2009 Ninth International Conference on Intelligent Systems Design and Applications

Hardware Architecture for Particle Swarm Optimization using Floating-point Arithmetic

Daniel M. Muñoz, Carlos H. Llanos
Department of Mechanical Engineering
Automation and Control Group/GRACO
University of Brasilia
Brasília, Brazil
e-mail: {damuz,llanos}@unb.br

Leandro dos S. Coelho
Industrial and Systems Engineering
Graduate Program, LAS/PPGPES
Pontificial Catholic University of Paraná
Curitiba, Brazil
e-mail: leandro.coelho@pucpr.br

Mauricio Ayala-Rincón
Departments of Computer Science and Mathematics
University of Brasilia
Brasília, Brazil
e-mail: ayala@unb.br

***Abstract*—High computational cost for solving large engineering optimization problems point out the design of parallel optimization algorithms. Population based optimization algorithms provide parallel capabilities that can be explored by their implementations done directly in hardware. This paper presents a hardware implementation of Particle Swarm Optimization algorithms using an efficient floating-point arithmetic which performs the computations with high precision. All the architectures are parameterizable by bit-width, allowing the designer to choose the suitable format according to the requirements of the optimization problem. Synthesis and simulation results demonstrate that the proposed architecture achieves satisfactory results obtaining a better performance in therms of elapsed time than conventional software implementations.**

Keywords-PSO; FPGA; Floating-point arithmetic

I. Introduction

Heuristics are important tools for nonlinear optimization problems in which the algorithms neither require continuity or differentiability of the objective function. Particle Swarm Optimization (PSO) is one of the heuristic algorithms that can be applied to nonlinear optimization problems [1].

PSO is a stochastic technique, bio-inspired on the social behavior of bird flocking, which provides several desired attributes, such as, simplicity, less computational requirements, parallelism [2], [3] and easy implementation without calculations of the gradient [4].

Although non-gradient based optimization algorithms present low computational cost, requiring only primitive mathematical operators, these techniques have a high elapsed time to solve large-scale engineering problems. Therefore, parallel implementations of population based optimization algorithms can increase the throughput and improve their performance.

Modern Field Programmable Gate Arrays (FPGAs) have hundred of thousand of logic elements, allowing the designers to develop, directly in hardware, complex algorithms that are commonly implemented in software. Hardware implementation of optimization algorithms can decrease the computation time by expressive processing speed-up and by performing simultaneous computations.

Most of the previous works regarding parallel implementations of the PSO algorithm consider clusters of networked personal computers [3], [5], [6]. Alternatively, FPGA implementations are a feasible and cheap solution for performing parallel optimization algorithms. Previous works covering FPGA implementations of the PSO algorithm consider the conventional fixed-point arithmetic; however, several optimization problems require to operate with a high degree of precision. Therefore, an FPGA implementation of parallel PSO algorithms using floating-point arithmetic is of great importance due to the large dynamic range for representing large and small real numbers, flexibility of architectures and customizable approaches, which allow the involved algorithms for a better performance.

This paper presents an FPGA implementation of a parallel floating-point PSO algorithm. All the architectures have been developed in Hardware Description Language (VHDL) and are based on the IEEE-754 standard. Also, the architectures are parameterizable by bit-width, allowing the designer to chooses the suitable format according to the precision requirements of the optimization problem. A detailed analysis over the performance and precision aspects of the floating-point operators involved in the PSO algorithm has been presented in [7] and [8], which must be considered in the hardware design, especially in the fitness function implementation. The proposed PSO architecture has been validated using well-known unimodal and multimodal benchmark functions, showing the effectiveness of the hardware parallel PSO algorithm.

Section II describes the PSO operation. Section III presents the related works. Section IV describes the hardware parallel PSO implementation and, before concluding, Section V presents synthesis and simulation results.

II. The PSO Operation

The particle swarm optimization (PSO) is a heuristic and stochastic algorithm, introduced in 1995 by Kennedy and Eberhart [9]. The PSO is based on the social behavior of social populations when adapting to the environment, in which the individual tends to return to the place that most satisfied it in the past [9].

