FPGA Implementation of a Cellular Univariate Estimation of Distribution Algorithm and Block-Based Neural Network as an Evolvable Hardware

Yutana Jewajinda and Prabhas Chongstitvatana

Abstract— This paper presents a hardware implementation of evolvable block-based neural network (BBNN) amd a kind of EDAs called cellular compact genetic algorithm (CCGA) in FPGA. The CCGA and BBNN have cellular-like and array-like structures which are suitable for hardware implementation. The implemented hardware demonstrates the completely intrinsic online evolution in hardware without software running on microprocessors. This work contributes to the field of evolvable hardware by proposing CCGA and a layer-based architecture to an integration of BBNN and CCGA as a kind of evolvable hardware. In addition, the proposed CCGA efficiently solves the scalable issues by scaling up to the size of BBNN. The presented approach demonstrates a new kind of evolvable hardware.

I. Introduction

Evolvable hardware (EH) is the integration of evolutionary computation and programmable hardware devices. The objective of evolvable hardware is to create "autonomous" reconfiguration of hardware structures in order to improve performance [1-3]. Recent research trend in EH directs towards functional approaches to the design of extrinsic and intrinsic EH [4,5]. For extrinsic EH, the evolutionary process is performed off-line. Then the results are downloaded into the hardware. On the contrary, for the intrinsic EH, the evolutionary process is performed wholly or partly in hardware [6-8].

The key concept of our focused evolvable hardware is to regard the configuration bits of programmable hardware architecture as the chromosomes of genetic algorithm (GA) [1,3,9]. By optimizing a fitness function to achieve a desired hardware function, the GA becomes a key to autonomous hardware configuration. There are a number of methods and techniques that propose to apply the genetic algorithm (GA) and evolutionary algorithms (EA) to be implemented in hardware for evolvable hardware (EH), especially implementation into FPGAs [10-13]. However, in order to accomplish the intrinsically on-line evolving in hardware and to utilize hardware resource efficiently pose a challenging research issue of how to modify or invent efficient and improved GA or EA algorithms that can be effectively implemented into hardware [13,14].

Yutana Jewajinda is with the National Electronics and Computer Technology Center, National Science and Technology Development Agency, Bangkok, Thailand (e-mail: yutana.jewajinda@ nectec.or.th).

Prabhas Chongstitvatana is with the Department of Computer Engineering, Chulalongkorn University, Bangkok, Thailand (e-mail: prabhas@chula.ac.th).

Estimation of distribution algorithm (EDA) is a family of evolutionary algorithms. EDA uses probability distribution to guide the search, thus a probability model is learned from the best individuals in each generation, and then subsequently sampled to generate the new population. Cellular Compact Genetic Algorithm (CCGA) is an EDAs that operates on probability vectors [15,16]. CCGA is developed from cooperative compact genetic algorithms [14]. In addition, Cellular CGA is also derived from the parallel EDAs [17-19]. Due to Cellular CGA employs two dimensional array structure like cellular automata, it is suitable for FPGA and hardware implementation [25]. The key feature of Cellular CGA is the capability to scale up to the problem size which is normally the limitation of genetic algorithms especially for hardware implementation.

Evolutionary artificial neural networks (EANN) use evolution to adapt the network structure and internal configuration simultaneously [20]. Since the goal of evolvable hardware is to change hardware architecture and functions without human intervention, EANN can be regarded as a class of evolvable hardware. Block-based neural network (BBNN) model provide simultaneous optimization of network structure and connection weights. The BBNN consists of a two-dimensional array with support integer weights and fixed-point arithmetic. The BBNN structure is suitable to be implemented in hardware especially using field programmable logic arrays (FPGAs). In addition, BBNN was successfully evolved using genetic algorithms to optimize weight and structure [21]. However, in the past research, the genetic algorithm for BBNN was done in software off-line on computer system or on-chip embedded processor [21]. This approach demands higher cost and larger FPGA since it needs on-chip processor and cannot deliver high performance online training in hardware.

This paper presents an FPGA implementation of a cellular compact genetic algorithm and block-based neural network. We propose cellular compact GA (CCGA) and the layer-based architecture, a new integration in hardware between cellular compact genetic algorithm and blocked-base neural network. The layer-based architecture for evolvable hardware consists of two layers. The top layer is for cellular genetic algorithms for evolving weight and structure. The bottom layer is the block-based neural network. With the cellular-like models of both block-base neural network and cellular compact genetic algorithm, this layer-based approach to hardware implementation provides modular

978-1-4244-1823-7/08/$25.00©2008 IEEE

block design in hardware and reduces interconnection length since the cellular cells only communicate to their neighbors. In addition, the cells of both block-based neural network and cellular compact genetic algorithm only interact with their neighbors.

The rest of this paper is organized as follows. Section II describes the cellular compact genetic algorithm. In Section III, the blocked-base neural network is described. Section IV presents the layer-based architecture. The FPGA implementation of cellular compact genetic algorithm and BBNN layers are presented in Section V. Section VI describes XOR problem as a case study. The paper concludes with a summary in Section VII.

