Prototyping a digital neural network System-on-Chip using an Altera Excalibur device

Lihua Ren, Peter Marshall and Steve Furber

Advanced Processor Technology Group
Department of Computer Science, The University of Manchester,
Manchester, M13 9PL, UK
{renl, marshall, sfurber}@cs.man.ac.uk

Abstract: *We present an implementation of an N-of-M code based sparse distributed memory using an Altera Excalibur device. Unlike a conventional memory, a sparse distributed memory responds not only to an exact input match, but also to any input within a specified Hamming distance of a particular input code. Therefore, it is very robust in noisy environments. The success of this prototype proves the feasibility of realising an N-of-M Kanerva memory in a system-on-a-programmable-chip, which is a good starting point for our research. In addition, it provides us with valuable design experience with the Altera Excalibur device.*

1. Introduction

In this section, we present our research objectives and general background information on basic artificial neural networks and the neural model we implemented.

1.1 Neural model

A biological neuron can be simulated by a device with many weighted inputs and a single output (figure 1). If the sum of the weighted inputs exceeds the threshold, the neuron will fire. The equation of a neural network is given as:

$$s_j = f_j\Big(\sum_i w_{ij} \cdot a_i - \theta_j \Big)$$

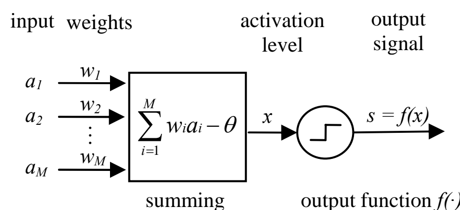

Figure 1: Model of a neuron

Here a_i is the ith input, w_{ij} is the synaptic weight connecting the ith input to the jth neuron, θ_j is the threshold of the jth neuron, f_j is the activation function and s_j is the output of the jth neuron. When a neuron is learning a particular input pattern, corresponding weights will be adjusted.

1.2 Project Objective

Our ultimate objective is to make a scalable neural network with billions of neurons in hardware. The network may be across many chips that communicate with each other. Prototyping a small-scale neural network in an FPGA at an early stage is an important step towards this goal. The implementation presented in this paper is an N-of-M coded Sparse Distributed Memory (SDM), which is based on ideas proposed in [1] developed earlier at the University of Manchester.

1.3 Kanerva's sparse distributed memory

Kanerva's SDM [2] is based on a high-dimensional binary space $\{0,1\}^n$. It stores and retrieves data by transforming a logical address into many physical addresses within a specified Hamming distance of that address. This is why the memory is 'distributed'. When a value is read, these physical addresses converge on the original stored data by applying a threshold to the sum resulting from an input address within a Hamming distance of the address that is used to store the data. When the memory is sparse and the binary space is high-dimensional, Kanerva shows that each location is far from most of the other locations, which allows a nearly-correct location to be much closer to the correct locations than to the wrong locations. Distribution and sparseness make the SDM nondeterministic and robust.

1.4 An N-of-M coded sparse memory

An N-of-M code is a possible representation of information when bundles of associated neurons cooperate to convey information, where N neurons in a bundle of M neurons fire at approximately the same time. The information content is defined by the selection of the N firing neurons. Using N-of-M codes in neural coding has many good properties, such as error recovery, power efficiency, self timing and a relatively high representational capability.

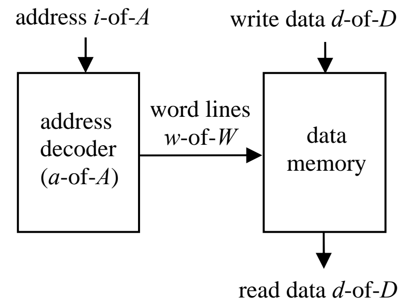

Figure 2: N-of-M Kanerva memory

The N-of-M Kanerva memory proposed in [1] is a refinement to Kanerva's SDM. It replaces his counters with simple binary weights and applies N-of-M codes to the weights and vectors of neural fascicles, making the original SDM simpler and immune to noise. As shown in figure 2, the N-of-M based SDM comprises two neural layers: an address decoder neuron layer and a data memory neuron layer. The input address and input data could be viewed as artificial layers that are external to the memory. N-of-M codes (possibly of different sizes) are employed in all neural layers (including input layers) and the distribution of weights in the address decoder layer.

