Evolutionary optimization of neural networks with heterogeneous computation

Study and Implementation

Jorge Fe, Ramón J. Aliaga,
Rafael Gadea-Gironés

Abstract In the optimization of Artificial Neural Networks (ANNs) via Evolutionary Algorithms (EAs) and the implementation of the necessary training for the objective function, there is often a trade-off between efficiency and flexibility. Pure software solutions on general-purpose processors tend to be slow because they do not take advantage of the inherent parallelism, whereas hardware realizations usually rely on optimizations that reduce the range of applicable network topologies, or they attempt to increase processing efficiency by means of low-precision data representation.

This paper presents, first of all, a study that shows the need of heterogeneous platform (CPU-GPU-FPGA) to accelerate the optimization of ANNs using genetic algorithms and, secondly, an implementation of a platform based on embedded systems with hardware accelerators implemented in FPGA (Field Pro-grammable Gate Array). The implementation of the individuals on a remote low-cost Altera FPGA allowed us to obtain a 3x-4x acceleration compared with a 2.83 GHz Intel Xeon Quad-Core and 6x-7x compared with a 2.2GHz AMD Opteron Quad-Core 2354.

Keywords Evolutionary Computation · Embedded System · FPGA · Neural Networks

1 Introduction

Artificial neural networks (ANNs) are widely used in many areas of research, delivering very promising results in some of them.

Institute for Molecular Imaging Technologies
Universitat Politècnica de València,
Camino de Vera s/n, 46022 Valencia, Spain
Tel.: +34-96-3877007
Fax: +34-96-3879609
E-mail: rgadea@eln.upv.es

However, the topology design of ANNs determines their usefulness because it significantly affects a networks performance [1]. When beginning to work with an ANN on a concrete problem, it is observed that too large a network usually tends to overfit the training data, affecting its generalization capability, whereas too small network usually encounters problems in learning the training samples due to its limited representation capability. Furthermore, whenever good results are achieved, there is always doubt as to whether these results are optimal for all network structures.

This uncertainty may be deemed acceptable by researchers if the other conditions when working with neural networks are fixed (for example number of learning samples and input variables), but when all these parameters need to be evaluated as part of the same research, the determination of the optimal topology requires a long period of experimentation. Curteanu and Cartwright [2] listed the main methods for the determination of optimal topology: trial and error, empirical or statistical methods, hybrid methods, constructive and destructive algorithms and evolutionary strategies.

In the structure design, Evolutionary Algorithms (EAs) are employed in two ways: to evolve the structures only [3] and to evolve both the structures and the connection weights simultaneously [4].

Many researchers focused on the simultaneous optimization of network structure and connection weights : Leung et al. [5] presented an improved genetic algorithm to tune the structure and parameters simultaneously. Tsai et al. [6] used a hybrid Taguchi-genetic algorithm to solve the problem of tuning both the network structure and parameters. Ludermir et al. [7] presented an approach combining simulated annealing and tabu search for the simultaneous optimization of multilayer perceptron (MLP) network weights and architectures. Palmes et al.[8] used mutation-based genetic neural network (MGNN) to replace BP by using the mutation strategy of local adaptation of evolutionary programming (EP) to effect weight learning. Li et al. [9] used an improved particle swarm optimization using optimal foraging theory (PSOOFT) to train an ANNs free parameters (weights and bias) and used a binary PSO algorithm to evolve the architecture, respectively. Lu et al. [10] propose a quantum bit representation to codify the network, indicating not the actual links but the probability of the existence of the connections, thus alleviating mapping problems and reducing the risk of discarding a potential candidate.

This use of EAs in this way face a major difficulty: the ability of an ANN to evolve into a superior ANN relies on the survival of the right topology. For this, it is necessary to ensure that individuals with the best topology are the best ranked and that these are retained as the best breeding individuals for the next generation. In the case of the simultaneous evolution of structures and connection weights, a good fitness value does not necessarily give an accurate representation of the quality of the structure. In the event that this fitness function is achieved, then the most efficient way to perform the crossover of these individuals should be determined.

The other aproach is to evolve the structures only, without any connection weights.Connection weights have to be learned after a near-optimal architecture is found. One major problem with the evolution of architectures without connection weights is noisy fitness evaluation [11][12]. Different random initial weights may produce different training results. Hence, the same representation (genotype) of a structure may have quite different levels of fitness. This one-to many mapping from genotype to the actual networks (phenotypes) may induce noisy fitness evaluation and misleading evolution. In order to reduce such noise, an architecture usually has to be trained many times using different random initial weights. The average result is then used to estimate the genotypes mean fitness. This method increases the computation time for fitness evaluation dramatically. It is one of the major drawbacks of this way of using EAs, and it is therefore the main justification for our study, because our implementation permits the distribution of the computation across a heterogeneous network, efficiently decreasing the computation load.

The second objective of our study is not to lose the perspective that the optimization of a neural network is not only to achieve a good approximation performance with the training data, but also with unseen data for the same problem (generalization).

The paper is organized as follows: Section 2 describes the evolutionary optimization method and Section 3 the implemented platform. The experimental results are presented in Section 4 and the conclusions in Section 5.

2 Evolutionary optimization method

The objective of our evolutionary optimization method was to get the best trained neural network in order to provide a solution to a problem with no apparent algorithmic solution.

The environment conditions for this objective used to be the following:

- We have a set of samples, with a number of clear targets, because it is the problem that we aimed to resolve (classification, regression, pattern matching in general)
- For each sample, we have several inputs. These inputs are not provided with clear evidence if they are necessary or important.
- And finally, we don't know whether all the obtained samples are suitable for the training, validation or testing of our neural network.

Our method consisted in the following phases:

Phase 1: Selection of the best inputs by means of an evolutionary computation based on the Delta Test [13].

Phase 2: Filtering of the samples through replicator neural networks [14].

Phase 3: Optimization of the topology of neural networks by means of heterogeneous evolutionary computation [15].

Phase 4: Optimization of the initial weights of neural networks by means of evolutionary computation.

Phase 5: Final training of the neural network with the topology obtained in phase 3 and with the initial weights obtained in phase 4.

