FAST AND EFFICIENT MULTI-LAYER CNN-UM EMULATOR USING FPGA

Zoltán NAGY and Péter SZOLGAY[1]

Department of Image Processing and Neurocomputing
University of Veszprém
Egyetem u. 10, H–8200 Veszprém, Hungary
e-mail: nagyz@almos.vein.hu

Received: August 30, 2002; Revised: May 27, 2003

Abstract

A new emulated digital multi-layer CNN-UM chip architecture called Falcon has been developed. Simulation running time can be hundred times shorter using the Falcon processor array compared to the software simulation. This huge computing power makes real time image processing possible. In this paper the main steps of the FPGA implementation and optimization are introduced. The Distributed Arithmetic technique is used to optimize the architecture on FPGAs. Using this technique, smaller and faster arithmetic units can be designed than using conventional approach where multiplier cores and adder trees are used to compute the state equation of the CNN array.

Keywords: cellular neural networks, reconfigurable computing.

1. Introduction

A Cellular Neural Network is a non-linear dynamic processor array. Its extended version, the CNN Universal Machine (CNN-UM) was invented in 1993, [1]. The main application area of this architecture is 2D signal or image processing. The most effective implementation of the CNN-UM architecture seems to be analog VLSI. The latest analogue CNN chip has a 128×128 pixel resolution and its equivalent computing power is 4 tera operation/second but its computational precision is about 7 or 8 bits [2]. In many applications these parameters are not high enough. If the resolution is higher we do not need to slice the images. If the precision is higher, less robust or more sophisticated analogical algorithms can be used [3].

A multi-layer CNN array can be used to solve the state equation of complex dynamical system. Currently the only method to solve the state equation of multi-layer CNN array is software simulation. If every layer has different time constant very small simulation step size must be chosen, thus software simulation is very slow. To achieve affordable runtimes the simulations have to be accelerated. This motivation came from the analysis of a retina model [4].

[1]Also affiliated to Analogic and Neural Computing Laboratory, Computer and Automation Institute of HAS, Kende u. 13–17, H–1111 Budapest, Hungary

The Falcon emulated digital CNN chip was designed to reach these goals. Special flexible emulated digital CNN-UM was developed where the accuracy, template size, cell array size and the number of layers can be configured. This paper describes the synthesis, implementation and optimization methods used to implement the Falcon processor array on FPGA.

2. Problem Statement

The Falcon architecture is designed to solve the full range model of a CNN cell (1).

$$\dot{x}_{i,j}(t) = \sum_{k=0}^{2\cdot n}\sum_{l=0}^{2\cdot n} A_{k,l}\cdot x_{i+k-n,j+l-n}(t) + \sum_{k=0}^{2\cdot n}\sum_{l=0}^{2\cdot n} B_{k,l}\cdot u_{i+k-n,j+l-n}(t) + I_{i,j} \quad (1)$$

Where x, u and I are the state, input and the bias values of the CNN cell, n is the neighbour value, A is the feedback, B is the feed forward template. The templates are $(2n+1)\times(2n+1)$ sized matrices. At the edges of the CNN array we use zero-flux boundary conditions, e.g. the value of the edge cells are duplicated.

The state equation of the CNN array is solved by forward Euler discretization. The h time step value can be inserted into the templates A and B, these modified templates denoted by A' and B'. Usually the input values do not change for several time steps so the state equation (1) can be broken into two parts: the feedback (2) and the input part (3), which is computed once at the beginning of the emulation.

$$x_{i,j}(m+1) = \sum_{k=0}^{2\cdot n}\sum_{l=0}^{2\cdot n} A'_{k,l}\cdot x_{i+k-n,j+l-n}(m) + g_{i,j} \quad (2)$$

$$g_{i,j} = \sum_{k=0}^{2\cdot n}\sum_{l=0}^{2\cdot n} B'_{k,l}\cdot u_{i+k-n,j+l-n} + h\cdot I_{i,j} \quad (3)$$