The N-of-M Kanerva memory operates in either a *recall* or a *learning* mode, similar to the way that a conventional memory operates in a *read* or a *write* operation. The weights between the input address and the address decoder layers are initialised to a random *a*-of-*A* code, while the weights between the address decoder and the data memory layers are initialised to zeroes. In the learning mode, an address input, in the form of an *i*-of-*A* code, and a data input, in the form of a *d*-of-*D* code, are presented to the memory. Then the connections between the fired address decoder neurons and the activated data memory neurons are established, with the connection weights setting to '1's. In the recall mode, only an input address is presented to the memory, in the form of an *i*-of-*A* code. Processing these inputs causes *w* address decoder neurons (whose activation levels reach or exceed a given threshold) to fire; in turn these firing address decoder neurons make some data memory neurons fire, if there are at least some connections between the address decoder and the data memory neuron layers. The *d* data memory neurons with the highest activation levels are selected as output.

In our implementation, the inputs, outputs and the input weights employ 11-of-256 codes. There are 1024 address decoder neurons (*W*); on average the number of firing address decoder neurons (*w*) is around 90. The threshold for all address decoder neurons is fixed as 2.

2. Implementation platform

The Altera Excalibur EPXA1 development kit includes a development board (figure 3) and software support. The development board is shipped with an EPXA1 device, 8 Mbytes of flash memory, 32 Mbytes of SDRAM, an Ethernet MAC/PHY, two UART port connections, a JTAG connector, etc. The EPXA1 device is an FPGA or PLD (Programmable Logic Device) based SoC

Figure 3: EPXA1 Excalibur device

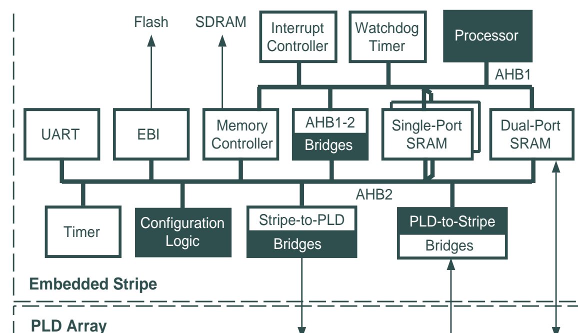

Figure 4: Excalibur device system architecture

platform; its architecture is shown in figure 4. As can be seen, the embedded stripe contains an ARM922T[TM] processor core, a memory subsystem and peripherals. The bus architecture is dual AMBA[TM] AHB buses. Interfaces between the embedded stripe and the PLD are the AHB2 bus, the Dual-Port RAM (DPRAM) and some General-Purpose I/Os (GPIOs). The device has 484 pins to communicate with off-chip devices, such as the SDRAM and the flash memory.

The system boots from the off-chip flash memory, which is programmed with an executable image using either the JTAG port or via Ethernet. Communication between the host and the embedded processor in the EPXA1 device is through a RS-232 serial port or *Telnet* via Ethernet.

3. Design Implementation

In this section, we present the design flow and the design implementation, focusing on the hardware neural-processing-element design.

3.1 Design flow

Based on the fact that the neural processing task is simple and highly repetitive, our partition strategy is to implement the neural processing task in hardware, while other parts of the system are implemented in software. The hardware design that does the neural processing task is known as a Neural Processing Element (NPE). As shown in figure 5, after partitioning, the system flow is divided into PLD and embedded software design flows. They are combined to generate a single slave binary file (**.sbi** file) for downloading to the EPXA1 device on the board. GNU tools are used to develop system and application software. The system runs the Redhat eCos operating system. Interaction between the host and the embedded software on the development board is via a custom command shell. As can be seen, the PLD design flow follows a block-based design methodology; it starts from *design entry* and ends with *place and route*, with *verification* throughout each design stage. Quartus® II software is an integrated tool environment for the complete design flow. It provides not only integrated tools for each stage in the PLD design flow, but also interfaces with a range of third party tools. A **.sbi** file can be generated after a full compilation. In the NPE design, Verilog HDL was used for *design entry*; both the integrated simulation tool and ModelSim (version 5.7a) were used for *verification* (functional and timing simulation); for the *synthesis* and *place and route* design stages, only the integrated tools (including static timing analysis tool) were used.