978-0-7695-3872-3/09 $26.00 © 2009 IEEE
DOI 10.1109/ISDA.2009.107

In the PSO operation, the population is called swarm and each individual is called *particle* (mass-less and volume-less). Each particle i has a current velocity vector $\boldsymbol{v}_i$, a personal best position vector $\boldsymbol{y}_i$ in the search space and a position vector $\boldsymbol{x}_i$, that represents a possible solution of the optimization problem. Considering a N-dimensional evaluation function and a swarm size of S particles, the position of the i^{th} particle of the swarm in the j^{th} dimension is updating by executing equations (1) and (2).

$$x_{ij}^{(t+1)} = x_{ij}^{(t)} + v_{ij}^{(t+1)} \quad (1)$$

$$v_{ij}^{(t+1)} = w v_{ij}^{(t)} + c_1 r_1 (y_{ij}^{(t)} - x_{ij}^{(t)}) + c_2 r_2 (y_s^{(t)} - x_{ij}^{(t)}) \quad (2)$$

where r_1 and r_2 are uniformly random numbers between 0 and 1, y_{ij} is the personal best position found by the particle i around the j^{th} dimension and y_s is the global best position amongst all the particles. There are three parameters: the inertia (w), the cognitive (c_1) and the social (c_2) coefficients.

The velocities v_{ij} are clamped to the range $[-v_{max}\ v_{max}]$ avoiding the particles leave the search space. Large values for cognitive coefficient (c_1) indicate particles with a high self confidence on their experience and large values for social coefficient (c_2) provide a particle with a high confidence on the swarm [10]. For unimodal functions is desirable a small cognitive coefficient and a large social coefficient; however for multimodal functions it can be usefully a trade-off between the cognitive and the social coefficients in order to improve the performance.

The inertia weight coefficient w is setting up for decreasing linearly from 1 to 0 until the stopping criteria is met. The inertia controls the exploration capabilities of the particles. Large values for w result in a global search and small values for w allow the particles to a local search [10].

III. Related Works

The standard PSO algorithm has several desired attributes such as, non-gradient calculation and easy implementation using only multiplication and add/sub operators. However, large computational times are required when solving large-scale problems. This problem can be solved by implementing parallel PSO algorithms, taking advantage of the intrinsic parallelism of this technique.

Several previous works regarding parallel PSO algorithms have been developed using a networked array of master/slave CPUs, demonstrating a faster convergence and improving the computational time of the algorithm [3], [5], [6], [11]. However, these solutions are expensive, taking into account that the PSO requires only primitive operations, where, the most computational cost is the evaluation of the fitness function.

An FPGA implementation of the PSO allows the algorithm to improve the performance: (1) by using parallel particles and (2) by performing as many as possible simultaneous operations in both the update process and the fitness function evaluation.

Previous works covering FPGA implementation of PSO algorithms demonstrate the feasibleness of the hardware PSO for solving large-scale optimization problems. Reference [12] implements a PSO algorithm in FPGAs for inversion of large neural networks and shows that the computing time in an FPGA is approximately 6 times faster than conventional computers. A high parallel PSO was implemented in [13] for dynamic optimization of array antennas. A matrix structure for synchronizing the particles to the fitness module was developed on FPGAs, achieving a high degree of parallelism of the PSO algorithm. Reference [14] presents a population-oriented hardware architecture for PSO with Discrete Recombination, validating the architecture with two 32-Dimensional fitness functions (Sphere and Rastrigin). An FPGA implementation of Simultaneous Perturbation PSO (SPPSO) is presented in [15], increasing the operation speed effectively by using the parallelism of the PSO. In [4] an FPGA architecture of wavelet neural networks with PSO is applied to a prediction problem, showing that the performance of the PSO is better than the SPPSO by working with a suitable number of particles.

Most of the previous works covering hardware architectures of the PSO algorithm perform the computations using a fixed-point arithmetic. However, many of the engineering applications require to represent numbers in a dynamic range, performing the computations with high precision, for instance using floating-point arithmetic.

