Particle Swarm Optimization: A Hardware Implementation

P. Palangpour[1], G.K. Venayagamoorthy[1], and S.C. Smith[2]
[1] Real-Time Power and Intelligent Systems Laboratory, Missouri University of Science and Technology, Rolla, USA
[2] Asynchronous Digital Design Laboratory, University of Arkansas, Fayetteville, USA

Abstract - *Particle Swarm Optimization (PSO) is a popular population-based optimization algorithm. While PSO has been shown to perform well in a large variety of problems, PSO is typically implemented in software. Population-based optimization algorithms such as PSO are well suited for execution in parallel stages. This allows PSO to be implemented directly in hardware and achieve much faster execution times than possible in software. In this paper, a pipelined architecture for hardware PSO implementation is presented. Benchmark functions solved by software and hardware PSO implementations are compared. The hardware PSO design is implemented on a Xilinx Virtex-II Pro Development Kit for evaluation. By implementing PSO directly on hardware an execution speedup of several orders of magnitude is observed.*

Keywords: particle swarm optimization, PSO, FPGA, hardware.

1 Introduction

Adaptive systems have become a large area of interest since many systems operate in changing, unpredictable environments. Evolutionary Algorithms (EAs) are well suited for adapting the behavior of many adaptive systems because of their simplicity; EAs only require a fitness function to provide a measure of the system behavior. Many different variations of EAs for adapting system behavior have been extensively explored. In principle, all EAs are population-based optimization algorithms. The population consists of candidate solutions to the problem being studied; during each iteration of the algorithm a series of operators are applied to each member in the population. After the operators have processed the population, the candidate solutions are evaluated and given a level of 'fitness' that represents their degree of performance for the problem being studied. Each of the operators is based on evolution and plays a role in combining and randomly modifying portions of the population. As the fitness of candidate solutions play a role in which solutions are selected to combine and modify, the population as a whole improves over time. Particle Swarm Optimization (PSO) is another population-based algorithm that begins with a population of potential solutions and continually evolves the solutions until they reach a desired level of fitness. While EAs and PSO are similar, PSO is simpler, requiring fewer operations. This is important for real-time applications where minimizing execution time is critical.

PSO is a swarm intelligence based optimization algorithm that has been shown to perform very well for a large number of applications. While PSO has been applied in a large number of applications, PSO is typically executed in software. While it takes less time to develop a software implementation of PSO, a hardware implementation is able to achieve much faster execution speed. Recently, there has been interest in using PSO for real-time applications [1, 2]. However, in order to meet the time constraints of some real-time applications, PSO must be executed directly in hardware.

Population-based optimization algorithms are relatively easy to partition into stages that execute in parallel. This makes population-based algorithms an excellent candidate for implementation directly in hardware or in a distributed fashion. Field Programmable Gate Arrays (FPGAs) are reconfigurable devices that are programmed to implement digital circuits. FPGAs are a popular platform for implementing soft computing methods due to their inherent flexibility as well as the advantage of relatively high-performance at low-cost.

Several hardware implementations of PSO have been reported in the literature. Reynolds *et al* implemented a hardware version of PSO for inverting a very large neural network [3]. One Xilinx XC2V6000 FPGA was used to execute the PSO algorithm while another was used for computing the fitness. The details of the hardware PSO architecture are not reported. A multi-swarm PSO architecture for the blind adaption of array antennas was proposed by Kokai *et al* [4]. Each swarm optimizes a single architecture and executes in parallel with the other swarms. The authors have not described the hardware architecture in detail or provided any performance measurements.

Farmahini-Farahani *et al* have implemented PSO within a System-on-a-Programmable Chip (SoPC) framework [5]. The authors utilized a hardware implementation of the discrete version of PSO, implemented on a Altera Stratix 1S10ES Development Kit, using a soft-core Altera NIOS II embedded

processor to compute the fitness function. Performance was sacrificed in exchange for flexibility by implementing the fitness function in software.

In this paper we propose a compact, pipelined architecture for implementing PSO on a FPGA. Two benchmark problems are formulated in hardware and implemented on the FPGA using the PSO core. Both the PSO algorithm and fitness function are implemented in hardware to yield maximum performance from the FPGA platform. The processing time for the hardware PSO is compared to the processing time of software PSO executed on a workstation.

The remainder of this paper is organized as follows: An overview of the PSO algorithm is given in Section 2. The proposed hardware PSO architecture is described in Section 3. The benchmark functions used and the experimental results are discussed in Section 4. The conclusions and future work are provided in Section 5.

2 Particle Swarm Optimization

The PSO algorithm was developed by Kennedy and Eberhart and is based on the social behavior of bird flocking [6]. Each particle in the population has a position vector that represents a potential solution to the problem. The particles are initialized to random positions throughout the search space, and during each iteration of the algorithm, a velocity vector is computed and used to update each particle's position. Each particle's velocity is influenced by the particle's own experience as well as the experience of its neighbors.

