Vol.2010-SLDM-145 No.11
2010/5/20

効率的なFPGA実装を指向した ニューラルネットワークのアーキテクチャ

林　圳　　董 宜平　　渡邊孝博

本稿では、多層ニューラルネットワーク（NN）をFPGAで実装する一般的なアーキテクチャを提案する。提案アーキテクチャは、リソースの使用効率を高めて、ネット遅延を削減するように工夫しており、NNのサイズが大規模になっても市販のFPGAチップ上に実現することができる。最も大きな特徴は層間マルチプレクシングと部分的なパイプライン方法を利用してマッピング方法を改善したことである。このアーキテクチャは、層の数と各層のニューロンの数が与えられる任意のNNに対して、適用することができる。実験の結果、従来の方法と比べて、提案した構造が非常にコンパクトで、高速度と低コストであることが分かった。

A General Neural Network Architecture for Efficient FPGA-based Implementation

Zhen Lin　Yi-ping Dong　Takahiro Watanabe

This paper presents a general architecture for a multilayer neural network (NN) to be implemented on FPGA. The proposed architecture is aimed at enhancing the efficiency of resource usage and reducing the net delay, so that a larger NN can be realized on a commercially available FPGA chip. A key feature is the mapping method, which has been exploited by using layer-multiplexing and partly pipeline manner. This architecture can be applied to any multilayer neural network composed of a given number of layers and a given number of neurons in each layer. Experimental results show that the proposed architecture can produce a very compact circuit and behaves the characteristics of higher speed and lower cost comparing with conventional methods.

1. Introduction

Artificial neural networks (ANNs) are characterized as an adaptive, robust, parallel computing model, which has the capability to learn from examples, approximate any given functions and classify nonlinear systems. It has been widely applied to the researches of signal processing, speech synthesis and analysis, pattern recognition, etc. [1]. Most of these applications require high-speed computation. But because of the inherent massive multiplication operation, the traditional software methods which are executed by serial computer cannot meet the requirements when the network size is large. So there is a necessary to develop such a hardware implementation that can exploit the inherent parallelism of neural network models.

There are many researchers trying to map NN into FPGA, because of the parallel implementation and reconfigurable ability supplied by FPGA [2, 3, 4]. However, there are still some challenges to implement NN in FPGA, such as the massive requirement for multiplier in synapses, the nonlinear calculation of active function and so on. The amount of hardware resource for them is so high that it's difficult to perform a complete NN in a single FPGA chip. One of the solutions for the problem is to use "time-multiplexing", that is, separate parts of the same system is implemented by time-multiplexing a single FPGA chip through run-time reconfiguration [5]. This method is not feasible when the application calls for high speed. Another typical solution called "layer-multiplexing" [6] can reduce the resource cost greatly at a higher level. This solution enables perform an 8-5-5-3 NN in the "XCV400hq240" FPGA board just with the total resource utilization rate of 62%. But this solution seems to be not so efficient when the net size is small because the resource utilization rate is not so high. On the other hand, when the number of layers grows up, the global forward speed measured by the interval starting from one input pattern imported to the obtainment of the output for this input pattern at the output layer comes down.

This paper presents an advantage in two basic respects with regard to previously reported neural implementations on FPGAs. The first is the mapping method which maps not one layer in layer multiplexing method but several layers. Among non-multiplexed layers, layer-pipeline manner is integrated. It can improve the resource utilization rate and enhance the global forward speed compared with the conventional layer multiplexing method. The second point we contribute is the flexibility which can adaptively construct the hardware structure according to the neural network parameter such as the neural topology, data structure and so on. We just need to modify these parameters during the compilation time but no other changes in modules, it can synthesis the architecture corresponding to the application automatically. We also develop a procedure to determine how many and which layers should

be mapped together at a time, according to the adopted FPGA chip and neural network topology. The upper limit of neurons count implemented in a single chip should be taken into consideration, and when the total number of neurons in the neural network to be implemented is smaller than this value, the physical neural architecture in FPGA chip maybe organized as whole pipeline manner to get the highest implementation speed, and on the other hand, the working manner specified as partly pipeline based on layer multiplexing that try to get the optimal tradeoff between the speed and cost.