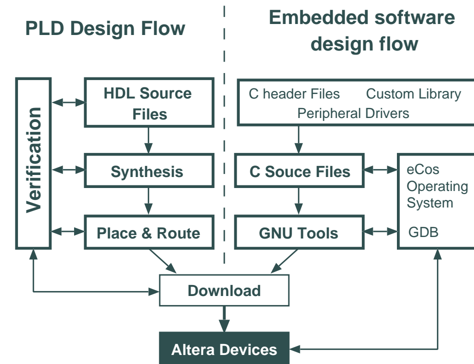

Figure 5: PLD and embedded software design flows

3.2 System architecture

Figure 6 shows the architecture of the N-of-M Kanerva memory at system level. As can be seen, software maintains a neural data structure for all neurons. Each neuron in the structure has an ID (neural number), a type (input address, input data and address decoder), a threshold (only for address decoder neurons), an activation level (only for address decoder and data memory neurons) and an output connection list (only for input address and address decoder neurons). The output connection list is a number of the IDs of the next-layer neurons that the current neuron connects to. Input weights in the neural data structure must be initialised. Before any recall session, learning is required to establish some connections between address decoder and data memory neurons. Then an input address can be written to an event queue (needs to be initialised) and the system can start processing the events in it. Each event is an ID of a neuron that fires; if it

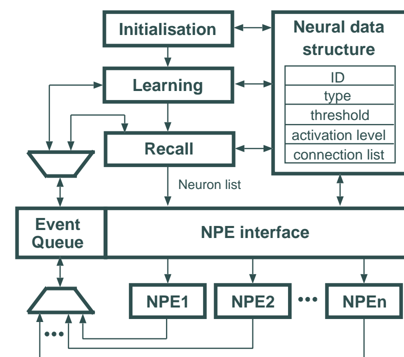

Figure 6: System architecture

connects to a list of neurons, each output neurons in the list will increase its activation level by 1 for the binary-weight implementation. Additions of the connection weights and the activation levels of the output neurons are accomplished by the hardware NPE(s), while the rest of the system is implemented in software. As shown in the figure, the event queue and the NPE interface delimit the software and hardware functions in the system. The hardware NPE(s) accesses the neural data structure directly; it reads the structure before processing and writes to it after processing. The updated activation levels are written back to the data structure; the neurons' IDs will be written to the event queue if they fire. The events in the event queue are later processed by the software and the operation is complete when the event queue is empty. As can be seen, the software implements input weight random initialisation, provision of address and data inputs, event queue processing, weight setting during learning modes, output-data selection and result-checking (the results are compared with those from a software-only version).

3.3 Design of the neural processing element

The NPE design acts as a hardware function for neural processing. In principle, the number of NPEs in the NPE design means that a number of functions get executed in parallel (although this is difficult to achieve in practice). In sections 3.3.1 and 3.3.2, we will describe two versions of the NPE design with different numbers of NPEs. As there will be millions of neurons implemented in one chip in the future, the off-chip SDRAM is used to store the weights and the neurons' activation levels. We access the SDRAM directly from PLD masters via the AHB2 bus. The event queue is implemented in DPRAM, allowing simultaneous access by both software and hardware. Signal *busy*, which shows the state of the NPE, is used as a handshake signal between the software and hardware via one of the GPIO lines.

3.3.1 Single NPE

Since there is only one neural processing element in the single NPE design (figure 7), it is relatively simple and small (15% of the total logic elements in the PLD). As shown in the figure, the processor communicates with the NPE through buses and the shared DPRAM. The software starts to write to the NPE the weight and the SDRAM address of the output neuron's parameters (activation level, threshold and neural ID) to the slave register file via the Stripe-to-PLD Bridge and the single-transaction slave interface. Then a signal (*start_read_trans*) will be generated to request the neuron's parameters (the address has been presented to both the single-read and the single-write interfaces). Once retrieved, the NPE starts computing. It adds the weight and the activation level, compares the result with the threshold to decide its firing state, writes the new activation level to the SDRAM, and writes the neuron's ID to the event queue if the neuron fires.