With this method, it is very important to be able to secure, and evaluate, the capacity for generalization of the optimal neural network obtained. For this objective, we set the incoming constraints:

– The set of samples after phase 2 filtering was divided into four subsets: training, validation, optimization and testing. The training subset was used to train the multilayer perceptron (MLP) in the fitness function of the evolutionary algorithm in phases 3 and 4. The validation subset was used for the early stopping of the MLP training in phases 3 and 4. The purpose of the optimization subset was to obtain the objective (a single objective or multi-objective) of the fitness function for each individual of the population. Finally the test subset was used to evaluate the final optimized neural network. We followed the recommendation that the test subset should not be used for the identification of the best-performing-trained neural network [16].
– In phases 3 and 4 many training runs for the MLP were necessary. The Resilient Backpropagation algorithm (RBP) was used, with this decision being based on two considerations: the speed of the algorithm and, importantly, the ease of implementation of the hardware for all the technologies used in the heterogeneous platform. In order to improve the generalization properties of RB algorithm, we performed early stopping on the validation subset.
– The final training run was carried out with the Bayesian regularization algorithm, with the union of the training,validation and optimization subsets.
– In phase 3, the optimization of the topology, we performed a multi-objective optimization, in which the second objective of the evolutionary computation was the reduction of the number of connections. This technique improved the generalization capacity of the resulting neural network [17]

2.1 Optimization of the topology

Phase 3 is the key part of our method. Our main contribution is to separate the feature selection (phase 1) and the identification of the initial weights (phase 4) from this optimization process. It is very common to combine the optimization of the three goals with the evolutionary computation [18], but this impairs the performance of the optimization of the topology.

In order to undertake the feature selection, it was very important to do without the intervention of a neural network, because in this way we became independent the selection of variables against the network topology. Until carrying out this study, we had used the optimization of the Delta Test using genetic algorithms, with regression problems with only one output [13], but this approach can be extended to classification problems with multiple outputs. The particular focus of this study did not concern this phase of the methodology. We are convinced that different, more efficient methods can be used, but, in our experiment, our aim was to have an orderly and efficient partition problem that could be optimized and implemented independently.

As has been said in the introduction section, the separation of the topology optimization and initialization of the weights is challenging because there is a very high dependency between them.

Our first approximation to the problem of separating the optimization problem was to perform several initializations of the weights in each individual of the population of different topologies, and to calculate the fitness function of the genetic algorithm with the average performance of these initializations.

Three main problems with this approach were:

- The enormous computational effort. This is the main contribution of our work and further details on this are provided in section 3.
- The control of the generation of the random numbers necessary for the initialization of the weights in the training algorithms. Without this control, the best individuals in the population may be lost because a good individual may decrease the value of the fitness function in subsequent generations of the evolutionary algorithm (see section 2.1.1).
- The choice of the best method to extract the best individuals when the different initializations are averaged (see section2.1.2).

2.1.1 Control of random number generation

The second problem was easily detected, because the best individual varied from generation to generation , and this behavior was independent of the method used to extract the best individual (third problem).

It is very important in our method to get the value of the fitness function of the best individuals will decrease in the evolution of the genetic algorithm. For this objective, the random generation stream of the individuals needed to be controlled, without affecting the random generation properties of the main genetic algorithm and preventing individuals from working with the same random number generator. All these properties were achieved by means of the following actions in the fitness function:

- The definition of a random generator stream with a defined seed particular to each individual. Therefore, three variables were necessary for the evolutionary computation of the topology of the neural networks: the number of neurons in the first hidden layer, the number of neurons in the second hidden layer and the seed of the random generator stream.

– The resetting of the number generator stream in order to reproduce the results of the best individuals.
– Different sub-streams for the different initializations of the weights for each individual.

2.1.2 Weight initialization

For the third problem, three different alternatives were used:

– A fitness function with the goal of the minimum mean of mse (mean squared error) of the same topology with the different weight initializations. This alternative was named GATOPOMEAN.
– A fitness function with the goal of the minimum mse of the same topology with the different weight initializations. This alternative is named GATOPOMIN.
– A fitness function with the goal of the best fitness value for the evolutionary computation of the initialization of the weights (as phase 4). In other words, the execution of an evolutionary computation for each individual as part of the main evolutionary computation. This alternative is named GATOPOGA.

The detailed assessment of these three alternatives was not an objective of this study. However, the computational effort required for each alternative is obvious, and the need of a high flexibility in the implementation of the individuals of the genetic algorithm.

3 Heterogeneous computational platform

From the point of view of implementation, the method of optimization of neural networks explained in section 2, and specifically with regard to phase 3 of the method, required a host processor in charge of the main genetic algorithm, which it is manager of the obtention of the topology of ANN.

The number of variables to be handled was reduced to 3 and therefore the population to be managed is realively limited. Each individual must implement the training of a neural network consisting of two hidden layers and various initializations of the weights (GATOPOMEAN and GATOPOMIN), with a nested genetic algorithm in the most complex case (GATOPOGA).

To implement the host processor, we opted for a generalist solution based on a multicore CPU using MATLAB, enabling us to easily adapt to platforms in order to obtain experimental samples, make quick implementation comparisons with existing toolboxes concerning global optimization methods and implementation of neural networks, and obtain a straightforward representaion of the results. This generalist host platform had access to the GPU through a graphics board connected to the main board. Our study does not focus on this host processor, as it is a very common platform, but evidently