The rest of this paper is organized as follows: Section II describes the circuit design of the single neuron and Section III presents the detail of the proposed architecture. Section IV validates the advantages of proposed method by the experiment. Finally, Section V concludes the paper.

2. NEURON DESIGN

2.1 Mathematical Model

A multilayer neural network is composed of one input layer which stores the input data, several hidden layers for computation and one output layer. Each layer consists of a set of processing elements called neuron and the main task of each neuron in neural networks is processing the following function:

$$y = f(x) = f(\sum_{i=1}^{n} \omega_i x_i + b) \quad (1)$$

where x_i stand for the *i*th input, and ω_i is the weight in the *i*th connection and *b* is the bias. The function *f(x)* is the nonlinear active function used in the neuron. Here we select the log-sigmoid as the active function due to its popularity, and it is described by the following:

$$f(x) = \frac{1}{1+e^{-x}} \quad (2)$$

Because the weights are stored in RAM that can be easily modified, we did not develop the online learning in our proposed architecture.

2.2 Neuron Circuit Design

As mentioned above, the computational resources required by a single neuron are multiplication block, accumulation block and active function block.

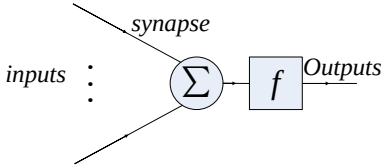

Fig.1. mathematical model of a neuron

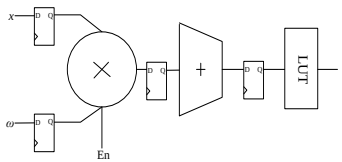

Fig.2. Neuron circuit block diagram

Figure 2 show the block diagram of the neuron circuit, where *En* is an enable signal that controls the neuron's state, working or not. The selection of word length/bit precision has an important impact on the output resolution, where longer bits means higher resolution but also larger resource cost. And in actually ANNs design, these parameters are set according to the application in order to achieve the efficient hardware implementation. In this paper, we perform the parameters as a variable, which can be modified by users in compilation time.

Assume that the input *x* normalized in the range of 0 to 1 and is N_i bits, 1 bit for sign and (N_i-1) bits for data. The weights are also represented in N_w bits: 1 bit for sign, N_{ww} bits for integer part and N_{wf} bits for fraction part, where $N_w = N_{ww} + N_{wf} + 1$. After multiplied and accumulated, the result becomes to N_r bits length: 1 bit for sign, $N_{ww} + N_{rw}$ bits for integer part and N_{wf} bits for fraction part, where $N_r = N_{ww} + N_{rw} + N_{wf} + 1$. The result of the active function is obtained as a signed N_i bits number, which is the same as the input.

The active function is realized by using LUT(look up table)[7] according to the fact that the modern FPGA chip has large number of built-in RAMs. As the active function is highly nonlinear, a general procedure to obtain an LUT of minimum size for a given resolution is as follows.

1) As mentioned before, the bit length of output from active function (formula 2) is N_i.

2) The actual output of active function is between 2^{-Ni} and $1- 2^{-Ni}$ when using the bit length N_i. Let *x1* and *x2* be the upper and lower limits of the input range, that is:

$$\frac{1}{1+e^{-x_1}} = 2^{-N_i}, \frac{1}{1+e^{-x_2}} = 1 - 2^{-N_i} \quad (3)$$

By solving (3) as *x* is variable, we can get:

$$x_1 = -\ln(2^{Ni} - 1), x_2 = +ln(2^{Ni} - 1) \quad (4)$$

3) Consider the fact that the step change in the output (Δy) is equal to 2^{-Ni}, and the corresponding minimum change in input is at the point of maximum slope, x = 0 in this case. So the minimal change value of input for the output change of 2^{-Ni} can be obtained from

$$\Delta x = \ln(\frac{0.5+2^{-Ni}}{0.5-2^{-Ni}}) \quad (5)$$

4) The minimal number of LUT values is given by

$$(LUT)_{min} = \frac{x_1 - x_2}{\Delta x} \quad (6)$$

3. Network Design

The function of network is to connect all the neurons and all the layers together, so the data can be forward through the connections from former layer to the latter layer. Fig.4 gives an example of multilayer neural network.