In the case of the multi-layer CNN we have the following set of state equations:

$$\begin{aligned}\dot{x}_{p,i,j}(t) &= \sum_{q=0}^{r-1}\sum_{k=0}^{2\cdot n}\sum_{l=0}^{2\cdot n} A_{p,q,k,l}\cdot x_{q,i+k-n,j+l-n}(t)\\ &+ \sum_{q=0}^{r-1}\sum_{k=0}^{2\cdot n}\sum_{l=0}^{2\cdot n} B_{p,q,k,l}\cdot u_{q,i+k-n,j+l-n}(t) + I_{p,i,j}, \quad (4)\end{aligned}$$

where r is the number of layers, x, u and I are the state, input and the bias vectors of one multi-layer cell. After discretization the following set of equations can be derived:

$$x_{p,i,j}(m+1) = \sum_{q=0}^{r-1}\sum_{k=0}^{2\cdot n}\sum_{l=0}^{2\cdot n} A'_{p,q,k,l} \cdot x_{q,i+k-n,j+l-n}(m) + g_{p,i,j} \tag{5}$$

$$g_{p,i,j} = \sum_{q=0}^{r-1}\sum_{k=0}^{2\cdot n}\sum_{l=0}^{2\cdot n} B'_{p,q,k,l} \cdot u_{q,i+k-n,j+l-n} + I_{p,i,j} \tag{6}$$

3. The Falcon CNN-UM Architecture

3.1. I/O Requirements

The first issue, which must be considered at design time, is the I/O requirements of the processor core. Even in the simplest case (when the neighbourhood $n = 1$) we need to load 9 template, 9 state and 1 constant values to update a cell: this is 19 values altogether. It is obvious that we cannot provide all these values from an external memory real time. On the other hand we cannot store the whole picture on the chip because of the memory limitations of the FPGAs. When the number of templates is low, it is evident to store the templates on the chip and this solution cuts the input requirements by half. But we still have to load 10 values to update one cell which requires very high bandwidth, so we must analyze the data-flow of Eq. (2).

When updating the cell array, the state value $x_{i,j}(m)$ of a cell must be loaded, when the upper left neighbour of the cell $x_{i-1,j-1}(m+1)$ is computed, the state value $x_{i,j}(m)$ is still required for the next two updates $x_{i,j-1}(m+1)$ and $x_{i+1,j-1}(m+1)$, in the following two lines $x_{i,j}(m)$ appears 3 times per line. Because $x_{i,j}$ appears when we update lines $j-1$, j and $j+1$ the simplest way to reduce memory bandwidth is to store these three lines of the picture on the FPGA, see *Fig. 1*. Similarly, if we used higher neighbourhood values we got the same result but we must store $2n+1$ lines. The only drawback of this method is that we have to store the constants on the FPGA, fortunately, only $n+1$ lines have to be stored. After these optimizations the I/O requirements of the processor are reduced to two input (one state and one constant) and two output operations per cell update. The memory size required to store the belt can be computed by the following equation:

$$((2\cdot n+1)\cdot sw + (n+1)\cdot cw)\cdot w, \tag{7}$$

where n is the neighbourhood value, sw and cw are the width of the state and constant value in bits and w is the width of the cell array.

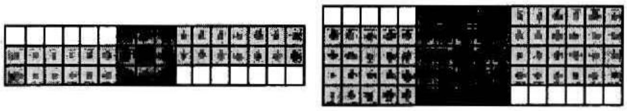

Fig. 1. The belt of the array stored on the FPGA in case of 3 × 3 and 5 × 5 templates

Table 1. I/O and memory requirements of the processor after optimization (assuming 256 cell wide array and 150 MHz clock frequency)

		State and constant width (bit)						
		4	6	8	12	16	24	32
Input bandwidth (GB/s)		0.15	0.225	0.3	0.45	0.6	0.9	1.2
On-chip memory requirements (kbit)	3 × 3	5	7.5	10	15	20	30	40
	5 × 5	8	12	16	24	32	48	64
	7 × 7	11	16.5	22	33	44	66	88

3.2. Conventional Arithmetic Unit

After successful reduction of the I/O requirements of the processor cores, the next question is how to organize our computing resources in the arithmetic unit. How many multipliers can we use efficiently? To achieve the highest performance, $(2n + 1) \times (2n + 1)$ multipliers can be used in the arithmetic unit. An adder tree is also required to sum the multiplied values. Using this arithmetic unit, one clock cycle is required to update a cell value. Unfortunately, this arithmetic unit is very huge because the multipliers require a large area, see *Table 2*.