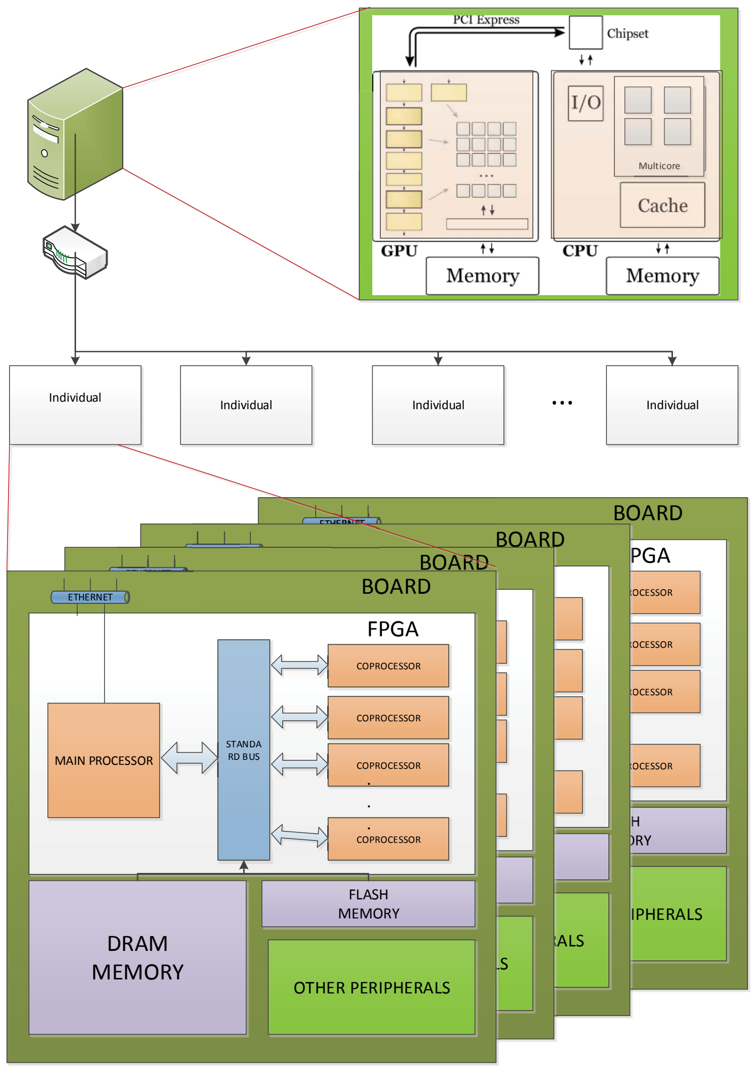


Fig. 1: Heterogeneous computational platform.

the minimum requirements are described. The structure can be seen at the top of Fig 1.

Our main contribution to this platform was the provision of the ability to connect the host processor (via simple socket clients) to a set of embedded processors (soft or hard) over an FPGA technology that provides us with many different solutions (systems) on a programmable chip (Fig. 1). This contribution enabled us to implement the individuals of the main genetic algorithm with the following alternatives:

- CPU cores
- GPU processors
- FPGA1 soft embedded processor + soft embedded processor with neural instructions
- FPGA2 soft embedded processor + programmable coprocessor by software
- FPGA3 soft embedded processor + programmable coprocessor by hardware
- FPGA4 hard embedded processor + soft embedded processor with neural instructions
- FPGA5 hard embedded processor + programmable coprocessor by software
- FPGA6 hard embedded processor + programmable coprocessor by hardware
- FPGA7 soft embedded processor
- FPGA8 hard embedded processor

Our group started with technologically possible solutions with devices using Altera's Cyclone IV family (FPGA1, FPGA2 and FPGA7). Evidently the FPGA technology enabling solutions (FPGA3, FPGA4, FPGA5, FPGA6 and FPGA8) had to be able to draw on the features provided by devices in Altera's Cyclone V family, which are shared by new devices from other FPGA vendors:

- Embedded hard cores
- Run time reconfiguration
- Partial reconfiguration

There are several precedents for implementing neural networks on FPGA. Sankaradas et al.[19] developed a coprocessor for convolutional neural networks, and they made a comparison of the acceleration in front of other technologies (CPU). Prado et al. [20] provided a training platform for reconfigurable topology. The drawback to this application is that it has to synthesize the topology each time it is changed and then implement it in FPGA. Another application implements a fixed-topology ANN and the backpropagation algorithm (BP) for training [21]. Wu et al. [22] proposed a BP algorithm implementation in FPGA, this is reconfigurable by means of software. Pinjare et al.[23] have implemented an FPGA accelerator for online training.

In the majority of these studies, the implementation of the ANN with FPGA (with online training) takes place with very complex and efficient processing elements (PE), but these implementations present great difficulties in terms of communication with them, their flexibility and, and above all, their adaptability to technological changes.

The first step for our platform that implements an individual was to resolve the issue of communication with the host, obviously for the particular type of algorithm running on this host. This requires that the portion of the platform that handles individuals, implementing a processor provided to be in charge of managing a socket server. Therefore, all our FPGA based solutions had one main processor (soft or hard macro) responsible for this task, usually requiring an operating system that helps management of the sockets.

On the issue of flexible platforms, we proceeded first with implementations that achieve the flexibility to allow the implementation of neural networks of different topology by using an architecture acting as a co-processor that acquire the instructions from the main processor. Therefore a new task was implemented in the main processor: the compilation of the instructions for the coprocessor. This was carried out for the implementations FPGA1, FPGA2, FPGA4 and FPGA5. In the future, we are convinced that the hardware reconfiguration of the coprocessor (partial reconfiguration) may well be a very good option for the creation of very efficient accelerators without a loss of flexibility (FPGA3 and FPGA6)

Finally, the issue of adaptability to new technologies was addressed by employing a standard interface (or a widespread interface) permitted the straightforward re-use of its intellectual property for different technological applications. We started with Altera Avalon, a proprietary bus system for Altera's Nios II SoCs, but in the future AMBA, a proprietary bus system from ARM, may provide the definitive solution for adaptability.

3.1 Training of neural networks

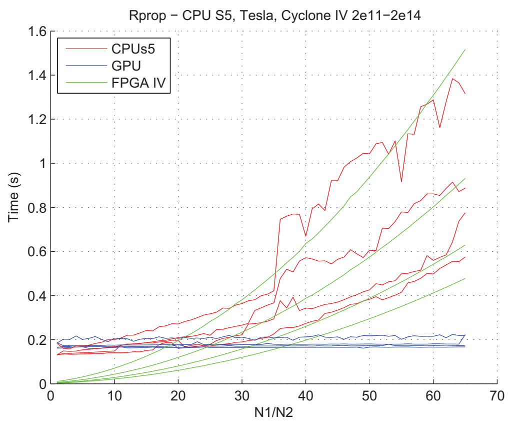
In the fitness function of each individual of the main genetic algorithm population, the computational effort was centered on the training of the neural network with resilient backpropagation.