Vol.2010-SLDM-145 No.11
2010/5/20

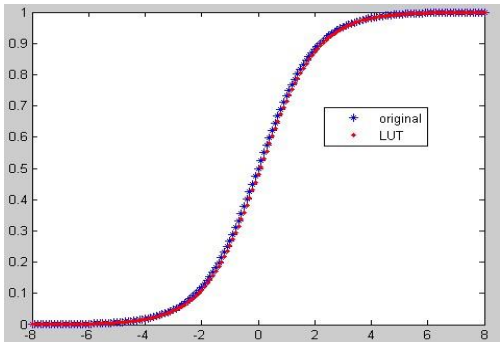

Fig.3. The experimental result between original and LUT

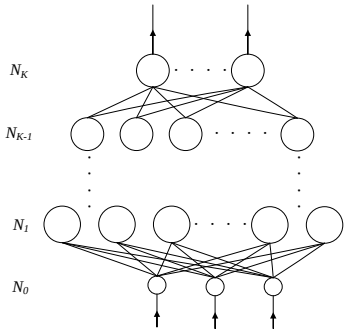

Fig.4. Multilayer neural network

Assume the symbols x_k^{ij} and y_k^i, stand for the input and output of neurons, respectively, where *k* is the layer number, *i* is the neuron number in this layer and *j* is the *jth* input of this neuron. Suppose the ID for input layer is *0*, then the forward process can be described as follows.

$$y_1^i = \sum_{j=1}^{N0} w_{ji}^{01} \cdot x_j, i \in [1, N_1], j \in [1, N_0]$$

$$x_k^{ij} = y_{k-1}^j, y_k^i = \sum_{j=1}^{N_{k-1}} w_{ji}^{(k-1)k} \cdot x_{ij} \quad (7)$$

$$k \in [2, N_K], i \in [1, N_k], j \in [1, N_{k-1}]$$

Where N_k is the total number of neurons in layer *k*, and $w_{ji}^{(k-1)k}$ stands for the weight between the *jth* neuron in layer *(k-1)* and the *ith* neuron in the layer *k*. Especially, the symbol N_k in Fig.4 means the total number of layers in this network and N_0 is the input number.

There are two concept of our proposed network architecture, including pipeline design and layer multiplexing design.

3.1 Pipeline Design

As mentioned above, the multilayer neural network has a characteristic that the neuron in the next layer depends on the neuron in the previous layer and there is no communication among neurons in the same layer. As shown in figure 5 (a, b, c), for an input pattern *m*, the traditional forward phase work as: firstly, the neuron in the first layer performs the neural computing for the input pattern *m* and neurons in the next layer are idle because they are waiting for the input coming from neurons in the previous layer. Secondly, when the neural computing has been completed, they send their result as input for the neuron in the next layer. So the neurons in the second layer become to work and the neurons in first layer are waiting for the end of this forwarding for input pattern *m*. At the last, when the signal arriving at the output layer, the other two would be in the idle or waiting state. Only the neurons in the output layer have accomplished they calculation and get the final result, the forwarding for the input pattern *m* could be over and there would be the next forwarding for another input pattern.

