Natural learning of neural networks by reconfiguration

L. Spaanenburg[a], R. Alberts[a], C.H. Slump[b] and B.J. vanderZwaag[b]
[a] Lund University, P.O. Box 118, 22100 Lund (Sweden)
[b] Twente University, P.O.Box 217, 7500 AE Enschede (The Netherlands)

ABSTRACT

The communicational and computational demands of neural networks are hard to satisfy in a digital technology. Temporal computing addresses this problem by iteration, but leaves a slow network. Spatial computing only became an option with the coming of modern FPGA devices. The paper provides two examples. First the balance between area and time is discussed on the realization of a modular feed-forward network. Second, the design of real-time image processing through a Cellular Neural Network is treated. In both examples, reconfiguration can be applied to provide for a natural and transparent support of learning.

Keywords: Field-Programmable Gate-Array, Reconfiguration, Modularity, Feed-Forward Neural Network, Cellular Neural Network. Temporal computing, Spatial Computing, Wave Computing.

1. INTRODUCTION

The Field-Programmable Gate Array (FPGA) has matured over the past years from an in-product personalization of Gate Arrays to a carrier of innovative computing styles. The initial gate array contained a prefabricated collection of transistors and low-level wire segments, that could be personalized by a last series of contact and interconnect fabrication steps. Often support was given from a library of final masks for logic cells. The main purpose was to bring the lumped logic elements of a computing architecture into a single container and thereby save valuable board space. With the advance of microelectronic technology, the capacity of the gate array increased to a level on which it could even contain entire application specific circuits, the ASIC. Still, personalization was performed at the foundry and, though the amount of prefabrication brought some of the cost benefits of mass production, a product series was required to make the concept effective.

With a further increase in capacity, it became acceptable to introduce memory elements into the array architecture. A logic function was softened by table lookup, while a signal path was established by the content of the attached memory element: the configuration bit. Writing such configuration bits into the on-chip configuration memory performed personalization of the ASIC: a process that can easily be performed by the application builder. The monolithic device architecture required programming of all memory locations in succession. The connected time penalty made on-the-fly reconfiguration still a dream.

Recent FPGA devices show a seemingly small but important change. With the still continuing increase in capacity, isochronity could not be maintained anymore. The architecture had to be divided into smaller clocking regions or modules. Making such regions separately addressable allows configuring a single module while using the others: reconfiguration on-the-fly. The effect is a hardware programming technology in the hands of the platform user, similar in scope to software programming[1]. But allowing every element of the ASIC to be fully programmable is both not needed and desired. It is not needed because system design has matured in abstraction, basing the design on more complex primitives. It is not desired because of the implied overhead in execution speed, area usage and power consumption. In this respect, the meaning of optimality and efficiency has changed, as the FPGA does not have to be fully used. Where full-size SRAM and multipliers are available as area efficient macros next to the more forgiving CLBs with their lumped logic primitives, it is imaginable that the design technology has to be adjusted.

André deHon has posed that the archetypical stage of FPGA-based design was characterized by the strict limitation on hardware resources. This made it necessary to use every hardware element as much as possible. The popular way to achieve this goal is by unraveling in time: the computational process is scheduled to execute in order on the few computational elements. This is called "temporal computing" in contrast to "spatial computing", where the process is unraveled in area to reduce any latency[2]. The facility of spatial computing makes the FPGA already very popular as hardware accelerator. The other innovation of partial reconfiguration of hard-wired modules as additional level of programming has been lesser utilized, though the potential benefit was already illustrated early on[3].

In this paper we will illustrate the potential of programming by reconfiguration while discussing two FPGA realizations of a neural network. Digital neural hardware has not enjoyed much popularity. The many synapses define a complex wiring scheme, which can only be simplified by temporal encoding[4] or multiplexing[5]. Further the sheer size of the multiplier, on which the synapse is built, is a concern, that seems to necessitate for temporal iteration on a limited amount of resources. It has been suggested in the past, that a spatial computing style would be appropriate. Unfortunately, microelectronic technology could only support such a realization as a board of single neuron ASICs[6]. But time has passed and meanwhile hardware complexity has become a lesser restriction. Recent FPGA architectures provide a large amount of Configurable Logic Blocks with interspersed optimized RAM and multiplier macro's, which can be efficiently utilized to map modular neural structures.

Feed-forward neural networks have become widespread because of their seeming simplicity and ease of use. Monolithic realizations tend to become instable in learning with increasing size. Large dataset statistics based on clustering and averaging induce a catastrophic cancellation in the neural network. It is found that for large datasets a modular arrangement of many small nets (the multi net) is to be preferred. Training a multi net is often a locally focused operation. A typical system is composed of a number of abstraction layers that range from basic to applied knowledge[7]. Different applications will differentiate in the mixture of basic knowledge only. Where the synaptic multiplication is conventionally a major hindrance to digital implementations of neural networks, adaptation in multi-nets can by its local nature be realized through configuration. This natural learning scheme spurs the computational efficiency of knowledge systems implemented on FPGA for such applications as process control and visual diagnostics. Such a multi-net can effectively be implemented on a per module basis on a Virtex-II chip[8].

