Stochastic Bitstream Based CNN and Its Implementation on FPGA

Ádám Rák, Gergely Soós and György Cserey

Abstract

In this paper, we present a new type of CNN (Cellular Nonlinear Network) model and FPGA implementation of CNN-UM (Cellular Nonlinear Network Universal Machine). Our approach uses stochastic bitstreams as data carriers. With the help of stochastic data streams more complex nonlinear cell interactions can be realized than conventional CNN hardware implementations have. The accuracy as indexed by bit depth resolution can be improved at the expense of computation time without influencing hardware complexity. Our simulation results prove the universality and utility of the model. Our experimental results show that the proposed model can be implemented on an FPGA hardware.

Index Terms

Stochastic bitstream, CNN (Cellular Nonlinear Network), parallel processing.

I. INTRODUCTION

In the past 15 years, since the invention of CNN [1], [2], many algorithms have been published and several hardware realizations have been introduced on the market [3], [4], [5], [6], [7]. The CNN community is constantly increasing and is still searching for the key application which could bring immense success and spread this technology in the field of everyday life.

After the first successfully functioning analog CNN chips (Ace4k, Ace16k and CACE1, sized 64x64, 128x128 and 32x32 [8], [9], [10]) and systems (e.g. Bi-i and for SCAMP3 [11], [12]) using these chips, the development and implementation of new chips and systems are in progress (EyeRis I and II [4]). Beside the analog implementations attempts have been made to design fully digital versions [6], [13], [14], [15]. The comparison of the various CNN implementations can be found in [13], which also presents an emulated digital CNN-UM solution for partial differential equations. Overviewing the referenced scientific literature, we can state that although some stochastic simulations have been done on CNN chips [16], [17], stochastic bitstream based CNN implementation has not been realized before.

The theory and mathematical foundations of stochastic bitstreams were published earlier [18]. Since then a few more applications have been implemented [19]. There are several examples in the related literature, where neural networks using stochastic bitstreams were implemented successfully [20], [21], [22], [23]. One of the main advantages of stochastic bitstreams is that their circuit realizations require few elements and the multiplication of two values can be realized by a single AND operation. It is important to note that while the analog VLSI does not follow the Moore law, the implementations of digital CNN and also STCNN do satisfy it, that is with the improvement of technology - decrease in size - we can implement more cells on the same area with higher communication speed.

Stochastic data are provided by almost every analog sensor. To extract information from this data stream, the standard practice is to integrate for relatively long time, then quantize (e.g. light sensors). However, if appropriately short integration time is used, the output is a 0-1 stochastic data stream where the carried information equals to the expected value. Then the signal of the sensor can be processed by a stochastic data stream based system without any conversion. Besides, if the output receiving module operates in switching mode then the requested 0-1 bit data stream is directly available.

We expect that with the help of stochastic representation, beside relatively small hardware complexity more sophisticated nonlinear cell interactions can be realized than in conventional CNN implementations have.

Because the computation is stochastic, an operation has to be performed many times in order that the statistics of the result should converge. In exchange, computational accuracy can be optional. The complexity is independent of accuracy, but it depends on speed in a nonlinear way.

Our model was tested on an FPGA system, which contains programmable logic units and connections. The hierarchic structure of programmable connections provides the possibility for implementation of any logic circuit within certain limits. The great advantage of FPGAs is the ability to implement the prototype without the need for VLSI chip fabrication. Disadvantages are bigger consumption, lower clock speed because of the length of connections and less realizable complexity on the same area.

Á. Rák and G. Soós are with the Faculty of Information Technology, Pázmány Péter Catholic University, Budapest, Práter u. 50/a. 1083, Hungary.
E-mail: [rakad,soos]@itk.ppke.hu

Gy. Cserey is also with the Infobionic and Neurobiological Plasticity Research Group, Hungarian Academy of Sciences - Pázmány Péter Catholic University - Semmelweis University, Budapest, Práter u. 50/a. 1083, Hungary.
E-mail: cserey@itk.ppke.hu
Our primary aim was to design a CNN structure which can be effectively realized on FPGA and VLSI environments, while running robust nonlinear operations. Universality and speed were also of great importance. Results show that both are tenable during implementation.

The structure of this paper is as follows: after the introduction, the second Chapter introduces the stochastic bitstream based CNN (STCNN) structure. The third Chapter gives an overview of the mathematical background of stochastic bitstreams, including the advantages and disadvantages. The fourth Chapter describes possible high level operations on bitstreams. The sixth Chapter introduces the STCNN nucleus and additional modules to extend our model to a universal machine can be found in Chapter six and seven. Chapter eight shortly summarizes the template design considerations. Simulation results and FPGA implementational aspects are presented in Chapter nine and ten respectively.

II. STCNN - THEORY AND STRUCTURE

The processing elements in a Cellular Nonlinear Network are cells arranged in regular grid. The main operation of the grid is defined on a small local neighbourhood. This local connectivity has several advantages, one of its main property is that the number of cells linearly correlates to the number of connections. The original CNN was defined by differential equations with time continuous update functions for the cell values and output (Figure 1). The input image gives feed forward excitation. If the image is constant while template running this can be represented a space dependent bias value called bias map. The initial value of the state can be considered as secondary input.

The structure of this paper is as follows: after the introduction, the second Chapter introduces the stochastic bitstream based CNN (STCNN) structure. The third Chapter gives an overview of the mathematical background of stochastic bitstreams, including the advantages and disadvantages. The fourth Chapter describes possible high level operations on bitstreams. The sixth Chapter introduces the STCNN nucleus and additional modules to extend our model to a universal machine can be found in Chapter six and seven. Chapter eight shortly summarizes the template design considerations. Simulation results and FPGA implementational aspects are presented in Chapter nine and ten respectively.

