Artificial Brains. An Inexpensive Method for Accelerating the Evolution of Neural Network Modules for Building Artificial Brains

Hugo de GARIS[a], Liu RUI[a], Huang DI[b], Hu JING[c]

[a] *Brain Builder Group, Key State Laboratory of Software Engineering, Wuhan University, Wuhan, Hubei Province, CHINA.*
[b] *Evolvable Hardware Group, School of Computer Science, China University of Geosciences, Wuhan, Hubei Province, CHINA.*
[c] *Computer Science Department, Utah State University, Logan, Utah, USA.*

profhugodegaris@yahoo.com, pnicholas@vip.sina.com, aaron192032@gmail.com, jinghu78@gmail.com

Abstract. This chapter shows how a "Celoxica" electronic board (containing a Xilinx Virtex II FPGA chip) can be used to accelerate the evolution of neural network modules that are to be evolved quickly enough, so that building artificial brains that consist of 10,000s of interconnected modules, can be made practical. We hope that this work will prove to be an important stepping stone towards making the new field of brain building both practical and cheap enough for many research groups to start building their own artificial brains.

Introduction

The primary research goal of the first author is to build *artificial brains* [1]. An artificial brain is defined to be a collection of interconnected neural net modules (10,000–50,000 of them), each of which is evolved quickly in special electronic hardware, downloaded into a PC, and interconnected according to the designs of human BAs (brain architects). The neural signaling of the artificial brain (A-Brain) is performed by the PC in real time (defined to be 25Hz per neuron). Such artificial brains can be used for many purposes, e.g. controlling the behaviors of autonomous robots.

There is at least one major problem with the above vision, and that is the slow evolution time of individual neural network modules. Typically, it can take many hours to even half a day to evolve a neural net module on a PC. Obviously, evolving several tens of thousands of such modules using only a PC to build an artificial brain will not be practical. Before such A-Brains can be built with this approach, it will be necessary to find ways to accelerate the evolution of such a large number of neural net (NN) modules. At the present time, the authors are pursuing the following approach. We perform the NN module evolution in hardware, to achieve a speedup factor (relative to ordinary PC evolution speeds) of about 10-50 (the topic of this chapter).

We use a Celoxica company's FPGA electronic board (containing a 3 megagate FPGA, i.e. Xilinx's Virtex II chip) to accelerate the evolution of neural network modules. One of the aims of this chapter is to report on the measurement of the speedup factor when using this Celoxica board to evolve neural network modules, compared to using a PC, as performed on the same evolution task.

The first experiment reported on in this chapter involved the evolution of a fairly simple neural net. This task was chosen to serve as the basis for the above comparison and is described in detail in section 3. The task itself was evolved 10 times and an average evolution time calculated. The evolutionary task used two different approaches and technologies, namely a) a standard genetic algorithm on a PC, and b) using the Celoxica board, and the high level language "Handel-C" [2] to program the evolution. Once the two execution time measurements were obtained, the speedup factor could be calculated. Final results are shown in section 7.

1. The Evolution of Neural Network Modules

This section gives a brief description of the approach we use normally to evolve our neural network (NN) modules, to be used as components in building artificial brains. We use a particular neural net model called"GenNet" [3]. A GenNet neural network consists of N (typically N = 12-20) fully connected artificial neurons. Each of the N^2 connections has a weight, represented as a signed binary fraction, with p (typically p = 6-10) bits per weight. The bit string chromosome used to evolve the N^2 weights will have a length of $N^2*(p+1)$ bits. Each neuron "j" receives input signals from the N neurons (i.e. including a signal from itself). Each input signal Sij is multiplied by the corresponding connection weight Wij and summed. To this sum is added an external signal value Ej. This final sum is called the activation signal Aj to the neuron "j".

$$Aj = \sum_{i=1}^{N} WijSij + Ej$$

This activation value is fed into a sigmoid function g that acts as a "squashing" function, limiting the output value Sj to have a maximum absolute value of 1.0

$$Sj = g(Aj) = \frac{Aj}{|Aj| + 1.0}$$

2. The Evolutionary Task

In order to compare the evolution times for the two different approaches (i.e. using an ordinary GA on a PC, and using the Celoxica board), the same fairly simple neural net task was evolved, namely to output a constant signal value of 0.8 over 100 clock ticks.

