A FAST SIMULATOR FOR NEURAL NETWORKS ON DSPs OR FPGAs[1]

M. Adé, R. Lauwereins[2], J. Peperstraete
ESAT-Laboratory , K.U.Leuven
K. Mercierlaan 94, B-3001 Heverlee, Belgium
Tel.: +32 -(0)16 - 22 09 31, ext. 1035 Fax: +32 -(0)16 - 22 18 55
email: ade@esat.kuleuven.ac.be

Abstract-This paper presents a brief description of our achievements and current research on the implemention of a fast digital simulator for Artificial Neural Networks. This simulator is mapped either on a Parallel Digital Signal Processor (DSP) or on a set of Field Programmable Gate Arrays (FPGA). We already developed powerful tools that automatically compile a graphical neural network description into executable code for the DSPs, with the flexiblity to adjust weights and thresholds at run-time. The next step is to realise similar tools for the FPGAs.

INTRODUCTION

The simulation of Artificial Neural Networks is mostly done on a digital computer. Yet it is commonly known that this remains a very time consuming job, even on a powerful workstation.
A possible answer to this problem is to implement the network on a parallel computer. We chose the latter to be either a Parallel Digital Signal Processor or Field Programmable Gate Arrays. The DSP (Motorola 56001) is chosen because its typical multiply-accumulate structure exactly reflects the weighted summation part of a neuron. With the alternative, the FPGA components (Xilinx 3090), we have the advantage of a hardware implementation, which is faster than its software counterpart.
We already developed tools that automatically translate a neural network description into executable code for a parallel DSP, and are developing similar tools to generate configuration code for a set of FPGAs. This allows for rapid

[1] This work has partly been sponsored by the Belgian Government under the Concerted Action on Applicable Neural Networks
[2] Senior Research Assistant of the Belgian National Fund for Scientific Research

0-7803-0557-4/92$03.00 © 1992

prototyping of various alternative neural networks, and for fast simulation with the possibility of changing parameters, like weights and thresholds, at run-time. Of course the result can also be used as a digital implementation as such. The class of neural networks that we chose to work with, is described next.

First, for the neuron model, we assume all inputs and outputs to be binary valued. Each neuron calculates a weighted sum of its inputs. To this summation, a parameterised threshold is added. Both the weights and the threshold have deterministic, digital values. The result is evaluated by a non linear function, for which we assume a binary step function.

Next, a network model is determined that can be realised on both DSPs and FPGAs. Because of the two dimensional implementation and the limited interconnection density on the FPGAs, the most suited choice is a layered feedforward model, also known as the perceptron model, described in [1].

A typical application is a pattern classifier, as described by [2], although any network that can be represented by the given model can be implemented.

It should be noted here that on DSPs almost any digital neural network could be simulated. At this moment however, we focus on the binary valued case since this is sufficient for the intended applications.

With the above described model, the functionality of a neural network can be determined completely at the design stage. If the application itself is also a strict function, all efforts can be concentrated on the implementation problem itself and on checking the correct behaviour after implementation. For this reason, our main goal is to realise any generalised symmetric function (GSF), [3], and some boolean functions. However, these can be regarded as elementary circuits to build larger functions. Hence, every function that can be realised by GSFs and some boolean functions only also belongs to the range of possible realisations. An example of this is an $n \times n$ digital multiplier. This is an extremely computation intensive application that covers all aspects of implementation, and hence is a good test case. Moreover, in [4] is shown that a neural implementation of a digital multiplier is even faster than a digital AND/OR gate implementations using the same chip area.

The GSFs are specified to the programming environment via a truth table, out of which it will compose the network by repeating and adapting a basic neuron.

The tools to synthesize an implementation on DSPs or FPGAs from the GSF truth table are part of GRAPE-II, the second generation of the GRAphical Programming Environment for parallel computers, developed in our group, [5]. The main idea is to generate executable code fully automatically, starting from the specification of the algorithm and the target hardware. Facilities for debugging and run-time evaluation are built in. Up to now, all efforts were concentrated in the domain of homogeneous parallel processor programming. Only recently, we began expanding the environment to the domain of programmable hardware.

In the next section more details are given about generalised symmetric functions, which are used as the example applications throughout the paper. The second

section describes the steps necessary to convert such a function into a neural network, while the last two sections describe the mapping of this network on DSPs and FPGAs respectively. Finally, the paper states some conclusions and points out the direction for further work.

PROPERTIES OF GENERALISED SYMMETRIC FUNCTIONS

Let $x=(x_{r-1},...x_1,x_0)$ be a boolean input vector.