We compared the speed efficiency of the different technologies in order to train the neural networks in accordance with the size of the topology of the ANN and the size of the training set (the number of samples). This comparison was very important for our evolutionary algorithm, because the different individuals of the populations in our method (phase 3) were going to work with topologies of different size.

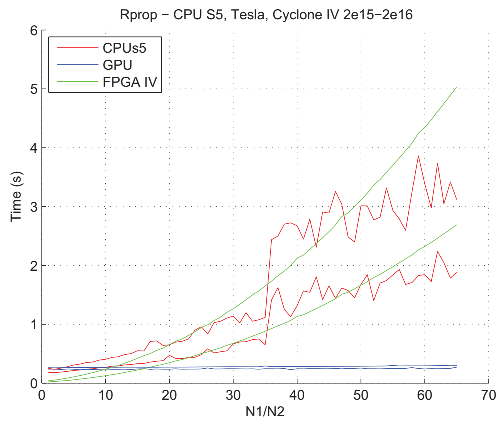
The results of these comparisons are shown in Fig. 2 and 3, and it can be seen that different technologies are suited to training different sizes of neural networks:

- For 2^{15} training samples (shown in Fig. 2a), our training run with an FPGA Cyclone IV (100 MHz coprocessor clock) was faster than a Tesla C1060 GPU (602 MHz core clock) when the number of neurons in the hidden layer was less than 20 (topology of the ANN: 6-20-20-1).
- For 2^{19} training samples (shown in Fig. 3b), our training run with a Xeon CPU (2534 MHz core clock) is faster than Tesla C1060 GPU (602 MHz core clock) when the number of neurons in the hidden layer was less than 6 (topology of the ANN: 6-6-6-1)

These results do not claim to be an exhaustive comparison of technologies for the specific task of training neural networks. This is not an objective of this article because there are many other variables that we have not taken into account, such as:



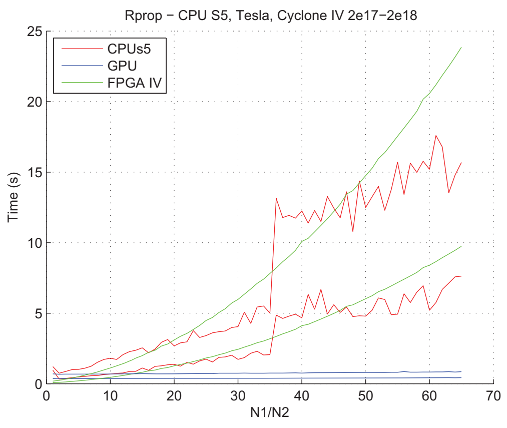
(a) Number of samples: 2^{12}, 2^{13}, 2^{14} and 2^{15}



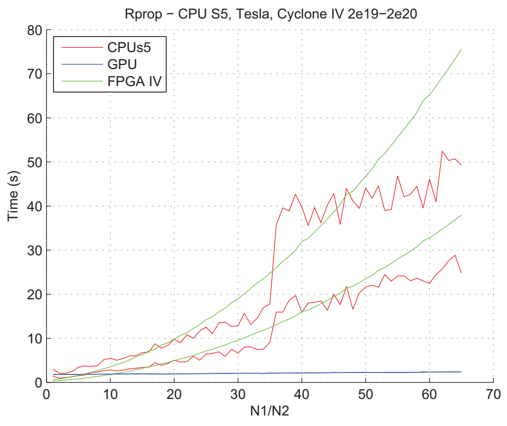
(b) Number of samples: 2^{16} and 2^{17}

Fig. 2: Training RBP, Comparative CPU(XEON)-GPU(TESLA1060)-FPGA(cyclone IV) with different topologies, 16 epochs

– The type of training algorithm;
– The different technological examples are not totally contemporary;
– The resolution with the FPGAs was the single precision IEEE754 floating point format and the CPU and GPU used the double precision IEEE754 floating point format;
– The importance of the software layers and operating systems used in the CPU and GPU;
– The programming mode (essential when working with CUDA);
– And the number of CPU cores used (only one of the four available cores in the Xeon series was used).



(a) Number of samples: 2^{18} and 2^{19}



(b) Number of samples: 2^{20} and 2^{21}

Fig. 3: Training RBP, Comparative CPU(XEON)-GPU(TESLA1060)-FPGA(cyclone IV) with different topologies, 16 epochs

Our proposal with these experiments is that the choice of a heterogeneous platform for the implementation of evolutionary computation for the identification of the optimal topology of a neural network makes sense when the objective function is essentially based on neural network training.

Of course, in terms of computational effectiveness, the FPGA solution is the best, but the availability of this technology, together with the design effort of reprogramming and its adaptability to new versions of training algorithms, is the main disadvantage of this type of technology solutions that only the emergence of standards like OpenCL may circumvent.

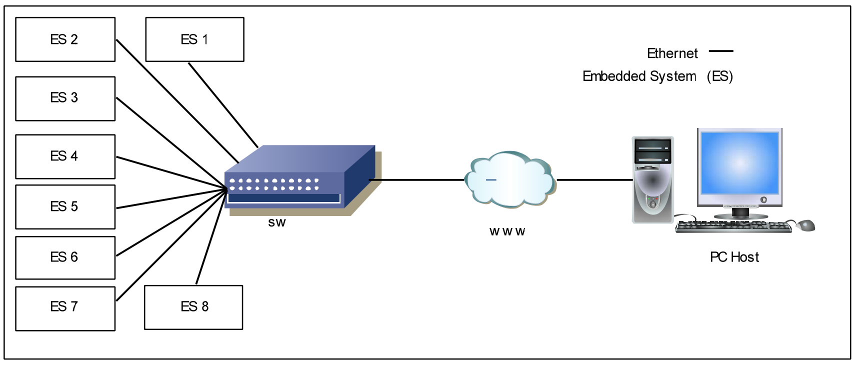


Fig. 4: ANN platform training.

4 Example of Platform FPGA2

The platform consists of eight embedded systems development boards (ES). The boards have an Altera Cyclone IV EP4CE115F29CN FPGA [24]. Each ES use the coprocessor [25] called the Neural Network Processor (NNP), having a training system in FPGA based on RBP algorithm. Each ES communicates with a host PC via Ethernet, this PC with MATLAB [26], Global Optimization Toolbox and Parallel Computing Toolbox, manages training with genetic algorithm (GA) and parallel tasks executed in each of the ES. Getting a 6x-7x acceleration compared with Quad-Core AMD Opteron 2354 and 3x-4x acceleration compared with Processor Intel Xeon Quad-Core E5540.