Locally connected networks have also received considerable interest. In 1988 Chua and Yang introduced the cellular neural network (CNN) as a novel class of information processing systems[9]. These systems are particularly suited for solving problems defined in space like image processing tasks and partial differential equations. Such problems are often characterized by the fact that the information necessary to compute the solution at a certain point in space is within a finite distance to that point. An example is edge extraction for digital images. Whether or not a certain pixel belongs to an edge depends only on the color of the pixels that are in the neighborhood of that pixel. Like a cellular automaton[10], a CNN is made of a regularly spaced grid of processing units (cells) that only communicate directly with cells in the neighborhood. This implies that cells are connected locally, which distinguishes CNNs from other neural architectures like the Hopfield network[11] and Kohonens' Self-Organized Feature-Mapping (SOFM) algorithm[12]. After the introduction of the Chua and Yang network, a large number of CNN models have appeared in literature. Harrer and Nossek have introduced the discrete-time version DT-CNN[13]. Despite the first-hand advantages for VLSI implementation, current hardware is either in analog technology or emulated on a standard DSP platform. Here, we show that an attractive mapping on a Virtex-II chip is possible.

The paper is structured along the following lines. In section 2, it is reviewed how a feed-forward neural network can be mapped on an FPGA. The design style is based on a logic-enhanced memory architecture with interspersed complex macro facilities, such as the Xilinx Virtex-II, but not necessarily confined to that. Spatial computing is enabled by the modular structure of the network, while vice versa partial configuration needs such a modularity to be cost effective. In section 3, we change over to the CNN as an example of cellular structures. The design style is based on the systolic array concepts as introduced by Kung at the start of the VLSI era[14]. Though the basic CNN cell is quite complex, the space requirements seem quite in balance with the module capacity featured by the Virtex-II. As the local computations can be made more efficient by a suitable compaction of the computational template[15], the FPGA realization promises a performance as desired for the future wave computers[16]. Subsequently we discuss in more depth the role of reconfiguration in support of the multi-dimensional programming needs of neural networks. Where, in conventional technology, learning is a separate phase, reconfiguration allows changing the network

settings on the fly, making the adaptation transparent to the network application. We conclude that the new generation of FPGA devices is extremely well suited to explore new programming paradigms on a flexible hardware platform.

2 FEED-FORWARD NETWORK

The most widespread implementation of neural functionality is by a feed-forward network. It is a network of simple identical nodes, in which the inputs are transported in one direction to the outputs. The connections between the nodes are manipulated by weights with values that are learned from applying and supervising a set of examples. The simplest realization of a neuron with incoming synapses is based on a multiplying adder. Many simultaneously active neurons can be imitated by executing the single instance for the many signal settings. Such different settings can be allocated in SRAM macros (Figure 1). In a typical FPGA architecture, the chip is divided into super blocks. For the purpose of the later discussion, we have shaped the neural module on a super block such that the SRAM macros are shared between consecutive networks to pass the values.

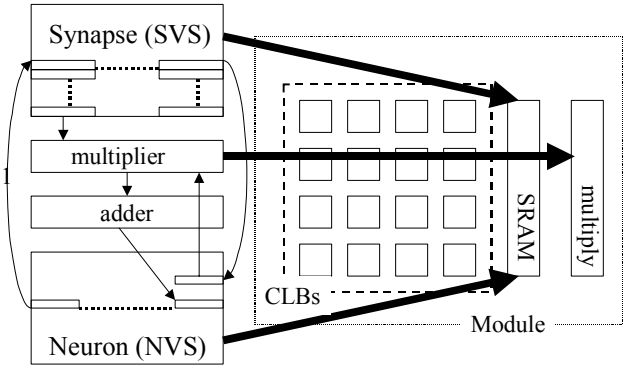

Figure 1 The neural implementation model mapped on an FPGA superblock.

The Neuron Value Store NVS contains the axon value and the start address in the Synapse Value Store SVS where the incoming synapses are administrated. The SVS contains the NVS address of the sourcing neuron, the weight value and the address of the next synapse. The bias can be either a value for a neuron or a synapse with no source. Model execution proceeds neuron by neuron, ordered from network input to output. For every neuron, the sourcing neurons are scanned over the incoming synapses to calculate the new output value. For a further detailed description of the function, see [17].

The FPGA 18k bit dual-port SRAM can be generated in various depth and width configurations. The monolithic neural network is a weighted connection of neurons with a characteristic transfer function. Design considerations of the network parameters, as described in [18], allow the values to be limited to 8 bits. Figure 2 gives an example from an experiment in character recognition of vehicle license plates. Shown are the effects of input redundancy and learning rates diversity on the degree by which weight accuracy influences the learning error. But also the transfer function is a design item. It can be explicitly imposed or locally created from a small sub-network. For instance, a sub-network of neurons with linear transfer can behave as a single neuron with a sigmoid transfer. In this sense, the monolithic neural network is already a modular one in disguise and one may therefore expect that the learning problems will be the same. From the observation that, with growing problem size, the monolithic network has increasing difficulty to learn with sufficient quality[19], one may expect not better from a multi-net.. The structural information on normal-size networks can then be stored in maximal 100 words of 16 bits, leaving ample room for supporting functions. For instance, the universally acclaimed sigmoid transfer is ineffective for the approximation of abrupt functions. When

the complexity of the circuitry to calculate the transfer is a hindrance, it is possible to keep the transfer function programmable in a lookup table. Such parameters are clear candidates for inclusion in the module SRAM. The choice is not only technology dependent, but also shows how a network can be replaced by its function table once learning has proven enough stability.