A single neuron of the network was randomly chosen to be the output neuron for the whole network. Its output signal S(t) was compared to a target (desired) signal value for each of T (=100) clock ticks. The fitness function was defined as follows.

$$f = \frac{1}{\sum_{t=1}^{100} (T(t) - S(t))^2}$$

In our evolutionary task, the target value T(t) is constant, i.e. T(t) = 0.8

In order to compare the evolution times of the above task, a concrete "cutoff" fitness value was used. This was found empirically, by evolving the standard GA version to what we felt was a reasonably "saturated" fitness value. This fitness value was then used in the Celoxica evolution experiments. Once the evolution, as specified by the Handel-C program executed on the Celoxica board reached the same fitness value, its evolutiontime (in seconds) was recorded. The task was evolved 10 times and the average value calculated.

3. Our Evolutionary Approach

The approach we pursue in this chapter to accelerating the evolution time of neural network modules is to perform the evolution in special hardware, e.g. using a Celoxica board [4]. In this approach, a high level language, called Handel-C [Handel-C 2006] is used, whose code is hardware compiled (silicon compiled) into the FPGA (Xilinx's Virtex II chip). We continue to work on this approach. In this first experiment, we have achieved an 8-fold speedup up compared with a standard GA algorithm on an ordinary PC. By using pipelining techniques and a larger FPGA, we hope to achieve a speedup factor of up to 50 times for the same task. (As will be seen in our more recent experiments, by evolving a smaller neural net, we were actually able to achieve this 50-fold speedup. See section 9).

In order to calculate the speedup factor of the Celoxica board relative to an ordinary genetic algorithm on a PC, the same evolutionary task specified in section 3 was used. The next section gives a brief description of the ordinary genetic algorithm. Section 6 described briefly the characteristics of the Celoxica board. Section 7 introduces the high level language Handel-C, used to program the Celoxica board. Section 8 presents the result of the comparison. Section 9 presents the results of our more recent evolutionary experiments on smaller neural nets. Section 10 presents ideas for future work, and section 11 summarizes.

4. Description of the Standard Genetic Algorithm

The standard GA (Genetic Algorithm) used to help calculate the speed up factor consisted of the following steps.

a) Randomly generate 100 bit string chromosomes of length $N^2*(p+1)$. Over the years we have used values, N = 16, p = 8, so our chromosomes (bit strings) were 2304 bits long.

b) Decode the chromosome into the N^2 signed binary fraction weights, and build the neural network for each chromosome.

c) Perform the fitness measurements for the task concerned. For details, see the next section.

d) Rank the fitnesses from best to worst.

e) Throw out the inferior half of the chromosomes. Replace with the superior half.

f) Mutate all the chromosomes except the top one. No crossover was performed in these experiments.

g) Go to b), until the evolution saturates at the target (desired) fitness value (of the elite chromosome).

5. The Celoxica Board

The aims of lowering the price of high-speed evolution, and achieving higher performance in evolving hardware led us to choose to use FPGAs (Field Programmable Gate Arrays). FPGAs are specially made digital semiconductor circuits that are often used for prototyping. The several million logic gates in modern FPGAs (e.g. Xilinx's Virtex II chip) make it is possible to have multiple copies of the same electronic sub circuit running simultaneously on different areas of the FPGA. This parallelism is very useful for a genetic algorithm. It allows the program to process the most time costly weight calculations in parallel, and this can speed up the overall evolution by a factor of tens to hundred of times.

We chose the Celoxica FPGA board for our project. "Celoxica" is the name of a UK company [4]. Our Celoxica board (an RC203) cost about $1500. With such a board, a design engineer is able to *program* electrical connections on site for a specific application, without paying thousands of dollars to have the chip manufactured in mass quantities. [4].

The RC Series Platforms of Celoxica are complete solutions for FPGA design and ASIC/SoC (system on a chip) verification and prototyping. The boards combine very high-density FPGA devices with soft-core & hard-core processors and an extensive array of peripherals. They provide easy access to programmable SoC's from the Electronic System Level (ESL) and consistently deliver fast and efficient access to very high-density reconfigurable silicon.

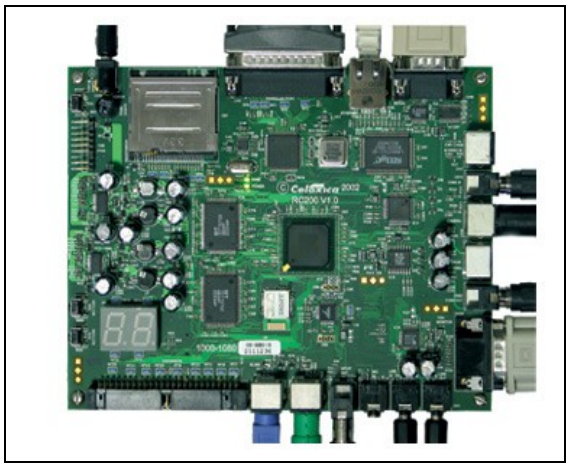

Fig.1. The Celoxica Board with central FPGA

We are currently using an RC203 FPGA board in our experiment. It's a desktop platform for the evaluation and development of high performance applications. The