Cells are indexed in row, column order. The size of the array is m by n. The state of $C_{i,j}$ cell ($i \in [0..m-1]; j \in [0..n-1]$) is $x_{i,j}(t)$. Input signal exciting the system is $u_{i,j}(t)$. The set of neighboring cells that are directly connected to $C_{i,j}$ is denoted by N. Using stochastic representation values are limited to $[0, 1]$ interval. Some limitations always exist in the physical realisation of any CNN network. States are updated using $g_{i,j,p} : [0, 1] \rightarrow [0, 1]$, $p \in [1..10]$ nonlinear interaction functions joined by the spatial integration functions $h_{i,j} : [0, 1]^{10} \rightarrow [0, 1]$. Eight of the interaction functions are for neighbouring cells, one for feedback and the last for an additional bias term $z_{i,j}$. Finally, the output of the array is the state after Nth iteration 1. When a $g_{i,j,p}$ function refers to a non valid $x_{i,j}$ value, boundary condition is applied to produce the argument.

$$x_{i,j,0} = u_{i,j}$$

$$x_{i,j,t+1} = h_{i,j}(g_{i,j,1}(x_{i-1,j-1,t}), g_{i,j,2}(x_{i-1,j,t}), ..., g_{i,j,9}(x_{i+1,j+1,t}), g_{i,j,10}(z_{i,j,t}))$$

$$y_{i,j} = x_{i,j,n}$$

To highlight the properties of the proposed structure, let us consider standard CNN configuration. The state update and output equation is

$$x'_{i,j}(t) = -x_{i,j}(t) + \sum_{(k,l):C_{k,l} \in \mathcal{N}_{i,j}} A_{i,j,k,l} \ast y_{k,l}(t) + \sum_{(k,l):C_{k,l} \in \mathcal{N}_{i,j}} B_{i,j,k,l} \ast u_{k,l}(t) + z_{i,j}$$

$$y_{i,j} = \sigma(x_{i,j})$$

Figure 1. The first image shows the standard STCNN structure. The input is influencing the state through linear weights. The output is calculated using $f(x)$ nonlinear function and after multiplication feeded back to the state. The state variable also gives direct feedback with a space independent bias term. If the input is static while execution, the input effect and the bias value can be represented by a space dependent bias map. This construction is shown in second image. The third image is the STCNN model. Bias map is used, and the output function is a simple identity function. The cell interaction function $g$ gives more complex local connection to cells then the weighted $f(x)$ function can realize.
where $\sigma$ is the nonlinear output function. To describe the cell dynamics zero condition for the states and some boundary condition is also declared.

Differential equations can be solved iteratively with Euler formula

$$x_{i,j}(t + dt) = x_{i,j}(t) + dt \ast x'_{i,j}$$

(4)

Substituting 2

$$x_{i,j}(t + dt) = x_{i,j}(t) + dt \ast \left\{-x_{i,j}(t) + \sum_{(k,l) : C_{k,l} \in N_{i,j}} A_{i,j,k,l} \ast y_{k,l}(t) + \sum_{(k,l) : C_{k,l} \in N_{i,j}} B_{i,j,k,l} \ast u_{k,l}(t) + z_{i,j}\right\}$$

(5)

if $dt$ is selected to be equal 1 we have the DTCNN equation [24]. Since time is discretized k index will be used.

$$x_{i,j}(k + 1) = \sum_{(k,l) : C_{k,l} \in N_{i,j}} A_{i,j,k,l} \ast y_{k,l}(k) + \sum_{(k,l) : C_{k,l} \in N_{i,j}} B_{i,j,k,l} \ast u_{k,l}(k) + z_{i,j}.$$  

(6)

If input is not changing during the transient we can accumulate constant parts into a bias map.

$$x_{i,j}(k + 1) = \sum_{(k,l) : C_{k,l} \in N_{i,j}} A_{i,j,k,l} \ast \sigma(x_{k,l}(t)) + w_{i,j}.$$  

(7)

This can be interpreted as follows: each cell state is updated using the cumulated effect of neighboring cells and an additional bias map. For example in case of 3x3 neighborhood:

$$x_{i,j}(k + 1) = h(g_{i,j,1}(x_{i-1,j-1}(k)), g_{i,j,2}(x_{i-1,j}(k)), \ldots g_{i,j,9}(x_{k+1,l+1}(k)), g_{i,j,10}(w_{i,j}))$$  

(8)

Although the functions could be different for all cells, in most models functions are space independent thus $g_{i,j,p} = g_p$ and $h_{i,j} = h$ for all cells. In standard CNN $p \in \{1, 9\}$ : $g_p(x) = A_p \ast \sigma(x)$ and $g(10)(x) = x$ and $h$ function is the sum of all attributes. One possible extension of the model to apply more complex $\sigma$ function with $A_p$ scales like in polynomial CNNs [25] [26]. Our stochastic representation allows us to have more freedom with moderate hardware complexity: the $g_p$ functions can differ not only by a scale factor but they can have different characteristics, even more the spatial integration function can be more complex then in an addition.

We will define a Stochastic Bitstream based representation for values in the next section, and will define the family of possible nonlinear functions in V-A. The operations of the standard CNN are described by the set of scaling constants called template. Our functions are more complex thus we need more values to describe them, we still call this set as template.

## III. STOCHASTIC BITSTREAMS - ADVANTAGES AND DISADVANTAGES

### A. Definition

Let $\xi_k$ a discrete time stochastic process consisting of independent random variables. Its possible values are:

$$\xi_k \in \{0, 1\}.$$  

(9)

Since there are two possible values they are also called 0 and 1 bit. We assign data content $g_k$ at each time instant:

$$g_k = \mathbb{E}\xi_k$$  

(10)

where $\mathbb{E}$ is the expected value. This means that the bitstream carries the data through its statistics. If the probability of the occurrence of the 1 bit is $p_k$

$$p_k = \mathbb{P}(\xi_k = 1),$$  

(11)

than its expected value is

$$\mathbb{E}\xi_k = 0 \ast (1 - p_k) + 1 \ast p_k = p_k,$$  

(12)

and

$$g_k = p_k.$$  

(13)

Which means that the carried information equals not only to the expected value but to the probability of the occurrence of the 1 bit.

