FPGA Based Implementation of a Hopfield Neural Network for Solving Constraint Satisfaction Problems

David Abramson † Kate Smith ‡ Paul Logothetis † David Duke †

† School of Computer Science and Software Engineering,
Monash University, Clayton, VIC 3168
E-mail: davida@dgs.monash.edu.au
Phone: +67 -7 (03) 9905 1183
Fax: +67 -7 (03) 9905 3574
WWW: http://www.dgs.monash.edu.au/~davida.

‡School of Business Systems,
Monash University, Clayton, VIC 3168
E-mail: kate.smith@fcit.monash.edu.au

Abstract

This paper discusses the implementation of Hopfield neural networks for solving constraint satisfaction problems using Field Programmable Gate Arrays (FPGAs). It discusses techniques for formulating such problems as discrete neural networks, and then it describes the N-Queen problem using this formulation. A prototype implementation of the a number of different N-Queen problems is described and results are presented that illustrate that a speedup of up to 3 orders of magnitude is possible using current FPGAs devices

1. Introduction

Many practical optimisation problems from business and industry can be formulated as standard mathematical programming problems using binary decision variables. Solution of these problems requires the use of heuristics or approximate algorithms due to the NP-hard nature of their complexity. Neural networks were proposed to solve such problems in 1985 [1], but the field has been plagued with problems of poor solution quality and inability to guarantee feasible final solutions [15]. These initial problems have now been overcome. Techniques have been proposed to help the Hopfield neural network escape from local minima of its energy function, and suitable construction of that energy function has been shown to guarantee the feasibility of solutions [16]. Using these improvements, neural network results have been obtained which compete effectively (and even outperform) other popular heuristics such as simulated annealing.

While most of the literature has focused on using Hopfield networks to solve the famous Travelling Salesman Problem, a range of practical problems have also been solved with neural networks [16][14]. The solutions to these problems were obtained by *simulating* the behaviour of the Hopfield neural network (designed to be implemented in electrical hardware) on a conventional computer. However, while the algorithms generate good solutions, the computation times are extremely slow. *If neural networks are to be applied routinely to practical problems, then the execution time must be reduced.*

There are a number of ways of accelerating the execution of the network algorithms, ranging from the use of high end parallel supercomputers, through to hardware implementations of the networks themselves using custom computing machines (CCMs). CCMs are attractive, because they have the potential to provide cheap high speed platforms for neural network based algorithms. However, until recently the cost of producing specific hardware has been high and the process error prone.

Recently, the advent of high density field programmable gate arrays (FPGAs), in combination with new synthesis tools, have made it relatively easy to produce programmable custom machines without building specific hardware. FPGA based CCMs can provide high performance on certain problems, demonstrating speedups of orders of magnitude over conventional machines [1][2][5]. There is great potential to apply these techniques to neural network based algorithms, however, research must be conducted to determine the appropriate methods.

This paper aims to demonstrate the potential of a custom computer based on FPGA technology for solving a classical constraint satisfaction problem: the N-Queen problem. The Hopfield neural network will be briefly described, and we will show how the N-Queen problem can be mapped onto the architecture. The issues involved in designing the custom computer will be discussed. Finally results will be presented which compare the computation times for the custom computer against the simulation of the Hopfield network run on a high end workstation. In this way, the speed-up can be determined.

2. Hopfield Neural Networks

Hopfield neural networks [8][9] are a biologically inspired mathematical tool which can be used to solve difficult optimisation problems. Their advantage over more traditional optimisation techniques lies in their potential for rapid computational power when implemented in electronic hardware, and the inherent parallelism of the network.

There are two types of Hopfield networks, discrete and continuous models, which permit different values for neuron states. Biological modelling of the human brain is attempted by utilising a fully inter-connected system of N neurons. Neuron i has internal state u_i and output level v_i (which can be either binary valued in the discrete model or real valued bounded by 0 and 1 in the continuous model). The internal state u_i incorporates a bias current (or negative threshold) denoted by I_i, and the weighted sums of outputs from all other neurons. The weights, which determine the strength of the connections from neuron i to j, are given by T_{ij} . The relationship between the internal state of a neuron and its output level is detemined by an activation function $g(u_i)$. The nature of this activation function depends on whether the Hopfield network is discrete or continuous. Commonly,

$$v_i = g(u_i) = \frac{1}{2}(1+\tanh(\frac{u_i}{\tau})) \qquad (1)$$

is used for the continuous model, where τ is a parameter used to control the slope (or gain) of the activation function. For discrete Hopfield networks, the activation function is usually a discrete threshold function:

$$v_i = g(u_i) = \begin{cases} 1 & if\, u_i > 0 \\ 0 & if\, u_i \le 0. \end{cases} \qquad (2)$$

