A full-parallel digital implementation for pre-trained NNs

Tamás Szabó, Lőrinc Antoni, Gábor Horváth, Béla Fehér
Technical University of Budapest, Department of Measurement and Information Systems,
H-1521. Budapest, Műegyetem rkp. 9, Bldg. R. I./113. Hungary,
Tel.: +36 1 463 2057, Fax.: +36 1 463 4112, E-mails: [szabo, antoni, horvath, feher]@mit.bme.hu

Abstract *In many applications the most significant advantages of neural networks come mainly from their parallel architectures ensuring rather high operation speed. The difficulties of parallel digital hardware implementation arise mostly from the high complexity of the parallel many-multiplier structure. This paper suggests a new bit-serial/parallel neural network implementation method for pre-trained networks. The method makes possible significant hardware cost savings. The proposed approach - which is based on the results of a previously suggested method for efficient implementation of digital filters - uses bit-serial distributed arithmetic. The efficient implementation of a matrix-vector multiplier is based on an optimization algorithm which utilizes the advantages of CSD (Canonic Signed Digit) encoding and bit-level pattern coincidences. The resulting architecture performs full-precision computation and allows high-speed bit-level pipe-line operation. The proposed approach seems to be a promising one for FPGA and ASIC realization of pre-trained neural networks and can be integrated into automatic neural network design environments. However, these implementation methods can be useful in many other fields of digital signal processing.*

1 Introduction and Motivation

Neural Networks (NNs) mean very attractive solutions for numerous real-world computational problems of digital signal processing. However, the lack of suitable implementation impedes the spread of NNs in everyday systems. There is a very wide scale of attempts using different implementation mediums. The main research directions are: analog neurochips, digital circuits, mixed signal (pulse-coded or other hybrid hardware) implementations and some optical devices [1] [2]. Each solution has its strengths and weaknesses.

Both analog and digital implementations have advantages and disadvantages. Analog implementations are quite sensitive to noise and thermal drift, and because of the variance of the parameters only a 5-8 bit precision can be attained. On the other hand adding, multiplication, non-linear functions etc. are relatively easy to implement using analog technique and their implementation operates at high speed.

Digital implementations assure arbitrary precision and they are not sensitive to noise and thermal drift. Nevertheless their operating speed is quite low and their realization is expensive as they need large chip-surface.

Our goal was to examine digital NN implementation techniques and develop new methods that exploit the advantages of digital implementations, but allow efficient high speed operation at reasonable hardware cost. The proposed methods are intended for dedicated (single chip) neurochip designs. However, these solutions are not only for NNs, but they can be very suitable to implement various digital signal processing algorithms.

The focal points of this paper are:

- to find an efficient digital implementation for fixed NNs,
- to develop an optimization algorithm for a bit-serial matrix-vector multiplier.

2 Background

There are numerous NN families in the literature. They have different structures, but a common feature is that they consist of neurons in a layered structure, typically using one or two hidden layers. A layer is composed of neurons working on the same inputs and generating different outputs. A neuron produces a function of the weighted sum of its inputs (a function over a linear combination of the input). This function is called activation or squashing function, which is typically some monotone nonlinear function (e.g. sigmoid, tangent hyperbolic, etc.).

The operation of a neuron can be interpreted as a function over a vector-vector scalar product (MISO system), where one of the vectors is composed by the inputs and the other one is the weight vector. If a whole NN layer is considered then it is a MIMO system where the input and output are also vectors. The projection between the input and output of an NN can be described as a function of a matrix-vector product:

*This work was supported by the Hungarian Fund for Scientific Research (OTKA) under contract T021003

0-7695-0619-4/00 $10.00 © 2000 IEEE

$$\mathbf{y} = f(\mathbf{W}\mathbf{x}) \tag{1}$$

where $\mathbf{x}$ and $\mathbf{y}$ are the input and output vectors, $\mathbf{W}$ is the weight matrix of the layer and function $f(\ .)$ represents the activation function of the neurons. NNs have two basic operation modes: the learning phase and the recall phase. If adaptivity is not required in a system then it is enough to implement a fixed network working in recall phase (fixed transformation). The elements of $\mathbf{W}$ are constant in this case, contrary to the learning phase where the weights -the matrix $\mathbf{W}$- change.

Here we deal with the implementation of constant matrix - variable vector multiplication for pre-trained NNs. We shall not deal with the implementation of the activation function in this paper. It is less difficult to implement the squashing function than the implementation of the matrix-vector multiplier, it can be done for example with Look-Up Tables (LUT) or a LUT-based interpolation technique.