A Boolean function is called generalised symmetric when its value only depends on $\|x\|$, the weighted summation of the input vector's elements, defined as $\|x\|=w_{r-1}.x_{r-1}+...+w_1.x_1+w_0.x_0$. This principle is described for a subclass, the purely symmetric functions, in [3]. Examples of GSFs are the majority function and the exclusive-or function.

It is clear from this definition that a GSF is well suited to be calculated by neurons. One advantage of using neurons instead of the classical two level AND/OR gate realisation for GSFs, is to keep the fan-in of all neurons only linearly dependent on the number of inputs of the function. This is not the case for the classical realisation, where the fan-in of the second level gates increases exponentially with the number of inputs. E.g. for an exclusive-or function, the fan-in of the second level gates can be shown to be 2^{n-1}, n being the number of inputs.

SPECIFICATION

The specification of such a neural network is done via the graphical interface of GRAPE, see [5]. When the neural network is already known, its graph is used as the primary input, together with a specification of the target hardware. On the other hand, a function could be given, e.g. as a truth table, for which the environment itself has to construct a neural network.

In the first case, when the graph is entered, it is transformed into code by a code generator. The code for the neurons depends on the choice of hardware, as will be explained in the last step for the function specification.

When the input is a function, the first step is to break it down into parts that can be implemented by (multiple) GSFs or by logic gates. This high level specification is written in coded form to a data base, which is the central part of GRAPE.

Then, a choice is to be made concerning the target hardware. If FPGAs are chosen, the function will be automatically converted down to neuron level before implementation. On the other hand, for DSPs, all parts can be implemented the way they are specified at the present level, or can be expanded further to lower levels, until the neuron level is reached. Expanding the parts at

the highest level is done by specifying the constructing GSFs. At the next level the GSFs can be expanded to neurons. This allows for simulation at different levels of the hierarchical network.
When this is finished, further partitioning will split (or group) the parts into entities that fit on one chip, being either a DSP or an FPGA. Next, code for each part of the network is generated. For the lowest level parts, the neurons, this is done by repeatedly retrieving out of a library the code for a general neuron, and adapting it to its specific data. Although basically the neuron model is the same for both DSP and FPGA, this general code is of course dedicated, so the library contains two coded neurons. Specific details about the neuron model for each target hardware are given in the respective sections. For the higher level parts, a file containing the assembly code for the DSP is generated.
The final result is a partially or complete neural network that models the specified function. All tools that are invoked next to map the network onto either DSPs or FPGAs, are target specific, since they need to take into account the specific properties of the target hardware. They are described in the next sections.

DSP IMPLEMENTATION

The basic principles of the neuron model we adopted, are already given in the introduction. Once the target hardware is specified, some more details are available. The general neuron for a DSP has the weights and the threshold built in as parameters. It is written in assembly language and defined as one macro, which is stored in the data base. The number of inputs is limited to 126, since the number of parameters that can be passed to a macro is limited. The weights as well as the threshold can be up to 24 bits long, which is the word length of the DSP, as long as the sum of the weighted inputs and the threshold does not exceed the 24 bits either.
For each neuron in the network, this macro is activated and the code is customised to the values for weights, threshold and inputs that were determined in the previous phases.
The resulting description is then compiled to executable code for the parallel processor. This roughly involves three steps : partitioning and assignment, scheduling, and routing.
Partitioning divides the network into subtasks, which are then assigned to the available processors. Care should be taken to group intercommunicating tasks as much as possible on the same processor. This way, the number of interprocessor communications is reduced to a minimum, so it is expected that the total interprocessor communication time is also kept low with respect to the computation time. In the scheduling phase, the execution order of the subtasks on each processor is determined, independent of their computation times. The data bits that are the inputs to the subtasks are also given a number to specify the sequence in which they will be used. This allows for handling both internal and

external unexpected interrupts when the application is being executed, since the exact duration of each part is not critical for proper execution. Tasks can be stopped and later on resumed, as long as the correct sequence is obeyed.
Control of the asynchronous communication is done by receive and send communication primitives, generated in the routing phase. Data for processors that can not be directly reached, is passed through other DSPs via buffers until it is at its destination.

The next step is evaluation of the network. An average speed of 10^6 connection updates per second and per DSP is reached. The values of the weights and thresholds can be changed at run-time. Of course, changing the neural network structure requires to go through all steps again, starting from specification, which is still fast and completely automatic.
In summary, these tools are capable of automatically generating out of a graphical description of a neural network the executable code for a very fast parallel machine.
This part of the simulator project is already finished and working properly, hence we started expanding it with the FPGA implementation, as described in the next section.