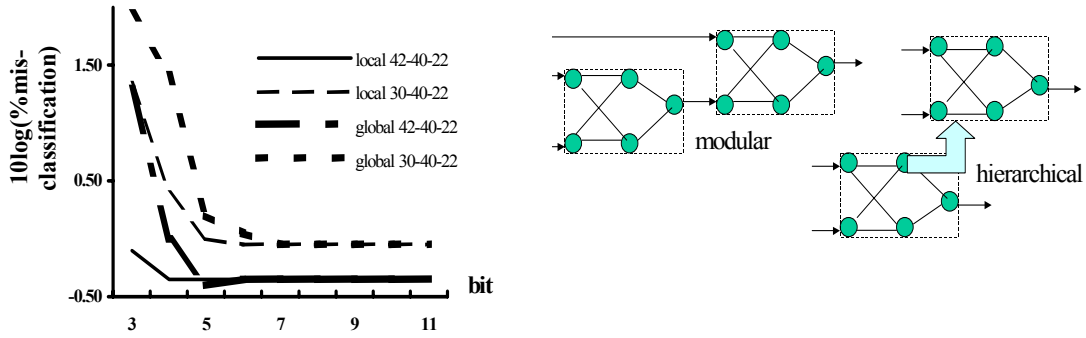

Figure 2 Structural composition (right) influences learning accuracy (left)

The best curve is obtained by input redundancy combined with separate learning speeds for the 1st and the 2nd neuron layer. Such differences come to bear when constructing a network by sub-networks (Figure 2) in hierarchy or modularity and are therefore the key to structured design techniques. Modular neural networks or more commonly called multi-nets are combinations of several neural networks[20]. They are of growing interest, as the implied feature redundancy is believed to make the overall net more accurate than the parts. Moreover, multi-nets can be easier to understand and to modify.

In both cases, the learning process suffers from the entropy in the example set. This can only be resolved (a) by data preprocessing, (b) by inclusion of pre-knowledge or (c) by domain structuring. Such can be achieved by using modular networks[21]. The claim that more than 80% of the development time for monolithic networks is spent on the data preprocessing underlines this observation[22]. A modular network can be interpreted as a multi-layer hierarchical network where ultimately on the highest level the weights are constant and equal to one (Figure 2). In other words, the top modular composition has lost its exclusive neural outlook and has become heterogeneous in nature by allowing for components of any fabric. By the expansion to multiple layers, and adding weights on the connections between networks, hierarchy is enabled as depicted in Figure 2.

Hierarchical networks go one step further in compositional sense. Each node in a neural network may be again a neural network[23], or a specialized function. This means that a neural network implements the evaluation function of the node, thereby preserving the weights on its inputs from the upper layer. An example is the fuzzyfication of singleton input variables, who would otherwise cause a classification problem. Similar to both types of networks is that both hierarchical and modular networks apply functional specialization, although in a different form. Specialization enables the fusion of existing knowledge into the neural network, as was shown for modular networks in [24]. In Figure 3, the knowledge about the operation of a float-glass furnace was collected as a set of small rule blocks. Subsequently each rule block was transformed into a neural module and then the modules were combined into a neural network. This example is later used for a spatial implementation.

The temporal design of a module contains already all the necessary ingredients for neural data processing. Scaling can easily be achieved by increasing the network representation within the SRAM, but this will soon lead to unwieldy long execution times. This was also the major hindrance for digital realizations of neural networks in the past. The alternative is the replication of the elementary module over the chip. In the past this was no option but with the coming of groundbreaking FPGA devices like the Xilinx Virtex-II family the option becomes very real. Whether such time unraveling can be utilized depends on other circumstances. For instance, when the network serves as a behavioral test generator and verifier for another design, it should be small in order to limit the overhead; for other purposes speed rather than footprint may be a major concern. Clearly this demands a fair degree of architectural scalability.