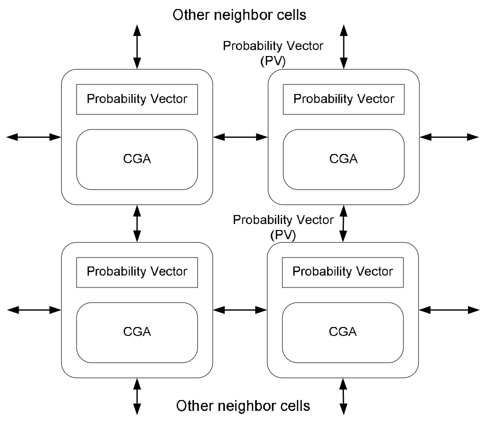

Fig. 1. Topology of compact GA

II. Cellular Compact Genetic Algorithm

The concept of cellular compact GA is to partition the search space into smaller sub-space. These smaller search spaces are then searched by separate GAs whose fitness is evaluated by combining solutions found by each of the GAs. The basic idea behind cellular compact genetic algorithm is to parallelize or divide a large problem into smaller tasks and to solve the task simultaneously using multiple genetic algorithms [27]. With this approach, the CCGA can scale up to the problem size and address scalability issue. The CCGA consists of uniform cellular compact genetic algorithm cells connected in a cellular automata space in which each CGA cell only exchange probability vectors to its neighbors.

A. CCGA topology

Fig. 1 illustrates the topology of the cellular compact GA. The topology of the proposed CCGA resembles the cellular automata (CA) system that cells only interact with their neighbors. However, the interactions between CA cells occur by exchanging the probability vectors instead of mating between individuals of sub-population directly. With this proposed CA topology, the hardware realization of the algorithm is not complicated to be implemented in term of scalability and signal wiring that greatly contribute to the increasing performance of the implemented digital circuit. In addition, CA architecture has capability of self-evolving and self-replicating [22]. Moreover, CA-like architecture can be practically and efficiently implemented into FPGAs or other reconfigurable devices because of the architecture of array of logic block [14].

Each coarse grained CCGA cell has a probability vector which represents a sub-population. Every CCGA cell is identical. In Fig. 1, Each CCGA cell with four neighbors exchanges probability vectors and key information between its neighbors. Every CCGA cell keeps adjusting its own probability vector toward the best probability. The confidence counter (CC) is introduced to help each cell evaluates recombination method of the probability vector from its neighbors. The key parameters for CCGA topology is the number of the neighbors of each cell.

```
1.  Initialize probability vector
       for i := 1 to L do p[i] := 0.5;

2.  Generate two individuals from the vector
       a := generate(p);
       b := generate(p);

3   Let them compete
      Winner, loser := evaluate(a, b);

4.  Update the probability vector toward the better one
     for i := 1 to L do
      if winner[i]  != loser[i] then
         if winner[i] = 1 then p[i] += 1/N
         else p[i] -= 1/N

5.  Check if the probability vector has converged
     for i := 1 to L do
      if p[i] > 0 and p[i] < 1 then goto step 2

6.  P represents the final solution
```

Fig. 2. Pseudocode of compact GA

B. CCGA Algorithm

The fundamental of the CCGA is the compact GA [23]. The compact GA represents the population as a probability distribution over the set of solutions. Each bit of the probability vector keep being adjusted according to the result of the tournament selection. The pseudocode of the compact GA is shown in Fig. 2.

Fig. 3 shows the pseudocode of the cellular compact GA. After probability vector of each cell in the cellular automata space is initialized to the mid-point range, two individuals are generated from the probability vector, then compete similar to a normal compact GA. The proposed algorithm is different from the normal compact GA and the cooperative compact GA [14] in four ways:

(1) The cellular-like topology employs uniform cell type. This allows flexibility and ease of implementation.

(2) The confidence toward the better probability vector is calculated as confident counters and passed directly to neighbor cells. In Fig. 3, the step 3, 4 and 5 are modified to the normal compact GA.

(3) Improved probability vector combination by local search and adaptive combination in step 6 shown in Fig. 3.

This combination scheme provides a solution to the greedy search characteristic of the cooperative compact genetic algorithm [14,18]. The local search is implemented through the use the confident counter that keeps the frequency of the updating to the probability vector of each cellular compact GA cell. The higher *confident counter* values contribute to higher chance to reach the better solution The probability vector combination use the following equation:

$$\mathcal{P}_i^{t+1}(x) = \beta \mathcal{P}_i^{t}(x) + (1 - \beta)\mathcal{P}_r^{t}(x)$$

Where β is an adaptive weight calculated from the best *confident counter* among neighbors

$\mathcal{P}_i^{t+1}(x)$ is a new inner probability vector of a CCGA cell

$\mathcal{P}_r^{t}(x)$ is the best incoming probability vector from neighbors

(4) Adaptive migration rate of the probability vector for each CCGA cell using *confident counter.* Since the updating rate of each CCGA cell depends on its *confident counter* which is different for each cell. This contributes to the different rate to exchange the probability vector.