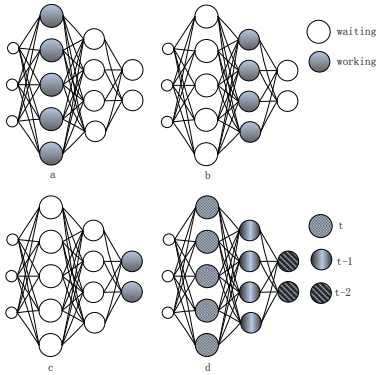

Fig.5. Non-pipeline vs pipeline

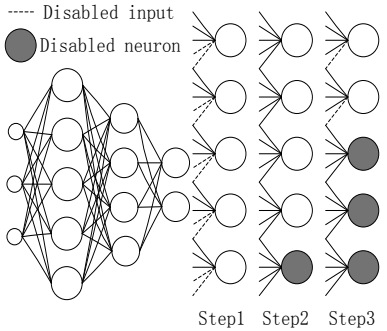

Fig.6. The process of layer multiplexing

In the conventional non-pipeline forward phase, there is only one layer working busily while the other layers just waiting. In order to save the cost, we integrate the pipeline manner in the layer architecture when forwarding. The fundamental pipeline algorithm is described as follows, where the symbol *t* is the time factor given by clock cycle.

$$y_1^i(t) = \sum_{j=1}^{N_0} w^{01}{}_{ji} \cdot x_j(t), i \in [1, N_1], j \in [1, N_0];$$

$$y_2^i(t) = \sum_{j=1}^{N_{k-1}} w^{12}{}_{ji} \cdot y_1^j(t-1) = \sum_{j=1}^{N_{k-1}} w^{12}{}_{ji} \cdot \sum_{j=1}^{N_0} w^{01}{}_{ji} \cdot x_j(t-1) \quad (8)$$

$$y_3^i(t) = \sum_{j=1}^{N_{k-1}} w^{23}{}_{ji} \cdot y_2^j(t-1) = \sum_{j=1}^{N_{k-1}} w^{23}{}_{ji} \cdot \sum_{j=1}^{N_{k-1}} w^{12}{}_{ji} \cdot \sum_{j=1}^{N_0} w^{01}{}_{ji} \cdot x_j(t-2)$$

As shown in figure 5 (d) and formula (8), when the first layer is under computing for the input pattern *t*, the second layer is busy calculating the result for pattern *t-1* which is transmitted by the first layer in the last cycle.

It doesn't need to wait for the end of some input pattern forwarding, but all the neurons in different layers are working simultaneously with different input pattern. By using the pipeline, the global forwarding speed would be much higher than the non-pipeline method. And comparing with non-pipeline method, there would not be much incidental cost or changes in the architecture by using proposed pipeline method, but just some modifications based on the old one, such as:

1) Enhance the clock rate for input memory module according to the pipeline depth.
2) Allocate a dependant memory for each neuron to store the weights.
3) Add a register to store the output of each neuron.

Unfortunately, due to the limited resource on FPGA, it is impossible to perform a whole neural network as pipeline manner in a single FPGA chip whose price is reasonable for commercial or industrial application. To solve this problem, we introduce the layer multiplexing into our pipeline method.

3.2 Layer Multiplexing with Partly Pipeline

Layer multiplexing was first proposed by S.Himavathi in 2007 [6], which aimed at reducing the resource requirement by multilayer neural network. Instead of realizing a complete network, only the single largest layer with each neuron having maximum number of input is implemented. The same layer behaves as different layers with the help of a control block. For example, consider a 3-5-4-2 network. The largest layer has five neurons and the maximum number of input is five, so in actually execution, there are only five neuron modules with each neuron having five inputs. A simple process of layer multiplexing for the given network is shown in figure 6. For the execution of the first layer, the whole five neurons would be enabled with disabling the other input synapses because the neuron in the first layer only need three input synapses. The working of the second and the third layer is almost the same as the first layer, identified as Step2 and Step3 in figure 6, respectively. The control block ensures proper functioning by assigning the enable signal, appropriate inputs and weights for each neuron.