One of the most essential property of the NNs is their inherent parallelism. Mainly this property is responsible for their high computational speed, power and fault tolerance. We can define full-parallel implementation in the following sense: the layer is computed full-parallel if all the products in every neuron are computed simultaneously. However, as it was mentioned, the implementation of digital parallel multipliers is very expensive. Usually the technical limits of the realizations do not allow full-parallel implementations because bit-level parallelism results in many connections that are difficult to implement. Besides multipliers, another bottleneck of the hardware realization of NNs is the high number of connections, because the implementation of wires consumes large chip surface. Bit-serial communication and computation is a good candidate to help us to overcome these problems. Bit-serial computation reduces the number of necessary arithmetic elements and keeps the number of connections tractable. The reduction -respecting the number of necessary wires and arithmetic elements- is about one order of magnitude depending on the precision of the computation. Unfortunately, bit-serial computation decreases the operation speed, too, but we hope that the proposed full-parallel implementation compensates this speed fall.

We can conclude from the properties described above that in our implementation we should exploit the inherent parallelism of NNs and that a good candidate to implement the matrix-vector multiplier (where the matrix is constant and the vector is variable) should be a structure which processes its input vectors parallely in viewpoint of the elements of the vector, but uses bit-serial data-flow. Certainly, the output vector of the multiplication ($\mathbf{y}$) is also available simultaneously in bit-serial manner. See Fig. 1.

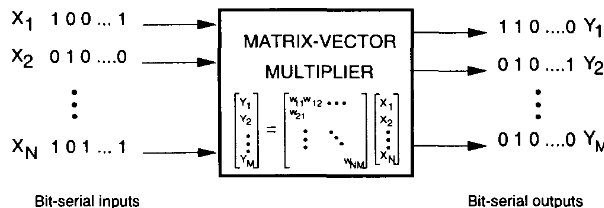

Figure 1: Bit-serial matrix-vector multiplier

In the following section we will present an approach to implement the constant matrix-variable vector multiplier. The proposed approach decreases the redundancies of the implementation in order to achieve efficient hardware.

3 The proposed constant matrix, variable vector multiplier architecture

A completely new approach has been published in [3] for vector-vector inner product computation used in digital filter implementations. This approach does not require the storage of the constant coefficients in the product but "builds" them into the structure of the multiplier. This technique can be called as a special type of Distributed Arithmetic (DA) approach. In this paper we present the generalization of this inner product arithmetic for matrix-vector products and propose a new optimization algorithm for matrix-vector multipliers and a new application area for DA, namely the field of NNs. One of our previous papers [4] deals also with a similar problem but the solution proposed there is less efficient than the current one.

It is possible to distinguish bit-serial/parallel and bit-parallel/parallel multipliers. Bit-serial/parallel multipliers read one operand bit-serially and the other one bit-parallely, while bit-parallel/parallel units read both operands parallely. Bit-serial/parallel multipliers can be derived from bit-parallel/parallel multipliers by the projection of spatial coordinates to time domain scheduling. The resulting bit-serial/parallel multiplier is composed of one-bit serial adders, delay elements and one-bit multipliers (AND gates). This transformation can be done in several ways but there is no significant difference between the hardware cost of the resulting architectures. Our method uses LSB first, bit-serial/parallel arithmetic as a constant coefficient multiplier. The coefficients are built into the multiplier "netlist".

Here we review the methods we applied to reduce the hardware cost of the implementation. The first technique that helps us to realize this reduction is Canonic Signed Digit (CSD) encoding [5]. It can be considered that if the coefficients are constant then it is not necessary to implement any hardware in the "0" digit positions of the constant numbers. The application of CSD encoding exploits this possibility. Besides this, we created an algorithm to attain bit-level optimization.

3.1 The CSD representation

CSD coding is a convenient method to reduce complexity in DSP hardware [5]. Allowing $\{0, +1, -1\}$ in the representation of the coefficients the relative frequency of non-zero elements in CSD representation equals to $B_c/3$ (against $B_c/2$ in two's complement or binary data), where B_c is the number of bits (word length) of this representation. Nevertheless, usage of ± 1 values does not mean extra hardware cost because $+1$ means bit-serial adder while -1 means bit-serial substractor and their hardware costs are the same. This implies that a constant multiplier can be implemented by less hardware using CSD coding while the dynamic range of the multiplier also slightly expands using the same number of bits. (The largest and smallest numbers which can be represented in a normalized CSD code are $-4/3$ and $4/3$). The CSD representation is unique, i.e. a given number may be represented in only one way. Let us see an example: $-0.4414_{DEC} = 0.0 - 1001000 - 1_{CSD}$ that is $-2^{-1}+2^{-4}-2^{-8}$. Its CSD representation contains three non-zero elements while its two's complement representation is composed by five ones.