The neurons update themselves (either sequentially or in parallel) according to the following rule:

$$\begin{aligned} u_i(t+1) &= u_i(t) + \Delta t(\sum_{j=1}^{N} T_{ij} v_j + I_i) \\ v_i(t+1) &= g(u_i) \end{aligned} \qquad (3)$$

and in doing so, the network of neurons will converge to a local minimum of the following energy function over time:

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} T_{ij} v_i v_j - \sum_{i=1}^{N} I_i v_i \qquad (4)$$

provided the weights are symmetric $T_{ij} = T_{ji}$.

If neurons are updated in parallel (or synchronously) then the possibility of convergence to a two-cycle exists. Both of the network states which comprise the two-cycle will be local minima of the energy function however. The discrete model has an advantage over the continuous model in terms of the number of updates required to converge to a local minimum. For this reason, and others related to hardware constraints which will be discussed later in this paper, we have chosen to use a discrete Hopfield network for solving the N-Queen problem. We have also chosen to update the neurons in a parallel operation rather than sequentially since it is our ultimate intention to solve large scale problems as rapidly as possible. Parallel implementation involves calculating all of the u updates then all of the v updates, as opposed to the sequential update which calculate the u and v update for each neuron one at a time.

Hopfield and Tank [7] showed that if a 0-1 optimisation problem can be expressed in terms of an energy function of the form given by (4), then a Hopfield network can be used to find locally optimal solutions of the energy function. This may translate to local minimum solutions of the optimisation problem. Typically, the network energy function is made equivalent to the objective function of the optimisation problem, while the constraints of the problem are included in the energy function as penalty terms. The network parameters can then be inferred by comparison with the standard energy function given by (4). The weights of the network, T_{ij}, are then the coefficients of the quadratic term, V_iV_j, and the external bias currents, I_i, for each neuron i, are the coefficients of the linear terms V_i in the chosen energy function. The network can be initialised by setting the activity level V_i of each neuron to an unbiased state. Updating the network according to equation (3) will then allow a minimum energy state to be attained, since the energy level never increases during state transitions. The derivation of the weights and external biases for the N-Queen problem are provided in the following section.

3. The N-Queen problem

The N-Queen problem is a classical constraint satisfaction problem, whose goal is to place N Queens on an NxN chess board in mutually non-attacking positions. Since a Queen can only attack horizontally, vertically and diagonally, the constraints can be stated as:

- there can be only one Queen in each row
- there can be only one Queen in each column
- there can be only one Queen in each diagonal (ascending and descending).

Suppose we define a binary decision variable:

$$X_{ij} = \begin{cases} 1 & \text{if a Queen is positioned in row i and} \\ & \text{column j of the chessboard} \\ 0 & \text{otherwise} \end{cases}$$

There are several ways of combining all of these constraints in an objective function as penalty terms, and one such function is:

$$\begin{aligned} f(\mathbf{X}) = {} & \frac{A}{2}\sum_{i=1}^{N}(\sum_{j=1}^{N} X_{ij} - 1)^2 + \frac{B}{2}\sum_{j=1}^{N}(\sum_{i=1}^{N} X_{ij} - 1)^2 \\ & + \frac{C}{2}(\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{\substack{p\neq 0 \\ 1\le i-p\le N \\ 1\le j-p\le N}} X_{ij}X_{i-p,j-p} \qquad (5) \\ & + \sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{\substack{p\neq 0 \\ 1\le i-p\le N \\ 1\le j-p\le N}} X_{ij}X_{i-p,j+p}) \end{aligned}$$

The first two terms ensure that the rows and columns sum to one, while the final term counts the cost of Queens on each diagonal. The value of this function is exactly zero for a non-attacking solution, and will be greater if some of the constraints are not satisfied. The values of A, B and C are selected to balance the relative importance of each constraint, and we have fixed these to unity.