C. Hardware Design

A CCGA cell is designed by adding additional modules to the hardware of the normal compact GA hardware. Fig. 4 shows the hardware design of the N-bit module of a CCGA cell. The design of CCGA bit-module is based on the design proposed in [14] integrated with the communication unit (COMM), the confident counter unit (CC) and the probability vector combination unit (VCOMBIN). In Fig. 4, the hardware design consists of four main blocks. The first block is the CCGA bit-module which can be cascaded to form N-bit chromosome. The second block is the additional units to compact GA hardware which consists of the confident counter (CC) and the communication unit COMM. The third block is the probability vector combination unit VCOMBIN. The fourth block is a simple finite state machine acts as the main controller for the whole block. The detail of these three additional modules is described as follows

COMM is a finite state machine that controls sending and receiving the probability vector as an 8-bit package between cells. For a chromosome of N-bit length, the compact GA needs to have N-bit of probability vector which each bit of the probability vector is represented as 8-bit. Thus, for N-bit length chromosome, N packages of 8-bit will be sent and received between cells by the COMM units of each cell.

CC is the *confident counter* designed as a 5-bit counter. During fitness evaluation, the counter is incremented every time when the fitness of the winner is better than the current best fitness. The value of the counter is passed to the neighbor cells with the current probability vector

VCOMBIN is the hardware block that implements the step 6 of the pseudocode in Fig. 3. A hardware part of the block consists of comparators and multiplexers for comparing incoming *confident counter.* The multiplication of β with the probability is implemented using shift register instead of using multipliers which occupy more hardware resource. With shift register implementation, the β values will be scaled down to multiple of 2. After multiplication, the values will be added using 8-bit adder. FSM_CONTROL is a finite state machine that controls.

```
L is chromosome length
N is population size
cc is Confident Counter
CA is Cellular Automata space

for each cell I in CA do in parallel
    Initialize each p[I]
        For  i  := 1  to L do
            [i] = 0.5;
    Initialize cc
            cc := 0;
end parallel for
for each cell  i  in  CA do in parallel
  while not done do
   1.  Generate two individual from the vector
              a := generate ();
              b := generate ();
   2.  Let them compete
           Winner, loser := compete (a, b);
   3.  Update the probability vector toward
       better one and Increment Confidence Counter
       3.1. Update probability vector
              for i := 1 to L do
              if winner[i]  != loser[i] then
                if winner[i] = 1
                then p[i] += 1/N
                else p[i] -= 1/N

        3.2 Increment Confidence Counter
              cc: = cc + 1;
   4.  Check if cc reaches a target level then
       Send p and cc to the neighbor cell
   5.  Receives  p and cc from neighbors
   6.  Use the adaptive convex recombination
       with the received p from the neighbor
       and its own p
       6.1  Select the highest cc from neighbors
              $cc_{max}$ := 0;
              for i := 1 to M do
                 if (cc[i] > $cc_{max}$ )
              $cc_{max}$ := cc[i];
              $p_{ccmax}$ := p[i]

       6.2  Convert $cc_{max}$ to $\beta : 0 \le \beta \le 1$
              $\beta$ := MAP( 1/ $cc_{max}$)

       6.3. Update $p_1$ with $\beta$
              for i := 1 to L do
                $p_1$[i] := $\beta$ $p_1$[i] + $(1 - \beta)$ $p_{ccmax}$[i]

   7.  Check if the vector has converged
              for i := 1 to L do
                 if p[i] > 0 and p[i] < 1 then
                    goto step 1
   8.  p represents the final solution
  end while
end parallel for
```

Fig. 3. Pseudocode of cellular compact GA

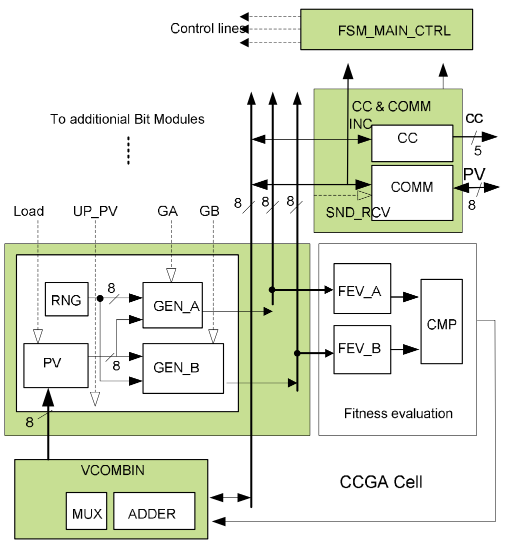

Fig. 4. Hardware design of a cellular compact GA cell

III. Block-Based Neural Network

A. Architecture of the Block-Based Neural Network

The Block-based neural network was proposed by S. W Moon et al in [20]. The BBNN model consists of a 2-D array of basic blocks. Each block is a basic processing element that corresponds to a feedforward neural network with four variable input/output nodes. Fig. 5 represents the structure of the BBNN model of *m* x *n* size with *m* row or stage and *n* column labeled as B_{ij} The label *i* denotes the stage. The last stage is denoted *m*. Any block in the BBNN is connected to its neighbors. The first row of blocks B_{11}, B_{12} to B_{1n} is the input layer and the blocks B_{m1}, B_{m2}, ... B_{mn} form the output layer. An input signal x = (x_1, x_2,…, x_n) propagates through the blocks to produce network output y = (y_1, y_2, …, y_3).