The experimental platform is presented in Fig. 4 and is composed of a host multi-core PC, and eight ESs, connected to the network via Ethernet. Using this mode of communication provides scalability, efficient data transfer, and the ability to quickly add further ESs.

As the number of variables for the determination of the topology or the initial weights varied, when a GA was used in the systematic search for both variables (phases 3 and 4 of our method), an initial population of individuals, n, was generated, and the calculation of each individual was performed in each ES, allowing the calculation of eight individuals simultaneously.

4.1 Software

This section presents the execution flow of the software in order to achieve optimal topology and determine the initial weights of an ANN. The software responsible for managing the training tasks was MATLAB with the Parallel Computing Toolbox (PCT), which enables the management of the tasks running on each processor core and the execution of the tasks of higher computational cost running on an ES. Besides using Global Optimization (GO),

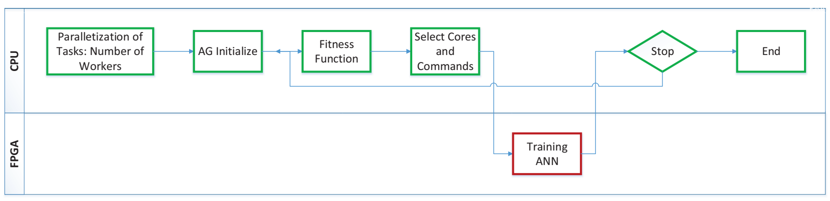

CPU
FPGA
Paralletization of Tasks: Number of Workers
AG Initialize
Fitness Function
Select Cores and Commands
Stop
End
Training ANN

Fig. 5: Software flow diagram.

this provides different methods for finding solutions to different problems and, one of these is with genetic algorithms, which are based on natural selection and the laws of genetics.

4.2 GA tasks

In GA the most important parameters to configure are the number of individuals, the number of generations and the fitness function. These parameters determine the execution time of the GA in this application. The computation time depends on the number of individuals to be computed in parallel and, this parameter is configurable in the algorithm by combining the PCT with GO. 1 to 8 individuals can be run in parallel. Fig. 5 shows a flowchart of the algorithm. Here is a brief description of each process:

- Initialize the number of cores used in the host PC for ANN training.
- Transmit the set of training vectors to SDRAM memory in the ES.
- Configure the genetic algorithm (number of generations, numbers of individuals, use PCT, etc).
- Called to the fitness function.
- The fitness function determines whether there is a free core and then generates an identifier, which is assigned an IP address.
- The fitness function communicates with each ES via the IP address.
- The process continues until all the individuals and generations set in the GA have been processed.
- The stopping method is by the number of generations.

4.3 Embedded System

Using a Terasic development board DE2-115 to implement the embedded system, the ES is mainly composed of a real-time RTOS MicroC/OS-II operating system, and NIOS II/f processor and an NNP coprocessor. The NNP is a soft-core single instruction, multiple data-path (SIMD) processor using single-precision IEEE754. In Fig.6 shows the implemented hardware.

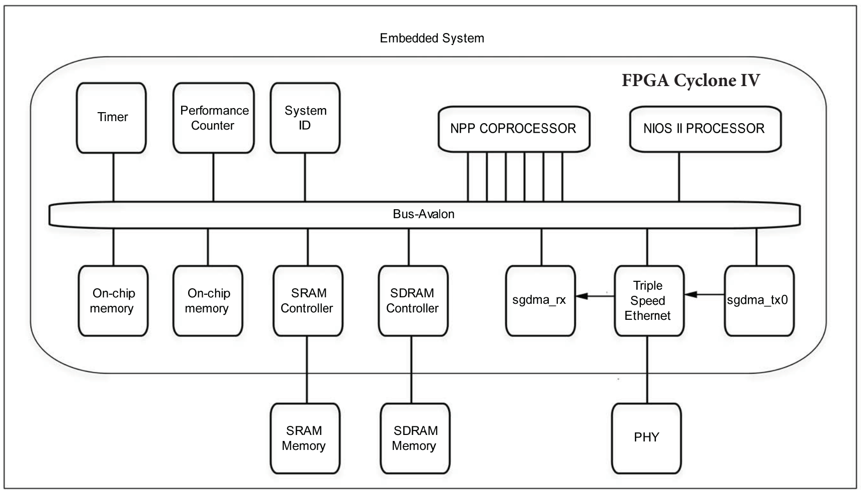
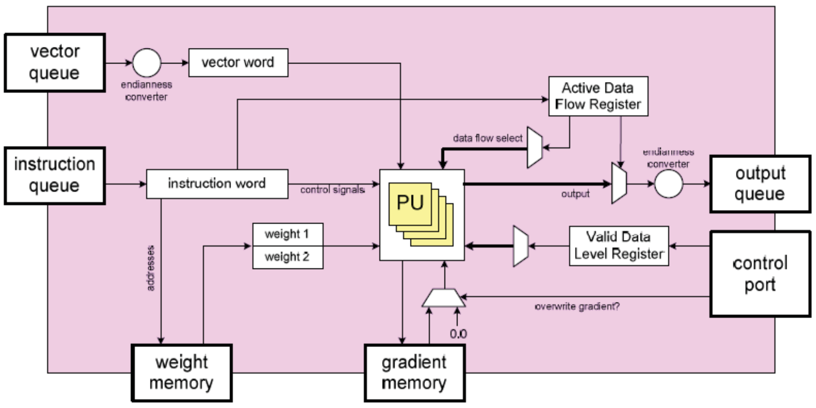


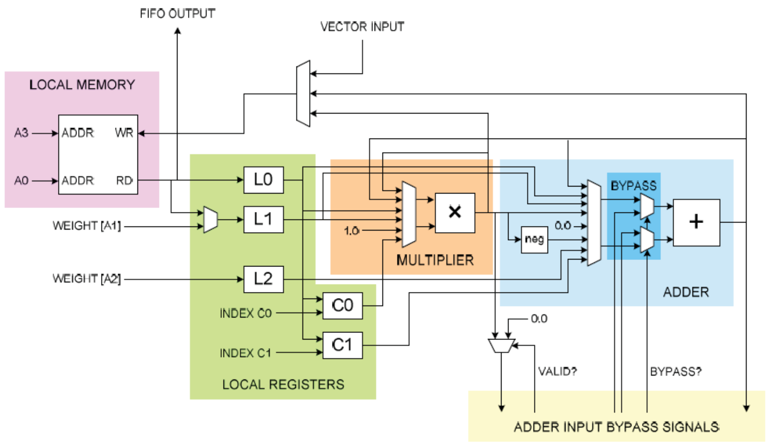
Fig. 6: Embedded System.