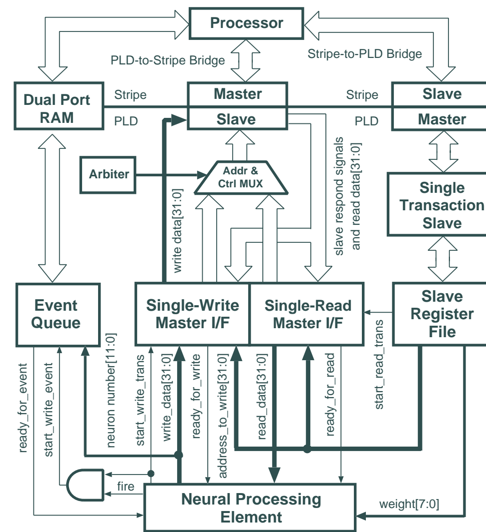

Figure 7: Single NPE block diagram

The event queue controller manages event-writing process (to the DPRAM). These events will later be read and processed by the software. Handshake signals between the NPE and the event queue controller, and between the NPE and the read and write interface, are used to control the read and write transactions. There are two bus master interfaces in the design: a single-read interface and a single-write interface. We chose *single* rather than *burst* mode for read and write transactions out of efficiency as well as making the design generic, because the connections between the input address and address decoder neurons are sparse (therefore the corresponding neurons' IDs are very likely incontiguous). Although the read and write interfaces do not operate simultaneously, we stick to arbitrating these interfaces instead of multiplexing them in order to improve reusability.

3.3.2 Multiple NPEs

Multiple neural processing elements are used in this NPE design (figure 8). The order of input executions in these NPEs is not important as the inputs to the memory are orthogonal. Due to the increased complexity, an 8-NPE design costs about 58% of the total logic elements in the PLD; for a 16-NPE design, it costs 85%.

The multiple NPEs design has a similar structure to the single NPE version. It is triggered by software via a start address (*start_address[31:0]*) and a number (*length-[3:0]*) that indicates how many processes are required in this session. The start address points to a value which encodes the weight and the address of the output neuron's activation level, threshold and ID. On triggering, the multiple NPEs design retrieves all the weights and the addresses using the given length in a burst-read operation; then, using these addresses, it again retrieves the neuron's parameters from the SDRAM in a series of single-read operations.

A dedicated control circuit is used to control the data retrieval from the SDRAM and the data writing to the SDRAM and DPRAM (indirectly). As shown in figure 9, there are four mains states in the state machine of the control circuit: *wait_phase*, *burst_read*, *single_read* and *single_write*, each with 0 to 4 sub-states. A single counter is used in all states except the *wait_phase* to control the number of read or write transactions.

The burst read interface accepts both burst read and single read transactions, controlled by the controller. A master register file is required to store the weights and the addresses of the activation levels temporarily; the addresses are needed later to retrieve the associated neurons' parameters. To save memory resources, parameters corresponding to each neuron are allocated to a NPE directly on retrieval.