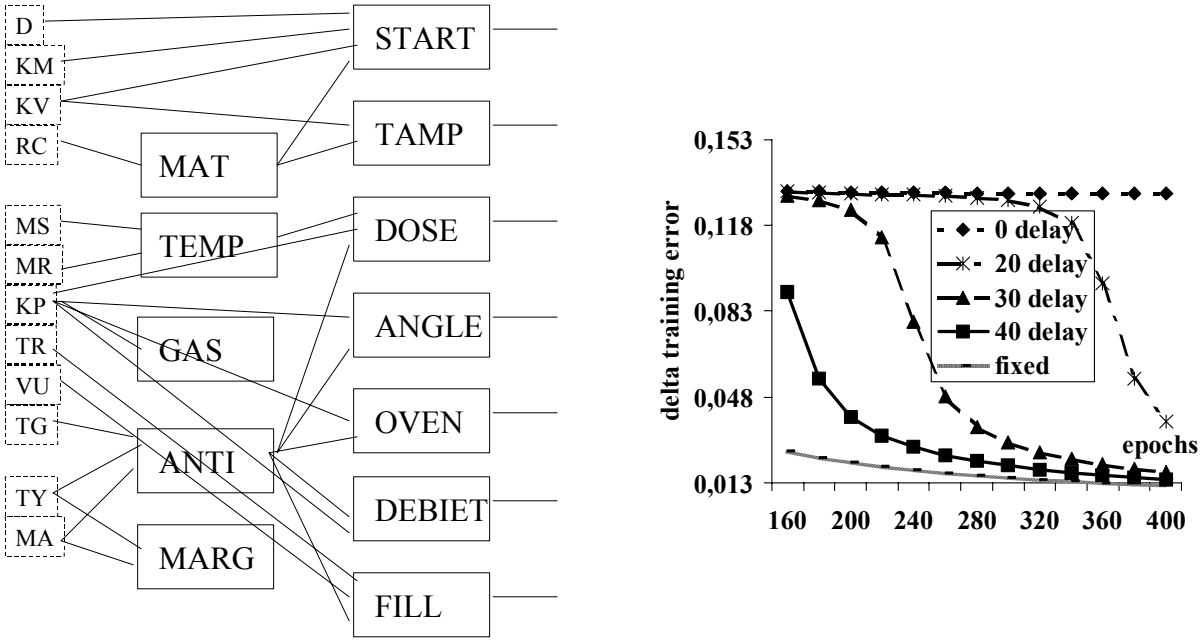

Figure 3 Typical rule set (left) and the impact of delayed block activation (right).

The added advantage of spatial computing is the opportunity to learn the modules of a neural network almost in parallel. This was first noted in [25] in the analysis of the unlearning potential of neural composition. When a network is assembled from trained and empty modules, it frequently happens that the inserted knowledge is swept away during the first epochs and the network continues as if nothing had been there. The remedy has proven to be a time ordering of the activation time of the individual modules. A simple example is given in Figure 3. The original circuit was next to impossible to learn, but already small delays between the activation of the modules brings learning time back to acceptable properties, wherein the overall network is trained in just slightly more time than a single module. As all inputs and outputs of the modules are handled by using the SRAM, passing information between modules views the SRAM as a blackboard. It is not necessary to signal new events by semaphores as a neural network is robust enough to allow for the occasional mixture of old and new values[26].

Such considerations make for a compact arrangement by mere concatenation of the modules. Figure 4 shows the floor plan of such a spatially unrolled neural network, executing the knowledge of Figure 3. The design is behaviorally constructed in VHDL using Xilinx WebPACK 4.2 and simulated by ModelSim XE5.5. The Xilinx Core Generator allows generating the multiplier and the RAM as facilitated by the chip. Then the logic synthesizer creates a mapping on the CLB's. Some manual intervention is required to confine the Place & Route to the envisaged area. Overall this results in the design shown in Figure 4, that uses 83 % of the available flip flops and 57 % of the available LUT's.

The only real variable is the RAM usage, which in turn relates to the network size. When only a small part of the FPGA is available for the neural network (as when the network is a drop-in on a different design), the network can only be unraveled to a limited degree and the RAM is heavily utilized. When more space is available, the RAM is freed for other purposes. This leads to using the average speed per line of RAM code (loc) as Figure of Merit. For our design this leads to 40 ns/loc. For a temporal design this number would be a constant, but for the spatial design the number of modules in the longest path divides the number. This leaves of course the latency of the design. When compared to a temporal software design on a Pentium-III, the acceleration is by a factor 20, as earlier reported in [1].

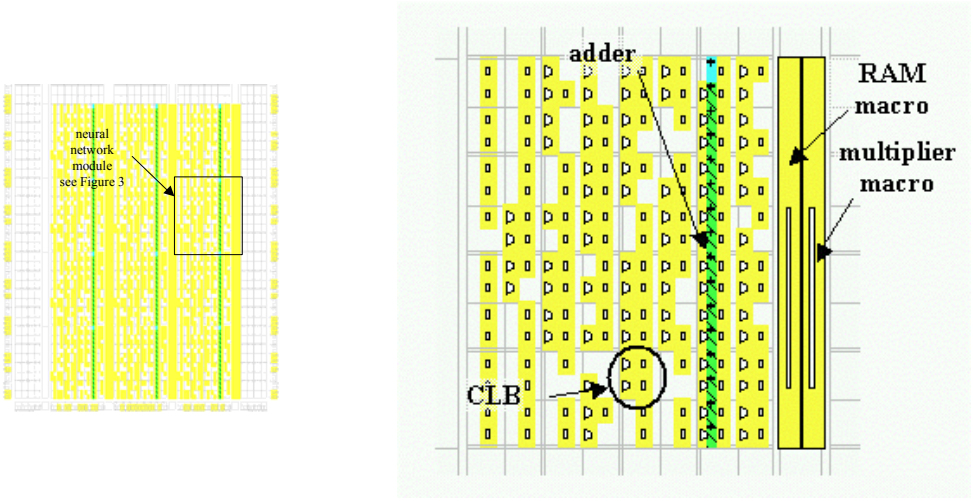

Figure 4 Floorplan of a spatial neural network (left) and a single module (right)

3. CELLULAIR NEURAL NETWORK

A DT-CNN presents a regular grid of locally connected cells. Theoretically, this grid can have any dimension. Focusing on image processing, we will restrict ourselves to the two-dimensional case, in which the cells are organized in a rectangular grid and in which each cell corresponds to a pixel in the image. A cell is identified by its position in the grid and will be denoted by the corresponding coordinate $c \quad c_r, c_c \quad Z^2$, where C_r denotes the row and C_c denotes the column (Figure 5a).

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

(a)

Cu	Bu	Au
C	B	A
Cd	Bd	Ad

(b)

G	F	E	D	C	B	A

(c)

Figure 5 Numbering of CNN cells (a), pixels (c) and in combination (b)

Though a specific CNN cell communicates only with its neighbors, it is part of an overall network. If the network is not as large as the image, the image will have to be partitioned. Commonly the network scans in stripes over the image, or, rather, the image passes in stripes over the network. To simplify the discussion on the CNN image processing system, we assume that the sequence of pixels are lexicographically numbered: A, B, C, and so on (Figure 5c). Bringing the cell numbering and the pixel numbering together, we come to a representation of pixel triplets, where pixel A and C are neighboring to B but so are the upper Bu and the lower Bd (Figure 5b).