This huge area can be reduced if the number of multipliers is decreased and a new cell value is computed serially, for example, in a row-wise order. In this case, $2n + 1$ multipliers, an adder tree and an accumulator register to store the partial results are used. The template operation is computed in $2n + 1$ clock cycles. A new block, called mixer, should be used between the memory and the arithmetic unit to simplify the control and the architecture of the memory unit. The mixer holds a vertical stripe of the cell array belt from the memory unit (dark grey area in *Fig. 1*). Storing these values in a small memory allows to use the same memory unit as in the previous case. The I/O requirements of the processor core are also reduced by this solution according to the clock cycles required for the cell update.

3.3. Distributed Arithmetic

Distributed arithmetic is a bit level rearrangement of a multiply accumulate to hide the multiplication [5]. It is a powerful technique for reducing the size of a parallel

Table 2. Area requirements of a 3 × 3 arithmetic unit in Virtex slices

Template	3 multipliers							9 multipliers						
width	State and constant width (bit)							State and constant width (bit)						
(bit)	4	6	8	12	16	24	32	4	6	8	12	16	24	32
4	66	87	102	138	174	246	318	211	276	323	435	547	771	995
6	87	135	165	219	273	381	489	276	422	514	680	846	1178	1510
8	102	165	192	252	312	432	552	323	514	597	781	965	1333	1701
12	138	219	252	396	486	666	846	435	680	781	1217	1491	2039	2587
16	174	273	312	486	603	819	1035	547	846	965	1491	1846	2502	3156

hardware multiply-accumulate that is well suited to FPGA designs. Distributed arithmetic is widely used in FIR filter implementations on FPGAs [6].

The conventional FIR filter computes the following convolution sum:

$$y(k) = \sum_{n=0}^{N-1} a(n)x(k-n), \tag{8}$$

where $y(k)$ is the response of the filter at time k, $a(n)$ are the filter coefficients, $x(k-n)$ is the input sample of the filter and N is the number of filter coefficients. The input sample can be written in the following fractional format:

$$x = -x_{B-1} \cdot 2^{B-p-1} + \sum_{b=0}^{B-2} x_b \cdot 2^{b-p}, \tag{9}$$

where x_b is a binary variable, B is the width of x and p is the position of the radix point. If *Eq.* (9) is substituted into *Eq.* (8), the following equation can be derived:

$$\begin{aligned} y(k) = {} & \sum_{n=0}^{N-1} -a(n)x_{B-1}(k-n) \cdot 2^{B-p-1} + \sum_{n=0}^{N-1} a(n)x_{B-2}(k-n) \cdot 2^{B-p-2} \\ & + \sum_{n=0}^{N-1} a(n)x_1(k-n) \cdot 2^{-p+1} + \sum_{n=0}^{N-1} a(n)x_0(k-n) \cdot 2^{-p} \end{aligned} \tag{10}$$

Eq. (10) can be computed serially by the architecture depicted in *Fig. 2*, which contains only look up tables, shift registers and one scaling adder.