This method presents an advantage that it can largely save the resource so that a larger network could be performed in a single FPGA chip so far as the largest single layer can be realized on this chip. But this success is achieved at the expense of losing the global forwarding speed, because there is only one layer being working and it has to reconfigure the network before performing neural computing in the next layer. When the network has more layers this problem becomes worse which may cause the timing violation in some speed-sensitive application.

From the synthesis report of [6], when implementing an 8-5-5-3 network by layer multiplexing, the utilization rate of slice is only 62%, that is, we still have enough slice resource to do some improvement to make the global forwarding speed higher. The main idea in this paper is adding the partly pipeline manner to the layer multiplexing method.

In our proposed method, we first calculate the maximum number of neuron modules to fit an adopted FPGA chip. And then taking this value and the neural network topology into consideration, we can get an optimal solution by assigning the appropriate mapping method, pipeline depth and layer multiplexing. Figure 7 gives an example of our method.

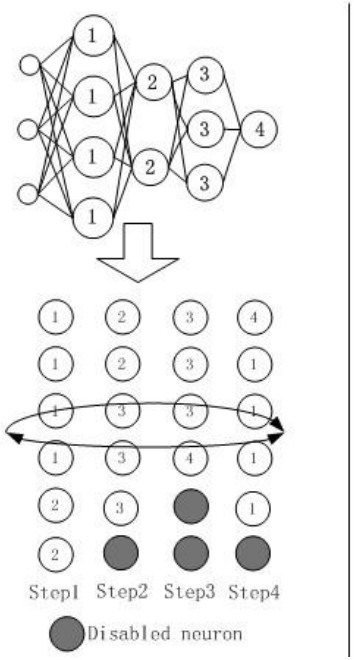

Fig.7. The schematic of our mapping method

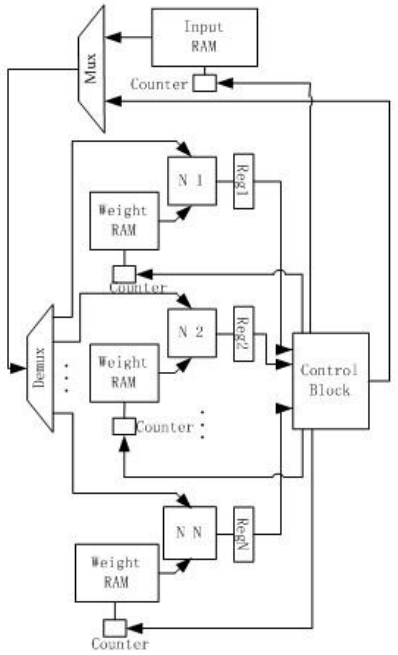

Fig.8. The network circuit

In this example, we suppose the network topology intends to be realized is 3-4-2-3-1, the maximum number of neuron modules the FPGA supported by is six. Numbers in the nodes in figure 7 mean the layer numbers. It performs the neural computing of two layers at a time with the pipeline manner between these two layers. In Step1, the first two layers are under

working, where the layer 1 is computing for the input pattern a^{m+1} and the layer 2 is serving for input pattern a^m, respectively. And in Step2, the 2nd and 3rd layers are configured by the layer multiplexing mechanism, where the 2nd layer is busy with input pattern a^{m+1} from the layer 1 in the previous step while the 3rd layer is computing for input pattern a^m from layer 2 by layer multiplexing. And the total number of neurons in the 2nd and 3rd layers is five, so there are disabled neuron modules to save the power. The next steps are almost the same as the first two steps, including enable the proper number of neuron modules, get the corresponding input from previous step, perform neural computing for the incoming data and then send the result to the next step by layer multiplexing. And if the computation in Step4 is completed, it would turn to the Step1, forming a loop. In this example, there are always two layers mapping the neuron modules in pipeline, which means the pipeline depth is two. By assigning different FPGA chip and neural network topology, the pipeline depth may be different.