main FPGA chip is a Xilinx Virtex II that can be configured without using an HDL (Hardware Description Language). Instead it uses a much easier high-level "C-like" language called "Handel-C" (after Handel the composer). This language is very similar to ordinary C (i.e. with approximately an 80% overlap between the two languages), with a few extra features, particularly those involved with specifying which functions ought to be executed in *parallel*.

With several million logic gates in the Virtex FPGA chip, it is possible to have multiple copies of the same electronic (sub) circuit running simultaneously. This parallelism allows a genetic algorithm for example to run much faster than on a PC. A Celoxica board attaches to a PC, with two-way communication, so that instructions to the board come from the PC, and results coming from the board can be displayed on the PC.

At the time of writing (October 2006), experiments are continuing to determine how much faster the evolution of a neural net module on the Celoxica board can be, compared to using a software approach in a PC. The value of this speedup factor is *critical* to this whole approach. Colleagues in the ECE (Electronic and Computer Engineering) departments at our respective universities, who are experienced in using Celoxica boards, estimate that the speed up factors will range between 10s and 100s of times. If these speedup factors can be achieved, it then becomes practical to evolve large numbers of neural net modules in a reasonable time, and connect them together to build artificial brains inside a PC.

6. The High Level Language "Handel-C"

Handel-C is Celoxica's high-level language used to configure the Virtex-II FPGA chip on the Celoxica board. This language is mostly ordinary C, with a few parallel programming features, e.g. the "par" statement, that takes the following general form:

```
par
{
   statement A;
   statement B;
   statement C;
}
```

The above Handel-C statement will cause all the statements (i.e. A, B, C) in the par block to be executed at once, by having each statement be configured, with its own separate set of logic gates in the Virtex-II chip on the Celoxica board. By using multiple copies of neural net modules, i.e. multiple electronic copies, each with its own area of programmable silicon, evolving in parallel, it is possible to speed up the overall evolution time, compared to ordinary software based PC evolution. In the following, we introduce the features of the Handle-C language.

Handel-C Programs

Handel-C uses much of the syntax of conventional C with the addition of inherent parallelism. You can write sequential programs in Handel-C, but to gain maximum benefit in performance from the target hardware you must use its parallel constructs.

Handel-C provides constructs to control the flow of a program. For example, code can be executed conditionally depending on the value of some expression, or a block of code can be repeated a number of times using a loop construct.

Parallel Programs

The target of the Handel-C compiler is low-level hardware. This means that you get massive performance benefits by using parallelism. It is essential for writing efficient programs to instruct the compiler to build hardware to execute statements in parallel.

Handel-C parallelism is true parallelism, not the time-sliced parallelism familiar from general-purpose computers. When instructed to execute two instructions in parallel, those two instructions will be executed at exactly the same instant in time by two separate pieces of hardware.

When a parallel block is encountered, execution flow splits at the start of the parallel block and each branch of the block executes simultaneously. Execution flow then re-joins at the end of the block when all branches have completed. Any branches that complete early are forced to wait for the slowest branch before continuing.

Channel Communication

Channels provide a link between parallel branches. One parallel branch outputs data onto the channel and the other branch reads data from the channel. Channels also provide synchronization so that a transmission can only be completed when both parties are ready for it. If the transmitter is not ready for the communication then the receiver must wait for it to become ready and vice versa.

Scope and variable sharing

The scope of declarations is based around code blocks. A code block is denoted with {...} .This means that:

- Global variables must be declared outside all code blocks.
- An identifier is in scope within a code block and any sub-blocks of that block.
- The scope of the variables is illustrated below:

```
int w;
void main(void)
{
    int x;
    {
            int y;
```

```
        }
        {
                int z;
        }
}
```

Since parallel constructs are simply code blocks, variables can be in scope in two parallel branches of code. This can lead to resource conflicts if the variable is written to simultaneously by more than one of the branches. Handel-C states that a single variable must not be written to by more than one parallel branch but may be read from by several parallel branches.