BBNN can implement both feedforward and feedback network configuration. A feedback configuration of BBNN architecture may cause a longer signal propagation delay. BBNN of size *m* x *n* can represent the input-output characteristics of any Multilayer perception (MLP) network for $n \leq 5$.

A block of BBNN consists of four nodes. These four nodes can be configured to represent four different types of internal configurations. Fig. 6 shows four types of internal configurations of one input and three outputs(1/3), three inputs and one output (3/1), and two inputs and two outputs (2/2). The four nodes inside a BBNN block are connected with each other through weights. A weight w_{ij} denotes a connection from node *i* to node *j*. A BBNN block can have up to seven connection weights and three the biases. For the case of two inputs and two outputs (2/2), there are four weights and two biases. The 1/3 case has three weights and three biases. There are three weights and one bias for 3/1 case. Generalization capability of BBNN network emerges through various internal configuration of a BBNN block.

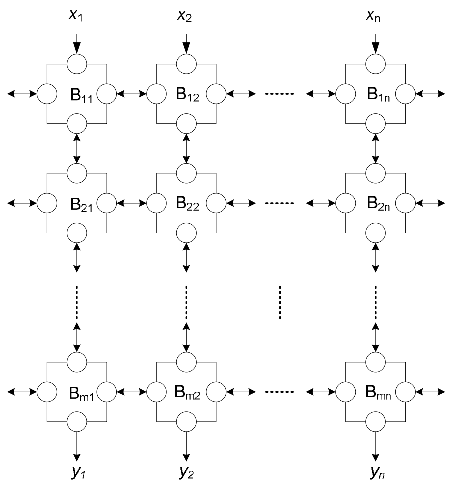

Fig.5. Structure of BBNNs

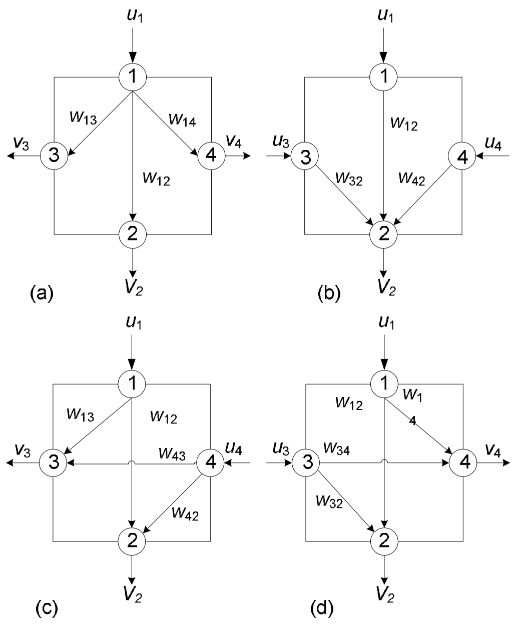

Fig.6. Four different internal configurations of a BBNN block (a) 1/3, (b) 3/1, (c) 2/2, (d) 2/2

If signal u_i is the input and v_j is the output of the block. The output v_j of a block is computed with an activation function h as follows.

$$v_j = h\left(\sum_{i \in I} w_{ij} u_i + b_j\right), \quad j \in J$$

I and *J* are sets for input and output node, respectively. The term b_j is the bias of the *j*th node. The activation function can be linear or nonlinear function. For hardware implementation direct implementation of nonlinear activation function requires more hardware resources. A more practical approach would be a piecewise-linear approximation of the non-linear activation function or using lookup tables.

B. BBNN Hardware Implementation

In [21], Merchant et al. propose the design of Smart Block-based Neuron (SBbN) that can be configured on-the-fly to emulate four types of the internal configuration modes of a BBNN neuron. However, the detail design of the SBbN is not presented. SBbN supports *gray mode* which the block is inactive and only pass the inputs to outputs.

In this paper, we propose the link-multiplexed block-based Neuron (LMBbN). This idea is similar to layer-multiplexed in [24]; however, in [24] the goal is to multiplex each layers of feed-forward network. The LMBbN is derived from the analysis of a basic block of block-based neural network which is shown in Fig. 7. In Fig. 7, there two types of routing switch: L and S types. These switches and four nodes form seven paths or "link" between two nodes. The architecture of the link-multiplexed BBNN is shown in Fig. 8. With this LMBbN, we can reduce number of multipliers by sharing it with other links since there are a limited number of hard-macro multipliers inside DSP blocks of FPGAs.