3.3 The Network Circuit Design

The operation of layer multiplexing and the partly pipeline is guaranteed by a control block, which is realized by the finite state machine (FSM). Figure 8 shows the whole circuit of neural network using our proposed method. The details of the block named NEURON is already presented in section II, so we will introduce the control block design here.

```
FSM (reset, clk, flag1~S, en1~N,con0~N, mux, demux)
   // input signal:  reset, clk, flag1~S
   // flag1~S:  the finish signal of computation in Step s
   // en1~N: the enable signal for each neuron module
   // con0~N: the address signals for each RAM
   // mux: the selection signal for the multiplexer
  // demux: the selection signal for the demultiplexer
always (current_state or flag1~S)
  case (current_state)
        IDLE:
             Do {...}
        Step1:
             Do { set en1~N, con0~N,mux,demux
                  next_state=Step2}
                  ……
        StepS:
             Do { set en1~N, con0~N,mux,demux
                  next_state=Step1}
       default: next_step=IDLE
  endcase
```

Fig.9. Pseudo code for the control block

```
Solve(topology, chiptype, datapath)
 // the topology information includes the number of
 // layers, the number of inputs, the number of neurons
// in each neuron
// the data path contains the data representation of input,
// weight that mentioned in section II.B.
   1. begin
   2. initial pipe_depth=layer_num
   3. size=est_neuron_size( datapath)
   4. max_num=cal_max_num(chiptype,size )
   5. searching(pipe_depth, max_num, topology)
   6. { stop_condition()
   7. if (stop_condition())
   8.   searching(pipe_depth-1,max_num, topology)
   9. else
  10.   return pipe_depth
  11.  endif}
  12. config_FSM(pipe_depth, topology, data_path)
  13. end
```

Fig.10. Procedure of topology generation

The main task of the control block is assigning proper signals for the input and weight RAMs, the multiplexer and demultiplexer, and performing the logical data transmission. The pseudo code of the FSM is given in figure 9.We also developed a MATLAB program to determine the optimal architecture for a given neural network and a selected FPGA chip. Firstly, it estimates the number of slices would be utilized by a neuron module when the data structure of the neuron is given. Secondly, the maximum amount of neuron modules allowed in the selected FPGA chip can be figured out due to we have integrated the information of several popular FPGA chip into the procedure. Then a searching function will be invoked to determine the optimal pipeline depth for the neural network. This procedure also provides the function that configures the parameters in the FSM. The pseudo code of the procedure is shown in figure 10.

4. Experiments

FPGA design flow is as follows: 1) design entry, 2) synthesis, 3) simulation and 4) devices programming. We choose the Verilog HDL as the coding language for design entry and Xilinx ISE 9.1i to do the synthesis and devices programming while the simulation is performed by Modelsim XE II 7.3a. The FPGA chip we selected here is Virtex XCV400hq240, which is widely used in commercial or industrial application as their lower price. The basic block of the Virtex is the Logic Cell (LC)[8], which is composed of a four inputs LUT, carry logic and a Flip-Flop. And two LCs form the slice, which is used to measure the resource cost.

Table I gives the synthesis report for a single neuron module by varying the data path in weights, because in most cases the input is always fixed at 9 bits with different weight bit lengths. From the table we can see that with respect to the increase of data precision, the maximum frequency comes down but at an acceptable level.

Table II is the synthesis report for the example neural network mentioned in Section III B. The value of pipe depth means the number of layers under execution at a time. From the table we can obtain that our proposed architecture is compact due to the simple module architecture and the effective control block, and also provides a flexible solution for a neural network to be implemented because the pipeline depth is adaptive to the selected FPGA chip and the network topology.

Table III shows a comparison between the traditional layer multiplexing method (LM) and our proposed method. We have integrated the partly pipeline manner into the layer multiplexing, there are several layers mapping the neuron modules by pipeline manner while LM only maps one layer at a time. As a result, the global forward speed of our method is much higher than LM with respect to the pipeline depth. The CPS (connection per second) is used to evaluate the throughput of neural network. Due to the pipeline manner, our method has much more neurons under working than the conventional LM, so it also shows an

advantage in CPS when compared with LM.