If you wish to write to the same variable from several processes, the correct way to do so is by using channels that are read from in a single process. This process can use a "prialt" statement to select which channel is ready to be read from first, and that channel is the only one that will be allowed to write to the variable.

```
while(1)
prialt
{
case chan1 ? y:
break;
case chan2 ? y:
break
case chan3 ? y:
break;
}
```

In this case, three separate processes can attempt to change the value of y by sending data down the channels, chan1, chan2 and chan3. y will be changed by whichever process sends the data first.

Handel-C uses much of the syntax of conventional C with the addition of inherent parallelism. You can write sequential programs in Handel-C, but to gain maximum benefit in performance from the target hardware you must use its parallel constructs. These may be new to some users. If you are familiar with conventional C you will recognize nearly all the other features.

7. Experimental Results

Evolving Neural Network Modules on a PC and on the Celoxica Board

The aim of the experiment described in this section is to evolve a "GenNet", i.e. a Genetically Programmed Neural Net that outputs a desired signal. The experiment was conducted both on a PC and on the Celoxica board. Based on the two evolution times of the GenNet using the two methods, a comparison was made to see how great a speed-up of the evolution can be achieved by using the Celoxica Board compared to using a software based evolution on the PC. The GenNet evolution program that was run on the PC was written in the "C" language, whereas the program run on the Celoxica board was written using the "Handel-C" language.

Handel-C is a C like language that can program the hardware, whereas C language is used to program the PC. The gate-level netlist output of Celoxica's "DK" (Designer Kit) software contains basic gates: OR, XOR, NOT and AND. The FPGA board is made of LookUp Tables (LUT, with 4 inputs and 1 output), flip-flops (FF) and other components. The output of the DK, which is the bit file, is downloaded to the board to configure its components. Technology mapping is the process of packing gates into these components. This is very different from traditional programming of a PC. A program in a PC is stored in memory. At each clock tick the CPU will fetch an instruction code and execute it, then fetch another code, etc. The program on the board is just a set of LUTs, so the execution of the program is at electronic speeds.

There are a few problems when coding the GenNet using Handel-C. Doing real number calculations is not as easy as in C. There is no variable type such as float, double etc in Handel-C. The fitness calculation, decoding of the chromosome into weights and chromosome fitness comparison all need real number calculators. Handel-C does provide a floating-point library and a fixed-point library. Float-point calculations will generate LUTs of great depth, which means longer propagation times, and slower running times. Fixed-point library calculations are preferred here because each real number variable will be represented as a sequence of bits of specified length. The first part of the bits will be interpreted as integers, and the second part will be interpreted as decimals. There are signed and unsigned fixed-point variables, while the signed fixed-point uses 2's complement representation. Attention must be paid too when using fixed-point library calculations. Using too many bits in each variable will increase the number of LUTs, while too few bits in each variable will decrease the precision of the calculation.

Handel-C has many features that are suitable for evolutionary computation. Firstly, the variable in Handel-C can be of any length. Handel-C also provides bit-wise operations. Thus using Handel-C to code chromosomes introduces great flexibility. Secondly, the "par" block provides parallelism, which reduces the running time of decoding each chromosome and the fitness calculation. Thirdly, the program is a set of LUTs on the board, so the running time is at electronic speeds. Fourthly, pipeline techniques can be applied to increase the throughput of some blocks of code.

The Handel-C program is tested and simulated in the PC. It is then placed and routed using Xilinx's ISE 7.1. Our Celoxica board is a Xilinx Virtex RC203. The GenNet program in C was run on a PC with a 1.69 GHZ Centrino CPU, and 1 GB of memory.

The program was run 10 times each on the board and the PC. Evolution time of each test was recorded. The average evolution time on the PC is 57.4 seconds, while the average evolution time on Celoxica board is 7.1 seconds. ***The speedup of the evolution time on the FPGA board is 8.1 times faster than on the PC.***

This is a great improvement compared with the evolution time on a PC. Although a great speedup is achieved, there are many possible improvements to the current method that can further increase the evolution speed on the board. The experiment on the current board has some limitations. If a larger and newer version of the board (e.g. the RC300) becomes available, results will be much better (i.e. a greater speedup becomes possible).

1. The current GA code's level of parallelism is not sufficient. Because the number of LUTs (the number of available gates) and Configurable Logic Blocks (CLBs) on the board is a limited resource (28,672 LUTs for Virtex RC203, 4 slices), the current GA code does not have enough parallel code blocks (e.g. using the "par" feature). Handel-C provides "par" blocks that can be used to express both coarse and fine-grained parallelism. (For example, see the Handel-C tutorial on the Celoxica website [4]).