This is our stochastic bitstream. [18], [27]. Further on if “bitstream” is mentioned it is meant to be stochastic bitstream.

The process consists of independent random variables. It means:

$$\mathbb{P}(\xi_a = x, \xi_b = y) = \mathbb{P}(\xi_a = x) \ast \mathbb{P}(\xi_b = y)$$  

(14)
If $g_k$ does not change in time, then the assumption of weak stationarity is true,

$$E\xi_i = E\xi_j, \ i \neq j$$

(15)

and the probability distribution is the same for all random variables. In this case our bitstream is a so called Bernoulli process. We want to represent real values ([0,1]) with a stochastic bitstream. For this purpose we need processes to generate an appropriate bitstream for a value, and to get the meaning of an unknown bitstream.

For a specific $g^*$ we can construct an appropriate bitstream $\xi^*_k$ using a reference random generator $\mu$ with uniform distribution $p_\mu$ between 0 and 1. In all time instance we get a value and compare it to $g^*$. If it is bigger than $g^*$ we emit 0 otherwise 1. This construction gives 1 bit with probability of $g^*$

$$P(\xi^*_k = 1) = P(\mu_k < g^*) = \int_0^{g^*} p_\mu(x)dx = (g^* - 0)1 = g^*.$$  

(16)

B. Measuring data content of a stream

For an unknown bitstream we need several samples in time to get the data content. If we measure values of a bitstream for several time instances, the law of large numbers states that we can estimate $g^*$ with the average of observations. Considering the central limit theorem we can describe our estimation. It says that sum of identically distributed independent random variables with finite variance converges to normal distribution.

C. Noise considerations

Let $b_t$ stochastic bitstream: $P(B_t = 1) = p.$ $S$ is a measurement of the expected value ($E_b_t$): $S = \frac{1}{M} \sum_{i=1}^M b_i.$ Deviance is $DS = \sqrt{\frac{P}{N^2}}.$ For convenience : $N = 2^n.$ In the worst case of the deviance: $p = 0.5$ $DS = \sqrt{\frac{0.25}{2^n}} = \frac{1}{2^{n/2}}.$ Because $S$ is the sum of many independent Bernoulli random variable, it should have gaussian distribution: $S \sim N(p, 2^{-n/2}).$ If we chose the most significant $\frac{n}{2}$ bits of $S$ as measured value, then the probability of precise measurement will be: $P(-\frac{1}{2^{n/2}} < S - ES < \frac{1}{2^{n/2}}) = P(-DS < (S - ES) < DS) = 0.68$ because of the gaussian distribution. With only seven bits only: $P(-D < (S - ES) < DS) = 0.95$ If $n = 16$ we have 8 or 7 bits of accuracy.

D. Basic operations with bitstreams

Bitstreams are consisting of binary values, thus logic operators can be applied to transform or join them. In the following at first we examine the logic gates with two independent inputs and one output [23]. Inputs are $\xi_n$ and $\eta_n$ two bitstreams, with probability distributions

$$p1_n = P(\xi_n = 1), p2_n = P(\eta_n = 1).$$

(17)

They are independent, thus

$$P(\xi_n = x, \eta_n = y) = P(\xi_n = x) * P(\eta_n = y)$$

(18)

1) Logic AND operation: The output is $\zeta_n$. The output is 1 if and only if both the inputs are 1. Therefore

$$E\zeta_n = P(\zeta = 1) = P(\xi_n = 1, \eta_n = 1) = p1_n * p2_n = E\xi_n * E\eta_n.$$  

(19)

We can observe that this logic operation is a multiplication between carried analog values of two stochastic bitstreams. Both multiplication and logic AND operation are associative and commutative, therefore an AND gate with arbitrary number of inputs can be replaced by some AND gates with two inputs, so an AND gate with arbitrary number of inputs multiplies the input bitstreams.

2) Logic OR operation: Let us examine the logic OR operation from a stochastic point of view. We have the multiplication but we would need the addition as well for creating linear functions.

$$E\zeta_n = P(\zeta = 1) = = P(\xi_n = 1, \eta_n = 0)$$

$$+ P(\xi_n = 0, \eta_n = 1) + P(\xi_n = 1, \eta_n = 1) =$$

$$= p1_n * (1 - p2_n) + (1 - p1_n) * p2_n + p1_n * p2_n =$$

$$= p1_n + p2_n - p1_n * p2_n = E\xi_n + E\eta_n - E\xi_n * E\eta_n$$

(20)

From this it follows that using OR operation for addition gives quite big error.
This description of a logical function can be extended to more input variables as well.

Arbitrary combinatory network can be built from these operations. We have the multiplication but the addition is not accurate, it can be approximated only. This is clear if we suppose that

$$P(\xi_n = 1) > 0.5, P(\eta_n = 1) > 0.5$$

then we are not able to represent the sum of two streams in a single bitstream.

E. Independence

We assumed that the stochastic bitstreams we worked with are independent, otherwise it is impossible to operate with them. It may happen that we need to clone one of the bitstreams. The typical case is, when we want to square a bitstream. By construction every bitstream is uncorrelated, therefore if we shift the stream with one time instant, then this shifted stream and the original one will be independent so we can calculate its square [21].

$$E \xi_n = E (\xi_n \land \xi_{n+1}) = E \xi_n \ast E \xi_{n+1}$$

This shifting can be useful in many case. We will denote shifting of a bitstream with $d(\cdot)$ function.