The propagation effects of the network dynamics enable communication with neurons outside the r-neighborhood. The network dynamics of a CY-CNN is described by a set of differential equations. The DT-CNN is a clocked system, whose dynamical behavior is described by a set of discrete state equations. At a discrete time k, the state x^c

of a cell c depends on the time-independent input u^d applied to its neighboring cells d and the time-variant outputs $y^d(k)$. The cell state equation is given by $x^c\ k \quad a_d^c y^d\ k \quad b_d^c u^d \quad i^c$, where the r-neighborhood is defined as $N_r\ c \quad d \quad Z^2 \left| \max \left| d_x \quad c_x \right|, \left| d_y \quad c_y \right| \right. \quad r$.

The real-valued coefficients a_d^c are called the feedback coefficients and determine how the state of c depends on the output of its neighboring cells. Similarly, the control coefficients b_d^c describe how the state of a cell depends on the input of its neighbors. For each cell, a real valued cell bias i^c is added to adjust its threshold. the functionality of the complete network is defined by two (2r+1)x(2r+1) matrices A and B and a cell bias i, such that $a_d^c \quad A_{c\ d}$, $b_d^c \quad B_{c\ d}$, $i^c \quad i$. Matrix A and B are called the feedback template and the control template respectively. Substituting this into the former equation gives the space-invariant state equation $x^c\ k = \sum_{d \in N_r\ c} A_{d-c} y^d\ k + \sum_{d \in N_r\ k} B_{d-c} u^d + i$.

For image processing purposes, the real-valued input range of a neuron is restricted to [-1, +1]. A value of -1 represents a white pixel, a value of +1 represents a black pixel, and all other values represent gray levels in-between. As mentioned above, the input of a single neuron is denoted by u^c. Similarly, the input of all neurons in the network is denoted by u. In fact, u is a function $Z^2 \rightarrow$[-1, +1]. Such a function represents a gray-scale image in the DT-CNN domain and will be called an activation pattern.

According to this equation, the functionality of a DT-CNN is completely defined by the control template A, the feedback template B and the cell bias i. The triple T=<A, B, i> is called the (cloning) template and is often thought of as an elementary DT-CNN program or a DT-CNN instruction. Together with the input activation pattern u and the initial output y(0), the template completely determines the dynamic behavior of the system. Solving a particular image processing problem therefore consists of finding the appropriate values for u, y(0), the template T, and the number of required network iterations n. As u, y(0) and T are real-valued, literature seems to prefer a floating-point representation. But fixed-point is more hardware friendly and quite sufficient in view of the auto-normalization in CNN nodes.

The design is targeted on the Xilinx Virtex-II 6000 and beyond. This provides room for 24 (6000) or 28 (8000) rows and 6 columns of modules. Every row calculates one pixel and every column makes one iteration. The constant U matrix and the iterating Y matrix are “alternating” to enter the FPGA. First U_1 enters the FPGA and in the next time cycle Y_1 enters the FPGA (simultaneously with U_2). This will make it possible for the image to be iterated five times before it exits the FPGA. If the image doesn’t stabilize after four iterations, the image must be further processed or considered as finished after the fifth iteration. If the image has changed between iteration four and iteration five, we do not know whether the image is stable or not. If more iteration is needed (the image isn’t stable), there are three options:

1. The pipeline will pause and back one step
2. The image will be reentered in the FPGA and pass for five new iterations
3. The image will be considered as finished anyway.

It’s fully possible to back the pipeline one step since all pipeline steps are equal, but it might need more logic than with other alternatives. Also the last value that exited the FPGA must be remembered if it shows that it will be necessary to back up. To reenter the image causes problem since the image is “constantly floating” through the FPGA. It must be decided from which point the image must be reentered. This leaves the last option as the most realistic for the moment.

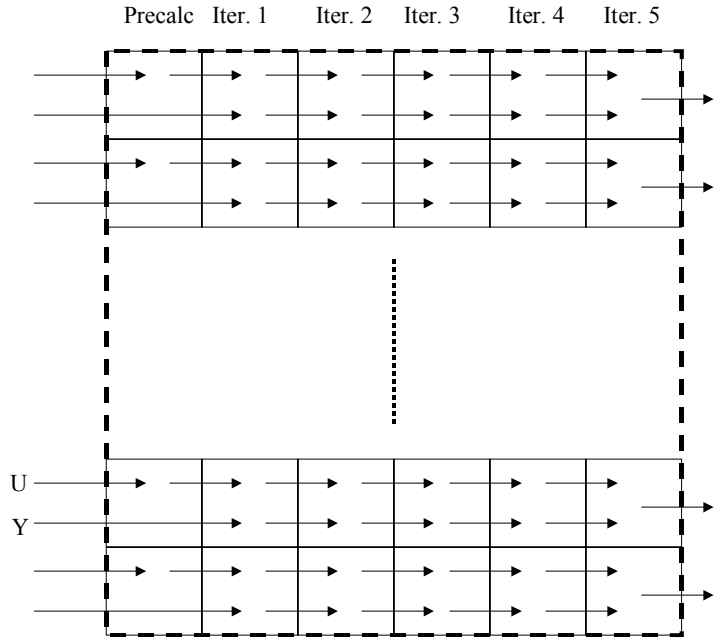

Figure 6 Dataflow of U- and Y-matrix.

