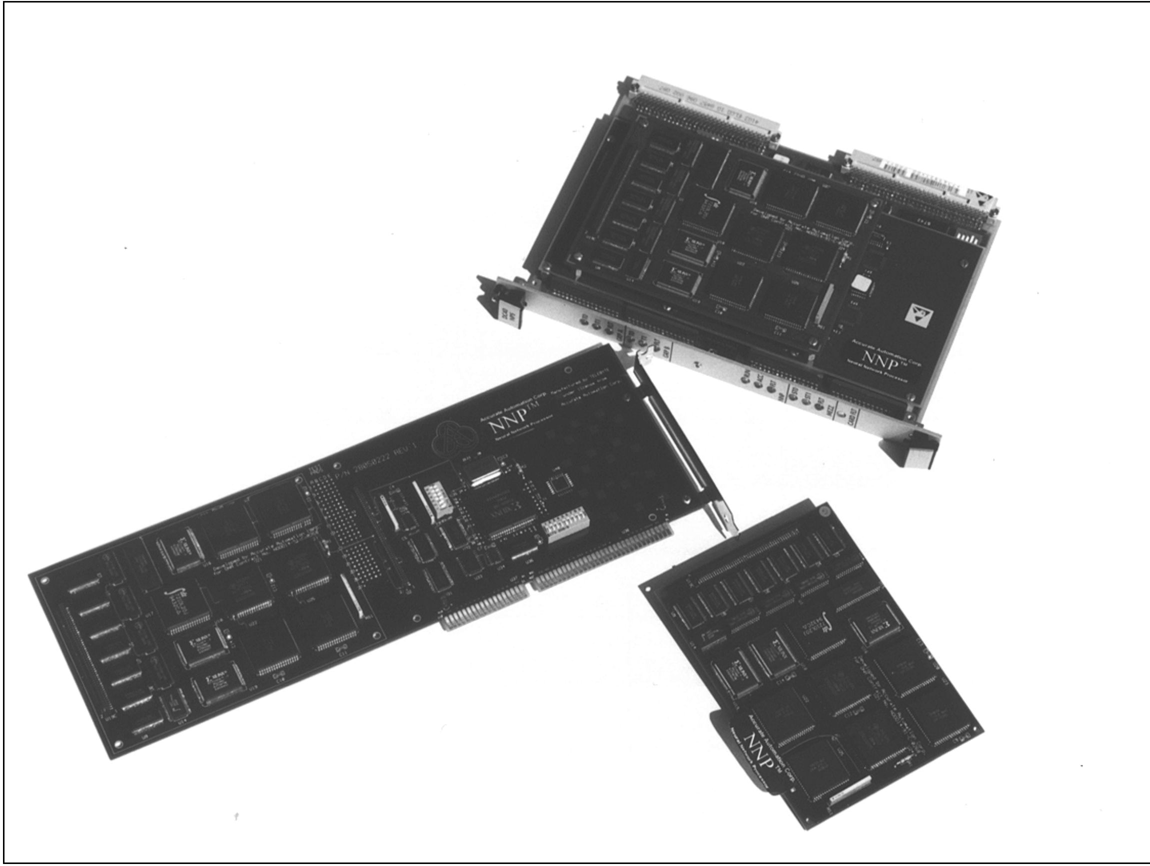

Figure 1: AAC MIMD Neural Network Hardware: NNP® Module (bottom); ISA PC Card (left); VME NNP® Board (top);

instruction multiple data (SIMD) matrix / vector multiplication architectures. Rather, our MIMD design fully exploits the intrinsic characteristics of the neural network topology (sparse, local, random). Moreover, by using an MIMD parallel processing architecture one can update multiple neurons in parallel with efficiency approaching 100% as the size of the neural network increases.[3,7,8] To achieve the desired efficiency we have adopted a design which:

- uses an instruction set which is optimized for neural network processing, allowing one to compute a neuron activation without arranging the weight matrix into linear arrays and/or inserting “artificial zero weighted connections”,
- uses an MIMD parallel processing architecture to permit neurons with totally different input topologies to be updated simultaneously without loss of efficiency, and
- uses dual neuron memories to virtually eliminate memory contention and maintain absolute memory coherence.