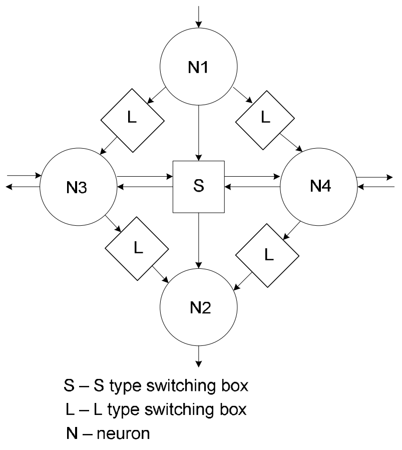

Fig.7. a BBNN block

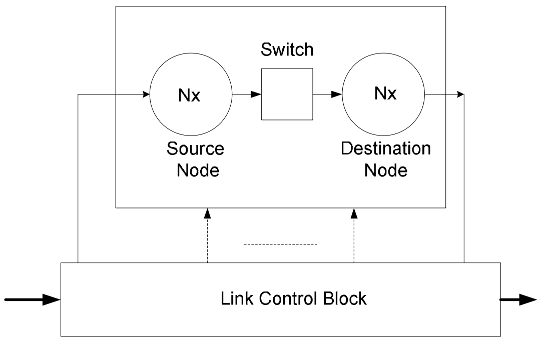

Fig.8.. Link-multiplexed BBNN block

Hardware design of the general neuron is shown in Fig. 9. The main controller receives configuration type. With configuration type, the general neuron can be configured to support case a, b, c, and d.

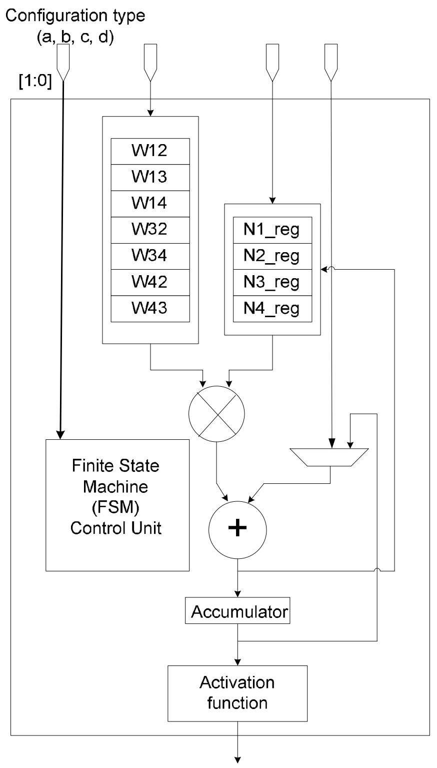

Fig.9. Block level hardware of LMBbN neuron with three inputs

When design the LMBbN block in Xilinx FPGA Virtex-5, we utilized the DSP hard macro inside the FPGA to implement the multiply-accumulate (MAC) function tospeed up the MAC operations. For the other blocks, we designed the register banks, activation using piece-wise linear approximation, and the FSM control block. There are two register banks: weight registers and bias registers. The FSM block control loading of weight and bias registers and DSP MAC according to 2-bit configuration type.

From Fig. 9, if we use fractional weight format size 10-bit for weights. The total number of bits to configure one LMBbN block with the proposed hardware implementation is 102 bits.

- Seven of 10-bit for weights
- Three of 10-bit for biases
- 2-bit for config-type

IV. Layer-based Architecture

In this section, we propose the Layer-based architecture for evolvable hardware based on the block-based neural network and the cellular compact GA. From Section III, to evolve the block-based neural network with one block using 10-bit fractional number, we need 102-bits genetic algorithms. However, to apply a block-based neural network to solve real-world problems, the more number of BBNN block will be required, for example if a BBNN network sized 3 x 4, it requires 1224-bits GA to evolve. The larger size of BBNN, the wider bits requires for GA. This turns to be the scalability problem for genetic algorithm. To solve this problem, we propose to use cellular genetic algorithm implemented in hardware since the CCGA can scale up with problem size since it has array-like architecture as the BBNN model. Fig. 10 shows the concept of the layer-based architecture.

From Fig. 10, the BBNN occupies different layer from the cellular compact GA. These two layers interact to each other between nodes of BBNN and CCGA. Fig. 11 shows the layer-based architecture with the eight nodes for each BBNN and CCGA layers. If l is the problem size that is the number of node of the BBNN, with n_p CCGA nodes, then each CCGA node contains l/n_p probability vectors.

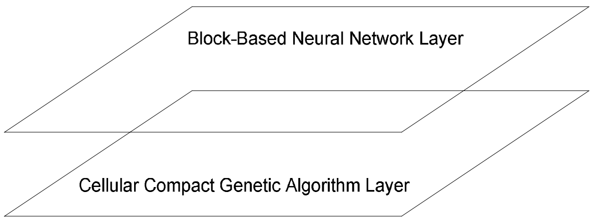
Block-Based Neural Network Layer
Cellular Compact Genetic Algorithm Layer

Fig.10. Layer-based architecture for evolvable hardware based on BBNN and CCGA

V. FPGA Implementation Results

We implemented the both BBNN and CCGA nodes in Xilinx ML501 Boards which has Virtex-5 LX50. The code was design and coded in synthesizable Verilog HDL. ModelSim Version 6.2 was used for simulation. Xilinx ISE 9.1 was used for FPGA implementation. For our initial tests of the implementation, "one max" problem with 32-bit was used to verify the operation of the CCGA. The simulation result is shown in Fig. 12. The hardware was also tested with F1 and F2 functions as follows:

$$F1(x) = \sum_{i=1}^{3} x_i \quad when \; -5.12 < x < 5.12$$

$$F2(x) = 100\,(x_1^2 - (x_1^2 - x_2)^2 + (1 - x_1)^2$$

when $-2.048 < x_i < 2.048$.