3.2 Bit-level optimization

The first idea was the application of CSD coding, the second one was bit-level reduction. To understand it let us follow this simple example. Assume an $N * M = 2 * 1$ dimension matrix-vector product:

$$y = \begin{bmatrix} \frac{13}{32}_{DEC} & -\frac{39}{32}_{DEC} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 & 1_{CSD} \\ -1 & 0 & -1 & 0 & 1 & 0_{CSD} \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{2}$$

This can be rewritten as:

$$y = (2^{-1} - 2^{-3} + 2^{-5})x_1 + (-2^0 - 2^{-2} + 2^{-4})x_2 = -2^0 x_2 + (2^{-2} - 2^{-4})(2^1 x_1 - x_2) + 2^{-5}x_1 \tag{3}$$

If we introduce $x_3 = 2^1 x_1 - x_2$ we obtain:

$$y = -2^0 x_2 + 2^{-2}x_3 - 2^{-4}x_3 + 2^{-5}x_1 \tag{4}$$

We can compare the original (corresponding to Eq. 3) and the optimized multiplier (Eq. 4) structure on Fig. 2. The difference is obvious. Both of them use CSD encoding but the reduced architecture performs the same operation using less building blocks. The original one contains six arithmetic and five delay elements while the reduced one needs only five arithmetic and three delay elements.

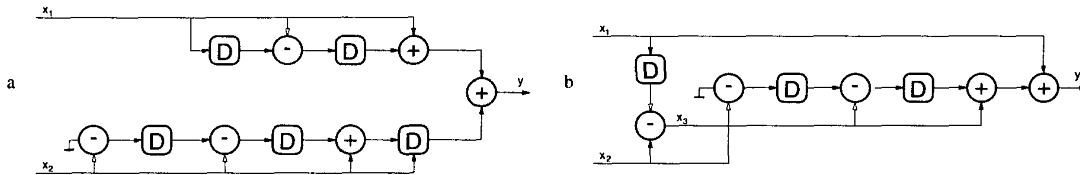

Figure 2: Bit-serial vector-vector product implementations: a) original, b) reduced

We shall not examine the mathematical formalism of the bit-level optimization algorithm in detail in this paper. The description of the formalism can be found in [3].

3.3 The optimization algorithm of the constant matrix, variable vector multiplier

In this section we introduce an optimization algorithms which utilize the ideas of sections 3.1 and 3.2 for bit-serial matrix-vector multipliers. First we introduce some notations. Given a decimal number $w_{i,j}$ in $[-4/3, 4/3]$. Its B_C-bit long CSD representation can be interpreted as a B_C-long vector $\mathbf{h}_{i,j}$, the elements of which are the bits of the CSD number $\mathbf{h}_{i,j} = \begin{bmatrix} h^0_{i,j}2^0 & h^1_{i,j}2^{-1} & \dots & h^{Bc-1}_{i,j}2^{-B_c-1} \end{bmatrix}$ (e.g. if $w_{i,j} = -0.875$, $B_C = 8$, $\mathbf{h}_{i,j} = [0. - 10010000]$). Similarly, let us denote B_x the word length of the input data ($\mathbf{x}$).

Thus, we rewrite the matrix-vector multiplication of Eq. 1 to a series of vector-vector products:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} = \begin{bmatrix} f(\sum_{k=1}^{N} w_{1,k}x_k) \\ f(\sum_{k=1}^{N} w_{2,k}x_k) \\ \vdots \\ f(\sum_{k=1}^{N} w_{M,k}x_k) \end{bmatrix} \tag{5}$$

In this equation the partial matrix-vector products, $\sum_{k=1}^{N} w_{i,k}x_k$ are actually inner products of the input vector $\mathbf{x}$ and a constant coefficient vector $\mathbf{w_i}$. A two-dimensional data structure in which the rows are the binary representations of the coefficient vector elements with an implicit + sign between the cell-array row elements can represent a one-dimensional coefficient vector. Eq. 6 shows this.