This work presents an FPGA implementation of a parallel floating-point PSO algorithm. The floating-point arithmetic provides a dynamic range for representing large and small numbers, guaranteeing that saturation will not occur for general purpose applications in comparison with the fixed-point arithmetic in which, the designer must chooses the suitable bit-width for the integer and fractional parts. Synthesis results show that the scalability of the PSO architecture depends on the complexity of the fitness function. The proposed PSO architecture was validated using the unimodal Sphere and Quadric functions and the multimodal Rastrigin and Rosenbrock functions. The results were contrasted with Matlab®, showing that the hardware parallel PSO achieves satisfactory results and a less elapsed time than software implementations.

IV. Hardware Implementations

The proposed parallel PSO architecture operates with a floating-point arithmetic and is based on the IEEE-754 standard to represents binary real numbers. This standard is characterized by three components: a sign S, a biased exponent E with Ew bit-width and a mantissa M with Mw bit-width, as shown in Fig. 1.

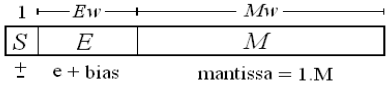

Figure 1. The IEEE-754 standard

A zero bit sign denotes a positive number and an one bit sign denotes a negative number. A constant (bias) is added to the exponent in order to make the exponent's range non negative. The mantissa represents the magnitude of the number. This standard allows the user to work not only with

the 32-bit single precision and 64-bit double precision, but also with a suitable precision according to the application.

A. The Hardware Parallel PSO

The following nomenclature is used: x_{pidj} means the current position of the i^{th} particle in the j^{th} dimension, y_{pidj} means the individual best position of the i^{th} particle in the j^{th} dimension and y_s means the global best position. The fitness value of the current position of the i^{th} particle is represented by $f(\boldsymbol{x}_{pi})$ and its respective fitness value of the individual best position is represented by $f(\boldsymbol{y}_{pi})$.

Fig. 2 shows the general hardware architecture of the parallel PSO. It can be observed a swarm of S particles updating their positions simultaneously in order to optimize a cluster of S fitness functions that represent an N-dimensional optimization problem.

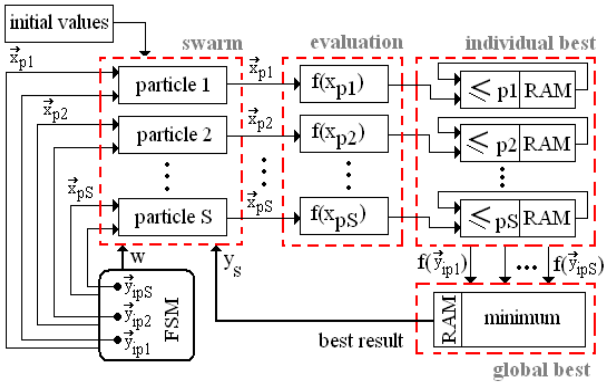

Figure 2. General hardware parallel PSO architecture

The architecture is composed of five main components: (A) *swarm unit*, (B) *evaluation unit*, (C) *individual best detection unit*, (D) *global best detection unit* and (E) a Finite State Machine (*FSM*). The *swarm unit* executes, for each particle, (1) and (2) equations in order to update the velocity and the position. The *evaluation unit* performs the fitness function evaluations in a parallel approach. Notice that each fitness function requires that all the positions in each dimension have been updated. The *individual best detection unit* compares the current fitness values of the particles with the respective best fitness values previously found. If the current fitness value $f(\boldsymbol{x}_{pi})$ is lower than the fitness value of the individual best position $f(\boldsymbol{y}_{pi})$, the individual best position (vector $\boldsymbol{y}_i$) is replaced by the current vector position (vector $\boldsymbol{x}_i$) and then, the individual best position is stored in a RAM. The *global best detection unit* calculates the minimum value among the S best fitness values and stores the corresponding global best position in a RAM. The *FSM unit* synchronizes all the hardware components.