This architecture allows us to implement a relatively simple single processor NNP® module and then string together multiple NNP® modules along a dedicated Interprocessor Bus with computational power (and cost) increasing “almost” linearly with the number of modules.[3,7,8]

The AAC MIMD Neural Network Processor:

- is designed to implement multiple interconnected neural networks of differing architecture simultaneously using 16 bit twos complement binary fixed point arithmetic, with up to 16k total neurons and 32k connection weights per module,
- is capable of running at 135,000,000 connections (byte wide multiply / additions) per second per module for a billion plus CPS with ten parallel NNPs®,
- supports two I/O buses, an Interprocessor Bus which can also be used for on-line I/O in parallel with the computational process, and a Memory I/O Bus through which the various processor memories may be mapped into the memory space of a host processor or DSP for downloading programs, connection weights, etc., and
- each processor in an NNP® array is controlled by a separate program written in a "RISC-like" instruction set supported by an Assembler, a "C" subroutine library to facilitate communications between the NNP® and the host processor, and the Accurate Automation Neural Network Tools.

2.1 Functional Overview: Single Processor Operation

A simplified schematic of a single NNP® is shown in Figure 2. The key functional components are the 32k by 16 bit Program and Weight Memories (PM and WM), the 64k by 16 bit Function Memory (FM), the 16k by 16 bit dual ported Neuron Memory (NM), and the Multiplier/Accumulator (MAC). During normal processor operation the PM and WM are configured as a single 32k by 32 bit Program/Weight Memory or PWM. Tthe NM is linearly subdivided into a 16k neuron memory, and an 16k buffer memory, while the FM is linearly subdivided into four 16k function memories used to store neuron transfer functions.