Individual statements and functions can be run in parallel. The more "par" blocks used, the more parallelism is achieved. However, when "par" blocks are mapped into hardware, their parallel execution requires more circuitry, which means more resources (i.e. logic gates, etc) will be consumed. The Celoxica board has a limited number of LUTs, so that not too many "par" blocks can be used. If too many parallel structures are used in the Handel-C code, the final number of LUTs will be too large to fit into the board (RC203) used in the experiment.

2. Decoding the chromosome into the weights, the fitness calculation and mutation can all be done in parallel. This requires many arrays to store temporary variables. Arrays in the Handel-C provide parallel access to all elements, which is suitable for "par" blocks. But too many indexed or two-dimensional arrays used will introduce the same problem, i.e. too many resources will be eaten up because of the multiplexing. Instead, we used block storage of RAM a lot in the code to avoid the problem of insufficient resources on the board. With RAM, at each clock tick, only one element of data can be accessed, which means the code related to the RAM must be sequential. However this decreases the level of parallelism of the GA code.

Our original Handel-C code had many parallel blocks, to such an extent, that it could not fit into the current board available for our experiment. It generated 37% more LUTs than the maximum capacity of the current board. In order to reduce the number of LUTs needed, many pieces of parallel code had to be rewritten. If most of the code in the GA can be parallel, then definitely much faster speedups will be achieved.

3. Pipeline techniques can be applied to the GA code to increase the throughput. Some complex expressions or calculations in the current code can be broken into smaller pieces and run simultaneously. Each stage in the pipeline reads the results of the previous stage while the new value is being written. This will result in increased data latency. But the downside of this approach is it results in increased flip-flop usage. Thus, all the above improvements are only possible when tested on a newer version or bigger board.

Martin performed an empirical study [5] with Genetic Programming using Handel-C and an FPGA board and reported a speed-up factor of over 400 times when compared with a software implementation of the same algorithm, but the PC used was an AMD 200 MHZ CPU and the problem had no real-number calculations. The problem and experimental setup was quite different from our evolution of GenNets because all our GenNet calculations had to been done in fixed-point, which consumed a lot of resources, thus preventing the program from being parallelized and pipelined easily.

8. More Recent Work

In an attempt to increase the 8-fold speed-up factor mentioned above, using the Celoxica board, relative to performing the same task on a PC, we undertook some recent experiments in which we were less constrained to make compromises (e.g. by having to use less PAR statements) than we were when undertaking the experiments discussed in earlier sections. We were less constrained as a consequence of performing simpler experiments that required less Handel-C code, and less logic gates on the Xilinx chip on the Celoxica board. We evolved neural nets that were smaller, having fewer neurons, and connections, and whose evolved tasks were simpler. As a result, we were able to take greater advantage of the intrinsically faster evolution speeds of the

Celoxica board, and confirmed our suspicion that approximately a 50-fold speed up would be possible, and indeed it was.

The first of these experiments (of 2 so far) was to evolve a fully connected neural network of only 3 neurons, hence 9 connections, whose binary fraction weights had only 4 bits to represent them (plus one bit for the sign +/- of the weight). The task was to output a constant "target" signal (of value corresponding to the binary fraction .1001, whose decimal value = 0.5625). The fitness was defined to be the sum of the squares of the differences between the actual output values and the target value, over 15 clock cycles. The population size was 20, and the number of generations was 16000. The evolution took 47 minutes, 36 seconds on the PC, and 51 seconds on the Celoxica board, ***a speed-up factor of about 56***. Only about 7% of the logic gates on the Xilinx chip of the Celoxica board were used. The Genetic Algorithm took up most of this 7%.

In a similar experiment, we then tried to evolve 4-neuron modules. Most of the parameter values of this second experiment were the same as before. This time the PC evolution time was 67 minutes 20 seconds, and the Celoxica board evolution time was 81 seconds, i.e. ***a speed-up factor of 50***. The percentage of the Xilinx chip occupied by the Handel-C program was approximately 223,000/3,000,000 approx 7.4%.

These speed-up factors were encouraging. However, we noticed something that dampened our enthusiasm somewhat, and that was the time needed to perform the ***routing*** of the electronic connections in the Xilinx chip of the Celoxica board. Xilinx provides special software to perform the routing of the electronic wiring that connects the logic gates on their programmable chip. We found that this routing time for our second experiment was about ***40 minutes.***

However, the whole point of using the Celoxica board is to accelerate the evolution of individual neural net modules, so that it becomes practical to evolve 10,000s of them to build artificial brains. If it takes about 1 hour (let's say) to evolve a neural net module on a PC, then what is the great advantage of using a Celoxica board to do the same thing, if the routing time is almost as long as the PC evolution time, even if the actual execution time of the routed circuit performs 50 times faster?!