If the input samples are represented with B bits of precision, B clock cycles are required to complete the calculation. Additional speed can be achieved by using two or more partial product LUTs and a scaling adder tree to sum partial products [7]. To achieve maximum performance, fully parallel distributed arithmetic FIR filter can be built which can compute the new result in a single clock cycle. The only drawback of the architecture is that space variant CNN templates cannot be

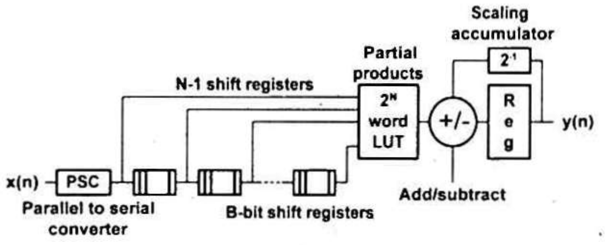

Fig. 2. Serial distributed arithmetic FIR filter

used because the coefficients in the partial product LUT cannot be changed when emulation is running. The cycle length of the arithmetic unit is determined by the parallelism of the FIR filters. Trade off can be made between speed and area, by using a fully serial or fully parallel FIR filter.

According to *Eq.* (2) this FIR filter architecture should be extended to 2 dimensions. This can be done by slightly modifying the shift register section and using larger partial product LUT as shown in *Fig. 3.* Inputs of the 2 dimensional FIR filter are connected to the cell memory which store a belt of the cell array. Increasing the number of inputs of the partial product LUT greatly increases its size. Using 3×3 sized templates the partial product LUT has 9 inputs and the area requirement is 9 slice for every bit of the partial product. This area requirement can be reduced to 3 slices/bit by using the architecture in *Fig. 3*, were the coefficients are grouped to fit into a 4 input FPGA LUT and adders are used to calculate the final partial products. Area requirements of the arithmetic unit built by various distributed arithmetic FIR filters are summarized in *Table 3*.

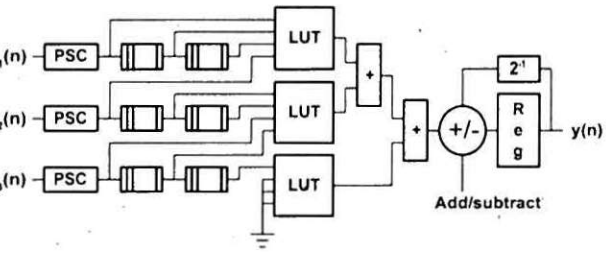

Fig. 3. 2 dimensional serial distributed arithmetic FIR filter

The main advantage of this approach is its easy scalability, while using the conventional arithmetic unit, the scalability is limited to 3 cases when $(2n+1) \times (2n+1)$ or $2n + 1$ or just one multiplier is used. In the case of distributed arithmetic, the cycle length is determined by the width of the state value, for example, in a 12 bit case the cycle length can be 1,2,3,4,6 or 12 clock cycles/cell. The area requirements of the distributed arithmetic units are usually smaller than the conventional approach, especially when the template width is high.

Table 3. Size of the 3 × 3 arithmetic unit in Virtex slices

Template width (bit)	2 clock cycle/cell							1 clock cycle/cell						
	State and constant width (bit)							State and constant width (bit)						
	4	6	8	12	16	24	32	4	6	8	12	16	24	32
4	113	149	174	259	324	504	649	148	216	266	412	520	824	1069
6	129	171	202	299	376	580	749	175	255	317	487	619	971	1264
8	145	193	230	338	428	655	849	202	293	368	563	719	1118	1459
12	177	237	286	418	532	807	1049	256	371	470	713	918	1412	1849
16	209	281	342	498	648	959	1249	309	449	572	863	1116	1706	2239

3.4. Achieving Even More Performance

Using the largest Virtex-II series FPGA form Xilinx which contains 46.592 configurable logical blocks (slices), several arithmetic units can be implemented. How can we use this huge amount of resources to achieve more performance?

The Falcon processors can be connected in a square grid on the FPGA. The performance of the array scales linearly according to the number of processors. The processed image is partitioned between the physical processors. Each physical processor column works on a long and narrow vertical stripe of the image. In one cycle a row of processor units gets the result of the previous iteration from the row of processor units above, calculates one iteration and sends the results to the row of processor units below. Adding additional columns to the grid increases the input bandwidth of the whole array and the available user I/O pins on the FPGA device limits the number of columns.