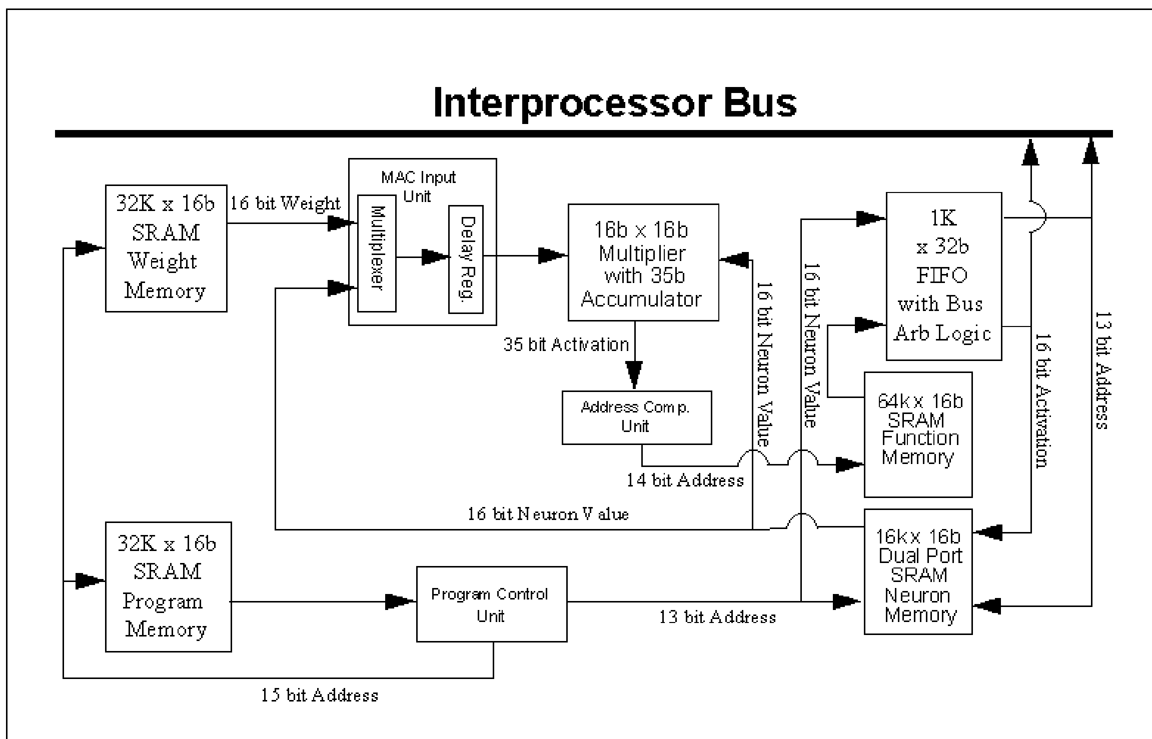

Figure 2. AAC Neural Network Processor (NNP®) Architecture

The instruction set consists of: four arithmetic instructions; three multiply/accumulate instructions, used to compute neuron activations; an instruction to pass a neuron activation (stored in the accumulator) through a transfer function and store the resultant neuron value; and five control instructions. The most commonly used instructions are the multiply/accumulate instructions which multiply a neuron value by a connection weight (or the neuron value fetched by the previous instruction). Since a neural network connection weight is fixed while a neuron value is a variable, the weight value is included in the multiple/accumulate instructions in immediate form while the neuron value is indirectly addressed by including its address in the NM's neuron memory in the instruction. The connection weight is passed directly to the MAC (with an appropriate delay) while the neuron value is fetched from the NM and then passed to the MAC. The MAC then computes the product of the weight and the neuron value and adds the result to the accumulator (with or without first clearing the accumulator) to obtain the desired neuron activation. The multiply/accumulate instructions are pipelined over ten clock cycles with a new instruction executed each clock cycle. As such, once the pipeline is filled, an n term neuron activation may be computed in n clock cycles. The last arithmetic instruction is used to pass a neuron activation stored in the accumulator through a transfer function and store the result in the NM's buffer memory. The instruction includes the address where the resultant neuron value is to be stored and the number of the desired transfer function stored in the FM and is pipelined over seven clock cycles.

In addition to the four arithmetic instructions the NNP® has five control instructions for looping, terminating a program, etc. Input/output (I/O) functions are handled externally via an I/O processor or off-line via the Memory Bus (MB).

2.2 Functional Overview: Multi-Processor Operation

The key to running the MIMD NNP® in a multiprocessor mode are the FIFOs, see Figure 2, and the buffer memories of the various processors. First one must initialize all processors with identical neuron memories and buffer memories. Then the processors run autonomously each with their own programming and weight set, reading neuron values from their local copy of the neuron memory, which is guaranteed to remain coherent with the other processors because it operates in a read-only mode. When any processor updates a neuron value it is loaded into its FIFO which requests access to the Interprocessor Bus, see Figure 2, and loads the new neuron value into all of the buffer memories of the parallel processors simultaneously via their second port. As such, the buffer memories remain coherent at all times. Although a single "broadcast bus" is used for interprocessor communication, because a processor needs to access the bus an average of only once every f cycles (where f is the average fan-in of the neural network) bus contention will not be a major factor, as long as the number of processors in a system is less than f/4,[7] which is consistent with realistic system designs and has been verified experimentally by Mathia[3].

The only point in the process where the parallel NNP®'s have to wait for each other is when an iteration has been completed or a neural network layer has been updated, and an instruction to interchange the neuron and buffer memories is executed. In this case, to guarantee coherency; i.e., that all processors operate on the same data; each processor must wait until all processors have executed an interchange instruction and have emptied their FIFO, at which point the buffer and neuron memories in all processors are interchanged simultaneously.

3. ANALYSIS OF NNP® IMPLEMENTATION OF A SIGNAL PROCESSING ALGORITHM

The NNP® has been successfully applied to several fault detection problems. One algorithm developed for determining a fault condition was based upon analysis of pressure transducer data. The best discriminant for determining when a fault condition had occurred was a sequence of values which were the sum of the magnitude of the autocorrelation coefficients for the data window used to evaluate the signal of interest. In our testing the sequence was 16 values long and the window was 256 samples long. The mathematical expression for the modified autocorrelation algorithm used in the problem is given in Equation 1.

$$\text{Autocorrelation_Mag_Sum(n)} = \sum_{i=k}^{n+k} \left| \sum_{j=i}^{n+k} x_i x_j \right| \tag{1}$$

where

$n \equiv$ Sample number

$k \equiv$ Window Size

Once the autocorrelation magnitude sum (AMS) is calculated, it is stored for 16 calculations with these 16 features then fed into a neural network for evaluation of a fault condition. The algorithm has been used to train a backpropagation neural network[9] to predict the fault condition. This analysis calculates the number of machine cycles as well as the methodology for incorporating both the preprocessing and the neural network on the AAC neural network processor (NNP®). The algorithm can be broken down into three phases: 1) Initial autocorrelation sum; 2) neural network inputs; and 3) modifications for speed and simplicity.