$$\begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,N} \end{bmatrix}^T = \begin{bmatrix} \mathbf{h^i_1} \\ \mathbf{h^i_2} \\ \vdots \\ \mathbf{h^i_N} \end{bmatrix} = \mathbf{H_i} = \begin{bmatrix} h^0_{i,1}2^0 & h^1_{i,1}2^{-1} & \cdots & h^{B_c-1}_{i,1}2^{-B_c-1} \\ h^0_{i,2}2^0 & h^1_{i,2}2^{-1} & \cdots & h^{B_c-1}_{i,2}2^{-B_c-1} \\ \vdots & \vdots & & \vdots \\ h^0_{i,N}2^0 & h^1_{i,N}2^{-1} & \cdots & h^{B_c-1}_{i,N}2^{-B_c-1} \end{bmatrix} \tag{6}$$

As we have seen in the previous section identical or contradictory bit patterns have to be found in these $\mathbf{H_i}$s. It can be seen that longer matching bit patterns result in a higher reduction rate, so we prefer the reduction step considering the longest bit-pattern. First of all, let us examine how these matching bit patterns can be found in a single $\mathbf{H_i}$ if we use CSD encoding.

3.3.1 Identification of matching bit-patterns

During this task we will exploit the feature of CSD encoding that in a CSD number there cannot be non-zero values on adjacent digits, that means we must search for patterns having non-zero values at the first and the last position. The shortest pattern of this kind is 3-bit long, for example {101}. The identification of bit-patterns must be carried out between the rows of $\mathbf{H_i}$, after the partial product corresponding to the bit-pattern can be separated that means the introduction of a new variable as we have discussed it in section 3.2. In our research we have focused on the identification of the identical bit-patterns only in one $\mathbf{H_i}$, we have not extended the algorithm to search for patterns in different $\mathbf{H_i}$s. This is because it would significantly increase the searching space and the complexity of the optimization algorithm but would not surely yield better result. However, the reductions are performed between the different $\mathbf{H_i}$s if necessary as we will see later. It means that our algorithm makes reduction respecting the whole matrix-vector product in every step.

Thus, we must search for identical bit-patterns between the rows of $\mathbf{H_i}$. When doing this we compare the rows $q, r = 1...N$ of $\mathbf{H_i}$ and store the result in so-called statistical matrices $\mathbf{S}_i p$s. The statistical matrices $\mathbf{S_i}p(q,r)$ contain the largest number of non-zero elements at the same position of coinciding identical (or contradicting) bit patterns from the q-th and the p-position right-shifted r-th rows of the actual $\mathbf{H_i}$ matrix with regard to the sign. The sign of the element of $\mathbf{S_i}p(q,r)$ will be positive if the sign of the corresponding bit-pattern is the same, and negative if it is different. The diagonals of $\mathbf{S_i}p$ are cleared to ignore self-coincidences.

With the statistical matrices we have determined the number of non-zero values of the corresponding bit-patterns in each row. In our algorithm, among the bit-patterns the one is chosen in each step which results the highest gain. From this point of view the algorithm is similar to the gradient-methods as it always chooses the steepest gradient. It is possible to find the optimal or nearly optimal solution with the $\mathbf{S_i}p(q,r)$ statistical matrices and a $\mathcal{C}_{\mathcal{AGG}}(\,)$ cost-function. The cost-function must consider two other important factors, as the possible gain depends not only on the number of non-zero elements of the bit-pattern chosen, but:

- on the gain the given bit-pattern results in the whole $\mathbf{Wx}$ matrix-vector multiplication (that means in all the $\mathbf{H}_i$s)
- on whether we must introduce additional delay units into the structure or not.

The delay units mentioned in the second point realize the possible 2^D delay (delay by D clock cycles). We must consider that we may have realized a certain delay for x_k already in a preceding step of the optimization, so we should consider only the ΔD_k difference between the existing and the desired delays. In this case the cost of the implementation of the needed delay can be described as:

$$\mathcal{C}_{\mathcal{D}}(p,k) = 1 + \Delta D_k C \tag{7}$$

where C is a technology-depending parameter that represents the cost-ratio of the implementation of the additional delay unit(s) and the arithmetic operator (1-bit adder or substractor).