B. The Swarm Unit

As stated by equations (1) and (2), the update process of each particle requires five multiplications, four add/sub operations and two random number generators. In this work, the cognitive (c_1) and social (c_2) coefficients are constants, and then, two multiplications are avoided by setting-up the random number generator in the range [0 to $c1$] or [0 to $c2$].

Fig. 3 presents the hardware architecture of one particle. The update process is performed on five stages (s_0 to s_4), sharing one floating-point multiplier (*FPmul*), one floating-point add/sub unit (*FPadd*) and one floating-point *LFSR* unit for implementing the random number generator (explained below). All the operations in each stage are performed simultaneously.

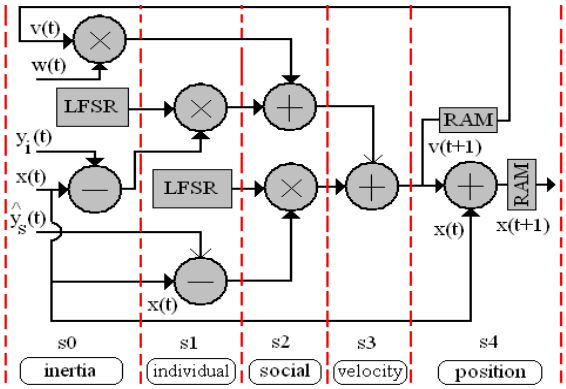

Figure 3. Particle architecture

C. The Floating-point Random Number Generator

A random number generator is required to maintain the stochastic behavior on the movement process of each particle. In this work an uniform pseudo-random floating-point number generator is performed by using two Linear Feedback Shift Register (LFSR), as shown in Fig. 4.

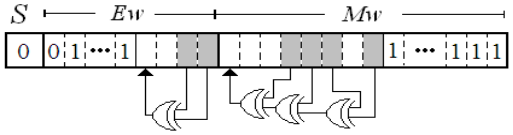

Figure 4. Floating-point random number generator

The LFSR is a shift register based technique, in which several bits, called *taps*, are chosen as a feedback function obtaining a new state. By selecting the appropriate *taps*, the bit sequences work as a pseudo-random number generator. Also, by selecting the register bits in both, the exponent and mantissa words, the random number generator works in the desired range. In this work the range U[0 to $c1$] and U[0 to $c2$] were used, avoiding additional multiplications.

D. The Evaluation Unit

In the *evaluation unit* the following well-known benchmark functions were implemented in hardware and were used in order to validate the proposed hardware parallel PSO architecture.

1) The Sphere function: it is a continuous and unimodal function as shown in (3). The global minimum is $f(\mathrm{x})=0$, $\boldsymbol{x}(i)=0$, $i=1...N$.

$$f(\vec{x}) = \sum_{i=1}^{N} x_i^2 \qquad (3)$$

It was implemented using two parallel *FPmul* units and one *FPadd* unit. An N-dimensional Sphere function can be

easily accomplished by sharing and synchronizing these hardware components.

2) The Quadric function: It is a continuous and unimodal function as defined by (4). The global minimum is $f(\boldsymbol{x})=0$, $\boldsymbol{x}(i)=0$, $i=1...N$

$$f(\vec{x})=\sum_{i=1}^{N}\left(\sum_{j=1}^{i}x_j\right)^2 \quad (4)$$

It was implemented using one *FPmul* unit, one *FPadd* unit and a Finite State Machine (*FSM*) for synchronizing these hardware components over a number of states that will depend on the dimensionality of the problem.

3) The Rosenbrock function: it is a multimodal function as stated by (5). The global minimum is $f(\boldsymbol{x})=0$, $\boldsymbol{x}(i)=1$, $i=1...N$

$$f(\vec{x})=\sum_{i=1}^{N/2}100\cdot(x_{2i}-x_{2i-1}^2)^2+(1-x_{2i-1})^2 \quad (5)$$

It was implemented using one *FPmul* unit, one *FPadd* unit, several registers and a Finite State Machine (*FSM*). Obviously, the number of states and latency will depend on the dimensionality of the problem.