3.1 Initial Autocorrelation Sum

The initial autocorrelation sum is calculated after the first "window size" number of samples have been recorded and stored. The autocorrelation is performed for only one-half of the actual number of available shifts because it is symmetrical. Suppose we have four samples in a buffer, (x_1, x_2, x_3, x_4). The sum of the magnitude of the autocorrelation coefficients described in Equation 1 is given by:

$$\left|x_1^2 + x_1x_2 + x_1x_3 + x_1x_4\right| + \left|x_2^2 + x_2x_3 + x_2x_4\right| + \left|x_3^2 + x_3x_4\right| + \left|x_4^2\right| \tag{2}$$

The total number of terms contained in Equation 2 is given by:

$$\text{Total number of product terms} = \sum_{i=1}^{\text{window size}} i = \frac{\text{window size(window size +1)}}{2} \tag{3}$$

However, the total number of nodes required to store the desired magnitudes for the autocorrelation is simply the “window size”. By careful analysis the total number of NNP® instructions required to preprocess the initial autocorrelation magnitude sum (AMS) is given by:

$$\begin{aligned}&\text{Total number of instructions} = \boldsymbol{k}(\boldsymbol{k}+2)+1\\&\text{where } \boldsymbol{k} \equiv \text{window size}\end{aligned} \tag{4}$$

With Equation 4, we can determine precisely the number of instructions required for any desired neural network architecture using the autocorrelation preprocessing. The next section will examine the requirements needed to implement the neural network and preprocessing network for the fault problem.

3.2 Hardware Implementation

The NNP® is designed to implement neural networks efficiently and quickly. The actual implementation is highly dependent upon the neural network designer for truly optimal speed and accuracy. To this end we will begin by showing the required implementation using a straightforward “brute force” approach followed by an approach which provides much faster processing. The window used for the autocorrelation sum had a length of $\boldsymbol{k} = 256$. The neural network input had a size of 16 autocorrelation sums. The “brute force” approach would be to compute each of the autocorrelation sums independently and then to splice them together to obtain the desired input to the neural network. Because the preprocessing of the autocorrelation sums takes the most instructions we will, for the present time, ignore the neural network calculations. Given the result of Equation 4, we can easily compute the total number of instructions to send an input to the neural network, as shown below in Equation 5.

$$\begin{aligned}&\text{Total number of instructions} = \boldsymbol{m}(\boldsymbol{k}(\boldsymbol{k}+2)+1)\\&\text{where}\\&\boldsymbol{k} \equiv \text{window size}\\&\boldsymbol{m} \equiv \text{number of inputs to neural network}\end{aligned} \tag{5}$$

Thus for $\boldsymbol{k} = 256$ and $\boldsymbol{m} = 16$ we obtain the following

$$\begin{aligned}&\text{Total number of instructions} = 16(256(256+2)+1)\\&\text{Total number of instructions} = 1,056,784\end{aligned} \tag{6}$$

The NNP® has a clock speed of 35 MHz with all instructions fully pipelined. Thus the time required to generate the 16 neural network inputs is given by

$$\text{Calculation Time} = \frac{\text{Total number of Instructions}}{\text{Clock Speed}}$$
$$\text{Calculation Time} = \frac{1,056,784}{35{,}000{,}000\ \text{Hz}}$$
$$\text{Calculation Time} = 30.193\ (\text{ms})$$
$$\text{Max Frequency} = 33.12\ \text{Hz} \tag{7}$$

While it is indeed impressive that over a million instructions are calculated so quickly, real-world pressure data is often sampled at a 28.8k samples/sec rate which is nearly three orders of magnitude faster than is achieved using the brute force method. How can we add new terms and subtract old ones without recomputing the entire autocorrelation each time new data is introduced? The next section describes how using knowledge about the common information within each of the autocorrelation terms to dramatically reduce the time required to introduce new inputs to the neural network classifier.

3.3 Reducing the Autocorrelation Calculation Time

The initial autocorrelation time cannot be reduced because we must first read several samples of data and then perform the autocorrelation. However, this can be considered an initial delay constant which will never have to be repeated. To see how this can be done suppose we have four samples in a buffer, (x_1, x_2, x_3, x_4) . The sum of the magnitude of the autocorrelation coefficients described in Equation 1 is given by:

$$|x_1^2 + x_1x_2 + x_1x_3 + x_1x_4| + |x_2^2 + x_2x_3 + x_2x_4| + |x_3^2 + x_3x_4| + |x_4^2| \tag{8}$$

Now we add a new data point (x_5) while removing (x_1) creating the set (x_2, x_3, x_4, x_5) . The sum of the magnitude of the autocorrelation coefficients described in Equation 1 is given by:

$$|x_2^2 + x_2x_3 + x_2x_4 + x_2x_5| + |x_3^2 + x_3x_4 + x_3x_5| + |x_4^2 + x_4x_5| + |x_5^2| \tag{9}$$