In Fig. 13 and Fig. 14, the simulation shows the comparison results between normal compact GA and cellular compact GA. The CCGA contains two CCGA nodes, each has 32-bit probability vectors while CGA has only one node with one 32-bit probability vector. The performance of CCGA outperforms the normal CGA term of speed and quality of the search results.

Table I shows the FPGA hardware resources required for BBNN and CCGA. Each BBNN block was implemented using 10-bit fractional number. We designed one 25x18 multiplier using DSP block and a Finite State Machine (FSM) to control how to multiplex computation between the four sub-nodes of a basic BBNN block. There are ten data inputs to a BBNN blocks. These are seven weights (w13, w12, w14, w43, w42, w34, and w32) and three biases (b2, b3, and b4). We implemented each CCGA that has 102-bit which each bit represented by an 8-bit probability vector. Due to one CCGA node supports 102-bit, the size of one CCGA node is about four times the size of a BBNN node as shown in Table I since the largest block of BBNN is the MAC which is already a hard macro in Xilinx Virtex-5 FPGA.

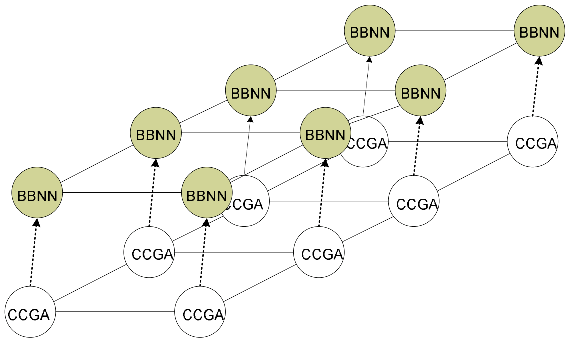

Fig.11. 2 x 4 BBNN layer and CCGA

From Table I, the speed for each BBNN and CCGA node is about 300Mhz regardless of the size of matrix like 1x1, 2x2, or even 3x3. The reason is that each node of BBNN and CCGA is quite independent in term of hardware implementation especially CCGA since each 8-bit probability vector of one bit of CCGA was parallelized in hardware implementation. In our implementation, each BBNN only requires one DSP hard macro in Xilinx FPGA. This saving can allow implementing more nodes of BBNN in one FPGA.

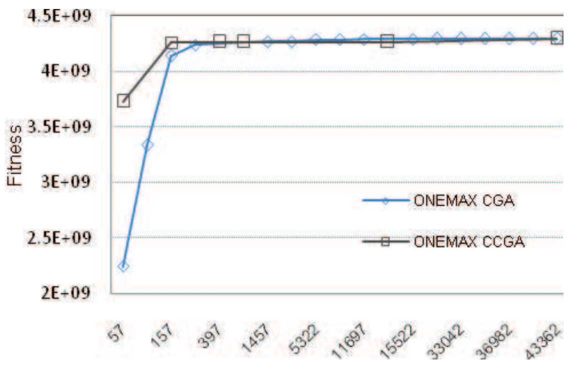

Fig.12. 32-bit "OneMax" simulation results

VI. A Case-Study

To demonstrate the capability of integration between BBNN and CCGA, we implemented a 2 x1 BBNN network which has three BBNN nodes and evolving the weights with three nodes of CCGA to solve the XOR problem with two inputs, x1 and x2, and one output, y1. The "off" is when x or y has value < 0.0625 and > 0.9375 when "on". Each CCGA has 102-bit which supports ten outputs; each has 10-bit, for seven weights and three biases for one BBNN node and two bits for four configuration types. In Fig 15 shows the block diagram of BBNN and CCGA to solve the XOR problem. The error calculation block computes using following equations:

$$Fitness = \frac{1}{1+e} \quad (1)$$

$$e = \frac{1}{N\, n_o}\sum_{j=1}^{N}\sum_{k=1}^{n_o} e_{jk}^2 \quad (2)$$

$$e_{jk} = d_{jk} - y_{jk}(x) \quad (3)$$

Where, N and n_o are number of training data and output. d_{jk} and y_{jk} are desired and actual output respectively.

Fig. 17 shows the hardware simulation results of training of the XOR problem. For one training pattern, the BBNN takes 60 clock cycles while CCGA takes only 3 clock cycles. At 549 epoch, the training was achieved with fitness 0.998 using 18-bit precision fixed point arithmetic. The number of bit in neural network has an impact on the performance of the hardware [26]. Fig. 16 shows a configuration of three BBNN Blocks that give the best fitness.