F. Complex logical functions

Arbitrary logical function (LF) $Q: \{0,1\}^n \rightarrow \{0,1\}$ can be described by a truth table. Let us examine the case of four inputs $R_1,..R_4$. If we consider them a binary number ($R_1$ is the most significant bit) we have a natural order to enumerate all possible 16 input cases. Let $Q_i$ denote (R1..R4) configuration when the decimal value of the binary numbers is $i$. We will use the notation $bit(i,k)$ to extract k-th bit from the integer $i$. $bit(i,1)$ gives the most significant bit. $Q$ can be equivalently described as

$$Q: Q_i \rightarrow L_i \in \{0,1\} \\ , i \in \{0..15\}$$

or

$$Q: \bar{Q} \rightarrow \bar{L}.$$ (25)

Let us apply $Q$ on $\xi_{1k}, \xi_{2k}, \xi_{3k}, \xi_{4k}$ independent stochastic bitstreams. Consider a time interval $[a,b]$ when streams are not changing their carried values. The probability of 1 bit in a bitstream is then

$$\forall k \in [a,b] : P(\xi_{jk} = 1) = p_j, j \in \{1,2,3,4\},$$ (26)

and certainly the probability of 0 bit is $(1 - p_j)$. As it was shown in section III-A, the expected value $E \xi_{jk} = p_j$.

Since variables are independent, probability of a specific input configuration for a given time instant $P(Q_i)$, is the multiplication of matching probabilities for each variables

$$P(Q_i) = \prod_j P(\xi_{jk} = bit(i,j)) = \prod_j \left(p_j^{bit(i,j)} \ast (1 - p_j)^{1-bit(i,j)} \right).$$ (27)

We can create $\bar{P}Q = [P(Q_0),..P(Q_{15})]$. Lines in the truth table are excluding events thus to calculate the probability of 1 bit output of the logical function we have to sum up probabilities of all lines with 1 bit output.

$$P(Q = 1) = \sum_{i:L_i=1} P(Q_i).$$ (28)

The associated output of the logical function is the expected value of the outcome stochastic bitstream:

$$fQ = EQ = 1 \ast P(Q = 1) + 0 \ast P(Q = 0) = 1 \ast \sum_{L_i=1} P(Q_i) + 0 \ast \sum_{L_i=0} P(Q_i),$$ (29)

or using $\langle x,y \rangle$ notation for scalar product

$$fQ(\xi_{1k},\xi_{2k},\xi_{3k},\xi_{4k}) = EQ(\xi_{1k},\xi_{2k},\xi_{3k},\xi_{4k}) = \langle \bar{F}Q, \bar{L} \rangle.$$ (30)

This description of a logical function can be extended to more input variables as well.
IV. COMPLEX OPERATIONS ON BITSTREAMS

A. Creating streams

In III-A section it was described how to create a bitstream with a specific data content. For this reason a good random generator with uniform distribution in [0..1] is needed. In this section we will describe a method to create streams using a random binary source that can emit 0 and 1 with probability of 0.5. It was shown in III-E subsection that from a bitstream with \( \mathbb{E} \xi_n = 0.5 \) we can clone more bitstreams with time-shift, and with simple AND gates we can create streams with carried value of 0.5\(^m \). We will call these streams reference bitstreams.

Let us connect four referential bitstreams \( \xi_1, \xi_2, \xi_3, \xi_4 \) with expected values of 0.5, 0.5\(^2 \), 0.5\(^3 \), 0.5\(^4 \) to a logical function \( Q(R_1, R_2, R_3, R_4) \). We summarize the logical truth table in Table I.

<table>
<thead>
<tr>
<th>( R_1 )</th>
<th>( R_2 )</th>
<th>( R_3 )</th>
<th>( R_4 )</th>
<th>( P(Q) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( 1 - p_1 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( 1 - p_1 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( 1 - p_1 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( 1 - p_1 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( 1 - p_1 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( 1 - p_1 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( 1 - p_1 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( 1 - p_1 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( (p_1) )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( (p_1) )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( (p_1) )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( (p_1) )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( (p_1) )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( (p_1) )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( (p_1) )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( (p_1) )</td>
</tr>
</tbody>
</table>

Table I

In the first five columns all possible input configurations are listed of a four input logical function. In the last column the possibility of this event is calculated using the separate possibilities of 1 bit for each input variable.

Using the four referential bitstreams \( \xi_1, \xi_2, \xi_3, \xi_4 \) and a logical function \( Q \), we can emit a bitstream coded by a 16 bit long \( L \) vector. Let us examine the possible expected value \( \mathbb{E}Q \) and equivalently the information content \( fQ \) of all possible bitstreams using equation 30. After sorting the \( \text{(code,value)} \) pairs in increasing order of the expected value we can have an overview displayed on Figure 2. The difference between two adjacent values is either \( 2^{-10} \) or 0. It means that we can generate arbitrary stream with acceptable precision.

Figure 2. If referential bitstreams with 0.5, 0.5\(^2 \), 0.5\(^3 \), and 0.5\(^4 \) expected values are connected to a logical function with 4 inputs and 1 output we can generate a custom bitstream. The expected value of the outgoing stream is displayed on the vertical axis. The function can be described by a 16 bit long code. All the possible decimal values of this code in [0..65535] are on the horizontal axis, (code,value) pairs are rearranged in increasing order of the expected values. The figure shows that the representation has no splits. All values are represented with appropriate accuracy. The representation falls off near by \( x=0 \) and \( x=1 \) values. The higher the gradient, the bigger quantification steps.
B. Multiply and Offset Unit

Using a logical function and reference bitstreams we can create Multiply and Offset Unit (MOU) unit. Let us consider a logical function with 4 input. The operand is connected to the first input, and three reference bitstreams with expected values of 0.5, 0.5², 0.5³ are also connected. The \( \vec{P} \vec{Q} = [\vec{P}(Q_0) .. \vec{P}(Q_{15})] \) can be constructed in similar way as it was done in section III-E. If we examine the rows of the truth table, \( Q_0 .. Q_{15} \) lines has \( p_1 \), \( Q_0 .. Q_8 \) lines has \( 1 - p_1 \) terms. If we create \( fQ \) for a given function description, \( \vec{L} \) can be split to \( L_{low} \) and \( L_{hi} \) representing \( Q_0 .. Q_8 \) and \( Q_0 .. Q_{15} \) respectively. After factoring out the common terms \( \vec{P} \vec{Q}' \) remains representing the lower three bits.

\[
fQ(\xi_1, \xi_2, \xi_3, \xi_4) = E\vec{Q}(\xi_1, \xi_2, \xi_3, \xi_4) = \left\langle \vec{P} \vec{Q}, \vec{L} \right\rangle = p_1 \left\langle \vec{P} \vec{Q}', L_{low} \right\rangle + (1 - p_1) \left\langle \vec{P} \vec{Q}', L_{hi} \right\rangle \tag{31}
\]

As it was seen in the previous subsection these scalar product terms with reference signals can describe constant values in 0..1 range. Since we only have 3 reference streams involved with less precision.

\[
fQ(\xi_1, \xi_2, \xi_3, \xi_4) = p_1(a - b) + b. \tag{32}
\]

C. Polynomial functions

If we connect \( I_1 \) stream with expected value \( x \) and three clones of it to a logical function \( Q \), polynomial transformations can be realized on the incoming stream (Table II).