There are many ways the build the pipeline. We are not bounded to build a 5-stage pipeline. For instance, it could also be 6 stages (using the last one to validate stabilization) if the U matrix is calculated on another part of the FPGA. This would reduce the throughput from 28 to 22 (or 24 to 18) pixels. (Six module rows on the chip are used for calculating the U matrix.) This will give us correct images that stabilize in five iterations

One *time cycle* is defined to be the time for one matrix to be multiplied with and summed with correct pixel values. Every pipeline stage remembers the last 2 values from pixels that passed by. The rest of the nine needed values (six values) are stored in the module directly above or the module directly below. A^0 is pixel A in iteration 0. Three values are stored in every pipeline stage. The arrows indicate which values are used to calculate which values. Now the FPGA will calculate 28 (or 24 for the 6000) pixels every time cycle with a latency of five time cycles minimum.

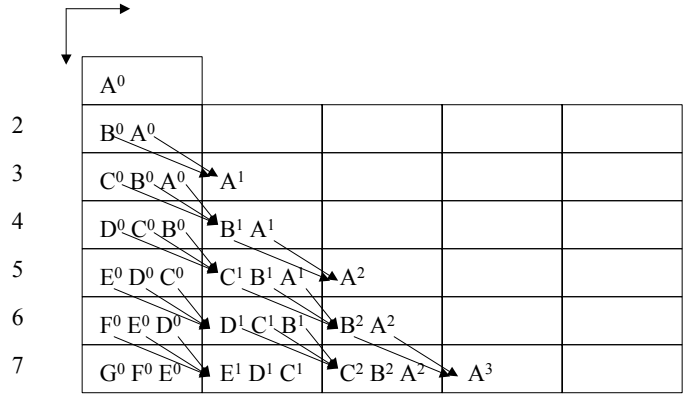

Figure 7 Fundamental systolic operation

As shown in Figure 7, the saved data in each module (3 pixels) are shifted to the right within the module. This is not taken into consideration in Figure 8; instead the address to this data is counting. One time cycle takes 13 clock cycles. The pixel line is defined with index U and the pixel line below is defined with index L.

Cycle	Cell R	Cell W	Mem R	Mem W	MAC A	MAC B	Multiply	Add
n.1	C-U	C→down	B	-	C	B(1,0)	P=C(n+1)*B(1,0)	S=P
n.2	B-U	B→down	A	-	C-U	B(0,0)	P=C-U*B(0,0)	S=S+P
n.3	A-U	A→down	C	-	B-U	B(0,1)	P=B-U*B(0,1)	S=S+P
n.4	C-D	C→up	B	-	A-U	B(0,2)	P=A-U*B(0,2)	S=S+P
n.5	B-D	B→up	A	-	C-D	B(2,0)	P=C-D*B(2,0)	S=S+P
n.6	A-D	A→up	-	-	B-D	B(2,1)	P=B-D*B(2,1)	S=S+P
n.7	-	-	B	-	A-D	B(2,2)	P=A-D*B(2,2)	S=S+P
n.8	C	-	A	-	B	B(1,1)	P=B*B(1,1)	S=S+P
n.9	-	-	-	C	A	B(1,2)	P=A*B(1,2)	S=S+P
n.10	-	-	-	-	0	0	P=0*0	S=S+P
n.11	K_C	k-B→right	-	k-C	0	0	P=0*0	S=S+P
n.12	-	-	-	-	0	0	P=0*0	S=S+P
n.13	Y(0)	B(n+1)→right	C	-	*I-C	1	P=i-C*0	S=S+P

Figure 8 Pipeline for constant initialization

The design of the CNN-based wave computer has a number of parameters that are still open to further optimization. In a project class "VLSI Design" some of such variations have been experimented with. In Figure 9 we show here as a proof of concept the floorplan of the Wickie computer. The multipliers and Block Select RAMs are organized in 6 columns and 24 rows. Therefore we can process image stripes that are 24 pixels wide, in a pipeline of 6 stages. The first contain the calculation of the U and B matrices to a constant that is propagated through the pipeline to the other stages. The five remaining stages operate on the Y and A matrices. Each stage represents one iteration on Y, yielding five iteration in total. Each pipeline stage consists of additions and multiplications, propagating and receiving 2 values and a table lookup grayscale threshold, in total 13 clock cycles. The pipelined design allows most neighboring values to be accessed locally from Block Select RAM. Thus, the need to propagate values is limited to only two cells, the ones directly above and below.

Handling 24 pixels per stage at an internal clock speed of 420 MHz requires an external data transfer rate of 30*24=720 MHz. This is obviously too much for most of the popular experimentation boards. Consequently we have to slow the system down by on chip series/parallel conversion, which still gives a comfortable camera-compliant speed of 125 frames/s. A further requirement for a comfortable wave computer is the on-line synthesis of efficient template as described in [15].