FPGA IMPLEMENTATION

The neuron for FPGAs is built on the same basic principles as the DSP neuron, but has to deal with restrictions on area and routing resources. Therefore we limit the number of inputs to some reasonable number like eight, and the weights to a five bit value. The threshold may have any value, but will be truncated to the maximum possible sum+1, which takes no more than eight bits.
Generally, the neuron for an FPGA consists of two separate parts. The first one performs the weighted summation on the inputs and produces the sum in binary coded form. The second part takes this binary number as input, together with a threshold value, and calculates the step function. The threshold and step orientation are defined as parameters.
When the desired network behaviour can be fully realised at the design stage, all parameters are completely known at compile time. With this knowledge it is preferable to generate neurons with minimal area occupancy, since this will speed up computation, and allow for larger networks to be implemented on a fixed number of FPGAs. This is done by a program that creates the summation part for each neuron taking into account the exact number of inputs and their weights. The logic is interconnected in a way that obtains the fastest execution possible, and that allows for the compilation program to remove all unnecessary logic. Then a comparator is added, for which the model is either a 'less than threshold' or a 'greater than threshold' function. Since the threshold is known at compile time, this logic can be simplified, which again leads to a minimal use of area.

An example for a neuron with four inputs being x0, x1, x2 and x3, with respective weights equal to 6, 9, 10 and 11, is given in Fig.1.

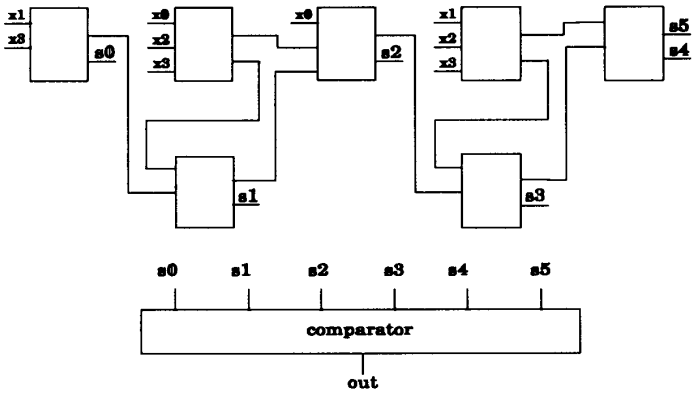

Fig.1 An example neuron implemented with minimal area

Each column in the summation part calculates a sum bit, where the rightmost bit represents the MSB of the sum. All sum bits needed go to the comparator, the result of which is the neuron output.

For each neuron, the program stores the required amount of area together with the maximal execution time in number of clock cycles. The area information will be used in the partitioning and assignment step to meet the restrictions on available area and routing resources per FPGA. Some extra area will be necessary on each FPGA to buffer off chip inputs and outputs. Note that this kind of neuron does not require any control logic, as long as the inputs are kept stable until the result is calculated. For the entire network however, some control logic will be needed to communicate with the outer world.

After partitioning and assignment, the maximum execution time for the complete network can be calculated.

A next point of investigation handles multiplexing several neurons in time on the same piece of hardware. When these neurons are not identical, it would be required to adapt the parameters of the implemented neuron before evaluation of each neuron to be multiplexed on it. Of course, these changes need to be done at run time. Since this is not feasible with the method presented, we can only allow for identical neurons to be multiplexed. This is not really interesting since generally only very few neurons in a network are identical, so the gain in area would be too small to compensate for the extra area needed to control the multiplexing operation.

Consequently, an other neuron model was developed in which all parameters are explicitly stored in registers that can be reloaded with other values at run time. This allows for implementing time multiplexing, and is also a well suited model

if parameters should not be fixed after compilation, which will permit studying the network behaviour for different values of weights and thresholds.
In this new model, the number of inputs is kept constant at eight, while the weights cannot go beyond five bits, but if the maximum weight of the application requires less bits, this can be specified. The neurons then will be implemented with that smaller number. The threshold is stored in an appropriate number of flipflops. The model is shown in Fig.2 and stored in a library for different numbers of weight bits.