This seems like a crushing objection to our overall approach, but there is a loophole fortunately, which is that one evolves a "generic" neural net that is then fed data into the circuit already routed on the Xilinx chip. In other words, the routing of the circuitry on the programmable chip occurs once, and is slow, but once the routing is done, the resulting circuit can be used many times over, in the sense that different neural net modules can be evolved by sending in different data to the same circuit. Thus one does not have to route each neural net module that is evolved in the chip. Sending in the data to the chip takes only seconds, compared to about 40 minutes to route the chip. Hence, we can take advantage of the much greater electronic speed of the Xilinx chip on the Celoxica board. Future work along these lines is currently being thought about and planned. See the next section on future ideas for further details.

9. Future Ideas

The immediate future work to be undertaken comprises two tasks. The FPGA on our Celoxica board has only 3 mega-gates. The Celoxica company field engineers, have very recently offered us to run our Handel-C code on one of their boards with a bigger chip (i.e. one with 6 mega-gates), so that our Handel-C code in our first experiment that originally had too many "par" statements can fit better into the larger chip, and hence run faster. This experiment is something we need to do.

Another immediate task is to translate the Handel-C code that we have already written into a more pipelined approach. Pipelining, a well-known technique in computer science, requires extra silicon (extra logic gates) but is faster to operate, due to its greater parallelism. We can send off this pipelined code to Celoxica also, or perhaps buy our own bigger FPGA board. This type of reasoning will remain valid as long as Moore's Law remains valid. We can expect every year or so, to be able to evolve neural net modules faster, due to larger FPGAs.

Greater speedups will allow neural net modules to be evolved considerably faster compared to using software techniques on an ordinary PC, so that a rather complex module containing multiple tests for its evolution (e.g. a module that responds with a strong signal if it sees any one of the 5 letters a, b, c, d, e, but a weak signal if it sees any of the other 21 letters, will require 26 "tests") that would take many hours on a PC, could be evolved in minutes using the Celoxica board approach.

Such speedups will revolutionize brain building. It would then become practical to evolve 10,000s of neural net modules in a time that would be practical within human patience levels. A relatively small brain building team (say of 10 people) could then build an artificial brain (of 10,000 modules) in 5 months, at a rate of 10 evolved modules per working day per person.

If the total speedup can be made closer to 1000 rather than 100, then it is conceivable that new multi-module evolutionary algorithms could be created, that would make use of this speedup to evolve not only the intra-module weights, but the inter-module connections as well. One of the weaknesses with the authors' current brain building approach is that the modules are evolved individually, irrespective of how they will interact with other modules in a network of modules. Very rapid evolution of individual modules would allow new approaches to multi-module evolution.

10. Summary

This chapter has presented results showing that a Celoxica FPGA board is capable of speeding up the evolution of neural network modules by a factor of about 10 to 50 times, depending on the size of the neural net being evolved. In the case of the first experiment with a large neural net (e.g. 20 neurons), which achieved only an 8-fold speedup, the possibility still exists of producing a speedup of perhaps up to 50 times using larger chips and pipelining techniques - an essential step if artificial brains, comprised of 10,000s of such modules are to be evolved in a reasonable time, and then run in real time in interconnected form in an ordinary PC. At the present time, the evolution of a single neural network can take many hours, a fact that makes brain building according to our PC-based approach quite impractical.

Smaller Gas

Most of the gates (flip flops) on the Xilinx chip on the Celoxica board, were taken up by the Genetic Algorithm. With the 3 and 4 neuron network experiments we tried, only about 7% of the gates were used. This is encouraging. We will try to evolve larger modules, i.e. with a larger number of neurons, and hence connections. The number of connections grows as the square of the number of neurons. There are also smaller GAs in the literature, usually called "Compact Genetic Algorithms" (CGAs) that by definition, are very simple and hence need fewer logic gates for their electronic

implementation. We may be able to evolve electronically, larger modules with small GAs, and hence really push up the size of the modules we can evolve (with a 50-fold speedup).