The Nios embedded processor is the master and controls the other system components, running the program memory from External 2MB SSRAM to FPGA. The training vectors are stored in the external SDRAM. The remaining components are internal to the FPGA, and in addition to the coprocessor, there are two DMAs that control the training vector transfer and instructions to the coprocessor, a memory of 16 KB which stores coprocessor instructions, and a memory of 4 KB (double the size of the memories of weights or gradients) for the variables needed for the RBP algorithm, corresponding to local increases and gradients of a previous epoch. Fig.7a shows the connection diagram and internal architecture of the NNP and Fig.7b shows the schematic for the contents of each processor unit (PU). It consists of an adder unit, a multiplier unit,a local memory block and a limited number of local registers.

The NIOS II/f processor receives instructions through the Ethernet connection to generate a training run. These instructions to configure the topology, if the generation of the initial weights is undertaken by the NIOS II, or should start with weights sent to memory via the Ethernet connection and the number of epoch. Next, with all data received, NIOS is responsible for generating the instructions for each new topology received, then, known [25] that the coprocessor is capable of processing a certain number n of parallel training vectors and the gradients generated automatically accumulate, ultimately appearing in the memory of gradients at the end of the epoch. The reading and processing of each group of n vectors are obtained by means of a set of instructions NNP.



(a) Overall structure of the copprocessor NNP



(b) Structure of each PU

Fig. 7: Neural Network Coprocessor.

4.4 Performance of the platform

To demonstrate the validity of the platform as an accelerator, we determined the initial weights of an ANN (evolutionary computation of phase 4 of our method). We worked with a fixed 6/5/3/1-topology (obtained in phase 3 of our methodology) with a sample set of 243,000 training vectors. This choice for the determination of the performance of our platform was necessary because a phase with a deterministic number of training runs with a fixed topology is necessary in order to allow us to compare technologies when they undertake the same computational effort.

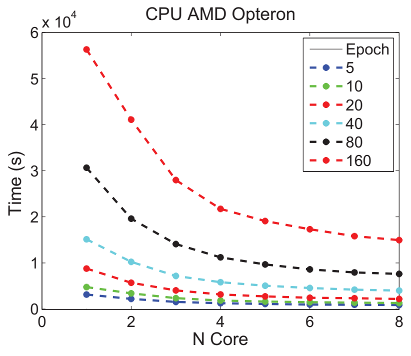
Table 1: Total Time - Command time

(a) Case I OPTERON&ES

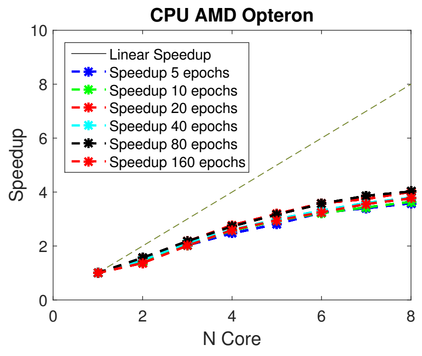
	Total Time	Command Time
5 epochs	1432.105	105.4
10 epochs	1749.802	100.2
20 epochs	2390.985	98.2
40 epochs	3663.150	97.4
80 epochs	6210.783	108.8
160 epochs	11301.815	109.3

(b) Case II XEON&ES

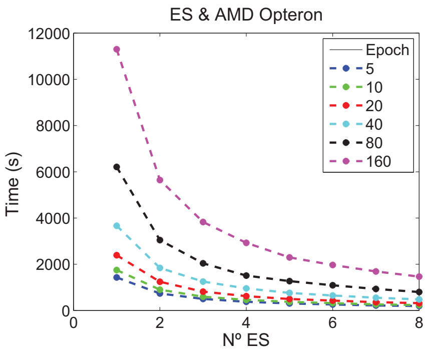
	Total Time	Command Time
5 epochs	1236.961	185.2
10 epochs	1578.712	166.9
20 epochs	2237.215	146.8
40 epochs	3520.052	132.0
80 epochs	6078.963	118.7
160 epochs	11136.066	118.3