By observation we can see that many of the products found in Equation 8 are in common with those found in Equation 9. We now generalize the terms of Equation 8 for incorporation into Equation 9:

$$\begin{aligned} a_1 &\equiv x_1^2 + x_1x_2 + x_1x_3 + x_1x_4 \\ a_2 &\equiv x_2^2 + x_2x_3 + x_2x_4 \\ a_3 &\equiv x_3^2 + x_3x_4 \\ a_4 &\equiv x_4^2 \end{aligned} \tag{10}$$

Now we can express Equation 9 using the substitutions of Equation 10.

$$|a_2 + x_2 x_5| + |a_3 + x_3 x_5| + |a_4 + x_4 x_5| + x_5^2 \qquad (11)$$

Now a little intuition into the problem before generalizing for any arbitrary window size. The data is fed to the NNP® in a serial manner. Observing Equation 9 we see that the (x_1) product terms have been dropped while the (x_5) product terms have been added. What is really happening is that the window has a finite length, in this example $k = 4$, which causes the (t - k)th term to be dropped into the “bit bucket” while adding the latest data sample (*t*). For convenience we will make the following definitions for the terms given in Equation 8:

$$\begin{aligned} x_{t-3} &\equiv x_1 \\ x_{t-2} &\equiv x_2 \\ x_{t-1} &\equiv x_3 \\ x_t &\equiv x_4 \end{aligned} \qquad (12)$$

Now we generalize for the case of Equation 8 using the definitions of Equation 12.

$$|x_{t-3}^2 + x_{t-3}x_{t-2} + x_{t-3}x_{t-1} + x_{t-3}x_t| + |x_{t-2}^2 + x_{t-2}x_{t-1} + x_{t-2}x_t| + |x_{t-1}^2 + x_{t-1}x_t| + x_t^2 \qquad (13)$$

Using similar substitutions for the coefficients yields the following equivalent expression for Equation 10:

$$\begin{aligned} output &= |a_1| + |a_2| + |a_3| + a_4 \\ output &= |b_1 + x_{t-3}x_t| + |b_2 + x_{t-2}x_t| + |b_3 + x_{t-1}x_t| + a_4 \\ &\text{where} \\ b_1 &\equiv x_{t-3}^2 + x_{t-3}x_{t-2} + x_{t-3}x_{t-1} = a_2\big|_{t=t-1} \\ b_2 &\equiv x_{t-2}^2 + x_{t-2}x_{t-1} = a_3\big|_{t=t-1} \\ b_3 &\equiv x_{t-1}^2 = a_4\big|_{t=t-1} \end{aligned} \qquad (14)$$

From Equation 14 we can conceptualize the neural network implementation as well as the simplicity of the solution. Even if we were implementing the algorithm on a traditional Von Neuman computer using a high order language, a tremendous cost savings in the computation of the autocorrelation would be realized. The implementation consists of the following steps:

- Compute initial autocorrelation.
- Store Coefficients (a's).
- Get next data sample.
- Compute and Store new (a's) using substitutions of Equation 14.
- Compute new autocorrelation.
- Get next data sample and so on.

When the data are exhausted then we would stop the program. Of course in dealing with the real world we expect that the data would be continuous. Hence the routine would be iterative. Because the NNP® is always under the control of the host processor we could reinitialize anytime the operator requests. To illustrate the principles just presented we will examine a simple AMS network with a data window of length *k=4,* and with data set x={1 1 1 1 0 0 0 0 0}.

The initial values for the *a* coefficients are $a_1 = a_2 = a_3 = a_4 = 0$. Also we must initially set all of the windowed data values to zero. This then gives us a data window which is {0 0 0 0} before any data is added. With this initialization complete we can then introduce the first data sample x_t.
The equations which now govern the calculation of the a coefficients and subsequently the AMS are given below:

$$\begin{aligned} a_{4,t} &= (buffer_t)^2 \\ a_{3,t} &= a_{4,t-1} + buffer_t buffer_{t-3} \\ a_{2,t} &= a_{3,t-1} + buffer_t buffer_{t-2} \\ a_{1,t} &= a_{2,t-1} + buffer_t buffer_{t-1} \\ output_t &= |a_{1,t}| + |a_{2,t}| + |a_{3,t}| + a_{4,t} \end{aligned} \quad (15)$$

3.4 Implementation of Modified AMS Algorithm on NNP®

A diagram of the neural network which implements the AMS algorithm is shown in Figure 3. At the time when the first data value is fed into the buffer the resultant buffer will be {1 0 0 0}. This results in an autocorrelation of 1 0 0 0, and AMS of 1; the next data sample results in a buffer of {1 1 0 0}with a resultant autocorrelation of {2 1 0 0}, and AMS of 3; the next sample results in a buffer of {1 1 1 0}with a resultant autocorrelation of {3 2 1 0}, and AMS of 6; the next data sample results in a buffer of {1 1 1 1}with a resultant autocorrelation of {4 3 2 1}, and AMS of 10 and so on as depicted in Figure 4.