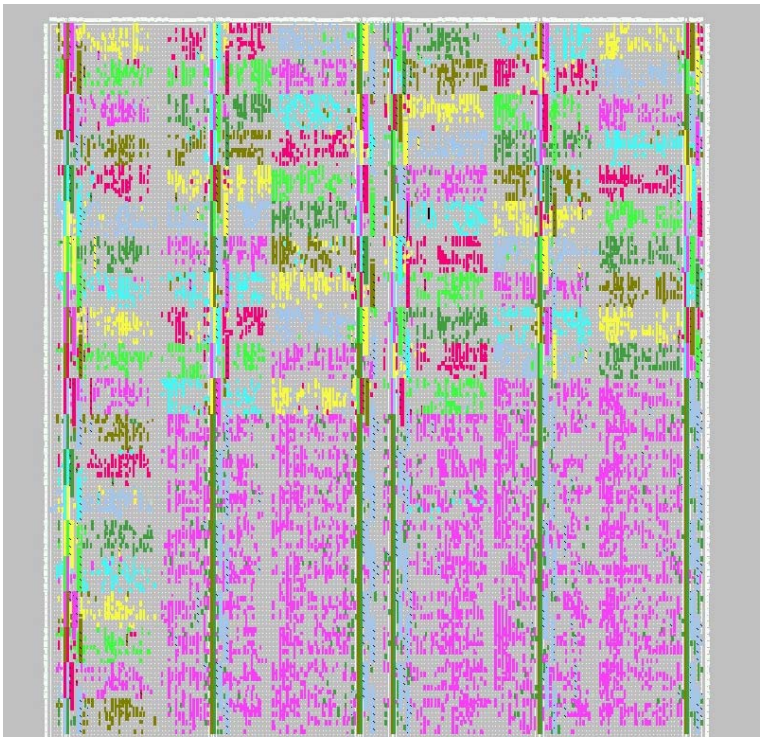

Figure 9 Floorplan of the WICKIE CNN wave computer mapped on a Virtex-II 6000

4 LEARNING BY RECONFIGURATION

The Field-Programmable Gate-Array receives its settings from an external store. A sequence of configuration bits is shifted into location in a similar way as test bits are placed via a scan path. The analogy goes even further. More complex designs lead to longer scan paths, making testing prohibitively more difficult. Along the same line, the more complex FPGA needs longer time to configure. Clearly, such a scheme can only be used as part of a bootstrap mechanism. The solution for testing is found in the utilization of internal structural modularity: the scan path can be applied to only an addressed module within the overall system. Along similar lines of thought, the modern FPGA uses internal structural modularity: the configuration can be applied to only an addressed module within the overall system. And the analogy goes even further, as the configuration process uses the same JTAG protocol as testing.

The advent of partial configuration (and therefore of in-line reconfiguration) poses a fundamental design question: to base the architecture on the movement of data or on the movement of functions. In the conventional technology, data is moved from logic to logic along the data path. Partial configuration allows leaving the data in local storage while reconfiguring the attached logic. A typical example is in the feed-forward neural network. Having the signal parameters of a neural node in the local SRAM, the structural definition of the neuron can be reconfigured to the needs of the next layer. As shown in Figure 2, such a local adaptation brings sizeable benefits.

Where the conventional FPGA development environment is targeted to hardware acceleration, there is little support for partial reconfiguration. The configuration bits are simply loaded on the chip at the start of a process. For partial reconfiguration, it is required that during the process the FPGA can actively retrieve new configuration bits from an external store. A typical set-up is shown in Figure 10. It is based on an on-chip Reconfiguration Management System (RMS) that maps the address of the module to be reconfigured together with a version number of the reconfiguration on the external Configuration Bit Store (CBS).

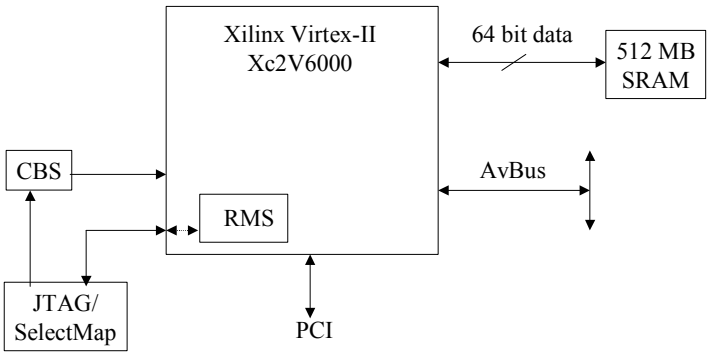

Figure 10 Active Reconfiguration Management

Along similar lines, the Cellular Neural Network can be made more versatile. In general, the CNN is designed rather than learned, as pre-designed templates take the role of the weight values in the trainable feed-forward network. In our architecture of the wave computer, such templates are presented flowing together with the pixels through the pipeline. The alternative is to configure the cells according to the templates. This need may occur due to the fixed length of the pipeline, which may require the effect of the template to be evened out over the pipeline depth in order to reach full utilization. Reconfiguration can also be applied to bring eventual pre-calculated U matrices in place. Furthermore, the discrimination of the gray values still requires a degree of adaptation.

5 DISCUSSION

The design is based on the Xilinx Virtex-II product family as it provides the required mix of macros and CLBs. These FPGAs come in various sizes. Intuitively the number of super blocks per FPGA imposes a maximum to the number of neural modules. This is not true as reconfiguration can also be used. The design presented here is based on programming with all the personalization data in the RAM macros. Dedicated networks can be smaller as the weights are not stored but hardwired etc.. Such networks can be reconfigured within the area spanned by the programmable version. Ultimately this leads to heterogeneous networks, where parts are configured and parts are programmed.