4) The Rastrigin function: it is a highly multimodal function, as shown in (6). The global minimum is $f(\boldsymbol{x})=0$, $\boldsymbol{x}(i)=0$, $i=1...N$

$$f(\vec{x})=\sum_{i=1}^{N}(x_i^2-10\cos(2\pi x_i)+10) \quad (6)$$

It was implemented using two *FPmul* units, one *FPadd* unit, two reduction argument units, two floating-point cosine units and a FSM for synchronizing all the hardware components. The cosine unit computes the cosine function using a Taylor series expansion as described in [8].

V. Results

The proposed hardware parallel PSO architecture was validated for a swarm composed of four parallel particles (S=4) optimizing a simple two-dimensional problem (N=2). Table I lists the experiment conditions.

TABLE I. Experiment Conditions

Parameter	Value
Max. number of iterations	500
Inertia weight	Linearly decrease [0.9 to 0.1]
Initial velocity	0.5
Maximum velocity	3
Domain range of the fitness functions	[-16 to 16] Rosenbrock function [-5.12 to 5.12] remain functions

A. Synthesis results

The hardware Parallel PSO was described in VHDL and synthesized for a Xilinx Virtex 5 family FPGA (xc5vlx330). Different bit-width representations were analyzed, taking into account the domain range of the fitness functions and their largest possible values. Synthesis results are important data in order to evaluate the feasibleness of the proposed circuits. This data summarize the resources consumption (area cost) and performance (frequency of operation) of the implemented circuits.

Table II presents the synthesis results for a swarm with four particles and a two-dimensional problem working with a simple precision format (32 bit-width).

TABLE II. Synthesis Results. 32 bits, 4 Particles, 2 Dimensions

Implemented FP-core	Slices Max 207360	LUTs Max 207360	DSP MUL Max 192	Freq. MHz
Particle	358 (0.17%)	1298 (0.63%)	2 (1.04%)	97.5
PSO-Sphere	4246 (2.05%)	12058 (5.81%)	26 (13.5%)	90.5
PSO-Quadric	4441 (2.14%)	12578 (6.07%)	18 (9.38%)	91.8
PSO-Rosenbrock	3377 (1.63%)	8423 (4.06%)	10 (5.21%)	94.7
PSO-Rastrigin	8474 (4.09%)	27067 (13.1%)	42 (21.9%)	90.6

As showed in Table II, all the proposed architectures are effectively implemented in hardware. At the worst case (PSO optimizing the Rastrigin function problem) there are available more than 70% of the FPGA resources for future implementations. The most critical parameter is the embedded multipliers consumption, for which the Rastrigin problem requires about 22%. However, these multipliers can be implemented using the logic blocs available resources in the FPGA. The frequency of the implemented cores is about 90 MHz.

In this architecture the dimensionality of the optimization problem has a minor effect in the area cost than the number of particles in the swarm. Notice that a new particle will require not only to implement the operations presented in Fig. 3 but also a new fitness function implementation and several connecting signals in the general architecture (Fig. 2). In contrast, a new dimension will require only several connecting signals in the particle architecture and in the general architecture; however, the latency of one iteration of the PSO algorithm is increased.

B. Simulation Results

The proposed hardware parallel PSO, in double precision, was simulated using the ModelSim® simulator tool and the results were contrasted with a Matlab® implementation using the same conditions listed in Table I. The double precision format (64 bit-width) was chosen to simulate the hardware parallel PSO, given that Matlab® uses the double precision format to perform the computations.

For each one of the benchmark functions, the hardware PSO was simulated over 10 runs. The same initial position of each single run was used in both hardware and software simulations. Figs. 5, 6, 7 and 8 show the evolution of the best fitness value against the number of iterations of the hardware PSO algorithm. The black line represents the average of the best fitness value. Figs. 9, 10, 11 and 12 present a comparison between the average of the best fitness values for hardware and software simulations.