The proposed AMS algorithm is considerably faster than the brute force method and can be implemented for real-time operation on the NNP® for the currently used sensor sampling rates. Of course this analysis has assumed only one sensor and a single NNP®. Additional sensors and NNP®s will affect the results.

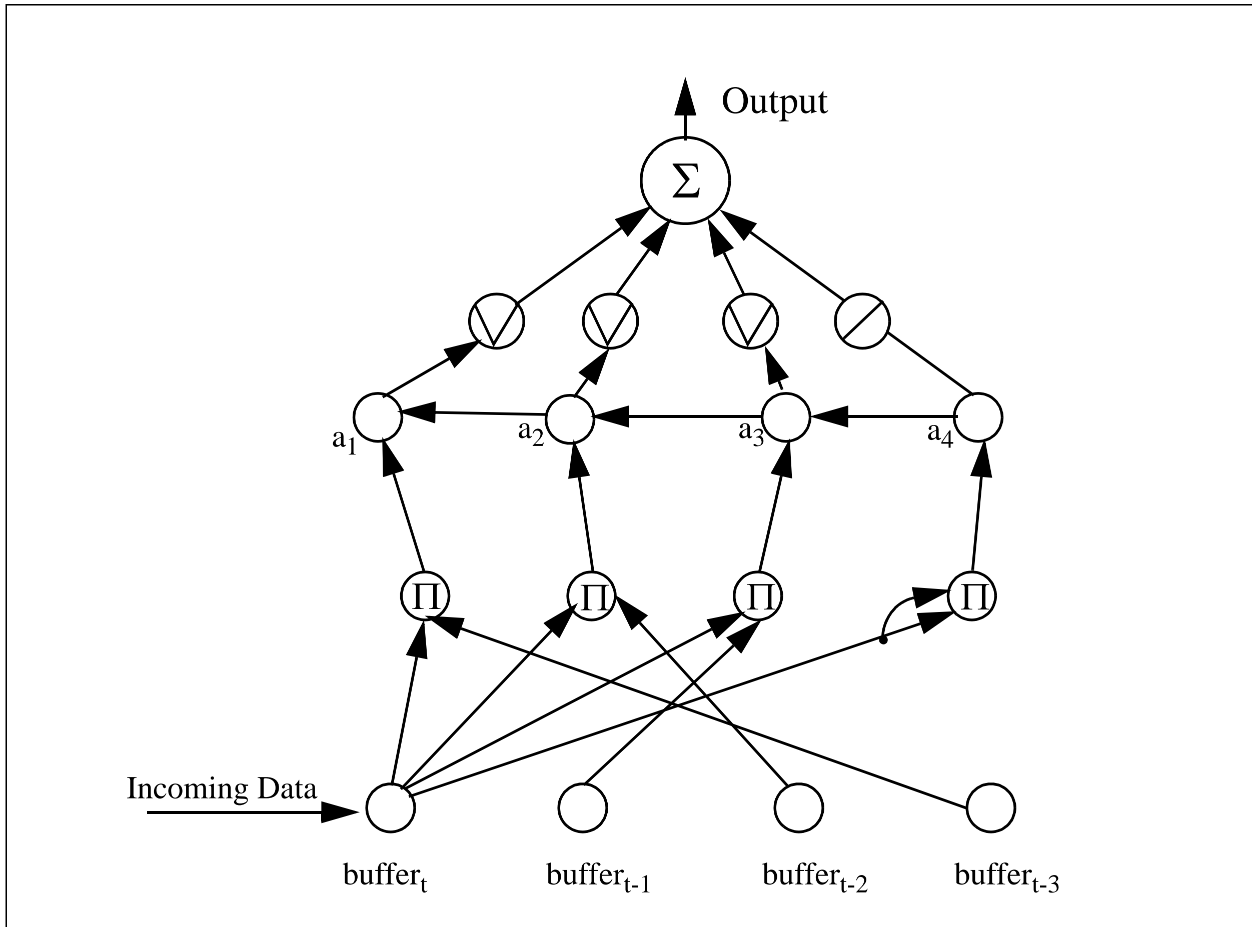

Figure 3: Neural Network Implementation of AMS Algorithm For Example Problem.