But reconfiguration seems to bring temporal computing back into action. This is not entirely true as setting new configuration tags for a limited area happens within nanoseconds, while the single run through a module takes microseconds. The reconfiguration of a module is a mere nuisance while the other modules can continue operation. In other words, reconfiguration allows the network to grow over the size of the FPGA.

ACKNOWLEDGEMENT

The FPGA design is performed at the Dept. of Information Technology of Lund University (Lund). Our gratitude goes to the students of the Project VLSI Design: Zalan Blenessy, Irina Fältman, Marcus Hast, Johan Hoberg, Tory Li, Andreas Lundgren, Suleyman Malki, Erik Montnémery, Sebastian Rasmussen, Anders Rångevall, Johannes Sandvall, Tor Silfverberg, and Milan Stamenkovic.

REFERENCES

1. A. de Hon, *Reconfigurable Architectures for General-Purpose Computing*, AI Techn. Rpt 1586, MIT, Cambridge (USA), 1996
2. A. DeHon, *Reconfigurable Architectures for General-Purpose Computing*, Ph.D. Thesis, MIT, Cambridge (USA), 1996.
3. J. Villasenor, C. Jones, and B. Schoner, "Video Communications Using Rapidly Reconfigurable Hardware," *IEEE Transactions on Circuits and Systems for Video Processing*, **5**, 565-567, December 1995.
4. A. Jahnke, U. Roth and T. Schoenauer, "Digital Simulation of Spiking Neural Networks", *Pulsed Neural Networks*, W. Maas and C.M. Bishop, chapter 9, MIT Press, Cambridge (USA), 1998.
5. P. Richert et al., "ASICs for prototyping with pulse-density modulated neural networks", *VLSI Design of Neural Networks*, U. Ramacher and U. Rueckert, 125 – 151, Kluwer Academic Publishers, 1991.
6. J. Quali et al., "A customizable neural processor for distributed neural network", *VLSI*, 167 – 176, 1991.
7. B.J. vanderZwaag, C.H. Slump and L. Spaanenburg, "Process Identification through modular neural networks and rule extraction", *Proceedings FLINS'02*, 268-277, Ghent, Belgium, 2002.
8. R. Alberts, *Nets In Space*, M.Sc. thesis, Rijksuniversiteit Groningen, Groningen (The Netherlands), 2002.
9. L. O. Chua and L. Yang, "Cellular Neural Networks: Theory", *IEEE Transactions on Circuits and Systems*, **35**, 1257-1272 and 1273-1290, October 1988.
10. K. Preston Jr., M.J.B. Duff, *Modern Cellular Automata: Theory and Applications*, Plenum Press, New York, 1984.
11. J.J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", *Proceedings of the National Academy of Sciences of the U.S.A.*, **79**, 2554-2558, 1982.
12. T. Kohonen, "Self-Organized Formation of Topologically Correct Feature Maps", *Biological Cybernetics*, **43**, 59-69, 1982.
13. H. Harrer and J.A. Nossek, "Discrete-Time Cellular Neural Networks", *International Journal of Circuit Theory and Applications*, **20**, 453-467, September 1992.
14. H.T. Kung, "Let's design algorithms for VLSI systems", *Proceedings 1st CalTech conference on VLSI*, 65-90, Caltech, Pasadena, 1979.
15. M.H. ter Brugge, *Morphological Design of Discrete-Time Cellular Neural Networks,* Ph.D. thesis, Rijksuniversiteit Groningen, Groningen (Netherlands), 2003.
16. T. Roska, "Computational and Computer Complexity of Analogic Cellular Wave Computers, *Proceedings 7th IEEE Workshop on CNNs and their Applications*, R. Tetzlaff, 323-338, World Scientific (Singapore), 2002.
17. M. Diepenhorst., M. vanVeelen, J.A.G. Nijhuis, and L. Spaanenburg, "Automatic generation of VHDL code for neural applications", *Proceedings IJCNN'99*, Washington (USA), 1999.
18. L. Spaanenburg et al. "Training neural nets for small word width", *Proceedings AmiRA'01*, 171-180, Paderborn, (October 2001.
19. W.G. Macready, A.G. Siapas, and S.A. Kauffman, "Criticality and parallelism in combinatorial optimization", *Science*, **271**, 56-59, 1996.
20. A. Sharkey, "Multi-Net Systems", *Combining Artificial Neural Nets*, A Sharkey, Springer, London, 1999.
21. T. Caelli, L.Guan and W.Wan, "Modularity in Neural Computing", *Proceedings of the IEEE*, **87**, No.9, 1497-1518, 1999.
22. B. Schuermann, "Applications and Perspectives of Artificial Neural Networks", *VDI Berichte*, **1526**, 1-14, 2000.
23. A.J.W.M. ten Berg and L. Spaanenburg, "Considerations of the Compositionality of Neural Networks", *Proceedings ECCTD*, **III**, 405-408, Helsinki, September 2001.
24. L. Spaanenburg, "Over multiple rule-blocks to modular nets", *Proceedings 23rd Euromicro* , 698-705, Budapest, 1997.
25. R.S. Venema and L. Spaanenburg, "Learning feed-forward multi-nets", *Proceedings ICANNGA'01*, 102-105, Prague, 2001.
26. J.A.G. Nijhuis et al., "Delay-insensitive learning in a feed-forward neural network", *Proceedings INNC'90*, Paris (France), July 1990.