It is not sure that the pattern containing the most non-zero values must be chosen during the optimization. It can be possible that a pattern containing less non-zero values but involved in more partial products (that means in more $\mathbf{H}_i$s) is more reasonable to choose as it results more gain, so the $\mathcal{C}_{\mathcal{AGG}}(\ )$ cost-function can be chosen in many different ways like 8 or 9:

$$\mathcal{C}_{\mathcal{AGG}}(p,q,r,C) = absmax_{p,q,r}\left[\mathbf{S}_i p(q,r)\right] - \mathcal{C}_{\mathcal{D}}(p,r), \tag{8}$$

$$\mathcal{C}_{\mathcal{AGG}}(p,q,r,C) = absmax_{p,q,r}\left[\sum_{\forall i, S_i p(q,r)>0} S_i p(q,r), \sum_{\forall i, S_i p(q,r)<0} S_i p(q,r)\right] - \mathcal{C}_{\mathcal{D}}(p,r) \tag{9}$$

The cost-function Eq. (8) always chooses the longest bit-pattern to separate with no respect how many times a shorter pattern occurs. The cost-function Eq. 9 adds separately the number of non-zero values at the same position of the bit-patterns and after chooses the locally maximum gain that can be obtained.

There is one more important step of the algorithm that must be considered. One must approximate the number p (the difference between the position of the identical bit-patterns) to know how many $\mathbf{S}_i p$-s must be generated. If we consider that in CSD representation the maximum number of non-zero values in a number is $\frac{B_c}{2}$ we can determine an upper limit to the value of p. According to practical experiences we can say that the optimal value of p is about $p_{max} = 2..3$.

When the algorithm is running the size of both the statistical matrices $\mathbf{S}_i p$ and the $\mathbf{H}_i$s changes dynamically. However, it can be seen that it is enough to re-compute in each iteration step the rows of the concerned $\mathbf{H}_i$ and the values of the concerned $\mathbf{S}_i p$. Like this the complexity of the algorithm will be linearly proportional with the number of the elements of $\mathbf{W}$.

3.4 Representative examples

The proposed approaches have been validated in this section by some representative examples. For these examples we assume that the weight distribution of a trained neural network is approximately Gaussian (see [6]). This is why the elements of the sample coefficient matrices were generated randomly in $[-(2^{Bc-1}), 2^{Bc-1}]$ with zero mean and($\sigma_2 = \frac{2^{Bc-1}}{3}$). (Approximate Gaussian distribution.)

The following table (Table 1) contains some representative results (averages of 5 runs using different matrices) for coefficient word length $B_c = 8, 12, 16$, respectively. The number of inputs N and the outputs M are $5, 10, 20, 40$. The columns contain the number of adders, substractors and delay units required. The hardware cost of the trivial realization can be approximated by $C_{TRI} = \frac{M*N*(B_c+1)}{2}$. The "R %" column shows hardware requirements in percent compared to trivial representation $(1 - \frac{C_{RED}}{C_{TRI}})$.

We can conclude from these examples that the reduction ratio increases proportionally to the dimensions (N and M). This can easily be explained because the more operations are in the matrix-vector product the more to reduce. However, it is not definitely true because the algorithm can execute different reduction steps, but it can be considered as a trend.

4 Application proposition

As it has been mentioned, our goal was to develop implementation techniques which are suitable for embedded applications. The proposed application area is the fixed, non-adaptive (single-chip) neural network implementation. The presented method can be very fast because of the full parallel design and at the same time can provide quite a large -theoretically arbitrary- commutational precision. Certainly, the parallel architecture can be exploited only if the inputs are presented in parallel manner. Such applications are, among