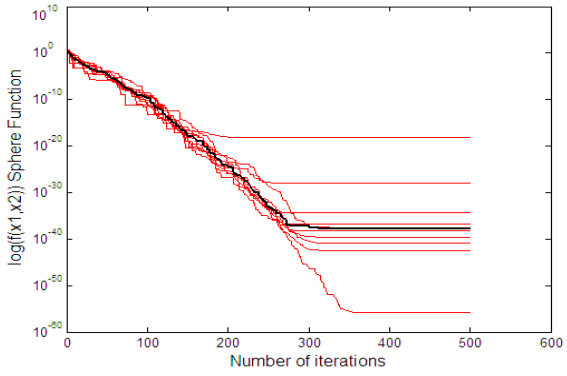

Figure 5. Best fitness value hardware PSO - Sphere function

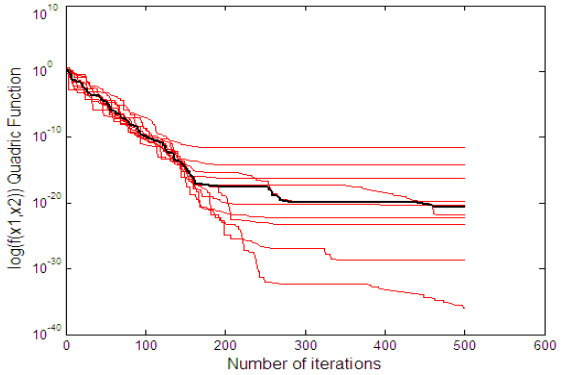

Figure 6. Best fitness value hardware PSO - Quadric function

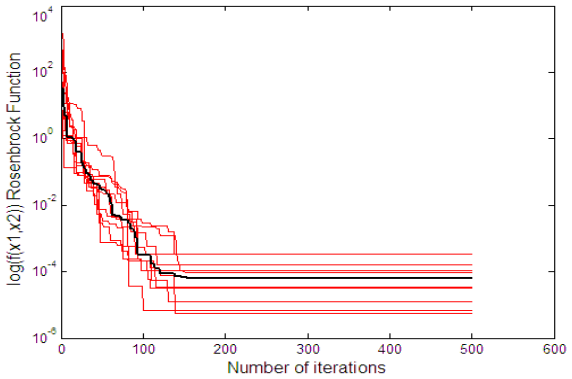

Figure 7. Best fitness value hardware PSO - Rosenbrock function

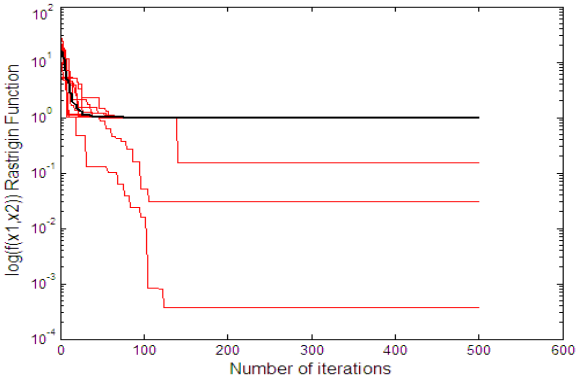

Figure 8. Best fitness value hardware PSO - Rastrigin function

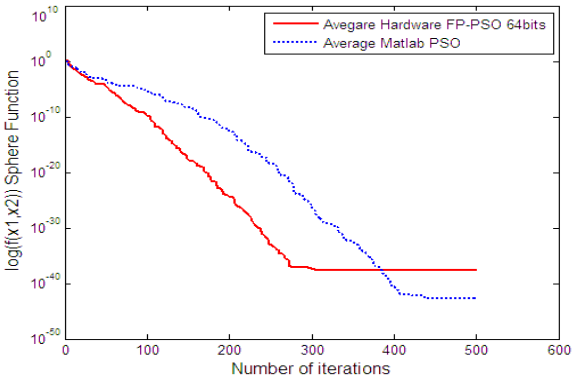

Figure 9. Hardware-software comparison - Sphere function

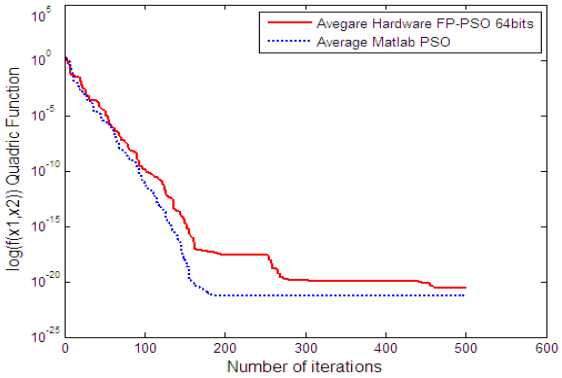

Figure 10. Hardware-software comparison - Quadric function

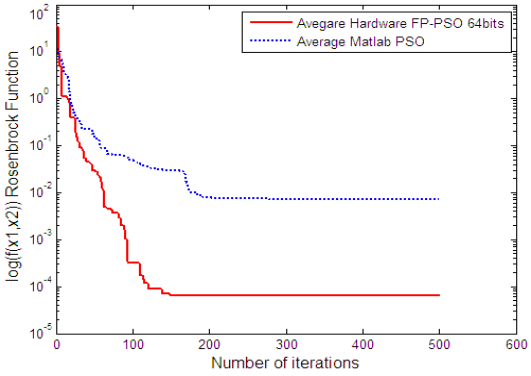

Figure 11. Hardware-software comparison - Rosenbrock function

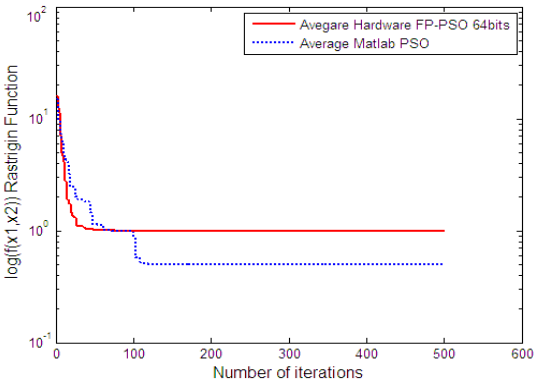

Figure 12. Hardware-software comparison - Rastrigin function

In order to validate the hardware implementations, both the hardware and software implementations were simulated without the random component of the algorithm. In this case, the best fitness function results were identical, demonstrating the correctness of the proposes architecture. Afterwards, the random component was included for the respective simulations.

As expected, for both the hardware and software implementations, the PSO algorithm optimizing unimodal problems achieves better results than optimizing multimodal problems. The hardware parallel PSO presents satisfactory results, about 1E-38 of the best fitness value, achieving this result faster than the software implementation (see Fig. 9). In the case of the Quadric function (see Fig. 10), the fitness function evolution is similar between the hardware and the software implementations, achieving the best results about 1E-21 and 1E-22 respectively; however the software implementation requires small iterations for achieving this results.