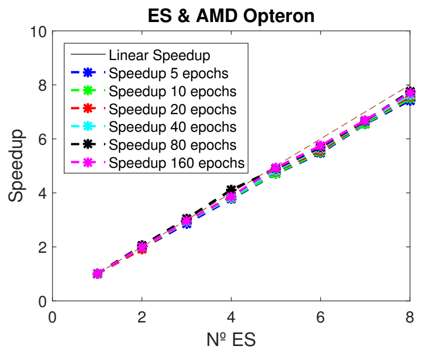
(a) Time: OPTERON with 8 cores



(b) Speedup: OPTERON with 8 cores

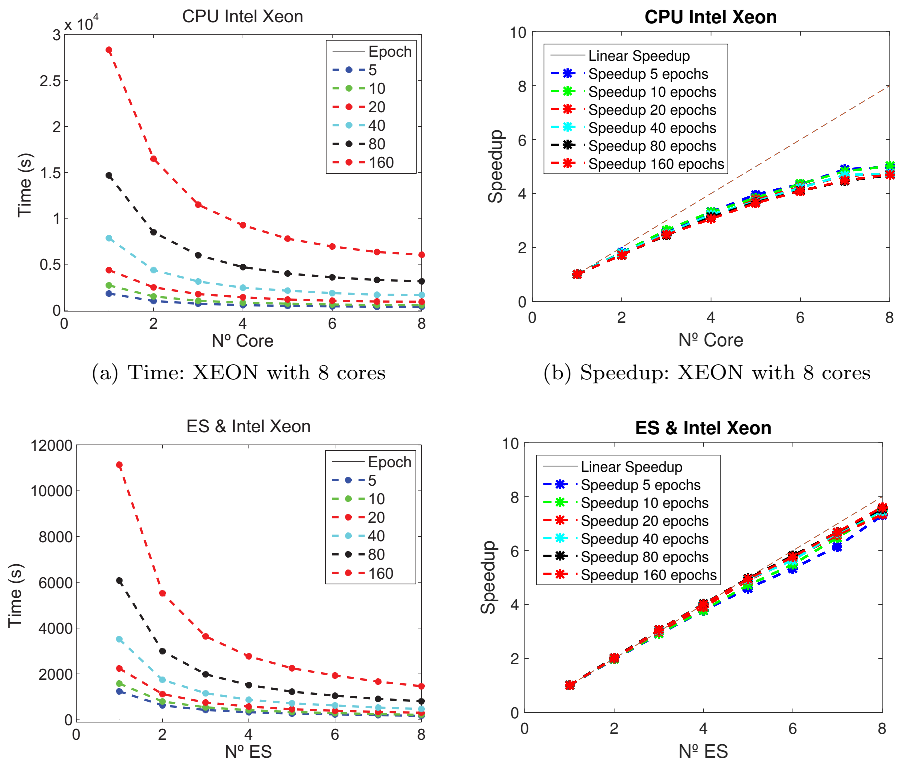


(c) Time: OPTERON with 8 ES



(d) Speedup: OPTERON with 8 ES

Fig. 8: Case I homogeneous and heterogeneous platforms with OPTERON



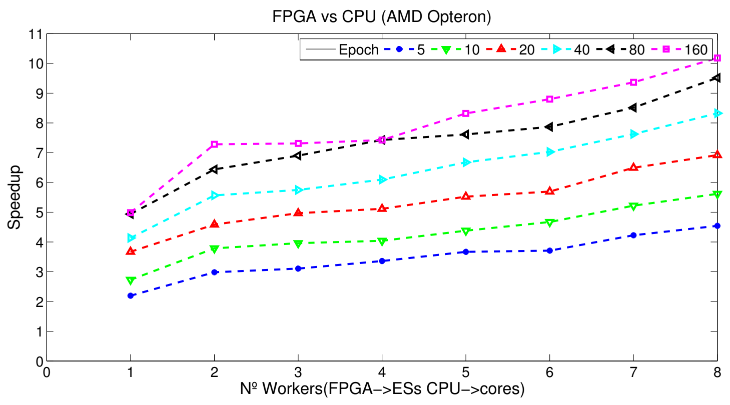
(a) Time: XEON with 8 cores (b) Speedup: XEON with 8 cores

(c) Time: XEON with 8 ES (d) Speedup: XEON with 8 ES

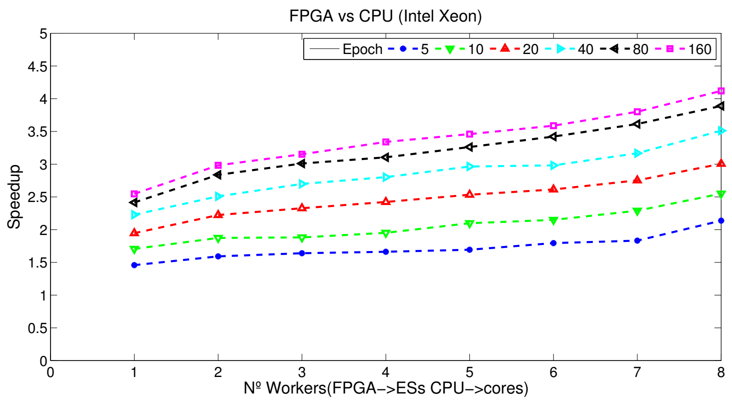
Fig. 9: Case II homogeneous and heterogeneous platforms with XEON

Fig. 8a shows the results for time taken by an evolutionary computation (phase 4 of our method of optimization of ANN) with only an AMD workstation (host processor and individuals) and in Fig. 8c with an AMD workstation (host processor) and ESs (implementing the individuals). We varied the number of cores used in case 8a and the number of ESs in case 8c. In Fig . 9a and 9c show the results of the same experiment with an Intel workstation.

Before we examine the acceleration of our proposed platform (with ESs), we can observe and conclude that our heterogeneous platforms (see Fig.8d and 9d) divide linearly the computational effort when the number of processing elements (remote ESs) for implementing the individual increases. This shows that the time dedicated to the calculation of the population of the different generations and the time dedicated to communicate commands and data with the individuals of the population (in our solution through simple sockets) are practically negligible for the calculation of the objective function of each individual. We detail these times in Table 1.



(a) Case I OPTERON&ES



(b) Case II XEON&ES

Fig. 10: Acceleration of our FPGA2 platform

Our proposed heterogeneous platform achieve an acceleration of 3x-4x compared with a 2.83 GHz Intel Xeon Quad-Core and 6x-7x compared with a 2.2GHz AMD Opteron Quad-Core 2354 (see Fig. 10).

It is very interesting to note in Figure 10 as improved acceleration as the number of workers increases. This convinces us about the suitability of our contribution: send computing of the fitness function of the individuals of the evolutionary algorithm to a remote computing system.

5 Conclusions

This paper has shown firstly the need for heterogeneous computing as we proceed to the optimization of neural networks using evolutionary algorithms.

In our optimization methodology the most expensive phase computationally is the selection of network topology, and we have shown in our section 3 that the best solution to address this evolutionary algorithm is that different individuals are implemented by hardware platforms with different technologies and different architectures.

Secondly we have shown that our solution is the distribution of computing via sockets (section 4) is scalable, flexible (to adapt to heterogeneous computing) and efficient compared to conventional solutions or homogeneous platforms.

Clearly it is an initial step that must be continued to incorporate GPUs. We have shown that GPUs are very efficient when topologies are large with a large number of learning samples; but we need to adapt this type of computing to our distribution system via sockets.