TABLE I
RESOURCE AND PERFORMANCE OF A NEURON WITH DIFFERENT WEIGHT PRECISIONS

No. of bits	10	11	12	13	14	15	16
Slices	73	78	82	86	90	93	98
Max *clk* (MHz)	105.3	104.2	103.6	103.0	102.5	102.1	101.7

TABLE II
SYNTHESIS REPORT FOR AN NN OF 3-4-2-3-1

Pipe Depth	Device Selected	XCV400hq 240		
	FEATURES	Present	Utilized	%
2	Slices	4800	691	14.4
	Flip Flops	9600	537	5.6
	LUTs with 4 input	9600	1282	13.3
3	Slices	4800	1021	21.3
	Flip Flops	9600	805	8.4
	LUTs with 4 input	9600	2102	21.9

TABLE III
COMPARISON BETWEEN LM AND OUR METHOD WITH VRIOUS NN TOPOLOGY

Network topology	Implement method	Slices	Utilization rate	Global forward speed	CPS
3-4-2-3-1	LM	478	9.96	0.197	406.8
	ours	1021	21.3	0.099	915.3
8-5-5-3	LM	532	11.1	0.162	508.5
	ours	1047	21.8	0.078	1017
4-8-3	LM	895	18.6	0.113	813.6
	ours	1231	25.6	0.532	1118.7

5. Conclusion

A general architecture for the implementation of multilayer neural network was proposed in this paper. The circuit for each new application can easily be generated by setting the parameter values to match the particular network size and running the synthesis. Similarly to a particular network, the best solution of the architecture design of pipeline and layer multiplexing is calculated by a MATLAB procedure, by returning the optimal pipeline depth and the control signal value in each state of the control block (FSM). We exploited the capability of a given FPGA chip by assigning the proper pipeline depth, so that a higher resource utilization rate and global forward speed were achieved.

It is easy to design a FPGA implementation for a given neural network at a short time by varying the data path. It also provides the feasibility to perform a larger neural network in a lower-performance FPGA chip at a relatively higher speed by using the partly pipeline method. So it is possible to develop a neural device for commercial or industrial application by our method.

Acknowledgement

This research was supported by CREST of JST, and partially supported by Intellectual Cluster Project and Waseda University Grant of Special Research Projects.

References

[1] Special Section: “Fusion of neural nets, fuzzy systems and genetic algorithms in industrial applications”, *IEEE Trans.on Ind.Electron., vol.46, no.6, pp.1049-1136,1999*.

[2] Jihan Zhu, Peter Sutton, “FPGA implementation of neural networks – a survey of a decade of progress”, Proceeding of 13th International Conference on Field Programmable Logic and Applications (FPL 2003), Lisbon, Sep 2003.

[3] Daniel Ferrer, Ramiro Gonzalez, “NeuroFPGA – Implementation artificial neural networks on programmable logic devices”, Proceeding of the Design, Automation and Test in Europe Conference (DATE’04).

[4] S. Oniga, “A new method for FPGA implementation of artificial neural network used in smart devices”, International Computer Science Conference microCAD2005, Miskolc, Hungary, March 2005, pp.31-36.

[5] J.G. Elredge and B.L. Hutchings, “RRANN: a hardware implementation of the backpropagation algorithm using reconfigurable FPGAs”, *Pro IEEE Int. Conf. on Neural Networks, June 1994*.

[6] S.Himavathi, “Feedforward neural network implementation in FPGA using layer multiplexing for effective resource utilization”, *IEEE Trans.on Neural Networks, vol. 18, no. 3, May 2007*.

[7] Venakata Saichand, “FPGA realization of activation function for artificial neural networks”, International Conference on Intelligent System Design and Applications, 2008.

[8] Xilinx, http://www.xilinx.com/