In the case of multimodal functions, the hardware parallel PSO achieves better and faster results (1E-4, 140 iterations) than the software implementation (1E-2, 200 iterations) when solving the Rosenbrock function (see Fig. 11). However, in the case of the Rastrigin function, both the hardware and software approaches present sub-optimal solutions.

Table III presents a comparison of the elapsed time of one PSO iteration (micro-seconds), between the proposed hardware parallel PSO and the software implementation. It can be observed that the elapsed time of the FPGA implementation, working at 50MHz, is about 78 times faster than the Matlab® implementation, working on a Intel Core Duo at 1.6 GHz and 1GB of RAM memory.

TABLE III. ELAPSED TIME COMPARISON - ONE ITERATION.

Case problem	Hardware	Software	Times faster
PSO-sphere	0.94 us	120.20 us	127
PSO-quadric	0.98 us	127.03 us	129
PSO-rosenbrock	1.10 us	105.74 us	96
PSO-rastrigin	1.64 us	128.40 us	78

VI. CONCLUSIONS

This paper describes an FPGA implementation of a hardware parallel PSO algorithm using the efficient floating-point arithmetic. The proposed architecture explores the parallel capabilities of the PSO by updating the particles and evaluating the fitness functions in a parallel approach. Synthesis results show that the scalability of the hardware parallel PSO depends on the complexity of the fitness functions.