There are two basic neighborhood variants of PSO, local and global. In this study the more common global version of the PSO algorithm is applied. The population consists of N particles. For each iteration, a cost function, f, is used to measure the fitness of each particle, i, in the population. The position of each particle, i, is then updated, which is influenced by three terms, the particle's velocity from the last iteration, the difference between the particle's known best position and the particle's current position, and the difference between the swarm's best known position and the particle's current position. The latter two terms are each multiplied by a uniform random number in [0,1] to randomly vary the influence of each term, as well as an acceleration coefficient to scale and balance the influence of each term. The best position each particle attained is stored in the vector, p_i, known as *pbest*, while the best position attained by any particle in the population is stored in the vector, p_g, known as *gbest*. The velocity vector, v_i, for each particle is then updated:

$$v_i^{t+1} = w \cdot v_i^t + c_1 r_1 \cdot (p_i^t - x_i^t) + c_2 r_2 \cdot (p_g^t - x_i^t) \quad (1)$$

where w, c_1, and c_2 are positive and r_1 and r_2 are uniformly distributed random numbers in [0,1]. The inertia coefficient, w, is used to keep the particles moving in the same direction they have been traveling. The value for w is typically in [0, 1]. The term, c_1, is called the cognitive acceleration term and c_2 is called the social acceleration term. These two values balance the influence between the particle's own best performance and that of the population.

The velocity is constrained between the parameters V_{min} and V_{max} to limit the maximum change in position:

$$v_i^{t+1} = \begin{cases} v_{Max} & \text{if } v_i^{t+1} > V_{max} \\ v_{Min} & \text{if } v_i^{t+1} < V_{min} \\ v_i^{t+1} & \text{else} \end{cases} \quad (2)$$

The position of each particle is then updated using the new velocities:

$$x_i^{t+1} = x_i^t + v_i^{t+1} \quad (3)$$

The position in each dimension is limited between the parameters X_{min} and X_{max}:

$$x_i^{t+1} = \begin{cases} x_{Max} & \text{if } x_i^{t+1} > X_{max} \\ x_{Min} & \text{if } x_i^{t+1} < X_{min} \\ x_i^{t+1} & \text{else} \end{cases} \quad (4)$$

In addition to the original continuous version of PSO, a binary version of PSO, Binary PSO (BPSO) has been developed [7]. In BPSO all values are encoded as binary strings and the velocity is redefined as the probability that a given bit will change. BPSO was developed for optimizing discrete problems and is not necessarily more efficient for hardware implementation as the velocity and position updates require the computation of a sigmoid and exponential function. Although BPSO could still be implemented in hardware, BPSO does not perform well on the continuous-valued problems used in this study [8]. For these reasons, the original continuous-valued version of PSO is selected for hardware implementation.

3 Hardware Implementation

Software implementations of PSO often use floating-point values. However, floating-point operations typically require several times the number of logic resources and processing time compared to a similar fixed-point operation. In addition, it is common for FPGAs to include a number of embedded multipliers that can be used to perform fixed-point

multiplications without using any of the FPGA's programmable logic. For these reasons, the hardware PSO implementation uses fixed-point representation for all values.

For hardware implementation, the PSO algorithm is decomposed into five operations that are performed on each particle: evaluate the fitness, update the particle's best position, update the global best position, update the velocity, and update the position. Each of these five operations are implemented in a separate hardware module. It should be noted that the constraints in (2) and (4) are not directly implemented; the results of the fixed-point arithmetic operations for (1) and (3) are set to saturate which indirectly constrains the values based on their fixed-point width. Since updating the *pbest* and *gbest* can be performed in parallel, the five operations can be organized in a 6-stage pipeline, including the initial fetch and final write stages. The hardware modules are shown in Fig. 1.

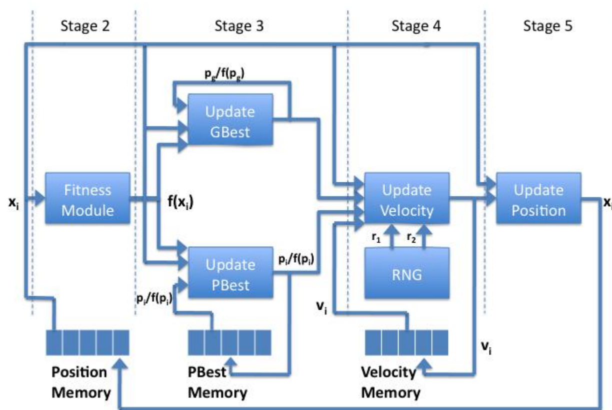

Figure 1. The proposed hardware PSO architecture.