3.5. Multi-Layer Extension: the Falcon-ML Processor

To emulate a multi-layer CNN array we have to make some modifications on the original Falcon architecture. The main structure is not changed and the processor cores are arranged in a square grid. In a multi-layer case the same optimizations can be made to reduce I/O bandwidth as in the single-layer case. The memory requirements and the required input bandwidth of the r-layer processor core are r times higher than the single layer architecture.

The Falcon-ML processor emulates a general multi-layer CNN array where every layer is connected together in all possible ways. This means that the arithmetic unit must do r^2 times more work than in the single layer case. Templates in the multi-layer case can be treated as $r \times r$ pieces of single-layer templates and r single-layer arithmetic units can compute the template operation for every layer. It is possible at high-precision cases that this arithmetic unit requires a huge area, in this case the parallelism is reduced, and one multiplier per single-layer template or serial distributed arithmetic FIR filters can be used in the arithmetic unit.

4. Features and Performance

After these design considerations we were able to make a synthesizable RTL-level (Register Transfer Level) VHDL (Very high speed Hardware Description Language) description of the Falcon and Falcon-ML architectures. We used Synopsis FPGA-Express to synthesize our processors. The processors can be configured via a configuration file before the synthesis.

The configurable parameters are the following:

- the bit width and displacement of the radix point for the state, constant and template values, possible values for width are between 2 and 64
- the neighborhood value of the templates
- the width of the cell array slice
- the number of processor core rows and columns
- the number of layers in the multi-layer case

The large number of configuration parameters makes it easy to synthesize the Falcon architecture, which is optimal for our requirements. If our requirements are changed, the same FPGA can be used but with differently configured Falcon processors.

The performance of the Falcon processor is compared to the speed of the software simulation and the CASTLE emulated digital CNN-UM [8]. In the software simulations, Intel Pentium IV with DDR RAM (Double Data Rate) and RDRAM (RamBus) and AMD Athlon XP processors are used. To simulate a CNN array, functions of the Intel Signal Processing Library [9] are used, which contain MMX optimized functions for various signal and vector processing tasks. The performance of the software simulation depends on the size of the cell array. If the size of the data set is greater than the Level 2 cache of the microprocessor, the performance drops to a lower level, which is determined by the FSB (Front Side Bus) frequency of the processor.

The Falcon and the CASTLE processor arrays do not make any rounding until the final step of the computation, thus the precision used inside the processor is higher than the input precision and this must be considered in the comparison. We select 24-bit precision to compare with the double precision floating point simulation.

Timing analysis of the implemented Falcon processor using distributed arithmetic shows that the processor core can run at 200 MHz using 24 bit precision and a new value can be computed in every clock cycle. The CASTLE processor runs on 125 MHz clock frequency and compute a new cell value in 3 clock cycles. *Table 4* shows the performance of the software simulation and the emulated digital architectures in CNN iterations/s in the case of different number of layers and cell array sizes. The Falcon processor seems to be faster than the CASTLE architecture but we have to note that the Virtex-II FPGAs use 0.15 μm technology while the CASTLE processor is manufactured with 0.35 μm. If the same technology is used, the custom VLSI chip will be faster. In the single layer case the Falcon emulated digital CNN-UM architecture offers 27.63 times more performance than the software simulation. In multi-layer configuration the speed up is more significant but a larger area is required to implement the processor cores. The results show that the Falcon and the CASTLE architectures are considerably faster than the software simulation, even in a single processor configuration.