<table>
<thead>
<tr>
<th>( I_1 )</th>
<th>( I_2 )</th>
<th>( I_3 )</th>
<th>( I_4 )</th>
<th>( N )</th>
<th>( \vec{P}(Q_i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( x^0 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( x^1 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( x^2 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( x^3 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( x^4 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>( x^5 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>( x^6 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>( x^7 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>( x^8 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>( x^9 )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>( x^{10} )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>( x^{11} )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>( x^{12} )</td>
</tr>
</tbody>
</table>

Table II

Four stream with the same expected value \( x \) is connected to a four input logical function. The table shows the probabilities for all 16 possible input configuration as polynomial function of \( x \) sorted in increasing order of the 1 bits in input.

The expected values of the input streams are the same, their value is \( x \). Probabilities for all 16 possible input configuration is listed in Table II. \( P(Q_i) \) depends on the count of 1 bits in the binary representation if \( i \), so we have 5 different \( Q \) rows.

\[
fQ(x) = K_0 \ast (4x^4 - 4x^3 + 6x^2 + 1) + K_1 \ast (-x^4 - 2x^3 + x^2) + K_2 \ast (x^4 - 2x^3 + x^2) + K_3 \ast (-x^4 + 3x^3) + K_4 \ast (x^4) \tag{33}
\]

\[
K_i \in \{0..4\} \tag{34}
\]

Using different \( \vec{L} \) values wide family of maximal 4th degree polynomials can be created.

V. ST-CNN Nucleus

A. Interaction functions

If we apply a polynomial transform function and then a MOU function (Figure 3) on a stochastic bitstream, we can create a wide range of polynomial transforms described by 2x16 bits.

To analyze the description power of the proposed construction we generated large amount of random \([L_{poly},L_{MOU}]\) values (more than 2 million pairs) and investigated the resulting \( g(x) \) functions and their first and second derivative in \([0,1]\) interval. The minimum and maximum values for all \( x \) were recorded and displayed (Figure 4). The next 8 different nonlinear weight functions that are presented on Table III were realized this way. In STCNN this construction can be used to form \( g_k \) functions.

Diagrams of different realized nonlinear (polynom) weight functions. From these we can empirically deduce the basic types and properties: there can only be one local extrema, steepness is limited, and constant offset is possible.
Table III

Diagrams of different realized nonlinear (polynomial) weight functions. From these we can empirically deduce the basic types and properties: there can only be one local extremum, steepness is limited, and constant offset is possible.
Figure 3. The proposed structure that can be used to create wide family of nonlinear interaction functions using two four input logical function. The lower part is a multiplication and offsetting unit, the upper one is the polynomial function part. It is also possible to pass the stream directly without any transform.

Figure 4. The limitations of the first and second derivatives of nonlinear functions are shown here.

B. Spatial integration function

Spatial integration function is a 10 input logical function. This can have $2^{10}$ different input configurations. In any CNN implementation there exist an interval for acceptable input range. The dynamics of the state variable is obviously bigger since information are spatially aggregated by summation. By construction our signals are limited into $[0, 1]$ and $g$ functions are mapping them to the same interval. If we want to sum 10 signals we need $[0, 10]$ range for the state variable. It is clear, that with a single stochastic bitstream we are not able to do that so we need extended representation. We can assign weights of $(2^3, 2^2, 2^1, 2^0)$ to four streams so we have a 4bits wide bitstream bus. To represent all possible $h$ functions we need $4 \times 2^{10}$ bits. To measure the data content of a bitstream we have to accumulate values and calculate the average as it was described in section III-B, III-C. Similarly we can measure the data content of a bus if we consider actual values as a 4bit long binary number and accumulate it for averaging using binary add. Since $g$ functions accept values only in $[0, 1]$ interval, the calculated value should be normalised before feeding it back.
C. Disadvantages

Following the mathematical background, there are difficulties to overcome. To achieve an appropriate accuracy, we need to average a large number of samples which means we have to repeat the operations several times. This is a trade off between speed, complexity and accuracy. To produce referential bitstreams we need to provide a number of uncorrelated bitstreams. For this task we need many real random generators or pseudo random generators.

VI. COMPUTATIONAL FRAMEWORK OF STCNN-UM

To make STCNN a useful system, it needs additional local and server modules beside the STCNN nucleus (Figure 5). These additional modules complete STCNN to an universal machine. The STCNN-UM is slightly different (main different modules: RNG, FSM, AIC, GRND) from the original CNN-UM structure. Let us examine the components:

- RNG (Random Number Generator): The function of this module is to create reference bitstreams. Real random generator or if there is no analog part, a pseudo random generator built from a shift register can be used. The initialization 'seed' can be retrieved from neighboring cells.
- LCCU (Local Communication and Control Unit): This module is responsible for programmability. Every cell receives global control via this module. (like LCCU in CNN-UM)
- FSM (Finite State Machine): This module provides local programmability. This can be seen as a simple state machine, that controls the template running (binary or grayscale) and registers. The LCCU can override this component. FSM is similar to the GACU and LCCU modules of CNN-UM, but they are not equivalent. From the view of hardware complexity 16 possible states seems to be reasonable.
- AIC (Analog Input Comparator): This is an analog-stochastic converter for external sensor input. Optional.
- LM (Local Memory): Local digital memory, which contains the registers and the counter (CNN-UM pairs: LAM, LLM)
- GPU (Global Programming Unit): Practically, this is a microcontroller which is able to upload templates.
- GCU (Global Control Unit): It handles the image upload/download.
- PMIC (Parallel Memory Interface Controller): Optional. It increases the writing and reading speed of the STCNN cell matrix, by multichannel DRAM access.
- GRND (Global Random Number Generator): It should be a really good random generator. This gives the initial 'seed' to other generators that are chained together, so this initialize all of them indirectly.