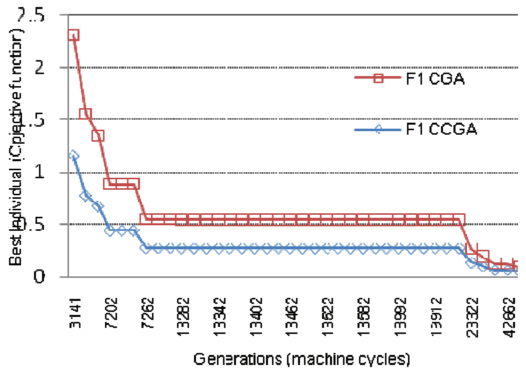

Fig.13. F1 simulation results

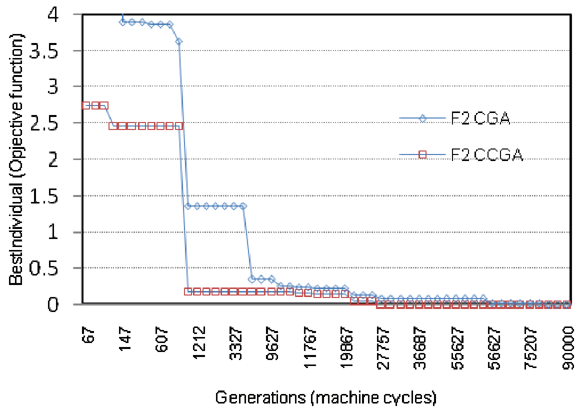

Fig.14. F2 simulation results

TABLE I
FPGA HARDWARE RESOURCE XILINX VIRTEX-5 LX50

Network size	FPGA resources for BBNN and CCGA on Xilinx Vertex-5 LX50		
		BBNN	CCGA
1x1	Slice Registers used Flip-Flops	341	621
	Slice LUTs used as Logic	263	1,932
	DSP48Es	1	0
	Total equivalent gate count	4,562	18,224
	Maximum Frequency	290Mhz	290Mh
2x2	Slice Registers used Flip-Flops	1326	1,642
	Slice LUTs used as Logic	974	5,506
	DSP48Es	3	0
	Total equivalent gate count	17,317	49,204
	Maximum Frequency	280Mhz	280Mh
3x3	Slice Registers used Flip-Flops	3,262	5,130
	Slice LUTs used as Logic	2,300	16,549
	DSP48Es	9	0
	Total equivalent gate count	36,952	147,614
	Maximum Frequency	270Mhz	270Mhz

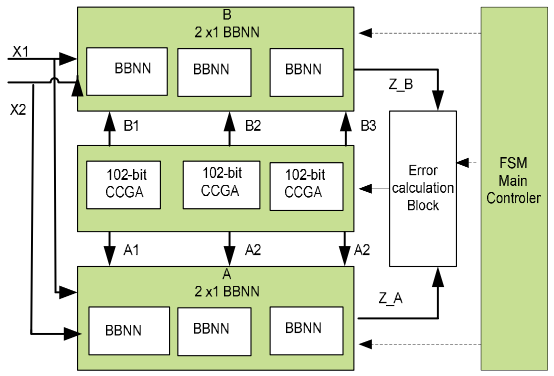

Fig.15. Block diagram of BBNN and CCGA for XOR

VII. CONCLUSION

In this paper, an approach to training BBNN in hardware using the cellular compact genetic algorithm which is a kind of EDAs is presented. We propose the cellular compact GA and the layer-based architecture for integration between the block-based neural network and cellular genetic algorithm in hardware. With the layer-based architecture, evolvable hardware based-on the integration between BBNN and CCGA is feasible and effective since both have array.like architecture. This approach provides a solution for scalability of genetic algorithm since CCGA can scale up to the size of the BBNN by adding more CCGA nodes without sacrifice the speed in term of clock period and cycles. The XOR problem was used as an example of the approach. It has been implemented in hardware and can classify the data successfully. The more difficult classification problems can be solved in real-time with this kind of evolvable hardware. We believe that the more hardware resource in future FPGA will create more applications of the block-based neural network and the cellular compact GA for real world problems.

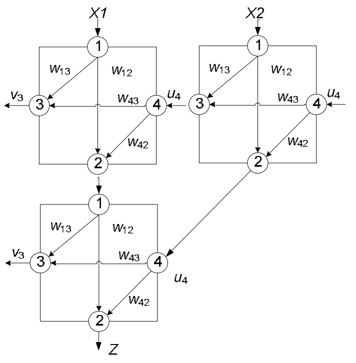

Fig.16. a structure of XOR that has fitness value of 0.998

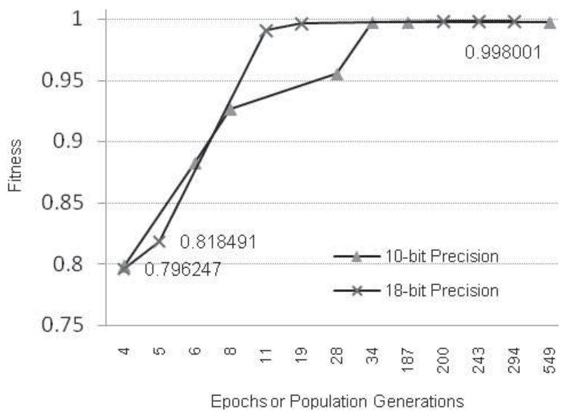

Fig.17. Fitness value in training process of XOR

REFERENCES

[1] T. Higuchi, Y. Liu and X. Yao, "Introduction to evolvable hardware", *Evolvable Hardware*, pp. 1-17, Springer 2006.

[2] J. F. Miller and P. Thomson, "Aspects of digital evolution: evolvability and architecture," *Proc. Parallel Problem Solving From Nature*, Amsterdam Netherland, 1998, pp. 927–936

[3] P. Haddow and G. Tufte, "An evolvable hardware FPGA for adaptive hardware," Proc. IEEE Congress on Evolutionary Compution, San Diego, CA, 2000, pp. 533-560.