The NNP® code to implement the updated autocorrelation, with buffers and the *a* coefficients set to zero can be computed on the fly. Careful examination shows that there are exactly $6(\boldsymbol{k}) + 4$ calculations for each additional autocorrelation term after the initial autocorrelation is computed. If we desire *m* autocorrelations to be used as the neural network input, the total number of instructions required using the revised methodology is given by

$$\text{Total Instructions (Revised Method)} = \boldsymbol{k}(\boldsymbol{k} + 2) + 1 + (\boldsymbol{m} - 1)(6\boldsymbol{k} + 4) \tag{16}$$

which for $\boldsymbol{k} = 256$ and $\boldsymbol{m} = 16$ results in 89,149 total instructions. The NNP® has a clock speed of 35 MHz with all instructions fully pipelined. The time required to generate the 16 neural network inputs using the revised method is 2.55 ms which is considerably faster than the "brute force" method. While this is considerably better, it is still not fast enough for real-time operation and throughput. The long pole in the tent is the initial autocorrelation calculation. After it is calculated, only $6\boldsymbol{k} + 4$ new calculations need to be made for each new autocorrelation. Thus if we view the initial autocorrelation calculation as a delay, "symbol latency", between the input and the output of the NNP, with each subsequent additional term added as we go along, the actual throughput will be 22.73k samples/sec. This is close to the data rate of the sensors and can be substantially improved by using more NNP®s or smaller data windows.

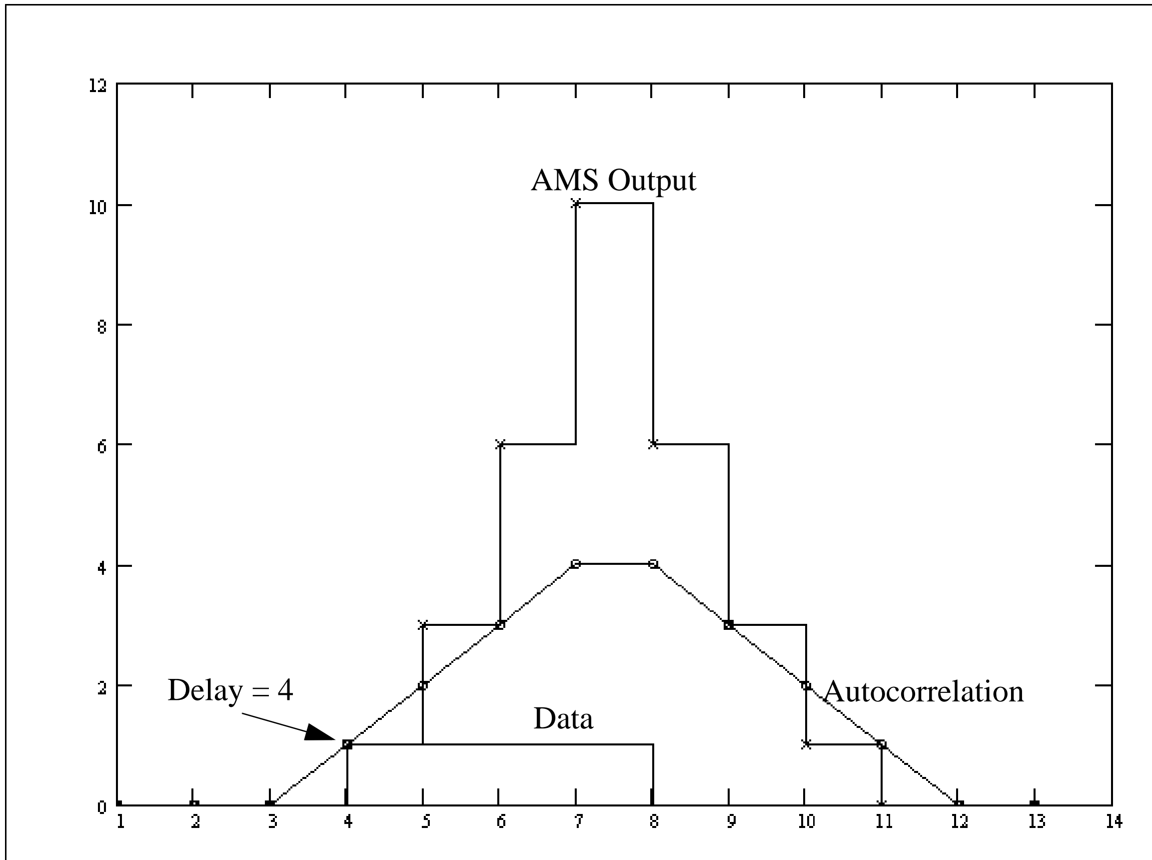

Figure 4: Data, Autocorrelation and AMS Output For Network of Figure 3.