Expanding and rearranging this objective function so that it is expressed as the sum of a linear component and a quadratic components yields:

$$\begin{aligned} f(\mathbf{X}) = {} & \frac{A}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N}\sum_{l=1}^{N}\delta_{ik}X_{ij}X_{kl} - A\sum_{i=1}^{N}\sum_{j=1}^{N}X_{ij} \\ & + \frac{B}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N}\sum_{l=1}^{N}\delta_{jl}X_{ij}X_{kl} \\ & - B\sum_{i=1}^{N}\sum_{j=1}^{N}X_{ij} + \frac{N}{2}(A+B) \\ & + \frac{C}{2}(\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N}\sum_{l=1}^{N}X_{ij}X_{kl}(1-\delta_{ik})\cdot(\delta_{i+j,k+l}+\delta_{i-j,k-l})) \\ = {} & -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N}\sum_{l=1}^{N}\begin{matrix}(-A\delta_{ik} - B\delta_{jl} - C(1-\delta_{ik})\cdot \\ (\delta_{i+j,k+l}+\delta_{i-j,k-l}))X_{ij}X_{kl}\end{matrix} \\ & - \sum_{i=1}^{N}\sum_{j=1}^{N}(A+B)X_{ij} + \text{constants.} \end{aligned} \qquad (6)$$

where δ_{ij} is the Kronecker-delta symbol equivalent to unity only if i=j (and is zero otherwise).

Comparing this expansion to the standard Hopfield network energy function (4), and noting the double subscript for this N Queen formulation, the weights and external biases can be read off as:

$$\begin{aligned} T_{ij,kl} = {} & -A\delta_{ik} - B\delta_{jl} - \\ & C(1-\delta_{ik})(\delta_{i+j,k+l}+\delta_{i-j,k-l}) \\ & \text{and} \\ I_{ij} = {} & A + B. \end{aligned} \qquad (7)$$

4. Implementation using FPGAs

4.1 Architecture

There have been many architectures proposed for the implementation of neural networks over the years, including both digital [10][11][6][3][4] and analogue circuits [12][13][17]. Most of these have concentrated on a hardware implementation of the neuron evaluation function, which consists of computing a number of matrix-vector products. Thus, these systems involve the parallel and pipelined executions of a number of multiply operations together with a reduction sum operator. A small amount of work has been focused on the training aspects of networks, which can be extremely time consuming.

The work described in this paper differs from general neural networks in three important ways. First, the weights are small and can be represented using small integers. This means that the hardware responsible for the accumulation can be optimised for small integer values. This not only reduces the size of the arithmetic units, but also reduces the carry propagation delays. Second, the neuron values are restricted to 0 or 1, rather than general fixed or floating point numbers. Consequently, the vector product becomes a set of conditional additions without the need to perform any multiplication operations. Multiplier units normally consume large amounts of logic, thus the savings here are dramatic. Third, because we are implementing a Hopfield network, the interconnection between neurons is fixed and dictated by the nature of the constraints in the problem. This means that it can be wired into the implementation, which is particularly relevant for FPGA based machines. This not only reduces the amount of interconnection hardware required, but also reduces the time spent accumulating the input values to a neuron.

The architecture chosen for this work consists of a set of neurons, as described by the VHDL code in Figure 1, which are then interconnected when the weights are non-zero as shown in Figure 2. Thus, neurons which have zero weights between them are not connected, limiting the number of inputs to a neuron to O(n) instead of O(N^2), where N is the number of neurons in the system. Whilst the code for each neuron is identical, each one is specified with a different `Weight_array`, and thus the hardware generated for each will be slightly different.

```
entity neuron_unit is
generic (  Ws : Weight_array:= (-1,-1,-1,0,-1,-2,-1,-1,-1,-1,-1,0,0,-1,0,-1); --weights
           Iij : in Weight_integer := 1;       -- constant external input to neuron
           Uij_init : in U_integer := 0);      -- startup value for U
port (     clk : in std_logic;              -- clock
           enable : in std_logic;           -- active high clock enable
           reset : in std_logic;            -- active high reset
           Xs : in X_array;                 -- n*n 1D vector of all neuron Xs
           Xij : out std_logic);            -- neuron output X, '1' if this neuron fires
end neuron_unit;

architecture   rtl of neuron_unit is
               signal deltaU : U_integer;
               signal Uij : U_integer;
begin
b1: block
begin
p1: process(Xs)                        -- compute the weighted sum of neuron's inputs
variable Wsum : U_integer;
   begin
         Wsum := Iij;                  -- Iij is external input
         rows_and_cols: for n in 1 to Cnum_neurons loop
         if (Xs(n) = '1') then
               Wsum := Wsum + Ws(n);   -- conditional summing of weights
         end if;
         end loop rows_and_cols;
         deltaU <= Wsum;               -- variable to signal for use outside process
end process p1;
p2: process(clk,reset)                 -- latch in new U value
   begin
         if (reset = '1') then    Uij <= Uij_init;     -- setup of initial pattern
         elsif (clk'event and clk = '1') then
         if (enable = '1') then   Uij <= Uij + deltaU; -- signal Uij is registered
        end if;
      end if;
    end process p2;

    Xij <= '1' when Uij > 0 else '0'; -- threshold for firing
end block b1;
end rtl;
```

Figure 1 - VHDL code for one neuron with default weights.