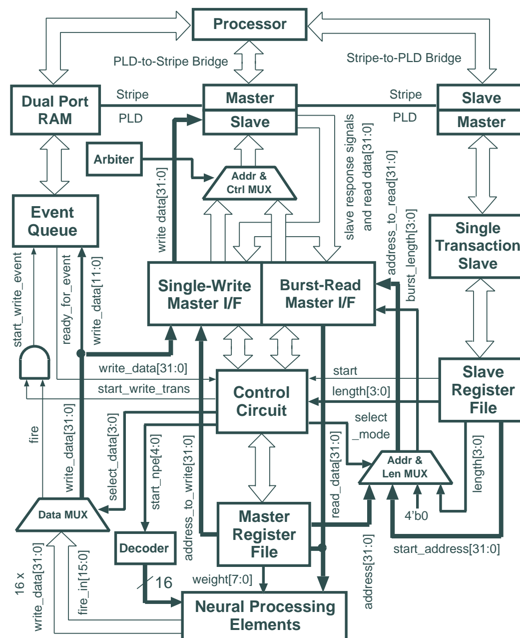

Figure 8: Multiple NPEs block diagram

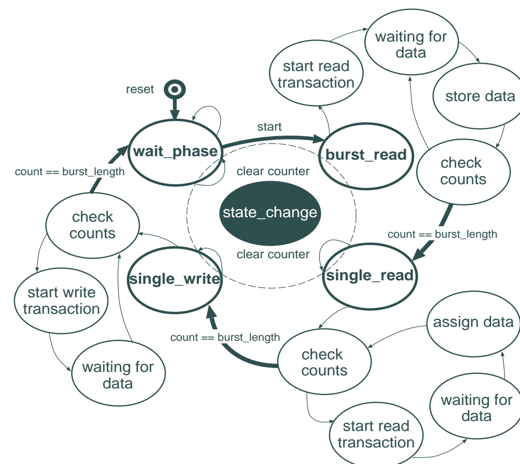

Figure 9: FSM of the control circuit

3.4 Results

Table 1 shows the performance of different versions of the NPE designs. It records the times it takes for both software and hardware versions of the design given the same task. The hardware version has three variants: single NPE, 8 NPEs and 16 NPEs; each runs across a range of clock frequencies. The software version has two cases recorded in terms of the state of the cache.

As can be seen, performance does not improve much as the number of NPEs increases. There are a number of reasons. Firstly, although there is parallel hardware, SDRAM access is still serial. The single-read and single-write transactions are inefficient – although initially a burst-read transaction is used to retrieve the weights and the addresses of the output neuron's parameters, the inefficiency caused by the single-read and single-write transactions outranks the benefits brought by the burst-read operation. Secondly, the single NPE design runs at up to 80MHz, while the

Table 1: NPE design performance

Hardware versions (cache on)							
Clock Speed (MHz)	25	40	50	55	60	70	80
1_NPE (ms)	236*	244*	223*	251*	225*	191	185
8_NPEs (ms)	266	219	208	202	196	-	-
16_NPEs (ms)	257	211	197	193	-	-	-
Software versions							
Cache On (ms)	172						
Cache Off (ms)	1645						

Note: Data marked with a '*' are measured with an added instruction to the software. The content of the instruction is not important; while the delay it brings to the software matters. The software needs to be delayed at the point of triggering the NPE design, before checking the value of the busy bit; since the bridges and clock synchronisation between the AHB2 and the PLD logic introduce delays between the software writing to hardware and hardware asserting the *busy* bit.

8-NPE version of the design can only run up to 60MHz, and the 16-NPE version, 55MHz. This results from fitting a design with increased complexity to a target device with limited resources (especially routing resources). Thirdly, the multiple NPEs versions have the overhead of a burst read to retrieve connection weights and addresses (they access the SDRAM three times in an add-compare operation; while only two single accesses are needed for the single NPE version). Lastly, more tasks are accommodated in software in the multiple NPEs versions, in order to avoid the 1k-address-boundary violations specified in the AMBA bus protocol. As a result, the single NPE design, running at 80MHz, has the best performance among different versions of the hardware NPE design.

When the cache is enabled, the software version of the system performs about 7% better than the fastest hardware version (172 vs. 185ms). The reasons are as follows. Firstly, the software runs at 160MHz, twice the clock speed of the hardware design in the PLD. Secondly, the software runs in an ARM922T™ processor; its cache system (64-way set associative Harvard cache architecture, allowing simultaneous instruction and data access) dramatically improves the performance (nearly 10 times). Lastly, the software and hardware in the system run serially instead of concurrently; this results from software/hardware partition and will be solved naturally as more functions are implemented in hardware.

4. Conclusions

An implementation of an N-of-M sparse distributed memory in an Altera Excalibur device has been presented. Although the performance is just comparable with the software version, the success of the system proves the feasibility of realising a digital neural network in a system-on-a-programmable-chip. As more functions are shifted to hardware, the system can be optimised as a whole (compared to an isolated NPE design) due to the possibility of a full mapping between system functions and available resources. As can be seen from the multiple NPEs design, system performance will not be improved noticeably by just increasing the number of NPEs. In order to achieve a better performance, memory bandwidth must be increased (by utilising burst-read and burst-write operations) to enable parallelism and pipelining in hardware.

The neural research project provides us valuable opportunities to use the Altera Excalibur device. Although considerable time has been spent on familiarisation with the device and the tools, once done, the design process is quite straightforward.

5. Acknowledgements

The authors gratefully acknowledge the funding from Cogniscience Ltd. and the support from Altera.

6. References

[1] S.B. Furber, W.J. Bainbridge, J.M. Cumpstey and S. Temple, "Sparse Distributed Memory using N-of-M Codes", submitted to *Neural Networks*.

[2] Pentti Kanerva, Sparse Distributed Memory, MIT Press, ISBN 0262111322, 1988

[3] Altera website: http//www.altera.com/