Table 4. Performance of the software simulation, the Falcon and the CASTLE architecture in CNN iteration/s

Array size	Single layer					3 layers			
	Athlon XP 1800 +	PIV 1.7G Hz DDR RAM +	PIV 1.8 GHz RDRAM +	CASTLE*	Falcon*	Athlon XP 1800+ +	PIV 1.7 GHz DDR RAM+	PIV 1.8G Hz RDRAM +	Falcon*
180 × 135	338.88	403.65	451.15	1,643.78	8,230.45	39.03	46.18	52.07	8,230.45
320 × 200	84.72	122.20	148.51	650.94	3,125.00	13.77	16.67	18.98	3,125.00
640 × 480	17.59	20.18	23.56	135.61	651.04	2.73	3.04	3.45	651.04
Speedup	**0.75**	**0.86**	**1.00**	**5.76**	**27.63**	**0.79**	**0.88**	**1.00**	**188.93**

*Performance of the single processor core configuration

The strength of the emulated digital architectures is to connect multiple processor cores to work parallel. The area required to implement a core processor depends on the accuracy, template size, cell array slice width and the number of layers. *Table 5* shows the number of implementable processor cores on different FPGAs. Using low precision, more than a hundred processor cores can be implemented on the largest FPGA. If an array of processor cores is used, the performance scales linearly correspond to the number of processors. The result is a 500-fold speedup compared to the software simulation using moderate accuracy.

Table 5. Number of implementable Falcon and Falcon-ML processor cores on different FPGAs

State and template width (bit)	Single layer			3 layers		
	6	12	24	6	12	24
v300	8	3	1	1	0	0
v1000	32	15	6	6	2	0
v3200	85	41	16	16	6	2
2v1000	13	6	2	2	1	0
2v8000	123	59	24	23	9	3

5. Examples

5.1. *Image Halftoning*

The first example is the 5 × 5 halftoning template from the CNN Template Library [10]. This template converts greyscale images into black and white images preserving the main features of the image. This function is implemented by the following template:

$$A = \begin{bmatrix} -0.03 & -0.09 & -0.13 & -0.09 & -0.03 \\ -0.09 & -0.36 & -0.6 & -0.36 & -0.09 \\ -0.13 & -0.6 & 0.05 & -0.6 & -0.13 \\ -0.09 & -0.36 & -0.6 & -0.36 & -0.09 \\ -0.03 & -0.09 & -0.13 & -0.09 & -0.03 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 0 & 0.07 & 0 & 0 \\ 0 & 0.36 & 0.76 & 0.36 & 0 \\ 0.07 & 0.76 & 2.12 & 0.76 & 0.07 \\ 0 & 0.36 & 0.76 & 0.36 & 0 \\ 0 & 0 & 0.07 & 0 & 0 \end{bmatrix}, \qquad z = 0.$$

This example was run on our prototyping board with a Xilinx Virtex-300 FPGA. Two processor configurations were used; in the first case one Falcon processor was used with 16 bit wide state and 8 bit wide template values. In the second case the template width remained 8 bit but the state width was decreased to 8 bit which enabled us to implement 4 Falcon processors. Each processor used 3 multipliers to compute the results. The simulation time step was 25/128 to make template value representation more accurate. 100 simulation time steps ran on an AMD Athlon XP 1800+ processor and on both Falcon processors.

The input, output and error images are shown in *Fig. 4* to *9* and the runtime and speed up of the computation are summarized in *Table 6*. The error images show the difference between the simulated and the emulated images, black pixel represents error larger than 0.01 and white pixel represents no error. In the 16 bit case, most of the errors are near to the edges of the image. The main source of these errors is the different boundary conditions. In the simulation, fixed boundary conditions were used, while the Falcon processor used zero-flux boundary conditions. In the 8 bit case the number of erroneous pixels is higher but these errors on the output image are not noticeable by a human observer.