4. IS CUSTOM HARDWARE BETTER?

As can be seen in the previous example, we have the capability of implementing neural network architectures in hardware. The big issue is whether or not custom hardware gains you anything compared to software algorithms. The first and foremost reason for using custom hardware is speed. If your application runs fast enough using a software neural network implementation then that is the route to go. Of course, if a really large problem comes your way then hardware is the answer. Often neural network hardware is often nothing more than a glorified matrix/vector multiplier. This type of hardware can speed up simple feedforward architectures but is hopelessly bogged down when more sophisticated neural network implementations are required. Many benchmarks have been suggested, but the best benchmark of all is the problem you are working on. For example, we compared our NNP® to a UNIX workstation and a Pentium computer for a series of recurrent Hopfield networks with the results displayed in Table 1. The results given in Table 1 show clearly that a single NNP® is faster than the Pentium and the UNIX workstation and that multiple NNP®s perform even better. One of the reasons for the improvement is due to the reduced overhead needed to operate multiple NNP®s as parallel processors. As can be seen, the UNIX machine performed well below its theoretical best due to software overhead.

CONCLUSIONS

In this paper we have presented two examples of how parallel neural network processors may be used to help solve the speed requirements of pattern recognition systems. We have presented the practical side of

implementing algorithms in a real-time system. The demands of real-time performance must still be met by a combination of high-speed hardware and a sound understanding of the problem at hand and the technology available to solve the problem.

Table 1: Performance in Million Connection-Bytes-Per-Second (MCBS) for a Continuous Hopfield Network.

System	Theoretical peak performance	Realizable sustainable peak performance	% of theoretical peak performance
1 NNP®	140	134.6 (128 neurons)	96.1
2 NNP®s	280	268.4 (128 neurons)	95.8
4 NNP®s	560	547.8 (256 neurons)	97.8
Pentium 90	(unknown)	28.5 (256 neurons)	(unknown)
SGI Onyx (1 Processor)	1600[a]	131.1 (256 neurons)	8.2
SGI Onyx (2 Processors)	3200[a]	184.4 (256 neurons)	5.8

a. Based on SGI's measure of 100 MFLOPS (1 processor), and 200 MFLOPS (2 processors).

ACKNOWLEDGMENTS

We wish to acknowledge Dr. Joel Davis and the Office of Naval Research for their support in the development of the AAC Neural Network Processor.

REFERENCES

1. Lindley, C. and Lindblad, T., "Review of Hardware Neural Networks: A User's Perspective," World Wide Web site: http://msia02.msi.se/~lindsey/elba2html/elba2html.html

2. Mathia, K., and Priddy, K., "Real-Time Geometrical Approximation of Flexible Structures Using Neural Networks," *Proc. IEEE Int. Conf. Syst., Man and Cyb.*, Vancouver, B.C., Vol. 3, pp. 2099-2102.

3. Mathia, K., Clark, J., Colbert, B., and Saeks, R., "Benchmarking an MIMD Neural Network Processor," Proc. *World Congress on Neural Networks 96,* San Diego, CA, pp. 1321-1326.

4. Murray, A., and Tarassenko, L., *Analogue Neural VLSI*, Chapman and Hall, London, 1994.

5. Rogers, G., Solka, J., Ellis, J., and Szu, H., (1992). "A Neurocomputing Benchmark for Digital Computers", Proc. SIMTEC-WNN '92, Clear Lake, TT., pp. 425-430.

6. Rogers, S., Colombi, J., Martin, C., Gainey, J., Fielding, K., Burns, T., Ruck, D., Kabriskey, M., and Oxley, M., "Neural Networks For Automatic Target Recognition," Neural Networks, Vol. 8, No. 7/8, pp. 1153-1184, 1995.

7. Saeks, R., Priddy, K., Pap, R. and Stowell, S., "On the Architecture of an MIMD Neural Network Processor", *Proc. of the SPIE Conf. on Neural Networks*, Orlando.

8. Saeks, R., Priddy, K., Pap, R., Schneider, K. and Stowell, S., "On the Design of an MIMD Neural Network Processor", Proc. of the World Conf. on Neural Networks, San Diego.

9. Werbos, P., *Beyond Regression: New Tools For Prediction and Analysis in the Behavioral Sciences.*, Harvard PhD Thesis, Committee on Applied Mathematics, 1974.