A. Execution cycle

Main steps of running a program

1) Initialization: uploading the given image and templates and FSM program
2) Starting iteration: running a transformation to the given input image to create the bias map
3) Store: save bias map to a register
4) Switch: possible template change for the A template
5) Iterations: running A template
6) Loop: return to step 1 or exit loop.
7) Readout: downloading image to host
B. Binary operations

In most image processing algorithms after grayscale filtering some features extracted as binary maps. In the previous section we described running of a grayscale operation. Before readout threshold can be applied using the threshold register, and result can be stored as a bit-map. All circuit parts are designed to process binary inputs as stochastic streams but if we measure only a single result we can use our system in deterministic mode. If we by-pass the $g$ function using appropriate control the spatial integration function turns to be a morphologic hit-and-miss operator.

VII. HARDWARE MODULES OF STCNN-UM

A. Main module

Figure 6. This is the structure of an STCNN-UM cell. This figure shows the FSM, RNG, LCCU, LM and STCNN nucleus modules and their interactions.
Let us review the elements of the main module (Figure 6):

- \( g(x) \): Nonlinear interaction function
- \( h(.) \): Spatial integration function. This realizes function \( h(.) \) and has 10 input bitstreams, 8 for neighboring cells, 1 for feedback and 1 for bias map. This component has four outputs, that can be accumulated by the counter.
- **Counter and divider**: Accumulates the output values of the \( h(.) \) block. It uses 4-output as a single binary number. After \( N \) measurements the average is calculated, and normalised. This is necessary in order to transform output to \([0, 1]\).
- **Bin. to stoch.**: This component converts a binary number into a stochastic bitstream.
- **Threshold register**: It’s an optional thresholding module to convert the image to binary.
- **Binary registers**: 1 bit registers, binary image can be stored here.
- **8 bit registers**: general registers storing 8 bit values, used mostly for grayscale operations.
- **8 bit registers (RO) from 4 neighbours**: Dedicated 8 bit registers connected to the neighbours. Can be used to implement fast spatial image shifting.
- **Binary registers (WO) from 4 neighbourhoods**: Dedicated binary registers connected to the neighbours. Can be used to implement fast spatial image shifting.
- **MUX**: It drives the bias input of the \( h(.) \). The possible input choices are stochastic bitstream, constant 0 or 1, or a binary register
- **Reference stochastic bitstream generator**: A pseudo random generator for generating uncorrelated stochastic bitstreams with expected value of 0.5. It can receive initialization value ‘seed’ from other cells’ random generator.
- **Programmer and readout module**: Since the whole program (template) consists of programmable logical functions, the programming is a memory write. This module is responsible for the template and image loading, and readout.
- **Control module**: This module controls the cell. It is responsible for register read/write iteration control, control wire settings.

**B. Reference random flow generator**

The greatest advantage of stochastic bitstreams is that basically an analog signal can be processed by minimal amount of digital components. The operations are deterministic on the streams, therefore it is possible to design well-defined circuits with predictable outputs. On a digital implementation, the operations can be implemented by logic functions, the delay can be realized by shift registers. For creating stochastic bitstreams, we need hardware white-noise generators or pseudo random generators. The random generators can be created by shift registers, using linear feedback shift registers. Every kind of pseudo random generator loops through a characteristic series of numbers. The accuracy of the STCNN is sensitive to the length of this loop.

Bitstreams fed to the same logical function must be independent, which is obtainable by shifting the bitstreams in time, using shift registers. Bitstreams of arbitrary value can be produced by logical function with several independent bitstreams. This makes it necessary that we have a vast amount of pseudo random bits available.

If we want to realize a real random generator, we cannot escape the complications of analog VLSI chip design during implementation.

Pseudo random number generators using shift registers proved to be the most efficient. On a programmable hardware, a huge amount of shift registers is available, and the statistical randomness of this method is still within acceptable limits, see Figure 7. Its working principle is that shift registers are connected sequently each after each, constructing a long one, which is

![Figure 7. Pseudo random generator using shift register (linear feedback shift register). We add up the tapped bits in GF(2) algebra, that is, we XOR them in pairs. The result will shifted to the first bit of the shift register. Output could be connected to any bit, usually the end of the register is used.](image-url)
It depends on the taps and register length how many states the pseudo random number generator can have before it starts the sequence from the beginning. A 17 bit LFSR (linear feedback shift register) on Figure 7 has \(2^{17} - 1\) possible stages. To run through all of its possible stages, the polynomial created from the sequential number of the tapping spots should be \(n\)th degree and it should be a primitive in GF(2) algebra. If the tapping spots are 17 9 5 4, then the polynomial is:

\[
x^{17} + x^9 + x^5 + x^4 + 1
\]

(35)

If there are two correlated bitstreams and we would like to make an operation on them, then with the help of a shift register we can insert a delay, as it was discussed in III-E. If the period of the random number generator is shorter than the longest delay on the stream, then there will be spurious correlations. Moreover, if the period of the random generator is too short, then the empirical average taken from the emitted bitstream will approximate the expected value badly.

\[
E\xi_n \approx \frac{1}{b - a} \sum_{k=a}^{b} \xi_k
\]

(36)

Let the period \(\frac{b-a}{2}\), in this case period < \(b - a\), then:

\[
E\xi_n \approx \frac{1}{b - a} \sum_{k=a}^{b} \xi_k = \frac{1}{b - a} \left( \sum_{k=a}^{a+b} \xi_k + \sum_{k=a}^{b} \xi_k \right)
\]

(37)

Because of the short period of the random generator, we add the same values twice:

\[
\sum_{k=a}^{a+b} \xi_k = \sum_{k=a}^{b} \xi_k
\]

(38)

\[
\frac{1}{b - a} \left( \sum_{k=a}^{a+b} \xi_k + \sum_{k=a}^{b} \xi_k \right) = \frac{2}{b - a} \sum_{k=a}^{a+b} \xi_k = \frac{1}{\text{period}} \sum_{k=a}^{a+\text{period}} \xi_k
\]

(39)

This way, the effective averaging interval will be determined by the period of the generator, thus we cannot achieve the desired accuracy.