	$B_c = 8$					$B_c = 12$					$B_c = 16$				
NxM	**ADD**	**SUB**	**DFF**	$\sum$	**R%**	**ADD**	**SUB**	**DFF**	$\sum$	**R%**	**ADD**	**SUB**	**DFF**	$\sum$	**R%**
The results of the simulation obtained using Eq. (8):															
5x5	19.6	23.8	25.2	68.6	39.0	28.0	34.0	41.1	103.4	36.4	38.2	45.0	48.8	132	37.9
5x10	39.6	42.2	31.4	113.2	49.7	62.4	60.6	44.4	167.4	48.5	82.8	77.2	55.6	215.6	49.3
10x5	36.0	38.0	51.8	125.8	44.1	54.0	58.4	76.8	189.2	41.8	70.8	81.0	99.8	251.6	40.8
10x10	69,4	82.8	56.4	208.6	53.6	106.8	118.0	78.0	302.8	53.4	138.8	153.2	103.0	395.0	53.5
10x20	141.8	147.8	73.8	363.4	59.6	202.6	225.6	106.4	534.6	58.9	271.4	289.2	139.2	699.8	58.8
10x40	266.6	279.4	101.8	647.8	64.0	410.2	409.8	138.4	958.4	63.1	525.2	549.8	183.6	1258.6	63.0
20x10	130.6	147.2	105.0	382.8	57.5	202.4	218.0	145.8	566.2	56.4	268.6	277.8	193.0	739.4	56.5
20x20	262.6	275.2	132.6	670.4	62.8	390.2	408.6	186.6	985.4	62.1	526.6	535.6	240.4	1302.6	61.7
40x10	249.0	267.8	203.2	720.0	60.0	382.8	396.0	287.2	1066.0	59.0	500.8	520.0	381.2	1402.0	58.8
The results of the simulation obtained using Eq. (9):															
5x5	19.6	23.6	25.6	68.8	38.8	28.0	34.8	41	103.8	36.1	39.4	45.4	53.2	138	35.0
5x10	39.6	42.6	31.6	113.8	49.4	64.6	58.6	44.4	167.6	48.4	81.0	81.2	56.4	218.6	48.6
10x5	35.4	39.4	44.4	119.2	47.0	53.0	61.2	75.0	189.2	41.8	74.2	75.8	100.2	250.2	41.1
10x10	73.4	77.0	58.4	208.8	53.6	108.2	115.2	84.2	307.6	52.7	144.8	149.4	105.6	399.8	53.0
10x20	150.4	139.4	70.8	360.6	59.9	202.6	226.2	106.2	535.0	58.8	270.4	298.8	132.2	701.4	58.7
10x40	270.6	280.8	110.4	661.8	63.2	408.4	413.2	148.2	969.8	62.7	514.0	570.8	183.8	1268.6	62.7
20x10	137.0	140.8	103.2	381.0	57.7	200.8	220.6	142.6	564.0	56.6	276.0	275.2	190.0	741.2	56.4
20x20	253.6	284.8	139.0	677.4	62.4	398.4	413.4	190.0	1001.8	61.5	518.6	548.6	233.4	1300.6	61.7
40x10	247.0	270.6	188.0	705.6	60.8	383.6	402.2	283.6	1069.4	58.9	501.4	526.2	387.4	1415.0	58.4

Table 1: Reduction example

others, the image processing tasks (e.g. OCR, Optical Character Recognition, and other pattern and texture recognition), control applications (MIMO, static and dynamic function approximation).

Beyond the Multiple-Layer Perception (MLP) which is the plausible application area of this matrix-vector multiplier structure we have suggested another interesting application area in one of our previous paper [7]: the tasks of the CNN (Cellular NN) -operations with space invariant templates- that can be redefined into a form which exploits the benefits of the new matrix-vector multiplier structure.

5 Conclusion

We have presented a new implementation approach with significant hardware-reduction to a matrix-vector multiplier in this paper. In our approach the reduction in the hardware cost is based on CSD coding and bit-level optimization of the matrix-vector product. The architecture obtained allows high-speed bit-level pipe-line operation. Our approach is convenient for FPGA and ASIC realization of pre-trained NNs. The proposed method can be used in an automatic NN design environment where the input (**W**) of the matrix-vector multiplier optimization algorithm comes from an NN simulator and its output is processed by a silicon compiler. Future research can be carried out on the optimization algorithm as it is possible to find different methods in the searching of identical bit-patterns.

References

[1] Manferd Glesner and Werner Pöchmüller, *Neurocomputers, an overview of neural networks in VLSI*, Neural Computing. Chapman & Hall, 1994.

[2] J. G. Delgado-Frias and W. R. Moore, Eds., *VLSI for Artificial Intelligence and Neural Networks*, Pelnum Press, 1991.

[3] Béla Fehér, "Efficient synthesis of distributed vector multipliers," *Microprocessing and Microprogramming*, vol. 38, pp. 345–360, 1993.

[4] Tamás Szabó, Béla Fehér, and Gábor Horváth, "Neural network implementation using distributed arithmetic," in *Proceedings of the International Conference on Knowledge-based Electronic Systems*, Adelaide, Australia, 1998, vol. 3, pp. 511–520.

[5] Richard I. Hartly and Keshab K. Parhi, *Digit-serial computation*, Kluwer Academic Publishers, 1995.

[6] I. Bellido and E. Fiesler, "Do backpropagation trained neural networks have normal weight distributions?," in *Proceedings of ICANN'93*, 1993.

[7] Tamás Szabó, Béla Fehér, and Gábor Horváth, "Dedicated digital implementation of the CNN universal machine," in *Proceedings of IEEE International Workshop on Intelligent Signal Processing*, 1999, vol. 1, pp. 194–199.