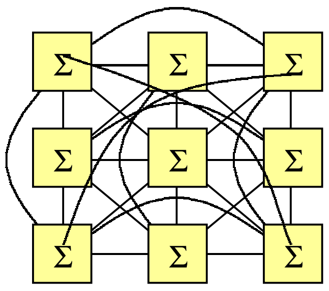


Figure 2 – Interconnection Architecture

4.2 Implementation

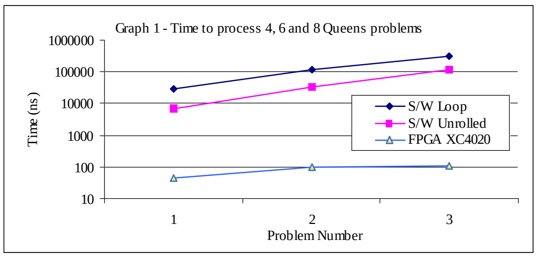
The consequence of the issues discussed in the previous section means that it is possible to implement the neural network using FPGA devices. FPGAs, like the Xilinx family of parts, consist of a number of Configurable Logic Blocks (CLBs) connected using a hierarchical, bus based, wiring scheme. The neurons and their interconnections are specified in VHDL, which is synthesised and simulated using Exemplar Logic's Galileo and V-System tools. Since all the weights are known at synthesis time, the synthesis tool's optimisation features are exploited to automatically remove any additions involving zero-valued weights. Xilinx's XACT Step software is used for FPGA mapping and routing. Our hardware platform, an Aptix AP4 reconfigurable logic board, routes the neuron outputs either to an array of LEDs or back to a host workstation for display. The AP4 board contains up to 16 XC4010 devices connected by 4 Aptix programmable switches (called FPICs). Using this configuration, it is possible to download a design consisting of up to 160,000 gates.

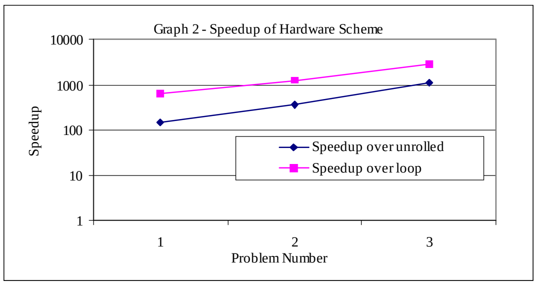
The use of VHDL has a number of advantages over conventional hardware design techniques like schematic capture. First, its high level syntax is not very different from conventional imperative programming languages, thus the design effort is not significantly different from writing a software simulation of a neural network. This is an important design consideration when considering the development cost of application specific hardware. Second, the VHDL software supports extensive optimisations, thus the performance of the underlying hardware is optimised for each neuron depending on the weights on its inputs. Finally, the VHDL compiler analyses the domain of the variables and generates optimal hardware without any further user interaction. For example, if the range of a variable is limited to the values from 0 to 3, then the data path used to transmit and manipulate the variable will be automatically constrained to 2 bits.

5. Performance Results

We have performed experiments using three different problems, 4 queens, 6 queens and 8 queens, named problems 1, 2 and 3. Two different software simulations were developed, both in C. The first uses a conventional loop structure which means that all inputs to a neuron are considered in the accumulation, even if the weight on the input is zero. However, the code is optimised for the 0-1 neuron output values, and thus it contains a conditional addition operation rather than a multiply. The main problem with this code is that it considers all possible inputs unlike the hardware architecture, and thus a comparison may seem unfair. Consequently, we developed another set of programs which unrolled the loops and removed all unnecessary addition operations. This code resembles the structure of the hardware solution, however, has the disadvantage that it does not use loops, and thus large problems do not fit in the instruction cache. The software was run an a Sun Ultra Sparc I workstation running at 140 MHz.

Graph 1 shows the relative performance of the 3 schemes, and Graph 2 shows the speedup of the hardware over each of the software approaches. In this case the hardware has been optimised for a Xilinx 4020EPG223-1 device. The results indicate that it is possible to gain between 2 and 3 orders of magnitude speedup, which is significant.