We would like to generate reference bitstreams with given reference values, from which we could later create the bitstreams that run STCNN-UM. We would like to create reference stochastic bitstreams with the value of \(2^{-i}, i = 1...n\). Meanwhile, the random generator gives a bitstream with the expected value 0.5, so all others should be generated from this by multiplication, i.e. AND gates.

\[
\begin{align*}
2^{-1} \cdot 2^{-4} &= \mathbf{E} \mathbf{A} \\
2^{-2} &= 0.5 \cdot 0.5 = \mathbf{E} \mathbf{B} \cdot \mathbf{E} \mathbf{C} \\
2^{-3} &= 0.5 \cdot 0.5 \cdot 0.5 = \mathbf{E} \mathbf{D} \cdot \mathbf{E} \mathbf{E} \cdot \mathbf{E} \mathbf{F} \\
2^{-4} &= 0.5 \cdot 0.5 \cdot 0.5 \cdot 0.5 = \mathbf{E} \mathbf{G} \cdot \mathbf{E} \mathbf{H} \cdot \mathbf{E} \mathbf{I} \cdot \mathbf{E} \mathbf{J}
\end{align*}
\]

(40)

Here, \(A, B, C, D, E, F, G, H, I, J\) are the outputs of the random number generators.

C. Binary to stochastic conversion

\[
\begin{array}{c}
\text{Digital comparator} \\
\text{binary number} \quad \text{>} \quad \text{stochastic bitstreams}
\end{array}
\]

Figure 8. This module can convert a binary value into a single stochastic bitstream with the help of identically distributed independent reference streams. The output bitstream carries the input value downscaled to \([0, 1]\) as its expected value.

For the feedbacks in STCNN, we have to make the values we gained during the previous iteration stochastic again. The mathematical model of this is discussed in III-A subsection. On a digital hardware we should use binary representation. The random binary value comes from reference stochastic bitstreams, and the target value is also binary with the same bit width.

An 8bit implementation can be seen on Figure 8. If the input binary number (target value) and the binary number created from the reference bitstreams in the given clock cycle are equal, then it is uncertain whether the output should be 0 or 1, but the chance of this is \(\frac{1}{2^8}\), thus this inaccuracy is insignificant.
D. Input layer

Stochastic data are provided by almost every analog sensor. To extract information from this data stream, the standard practice is to integrate for relatively long time. We can attach analog sensors this way, but then we need to convert the signal to stochastic representation (Figure 9). This can be done like it was discussed in III-A. However, if an appropriately short integration time is used, the output is a 0-1 stochastic data stream where the carried information equals to the expected value. Then the signal of the sensor can be processed by a stochastic data stream based system without any conversion. This functionality is realized by the AIC - Analog Input Comparator using a white noise signal as reference.

![Figure 9. AIC - Analog Input Comparator. This module can be used to convert analog signal produced by external sensor, like CMOS light sensor or thermometer](image)

VIII. STCNN TEMPLATE DESIGN

The template of STCNN is the description of all $g(x)$ cell interaction function and $h(.)$ spatial integration function. $g(x)$ is a composition of a polynomial and a multiply and offset function. Basic implementation for 4th degree polynom needs 2 times 16bit for $L_{poly}$ and $L_{MOU}$. The $h$ function is a 10 input logical function. As it was mentioned in V-B to represent all possible $h$ function 4 times 1024 bit is needed.

We can generate $L_{poly}$ and $L_{MOU}$ values for a given $f(x)$ polynom using euclidian distance measure and simple automated try and error method. For $h$ function we can use simple summation. This needs very low hardware complexity compared to generic realisation. It is practical to define symmetrical $h$ function and place asymmetries to $g$ functions. This gives useful constraints that reduces the problem space so we can algorithmically generate the description. Of course most of the problems still need high computational effort. Hopefully, a template library will be started soon, which contains the basic image processing operations and filters.

IX. SIMULATION RESULTS

Basically, there are two possible levels of simulations. First, the system can be simulated on low level tracking the bitstreams through logical functions, or high level tracking expected values through the transformation functions. It is easier to implement simulation environment on the level of expected values, and the simulation speed is also higher compared to low level. In the course of our work, we implemented both environments in C++ language. For bitstreams we were also using 3rd party VHDL simulator as well. Our measurements showed that low level simulation results converged to expected value level simulations while we were increasing the averaging sample count the results converged fulfilling the expectations that were described in section III-C.

The Game Of Life (GOL) cellular automata can be implemented on the system using binary operations in deterministic mode (Figure 10). The GOL can implement a universal turing machine, so the turing machine can be implemented on STCNN-UM also.

Further results can be seen in the diagrams below (Figure 11 and 12) for random generated templates, showing the effects of nonlinear interaction and some steps of a wave propagation.

To prove the affectivity of our system some basic algorithms were tested. In Figure 13, two selected experiments are introduced. The first one is an edge detection experiment. The templates were generated intuitively and then $g$ functions were improved by optimization. The second row shows the results of the different steps of the algorithm.

A correlation estimation algorithm was also designed. First the input image is saved to the bias map. Then the image is shifted using register interaction. As we have two operands we can join the center elements to enhance correlation. Finally after some spatial lowpass filtering the image was thresholded to give final binary result.
Figure 10. Sample image sequence of the Game Of Life cellular automata, which can be implemented on the system with a single template in binary mode.