The flow for an execution of a single particle is described as follows. In the first stage, the position for particle i, x_i is fetched from memory. Then in the second stage, the fitness module computes the fitness, $f(x_i)$, based on the position of particle i. The current *pbest* values, p_i and $f(p_i)$, are also fetched from *pbest* memory. In the third stage, *gbest* and *pbest* are updated. x_i and $f(x_i)$ are sent to both the update *gbest* and *pbest* modules. In addition, p_i and $f(p_i)$ are provided to the update *pbest* module, while p_g and $f(p_g)$ are sent to the update *gbest* module. Each module selects the lowest fitness and associated positions for their output, which are the new *gbest* and *pbest*. In addition, the old velocity, v_i, is fetched from velocity memory. Now in the fourth stage of the pipeline, the new p_i and $f(p_i)$ are stored in *pbest* memory. The update velocity module uses v_i, x_i, p_i, p_g, r_1, and r_2 to compute the new velocity, v_i. In the fifth stage, the update position module uses x_i and v_i to compute the new position, x_i. The new velocity, v_i, is stored in velocity memory. In the final stage, the new position, x_i, is stored in position memory.

3.1 Hardware Velocity Update

In PSO, the velocity update equation involves the largest number of arithmetic operations. As shown in (1), there are five multiplications, two additions, and two subtractions. In hardware, multiplications require a large amount of logic and processing time, and therefore are to be avoided if possible. Since the inertia, w, is typically set to 0.8, the first term of the velocity update can be simplified in two different ways. The first is to replace the term with an arithmetic shift to the right of v_i. This effectively changes the inertia to 0.5 and eliminates the multiplication. An alternative is to remove the inertia entirely and substitute v_i for the first term.

The cognitive and social acceleration coefficients, c_1 and c_2, are typically set to 2.0. Performing arithmetic left shifts on r_1 and r_2 would effectively multiply each value by 2. However, since the values for c_1 r_1 and c_2 r_2 are just uniform random numbers in [0, 2], the fixed-point pseudo random numbers can just be extended to fulfill the range by incorporating another bit from the PRNG.

3.2 Random Number Generation

Two random numbers are needed for each velocity update. This means $2Nm$ random numbers are required for a complete PSO execution of N particles and m iterations. While PSO is still able to find solutions in the absence of the random influence, it isn't guaranteed PSO will converge as fast or with as high quality of solutions [9]. Pseudo random numbers are generated in hardware using Pseudo Random Number Generators (PRNGs). Typically, linear feedback shift registers and cellular automata based PRNGs are used due to their simplicity. In this work, a neighborhood-of-four cellular automata random number generator is used [10]. This PRNG produces a pseudorandom 64-bit string each cycle.

3.3 Memory Requirements

Each particle is required to store a position, velocity, best position, and best position fitness. In addition, a global best position and global best position fitness are stored. The values that need to be stored are categorized into two types, positions/velocities and fitness values. The bit-width of the fixed-point position values is b_p, while the bit-width of the fitness values is b_f. Therefore, each particle is required to store $3\ (b_p + b_f)$ bits, while the *gbest* values require $(b_p + b_f)$ bits. The total number of memory storage required is $(3N+1)\ (b_p + b_f)$ bits.

3.4 Control Module

The control module is used to initialize the memory and generate the control signals for the modules. Upon reset, the

control module enters an *Init* state that is used to initialize the counters to their respective starting states. The control module then enters an *Init-PRNG* state to initialize the PRNG. The *Init-Particles* state cycles through the particles and sets each particle's initial position, velocity, and *pbest* position to a random value from the PRNG. The *gbest* position is initialized in the same manner. The next state, *Init-Pipeline*, is used to prepare the module inputs. The *Execute* state is responsible for shifting each particle through the pipeline and properly passing the modules the correct particle information from memory while storing each particle's new values as they are updated. Upon either reaching a defined fitness or number of iterations, the *Finished* state is entered and execution halts.

4 Results

The hardware PSO design is developed in VHDL and implemented on the Xilinx Virtex-II Pro Development Kit (Xilinx XC2VP30). In order to assess the performance of the hardware PSO implementation, the hardware implementation is compared to a software implementation developed in Matlab. The Matlab implementation is executed on a system with a 2.13 GHz Intel Core 2 Duo processor. First, the performance of the two implementations is compared with respect to the lowest fitness that the implementation is able to achieve for the benchmark problems. Then the execution speed of the two implementations is compared. Finally, the FPGA resource requirements for the hardware implementation are discussed.