Another factor assisting the growth in the size of modules is of course Moore's Law. For example, the next generation Celoxica board, beyond the RC203 that we are currently using, has a Xilinx FPGA chip that contains 6 million logic gates, i.e. a doubling compared to our RC203 Celoxica board. Celoxica does indeed have a new board based on the new Xilinx chip"Virtex 4". So, our next research project will be to see how large our neural modules can become, i.e. just how many fully connected neurons can be evolved electronically on the Celoxica board? More interestingly is the question, "How many more years of Moore's Law will be needed before it will be possible to evolve electronically, neural net modules of reasonable size (i.e. having about 12-20 neurons)? It looks as though the answer is only a few years away (or none at all?). Once such powerful modules are readily evolvable, then the real work of building artificial brains can begin. The underlying technology, i.e. the electronic evolution of large numbers of neural net modules, will make the production of 10,000s of evolved modules needed to build an artificial brain, practical. The real challenge then of designing an artificial brain can then begin, and result hopefully in the creation a new research field, namely "Brain Building" or "Artificial Brains".

As mentioned at the end of section 9, one other research challenge remaining is how to design a "generically evolvable" neural net to overcome the slow routing problem. For example, one evolves the generic circuit *once*, and then sends in different fitness definitions as external data from the PC to the circuit. To change the fitness definition, simply means changing the data that is input to the "changeless" generic circuit. Fleshing out the details of these initial ideas remains a future research topic.

Postscript : Answering a Reviewer

This postscript contains questions from a reviewer to an earlier version of this chapter. We thought some of the questions were interesting, so we include them here, with our answers. The questions are in italic, beginning with a Q, and our answers begin with the letter A.

Q : This is a very interesting chapter that presents results that seem to indicate that computational power is not going to be a major obstacle to creating generally intelligent computers in the mid-term.

A: We think computational power is a necessary condition at least. Modern electronics is already capable of creating artificial brains with tens of thousands of evolved neural net modules. Now that it is possible, the authors would like to actually do it.

Q: Some questions about comparisons with more conventional hardware. A quad-core processor on a four-processor machine could in theory give a 16x speedup over a single processor. If that turns out to be approximately feasible, what will the cost-benefit tradeoff be between this approach and the approach of using conventional hardware? Will the underlying FPGA speed grow as fast as processor speed? If not, then Moore's law could potentially obviate this approach.

A: Special hardware, such as a quad core processor and the like, are definitely viable alternatives to what we propose in this chapter, but are more expensive. Our Celoxica board costs only about $1000, so seems a cheap way to get a 50-fold speedup over an ordinary PC. If the speeds of future FPGA boards do not grow faster than the speeds of PCs, then that would be something we would welcome. The FPGA board is

only a tool for us to accelerate the evolution speed of the neural net modules. If a faster cheaper way can be found, that's fine by us.

Q: The authors seem to suggest that only real barrier to achieving general intelligence is affordable hardware speed. Do they really believe that all the intellectual problems have been solved?

A: If we appear to be suggesting that, then that is not our intention. Of course, affordable hardware speed is only the initial and necessary condition to achieving general intelligence. Once the hardware speed is achieved, then our real research challenge of designing interesting capable artificial brains can begin, with the emphasis on the word *begin*.

Q: There are many other approaches to hardware acceleration for neural systems. Could the authors say a bit about how their approach compares?

A: There are approaches such as ASIC chips (i.e. custom designed chips for specific tasks). Of course the great advantage of the FPGA approach is it flexibility, its reprogrammability. If we want to evolve a different neural net model in the FPGA, we can do that easily. With a non reprogrammable approach we couldn't. ASICS may be faster than FPGAs, but their flexibility far out ways their relatively slower speed relative to ASICS, we feel.

Postscript : Last Minute Results

As proof that the work of this chapter is ongoing, we report here some very recent results concerning two experiments. The first was to see how many neurons N we could fit into the FPGA chip, using a more compact GA (i.e. the CGA, "Compact GA" mentioned in section 11 (Summary). By using fewer lines of code to specify the GA, as is the case in a CGA, there was a lot more room on the FPGA for more neurons N in the neural network. For the same experiment mentioned in section 9, using the CGA, we were able to place N = 28 neurons on the chip. This is far more than we need, so we undertook a more demanding experiment. The aim this time was to evolve (using the CGA mentioned above) a sine curve output for half a wavelength. The number of ticks of the clock used for the curve (i.e. one tick is one cycle of calculating the output signal for each neuron in the network) was 45. The number of bits in the weights was increased to 8 (i.e. 1 for the sign, and 7 for the weight value). The fitness definition was the sum of the squares of the errors between the target sine half curve (i.e. y(t) = sin(pi*t/45)), and the actual output signals over the 45 ticks t. The number of neurons as 12, population size was 256, bit string chromosome length was 12*12*8 = 1152 bits. The number of generations used was 128,000. This half sine curve evolved well, showing that a non trivial neural net could be evolved in the 3M gates available in the FPGA of the Celoxica board. The speedup factor was about 57 times compared to evolving the same task on the same PC used to control the Celoxica board.