Figure 11. Randomly generated templates running on a test image, presented in the first row. The second row shows the results of different templates executed on the input image once. The third, fourth and fifth row show the results of 3 different templates, executed for 3 iterations they show diffusion like property.

Figure 12. In the first row, the input image is shown. In the second row, the results of random template executed on the input image. They produce center-surround like operations especially the last one.
Figure 13. In the first row, the input image of the edge detection experiment can be seen. The second row shows the results of the different steps of the algorithm: the first image is the output of a spatial filtering. On the second result for absolute value estimation is presented. The third is the result after thresholding. The third row shows the input image of the correlation experiment. The fourth row shows the results of the different steps of the algorithm belonging to this.

X. FPGA IMPLEMENTATION

FPGA arrays has special cellular structure. The main building blocks are CLBs - Configurable Logic Block and additional block RAM memory modules. All CLB consist of few slices. In the Xilinx Spartan-3 family slices are made up from 2 four input Look up tables (LUT) and 2 D-flipflop registers. A look up table can hold any 4bit → 1bit logical function. This is similar the the truth table representation. In the theoretical part of this paper we were defining operation using 4bit → 1bit logical functions. This was a practical consideration for efficient hardware mapping.

The syntheser module of the FPGA developer environment is responsible for mapping more complex logical functions to hardware structures thus more complex cell interaction functions can be used but 4 → 1 is the minimal reasonable complexity. In some cases functions are mapped to LUT in other cases to block memory. If it is mapped to memory it can be changed in run time, otherwise it ishardwired and possibly optimized using logical equivalences. If we would like to use more than one or dynamic h function the truth table of it should be realized by memory. It consumes 4 times 1024 bits so it can be mapped
only to block RAM by the synthesizer. The size of the block RAM is limited depending on the type of the FPGA. If we can use predefined $h$ function the hardware complexity of a single cell can be reduced significantly.

In our experiments we were using a Xilinx Spartan-3 FPGAs (XC3S1500 and XC3S50). We made three hardware realisations. In first experiments we implemented a fully programmable $g$ and $h$ nucleus structure. A small 7 by 7 array was synthetized utilizing 100 percent of the block memories and 68 percent of the slices. The second experimental design was simplified to use summation as $h$ function. Summation for 10 inputs and 4 outputs can be realized using 14 LUTs. A 15 times 15 cell array can be placed to the same FPGA. The third realisation was implemented on the smallest Spartan-3 FPGA.

The following modules were also implemented to complete the functionality of the STCNN nucleus: control unit, Local Memory, LFSR RNG - linear feedback shift register random generator, Global Random Number Generator and the Programmer&readout module. We sum up design considerations for all modules. Because of the size limitation of the available FPGA and the high VHDL simulation time, significant simplification of the units were necessary.

- Local memory: Two feedbacks, 8 registers and the counting dividing unit remained, and the direct connection between the cells and the binary registers were omitted. To make the programming of the cells easier, we included an 8 bit wide shift register chain. This structure can be used for uploading and downloading images.
- Local control unit, Global control unit, Programmer&readout module, Global Programming unit: We were using small array size so we could insert a single controlling unit and it was able to fulfill all tasks without any wiring problems. We were using RS232 connection to the host PC.
- LFSR RNG: this is ordinary pseudo random generator using linear-feedback shift register. The output of the generator was connected to an adequately long (272 bit) shift register, which is tapped at every 16 bits so we get 18 independent bitstreams. As a consequence of at least 16 bit time shift between any of them, these bitstreams can be regarded as statistically independent. Three units like this, run with different starting values generates 3*18 referential bitstreams, which supplies the LM and the STCNN nucleus appropriately.
- Global Random Number Generator We were using the Rx pin of the RS232 port to initialize all ‘seeds’. During uploading images and templates this line has many transients that can initialize LFSRs.

**Performance**

On an XC3S50 FPGA, a few STCNN cells were implemented, it runs at 93MHz with minimal error, consumed 793 Flip Flops of 1536 and 642 four-input LUTs of 1536. The implementation actually contained 15 cells but only 3 were real non-boundary STCNN cells. Tiling was applied for processing bigger images. We tested it on different clock frequencies 93MHz, 100MHz, 120MHz as you can see in Figure 14. The cells contained only the most necessary modules for the operation. It was non-programmable in the sense that after programming the FPGA the template remained the same until the next FPGA programming. The big LUT was implemented as optimized logic consuming much less resource than the theoretical maximum. One local memory and a 16bit counter were implemented, with a shift register chain for readout.

![Figure 14](image-url) Gray-scale edge detection results where the STCNN was implemented on an XC3S50 FPGA with different clock frequencies. A) input image, B) result at 93MHz, C) result at 100MHz is significantly noisy driving the STCNN over the safe frequency. D) result at 120MHz is complete noise without actual data, the STCNN implementation completely fails to work at this frequency.

**XI. Conclusion**

This paper presented a new type of CNN architecture and FPGA implementation of CNN-UM built on stochastic bitstreams. We compared the structure of STCNN to conventional CNN architecture. Using this method, different nonlinear polynomial weight functions can be implemented for all cell interaction instead of using weighted form of a uniform nonlinear function. We designed additional modules to complete STCNN nucleus to a universal machine. A simulation framework was created to analyze this UM in an efficient way. Results for two different algorithms were presented. We also implemented a small array in real FPGA hardware that were working as it was expected from simulations.
ACKNOWLEDGMENT

The Operational Program for Economic Competitiveness (GVOP KMA) which supports the multidisciplinary doctoral school at the Faculty of Information Technology of the Pázmány Péter Catholic University is gratefully acknowledged. The authors are also grateful to Professor Tamás Roska, Gaurav Gandhi, Ferenc Lombái and Alpár Sándor for the discussions and their suggestions. And special thanks go to Zsófia Cserey and Éva Bankó.

REFERENCES