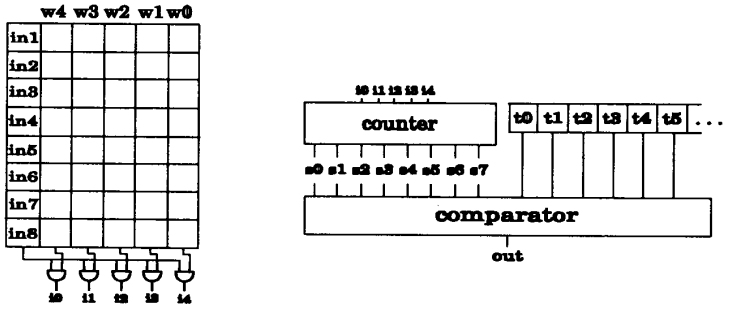

Fig.2 Model of a neuron allowing for changing weights and threshold at run time

Binary inputs are 'anded' with the digital weight bits and then shifted into the counter, one at a time, to produce the sum word, which is then compared with the threshold. The shifting of the inputs only begins when all inputs have arrived.
New weights and thresholds can be loaded by using special control lines, one for the weights, and one for the threshold, and using one input line through which the values are shifted in bit by bit.
The amount of area required for the neurons and controllers will again be generated by the program, as will be the execution times. For the same neuron execution times seem to be competitive with those for the minimal area neurons. Of course, these neurons will be mostly larger than is strictly necessary. In addition, a rather large controller is required for each neuron. This inconvenience can be eased by using only one controller per layer in the feedforward network, at the expence of execution time. Of course, when a layer is implemented on several FPGAs, each one of them needs a controller. This should be kept in mind at the partitioning and assignment phase.
Some extra control logic will be needed here also to take care of the communication with the outer world.
For both approaches, compilation of the global code will then generate a configuration file for each of the FPGAs, very similar to the compilation for the parallel DSP to executable code. The first step is also partitioning of the network followed by assignment to the available components. The main criterion in this

case is the amount of area occupied on one FPGA, rather than computation and communication time.
The routing phase is simplified with regard to the DSPs by allowing only directly connected Xilinx components to communicate with each other.
Because the execution parallelism of the programmed tasks on one FPGA is inherent to its structure, the scheduling phase can be completely omitted, except when multiplexing multiple neurons on the same hardware is considered. For the latter case, scheduling will give rise to the design of extra control logic.
With these limitations on routing and scheduling, a basic version of the simulator can be developed that will mainly concentrate on the assignment and partitioning part. We plan to add the omitted options on routing and scheduling in a more advanced stage.
When compilation is done, evaluation can start. Inputs and outputs are directly available at the I/O pins, while the system can also provide for in circuit test points by bringing them out through unused I/O pins of an FPGA. These signals are all automatically redirected by the environment to the user interface.
At this point, the first neuron model is already available from the library. Partitioning and assignment are still done manually, but from that point on, the tools we developed perform all further steps. The other model is presently under development.

CONCLUSION

In this paper we explained how we succeeded to implement a flexible and fast neural network simulator on DSPs, answering an existing need in the field of neural network research. At present, the first steps in expanding the environment to provide for an implementation on FPGAs are already set. The global strategy followed for this case is very similar to the DSPs, with adaptation to the properties and limitations of the new components.
We started with the development of two neuron models, one for minimal area use and one to provide for changing weights and thresholds at run time. Next, a tool has been worked out that translates the network once it has been partitioned and assigned, into suitable code to program the components. It does not yet support time multiplexing. The tools for the interfacing with GRAPEII, and for the partitioning and assignment, and for scheduling are under development.

REFERENCES

[1] M. Minsky and S. Papert, Perceptrons : an introduction to computational geometry, Cambridge, MA: MIT PRESS.

[2] C. E. Cox, W. E. Blanz, "Ganglion - A fast field programmable gate array implementation of a connectionist classifier" , in RJ 8290 (75651), IBM research division, Almaden, November 9, 1990.

[3] M. Paterson, N. Pippenger, U. Zwick, "Faster Circuits and Shorter Formulae for Multiple Addition, Multiplication and Symmetric Boolean Functions", in Proceedings of the 31st Annual Symp. on Foundations of Comp. Science, St. Louis, Missouri, Oct. 22-24, 1990, pp. 642-650.

[4] R. Lauwereins, J. Bruck, "Efficient Implementation of a Neural Multiplier", in Proceedings of the Int. Conf. on Microelectronics for Neural Networks, Kyrill & Method Verlag, Ed. U. Ramacher, U. Ruckert, J. Nossek, Oct. 16-18, 1991, Munchen, Germany, pp.217-230.

[5] M. Engels, R. Lauwereins, J. Peperstraete, "Rapid Prototyping for DSP Systems with Multiprocessors" , IEEE Design&Test of Computers, Vol.8,No.2,June 1991,pp.52-62.