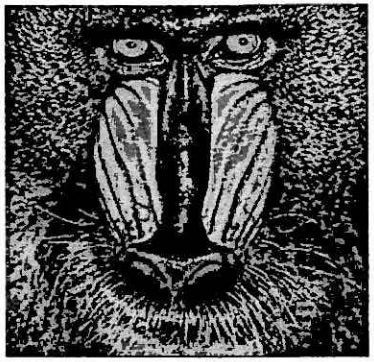
Fig. 4 Input and initial state

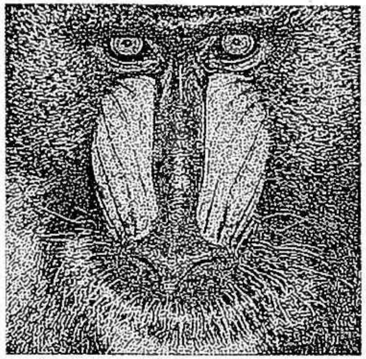
Fig. 6 Output of the Falcon processor using 16 bit state precision

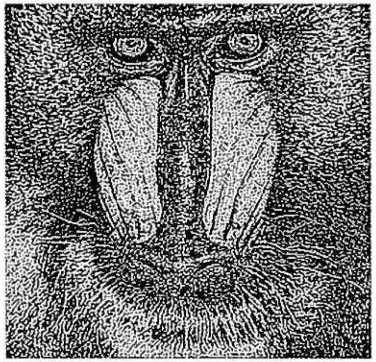
Fig. 8 Output of the Falcon processor using 8 bit state precision

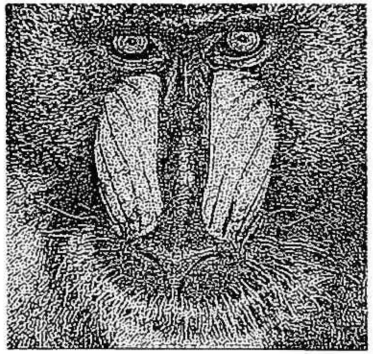
Fig. 5 Simulation result

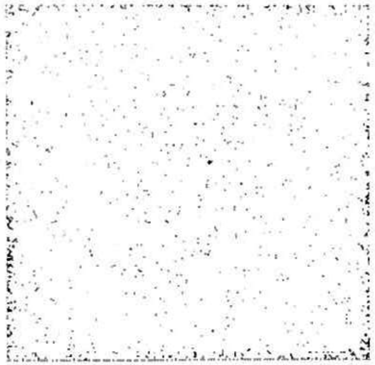
Fig. 7 Difference of the simulated and the emulated output in the 16 bit case

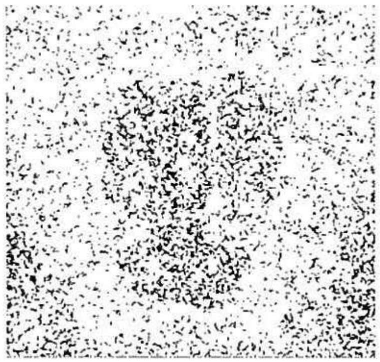
Fig. 9 Difference of the simulated and the emulated output in the 8 bit case

Table 6. Runtime and speedup of the examples

	Runtime of 100 iterations(s)	Speedup
Athlon XP 1800+	1.97475	1
1 Falcon processor 16 bit wide state	0.65497	3.015
4 Falcon processors 8 bit wide state	0.163782	12.0571

5.2. Emulating a 3-Layer Retina Model

The simulation of a retina model motivated the development of the multi-layer Falcon-ML architecture [4]. The retina is modelled with 3-layer CNN array where every layer has different time constant. One control and six feedback templates describe the connections between the layers. Some template elements have large values (±60,000) while others require fine resolution (0.2). The minimum timestep required to simulate the array is 0.0001, and the size of the input images is 180×135 pixels.

The implementation of the Falcon-ML processor which can emulate such a CNN array on our prototyping board was a very challenging task. After examining the templates we found that at least 28 bit wide state and 19 bit wide template values should be used. Unfortunately, the Virtex-300 FPGA in our prototyping board has very limited resources, so some modifications were required to implement the Falcon-ML architecture with these parameters.

At the first step the memory unit is changed and the on-chip SRAM memories are used to store a belt of the image. The height of the belt stored in the memory unit can be reduced to two lines instead of three because only one processor core is used. The size of the arithmetic unit is also reduced, using two bit a time serial FIR filters and 14 clock cycles are required to update a cell. The displacement of the template values can be configured independently for every layer and the template precision can be reduced to 9 or even 4 bits depending on the values used in the templates.