6. Conclusions

The aim of this work was to establish whether it was possible to achieve a reasonable speedup by implementing FPGA based Hopfield neural networks for some simple constraint satisfaction problems. The results are significant – our initial implementation using standard Xilinx FPGAs yielded 2-3 orders of magnitude speedup over a Sun UltraSparc workstation.

The main problem with the work to date is that the problems are both unrealistically small and simplistic. We have built real hardware for the 8 queens problem, but this requires 4 XC4020 devices (each of 20,000 gates) interconnected using an Aptix FPIC switch. Thus, larger more realistic problems will require more, or larger, FPGAs. Fortunately, a number of companies are currently developing larger devices, including the Xilinx Virtex series which will contain around 1 million gates. Further, the constraints on the N-Queen problem are simpler than those found in many real world scheduling applications. Thus, it is not clear whether we will be able to optimise the neuron structure for more complex problems since the weights matrix may not contain as many zero elements. We are currently investigating a more complex planning problem to gain further insight into this problem. In the mean time, the current results are extremely promising.

Acknowledgements

This project has been funded by the Australian Research Council.

References

[1] Abramson, D. Logothetis, P, Postula, A., Randall, M, "FPGA Based Custom Computing Machines for Irregular Problems", Fourth International Symposium on High-Performance Computer Architecture, (HPCA98), February 1-4, 1998, Las Vegas, Nevada.

[2] Abramson, D.A., "A Very High Speed Architecture to Support Simulated Annealing", IEEE Computer, May 1992, pp 27 - 34.

[3] Arima, Y, et al, "A 336-Neuron, 28k-Synapse, Self Learning Neural Network Chip with Branch Neuron Architecture", ISSCC Dig. Techn. Papers, Proc IEEE Intn'l Solid-State Circuits Conf., 1991, pp 182-183.

[4] Boser, B., Sackinger, E., Bromley, J., IeCun, Y amd Jackel, L. "Hardware Requirements for Neural Network Pattern Classifiers", IEEE Micro, Feb 1992, pp 32 – 39.

[5] Buell, D., Arnold, J. and Kleinfelder, N. "SPLASH2: FPGAs in a Custom Computing Machine", IEEE Press, 1996, ISBN 0-8186-7413-X.

[6] Graf, H. P. and Henderson, "A Reconfigurable CMOS Neural Network", ISSCC Dig. Tech Papers, Proc IEEE Intn'l, Solid State Circuits Conf., Vol 33, 1990.

[7] Hopfield, J. J. and Tank, D. W. "Neural Computation of Decisions in Optimization Problems", Biological Cybernetics, vol. 52, pp. 141-152, 1985.

[8] Hopfield, J. J., "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", Proceedings National Academy of Sciences, vol. 79, pp. 2554-2558, 1982.

[9] Hopfield, J. J., "Neurons with graded response have collective computational properties like those of two--state neurons", Proceedings National Academy of Sciences, vol. 81, pp. 3088-3092, 1984.

[10] Jones, S. "Learning in Systolic Neural Network Engines", Proceedings of 26th Annual Hawaii International Conference on System Sciences, Jan 1993, pp 161 – 167.

[11] Kumar, S., Forward, K. and Palaniswami, M. "Performance Evaluation of a RISC Neuro-Processor for Neural Networks", Proceedings of 3rd IEEE/ACM International Conference on High Performance Computing, December, 1996, India, pp 315 – 320.

[12] Masa, P., Hoen,, K. and Wallinga, H. "A High Speed Analogue Neural Procesor", ", IEEE Micro, Vol 14, No 3, June 1994, pp 40 - 50.

[13] Moreno, J. et al, "An Analogue Systolic Neural Network Processing Architecture", ", IEEE Micro, Vol 14, No 3, June 1994, pp 51 – 59.

[14] Smith, K, and Palaniswami, M., "Static and Dynamic Channel Assignment using Neural Networks", IEEE Journal on Selected Areas in Communications, vol. 15, no. 2, pp. 238-249, 1997.

[15] Smith, K. "Neural Networks for Optimisation: A review of more than a decade of research", to appear, INFORMS Journal on Computing.

[16] Smith, K., Palaniswami, M. and Krishnamoorthy, M., "Neural Techniques for Combinatorial Optimisation with Applications", to appear, IEEE Transaction on Neural Networks.

[17] Verleyson, et al, "An Analogue Processor Architecture for a Neural Network Classifier", IEEE Micro, Vol 14, No 3, June 1994, pp 16 – 28.