In the field of FPGA implementations, our future work will be directed to the use the partial and dynamic reconfiguration. That is, several systems could be loaded in flash memory in order to enable the hardware to be reconfigured during the execution of the evolutionary algorithm. This method would permit us to use for the different individuals, the optimal hardware implementation for each topology, thereby increasing the overall efficiency of the system.

Acknowledgments

The translation of this paper was funded by the Universitat Politècnica de València, Spain

References

1. A. Farmahini-Farahani, S. Vakili, S. M. Fakhraie, S. Safari, and C. Lucas, “Parallel scalable hardware implementation of asynchronous discrete particle swarm optimization,” *Engineering Applications of Artificial Intelligence*, vol. 23, no. 2, pp. 177–187, 2010.
2. S. Curteanu and H. Cartwright, “Neural networks applied in chemistry. i. determination of the optimal topology of multilayer perceptron neural networks,” *Journal of Chemometrics*, vol. 25, no. 10, pp. 527–549, 2011. [Online]. Available: http://dx.doi.org/10.1002/cem.1401
3. M. M. Islam, M. A. Sattar, M. F. Amin, X. Yao, and K. Murase, “A new adaptive merging and growing algorithm for designing artificial neural networks,” *Ieee Transactions on Systems Man and Cybernetics Part B-Cybernetics*, vol. 39, no. 3, pp. 705–722, 2009.
4. K. H. Han and J. H. Kim, “Quantum-inspired evolutionary algorithms with a new termination criterion, h-epsilon gate, and two-phase scheme,” *Ieee Transactions on Evolutionary Computation*, vol. 8, no. 2, pp. 156–169, 2004.
5. T. B. Ludermir, A. Yamazaki, and C. Zanchettin, “An optimization methodology for neural network weights and architectures,” *Ieee Transactions on Neural Networks*, vol. 17, no. 6, pp. 1452–1459, 2006.
6. L. Y. Ma and K. Khorasani, “Constructive feedforward neural networks using hermite polynomial activation functions,” *Ieee Transactions on Neural Networks*, vol. 16, no. 4, pp. 821–833, 2005.
7. V. Maniezzo, “Genetic evolution of the topology and weight distribution of neural networks,” *Ieee Transactions on Neural Networks*, vol. 5, no. 1, pp. 39–53, 1994.

8. P. P. Palmes, T. Hayasaka, and S. Usui, "Mutation-based genetic neural network," *Trans. Neur. Netw.*, vol. 16, no. 3, pp. 587–600, May 2005. [Online]. Available: http://dx.doi.org/10.1109/TNN.2005.844858
9. T. Mu, J. Jiang, Y. Wang, and J. Y. Goulermas, "Adaptive data embedding framework for multiclass classification," *Ieee Transactions on Neural Networks and Learning Systems*, vol. 23, no. 8, pp. 1291–1303, 2012.
10. T.-C. Lu, G.-R. Yu, and J.-C. Juang, "Quantum-based algorithm for optimizing artificial neural networks." *IEEE Trans. Neural Netw. Learning Syst.*, vol. 24, no. 8, pp. 1266–1278, 2013.
11. X. Yao, "Evolving artificial neural networks," *Proceedings of the Ieee*, vol. 87, no. 9, pp. 1423–1447, 1999.
12. X. Yao and Y. Liu, "A new evolutionary system for evolving artificial neural networks," *Ieee Transactions on Neural Networks*, vol. 8, no. 3, pp. 694–713, 1997.
13. F. Mateo, D. Sovilj, and R. Gadea-Gironés, "Approximate k-NN delta test minimization method using genetic algorithms: Application to time series," *NEUROCOMPUTING*, vol. 73, no. 10-12, Sp., pp. 2017–2029, JUN 2010.
14. S. Hawkins, H. He, G. Williams, and R. Baxter, "Outlier detection using replicator neural networks," in *In Proc. of the Fifth Int. Conf. and Data Warehousing and Knowledge Discovery (DaWaK02*, 2002, pp. 170–180.
15. J. Fe, R. J. Aliaga, and R. G. Gironés, "Experimental platform for accelerate the training of anns with genetic algorithm and embedded system on fpga," in *IWINAC (2)*, 2013, pp. 413–420.
16. L. Prechelt, "Proben1 - a set of neural network benchmark problems and benchmarking rules," Tech. Rep., 1994.
17. H. A. Abbass, "An evolutionary artificial neural networks approach for breast cancer diagnosis," *Artificial Intelligence in Medicine*, vol. 25, pp. 265–281, 2002.
18. F. Ahmad, N. A. M. Isa, Z. Hussain, and S. N. Sulaiman, "A genetic algorithm-based multi-objective optimization of an artificial neural network classifier for breast cancer diagnosis," *Neural Computing and Applications*, vol. 23, no. 5, pp. 1427–1435, 2013.
19. M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. Graf, "A massively parallel coprocessor for convolutional neural networks," in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, july 2009, pp. 53 –60.
20. R. Prado, J. Melo, J. Oliveira, and A. Neto, "Fpga based implementation of a fuzzy neural network modular architecture for embedded systems," in *Neural Networks (IJCNN), The 2012 International Joint Conference on*, june 2012, pp. 1 –7.
21. M. avulu, C. Karakuzu, S. ahin, and M. Yakut, "Neural network training based on fpga with floating point number format and its performance," *Neural Computing and Applications*, vol. 20, pp. 195–202, 2011. [Online]. Available: http://dx.doi.org/10.1007/s00521-010-0423-3
22. G.-D. Wu, Z.-W. Zhu, and B.-W. Lin, "Reconfigurable back propagation based neural network architecture," in *Integrated Circuits (ISIC), 2011 13th International Symposium on*, dec. 2011, pp. 67 –70.
23. S. L. Pinjare and A. K. M, "Article: Implementation of neural network back propagation training algorithm on fpga," *International Journal of Computer Applications*, vol. 52, no. 6, pp. 1–7, August 2012, published by Foundation of Computer Science, New York, USA.
24. http://www.altera.com.
25. R. Aliaga, R. Gadea, R. Colom, J. Cerda, N. Ferrando, and V. Herrero, "A mixed hardware-software approach to flexible artificial neural network training on fpga," in *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS '09. International Symposium on*, july 2009, pp. 1 –8.
26. http://www.matlab.com.