This specialized Falcon-ML architecture can be implemented on our prototyping board. The processor runs only on 100'MHz clock frequency because of the limitations of the Virtex-300 FPGA on our prototyping board and it computes a new value every 14th clock cycle. The performance of this slow processor is 293 CNN iteration/s on a 180×135 pixels sized image, which is 6 times faster than a Pentium IV 1.8 Ghz processor, see *Table 5.* Using faster memories, higher speed grade FPGA or using the more advanced Virtex-E and Virtex-II FPGAs, 30 times higher performance can be easily achieved.

6. Conclusions

The implementation of the Falcon architecture was successful on our prototyping board, using a Virtex-300 FPGA from Xilinx Inc. The performance of the architecture was encouraging, even in a single processor configuration a 27-fold speedup can be achieved. The easy scalability of the array makes it possible to connect the processor cores and achieve even more performance. Using re-configurable devices to implement the Falcon architecture provides us more flexibility compared to the conventional emulated digital architectures, e.g., different configurations can be used on the same hardware and extra design effort is not required to implement it.

If forward Euler method is used, very small time step is required for precise emulation of a CNN dynamics, mainly in a multi-layer case. Instead of computing thousands of iterations, better numerical method should be used, where the final value of the iteration is computed from several substeps or adaptive stepsize control can be used. Software implementation of a sophisticated numerical method can be slower than the forward Euler method. Re-configurable hardware can be used to implement such an algorithm to improve performance and make very precise and fast emulation of various CNN architectures possible.

Acknowledgements

This research is sponsored by the National Research and Development Funds of the Széchenyi Plan under the consortium NKFP OM-2/052/2001 and OTKA #029609.

References

[1] ROSKA, T. – CHUA, L. O., The CNN Universal Machine. An Analogic Array Computer, *IEEE Trans. On Circuits and Systems-II*, **40** (1993), pp. 163–173.

[2] LIÑÁN, G. – DOMÍNGUEZ-CASTRO, R. – ESPEJO, S. – RODRÍGUEZ-VÁZQUEZ, A., ACE16k: A Programmable Focal Plane Vision Processor with 128x128 Resolution, in *Proc. of the 15^{th} European Conference on Circuit Theory and Design*, **1** (2001), pp. 345–348

[3] SZOLGAY, P. – TÖMÖRDI, K., Analogic Algorithms for Optical Detection of Breaks and Short Circuits on the Layouts of Printed Circuit Boards Using CNN, *Int. J. of Circuit Theory and Applications*, **26** (1998).

[4] BÁLYA, D. – ROSKA, B. – ROSKA, T. – WERBLIN, F. S., A CNN Framework for Modeling Parallel Processing in a Mammalian Retina, *Int. J. on Circuit Theory and Applications*, **29** No. 3, 2002.

[5] LIU, P. – LIU, B., A New Hardware Realization of Digital Filters, *IEEE Trans. on Acoust., Speech, Signal Processing*, **ASSP-22** December 1974, pp. 456–462.

[6] MINTZER, L., FIR filters with the Xilinx FPGA, in *Proc. of FPGA '92 ACM/SIGDA Workshop on FPGAs*, pp. 129–134, 1992.

[7] WHITE, S. A., Applications of Distributed Arithmetic to Digital Signal Processing, *IEEE ASSP Magazine*, **6** (3) (1989), pp. 4–19.

[8] KERESZTES, P. – ZARÁNDY, Á. – ROSKA, T. – SZOLGAY, P. – HÍDVÉGI, T. – JÓNÁS, P. – KATONA, A., An Emulated Digital CNN Implementation, *Int. J. of VLSI Signal Processing*, Kluwer, 1999 September 9.

[9] Intel Performance Libraries homepage, http://www.intel.com/software/products/perflib/

[10] CNN Software Library, http://lab.analogic.sztaki.hu/

[11] Xilinx products homepage, http://www.xilinx.com/