The proposed PSO architecture was validated using four parallel particles optimizing two-dimensional benchmark functions, demonstrating the effectiveness of the hardware parallel PSO. The better results were achieved in case of the unimodal Sphere function and the multimodal Rosenbrock function, requiring less iterations than the software implementation. Also, simulation results demonstrate than the hardware implementation is about 78 times faster than the Matlab® implementation.

As future works we intend to implement a hardware adaptive architecture of the parallel PSO, as well as, to perform a scalability study of the FPGA implementation of the parallel PSO algorithm using floating-point arithmetic.

REFERENCES

[1] A. Naka, and Y. Fukuyama, “Practical distribution state estimation using hybrid particle swarm optimization,” IEEE Trans. Power Eng. Soc. Vol. 2, pp. 815-820, 2001.

[2] F. van den Bergh and A. Engelbrech, “A cooperative approach to particle swarm optimization,” IEEE Trans. Evolutionary Computation, Vol. 8, No. 3, pp. 225-239, 2004.

[3] G. Venter and J. Sobieszczanski, “A parallel PSO algorithm accelerated by asynchronous evaluations,” Proc. Structural and Multidisciplinary Optimization Conference, Jun. 2005, pp. 123-137.

[4] C. Lin and H. Tsai, “FPGA implementation of a wavelet neural network with particle swarm,” Trans. Math and Computer Modeling, Vol. 47, pp. 982-996, 2007.

[5] J. Schuttle, J. Reinblot, B. Fregly, R. Haftka and A. George, “Parallel global optimization with the particle swarm algorithm,” Int. J. Numer. Methods Eng. Vol. 6, No. 13, pp. 2296-2315, 2004.

[6] B. Koh, A. George, R. Haftka and B. Fregly, “Parallel asynchronous particle swarm optimization,” Int. J. Numer. Methods Eng. Vol. 67, pp. 578-295, 2006.

[7] D. Sanchez, D. Muñoz, C. Llanos, M. Ayala-Rincón,. "Parameterizable floating-point library for arithmetic operations in FPGAs", Proc. ACM Symp. Integrated Circuits and Systems Design, Aug. 2009, pp. 40-46.

[8] D. Muñoz, D. Sanchez, C. Llanos, M. Ayala-Rincón,. "Tradeoff of FPGA design of floating-point transcendental functions", in Press, Proc. IEEE Conf. Very Large Scale Integration, Brazil, Oct. 2009.

[9] J. Kennedy and R. Eberhart, “Particle swarm optimization,” Proc. Conf. Neural Networks, vol. 4, 1995, pp. 1942-1948.

[10] F. van den Bergh, "An analysis of particle swarm optimizers," PhD Thesis, Department of Computer Science, University of Pretoria, South Africa, 2002.

[11] K. Parsopoulos, D. Tasoulis, M. Vrahatis, "Multiobjective optimization using parallel vector evaluated particle swarm optimization," Proc. IASTED Conf. Artificial Intelligence and Applications, Feb. 2004, pp. 823-828.

[12] P. Reynolds, R. Duren, M. Trumbo, and R. Marks, "FPGA implementation of particle swarm optimization for inversion of large neural networks," Proc. IEEE Swarm Intelligence Symp., Jun. 2005, pp. 389-392.

[13] G. Kókai, T. Christ and H. Frhauf, “Using hardware-based particle swarm method for dynamic optimization of adaptive array antennas,” Proc. NASA/ESA Conf. on Adaptive Hardware and Systems, Jun. 2006, pp. 51-58.

[14] J. Peña and A. Upegui, "A population-oriented architecture for particle swarms," Proc. NASA/ESA Conf. Adaptive Hardware and System, Aug. 2007, pp. 563-571.

[15] Y. Maeda and N. Matsushita, "Simultaneous perturbation particle swarm optimization using FPGA," Proc. IEEE Conf. Neural Networks, Aug. 2007, pp. 2695-2700.