In the near future, we will evolve many other single neural network modules using our Celoxica board, to show that they can be evolved. The next immediate task is to create an approach we call "Generic Evolution". By this we mean creating a generic evolvable model, that is routed once (or only a few times) in the FPGA and then used multiple times, by sending in data signals to the FPGA. Each data set that goes into the chip is used to evolve a neural net module. This data contains such things as the number of positive and negative examples of training vectors for the neural net evolution, and the training vectors themselves. The model expects data input to have a given dimensionality, e.g. 1 dimensional bit strings, or 2D pixel grid input from a digital camera, etc. We are currently working on this, to overcome the problem of

having to route the FPGA for each module evolution. Since the routing takes about 40 minutes for a full chip, this is too slow. So by having a generic model routed once only in the chip, we can send it different data for different neural net module evolutions. Sending in data to the already routed chip takes only a few seconds.

As a more concrete illustration of this "generic evolution" idea, we provide the following example. Assume we want to evolve several hundred 2D pattern recognition neural net modules. The pattern to be detected is "shone" onto an 8 by 8 pixel grid of photo-cell detectors, each of whose signal strength outputs is strong if strong light falls on the photo-cell, and is weak if the light falling on it is weak. These 64 light intensity output signals are fed into a fully connected 16 neuron neural network. Hence each neuron receives 4 external signals from the pixel grid. If the pattern on the grid is of the type that the neural net module has been evolved to detect, then the module will output a strong signal, otherwise a weak signal. To evolve such a detector module, we create a set of positive examples of the pattern, e.g. a "vowel" detector (i.e. the letters A,E,I,O,U). Similarly, we create a set of negative examples (e.g. the 21 consonants). We then shine the 5 vowels onto the grid, say for 100 ticks each. (One tick is defined to be the time needed for all neurons in the module to calculate their output signals.) We then shine the 21 consonants onto the grid for 100 ticks each.

The 64 (8 by 8) pixel values from the photocell grid are converted into a 1 dimensional (64 element) vector, by concatenating the 8 rows of the grid. There will be 26 (5 + 21) such vectors. Thus the data to be sent to the generic circuit will take the following form :- (N1, i.e. the number of positive examples, N2, i.e. the number of negative examples, C, i.e. the number of clock ticks per example, and then the (N1 + N2) 64-element input vectors.

The Handel-C code is written such that it expects the above parameters to be sent to it once the routing of the code has been completed. One can evolve hundreds of different 2D pattern detection neural net modules, in this way, without having to reroute the chip for each module evolution. The various modules have generic features, e.g. they all use a 16 neuron neural network, with a 64 input signal. The N1 positive examples are input first, then the N2 negative examples. The fitness definition of the neural net module is also generic. The target (i.e. desired) output signal of the evolving neural net module is high if a positive example is input, and low if a negative example is input. Hence the fitness definition can be generalized, e.g. to be the inverse of the sum of the squares of the differences between the target signal values and the actual signal values for C*(N1+N2) ticks. This fitness definition will be a function of the various parameters (N1, N2, C, etc) that are sent in as data for each neural net module evolution. Hence this generic fitness definition need only be coded and compiled and routed once for the evolution of hundreds of 2D pattern recognition neural net modules.

If the generic fitness definition changes (e.g. by having a new neural net model, or different input vector formats) then another routing can be performed, costing 40 minutes. But in practice, much time is saved by using this "generic evolution" approach that is new, and an invention of our research team. It is becoming an important theme in our use of the Celoxica board to accelerate the evolution of tens of thousands of neural net modules to build artificial brains.

References

[1] Hugo de Garis, Michael Korkin, THE CAM-BRAIN MACHINE (CBM) An FPGA Based Hardware Tool which Evolves a 1000 Neuron Net Circuit Module in Seconds and Updates a 75 Million Neuron Artificial Brain for Real Time Robot Control, Neurocomputing journal, Elsevier, Vol. 42, Issue 1-4,

February, 2002. Special issue on Evolutionary Neural Systems, guest editor: Prof. Hugo de Garis. Downloadable at http://www.iss.whu.edu.cn/degaris/papers
[2] www.celoxica.com
[3] See http://www.iss.whu.edu.cn/degaris/coursestaught.htm Click on the CS7910 course.
[4] www.celoxica.com
[5] Peter N. Martin, Genetic Programming in Hardware, PhD thesis, University of Essex, 2003, http://homepage.ntlworld.com/petemartin/HardwareGeneticProgramming.pdf