[4] S. L. Smith, D.P. Crouch and A. M. Tyrrel, "Evolving image processing operations for an evolvable hardware environment," *Proc. Evolvable systems: from biology to Hardware ICES2003*, 2003, pp. 332–343.

[5] L. Sekanina, "Virtual reconfigurable circuits for real-world applications of evolvable hardware," *Proc. Evolvable systems: from biology to Hardware ICES2003*, 2003, pp. 332–343.

[6] T. Higuchi, M. Iwata, D. Keymeulen, H. Sakanashi, H. Murakawa, I. Iajitani, E. Takahashi, K. Toda, M. Salami, N. Kajihara, and N. Oesu, "Real-world applications of analog and digital evolvable hardware," IEEE Transactions on Evolutionary Computation, vol. 3, pp. 220-335, Sept. 1999.

[7] G. Hollingworth, S. Smith, and A.M. Tyrrell, "Safe intrinsic evolution of virtex devices," Proc. NASA/DoD Conference on Evolvable Hardware, July 2000, pp. 195-202.

[8] H. Liu, J.F. Miller, and A.M. Tyrrell, "Intrinsic Evolvable Hardware Implemenatation of a robust biological development model for digital systems, Proc. NASA/DoD Conference on Evolvable Hardware, July 2005, pp. 87-92.

[9] Y. Zhang, S. L. Smith, and A. M. Tyrrell, "Digital circuit design using intrinsic evolvable hardware," Proc. NASA/DoD Conference on Evolvable Hardware,July 2004, pp. 55-62.

[10] S. Scott and A. Seth, "HGA: A hardware-based genetic algorithm," Proc. ACM/SIGGA 3rd Int. Symp. Field-Programmable Gate Array,1995, pp. 1-12.

[11] T. Kajitai et al, "A gate level EHW chip: implementating ga operations and reconfigurable hardware on a single LSI," Proc. Int. Conf. Evolvable System, 1998, pp. 1-12.

[12] C. Aporntewan and P. Chongstitvatana, "A hardware implementation of the compact genetic algorithm," Proc. IEEE Congress on Evolutionary Computation, Seoul, Korea, 2001, pp. 624-629.

[13] J. C. Gallagher, S. Vigraham, and G. Kramer "A family of compact genetic algorithms for intrinsic Evolvable Hardware," IEEE Transactions on Evolutionary Computation, vol. 8, pp. 111-126, April 2004.

[14] Y. Jewajinda and P. Chongstitvatana, "A cooperative approach to compact genetic algorithm for evolvable hardware," *Proc. IEEE Congress on Evolutionary Computation*, 2006, pp. 624–629.

[15] P. Larranaga and J. A. Lozano, *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*, Kluwer Academic Publishers, 2001.

[16] M. Pelikan, K. Sastry, and E. Cantu-Paz, *Scalable Optimization via Probabilistic Modeling*, Springer, 2006

[17] C.W. Ahn, D.E. Goldberg, and R. Ramakhrishna, "Multiple-deme parallel estimation of distribution algorithms: basic framework and application. In *Proceedings of Parallel Processing and Applied Mathematics, PPAM 2003, LNCS 2774,* pp544-551, Springer, 2004

[18] L. DelaOssa et al., "Improving model combination through local search in parallel univariate EDAs," *Proc. IEEE Congress on Evolutionary Computation*, 2006, vol 2, pp. 624–629.

[19] K. Sastry, D.E. Goldberg, and X. Liora "Towards billion-bit optimization via a parallel estimation of distribution algorithm," Proc. GECCO 2004, 2004, pp. 412-413.

[20] S. W Moon and S. G. Kong, "Block-based neural networks," *IEEE Transaction on Neural Networks,* vol 12, pp. 307-317,2001

[21] S. Merchant et al., "FPGA implementation of evolvable block-based neural network," *Proc. IEEE Congress on Evolutionary Computation*, 2006, vol 2, pp. 3129–3136.

[22] M. Sipper, Evolution of parallel cellular machines: the cellular programming approach, Berlin: Springer-Verlag, 1997.

[23] G. Harik, F. Lobo, and D. Goldberg "The compact genetic algorithm," IEEE Transactions on Evolutionary Computation , vol. 3, pp. 287-309, Nov. 1999.

[24] S. Himavathi et. al, "Feedforward neural network implementation in FPGA using layer multiplexing for effective resource utilization," *IEEE Transaction on Neural Networks,* vol 18, no. 3, pp. 880-888, 2007

[25] V. Betz, J. Rose, and A. Marquardt, Architecture and CAD for Deep-Submicron FPGAs, Boston, Springer, 1999.

[26] A. W. Savich et. al, "The impact of arithmetic representation on implementing MLP-BP on FPGAs: a study," *IEEE Transaction on Neural Networks,* vol 18, no. 1, pp. 240-252, 2007

[27] E. Cantu-Paz, *Efficient and accurate parallel genetic algorithms*, Boston, MA:Kluwer Academic Publisher, 2000.