4.1 The Benchmark Problems

Two well-known benchmark optimization problems have been selected for comparing the two implementations. Both functions are multidimensional, where n represents the number of dimensions. The first benchmark problem is the sphere function (5).

$$f(x) = \sum_{i=1}^{n} x_i^2 \qquad (5)$$

The second benchmark problem is the Rosenbrock function (6).

$$f(x) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2 \qquad (6)$$

The sphere function is a simple unimodal function that is typically used to test local optimizers. The Rosenbrock function is multimodal for n of 4 and higher and is more difficult to optimize. These functions are often used to assess the performance of EAs. The global minimum for the sphere function is clearly located at $x_i = 0$, for all i. The global minimum for the Rosenbrock function is located at $x_i = 1$, for all i. The global minimum for each function will produce a fitness value of 0. In this sense, the best position is the position that produces the lowest fitness value.

4.2 Simulation Results

For the benchmark functions, the values for X_{min} and V_{min} are set to -128, while X_{max} and V_{max} are set to 127. While the values are typically constrained in [-100, 100], these values are selected for their ease in hardware representation. The values for c_1 and c_2 are selected to be 2.0. The inertia w is selected to be 0.5. The number of particles is 20. While the hardware and software PSO implementations use the same parameters, each utilizes a different random number generator; as a result, each execution of PSO is unique. PSO is used to solve each benchmark function, for dimensions of 1, 5, and 10. In each case, PSO executes for 1000 iterations before terminating.

The achieved fitness for the hardware and software implementations of PSO on the benchmark problems is listed in Table I. Both the software and hardware implementations result in very similar fitness values after 1000 iterations. A comparison of the hardware and software PSO fitness values for the sphere function of 10 dimensions is shown in Fig. 2. Similarly, a comparison of the hardware and software PSO fitness values for the Rosenbrock function of 10 dimensions is shown in Fig. 3.

TABLE I. MINIMUM ACHIEVED FITNESS FOR SOFTWARE AND HARDWARE PSO

Function	***n***	**Fitness Achieved**	
		Software PSO	***Hardware PSO***
Sphere	1	0.000	0.000
	5	0.000	0.000
	10	0.014	0.001
Rosenbrock	1	0.000	0.000
	5	0.044	0.007
	10	9.679	8.611

The execution time for the software and hardware PSO implementations is listed in Table II. The hardware PSO implementation is clearly several orders of magnitude faster than the Matlab PSO implementation. Even for the worst-case improvement, the single-dimension Rosenbrock function, the hardware implementation is 6,220 times faster. The best-case improvement is for the 10-dimension sphere function, which achieves a speedup of 28,935X.

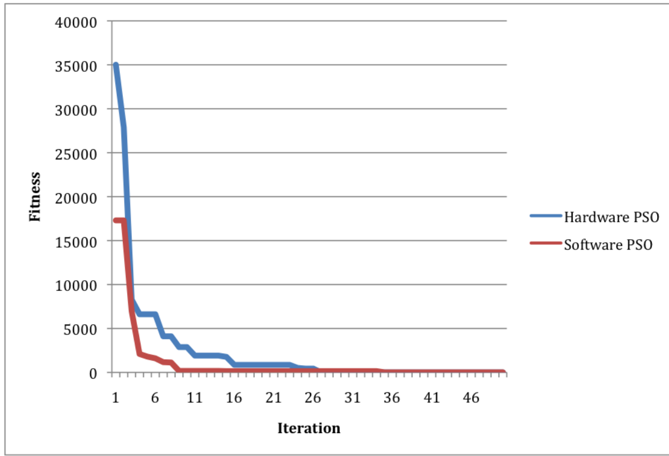

Figure 2. The *gbest* fitness for the sphere function of 10 dimensions.

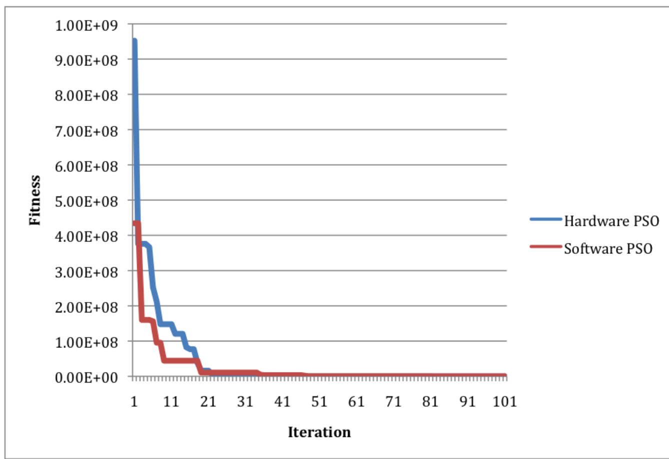

Figure 3. The *gbest* fitness for the Rosenbrock function of 10 dimensions.

TABLE II. SOFTWARE AND HARDWARE PSO EXECUTION TIME

Function	*n*	Execution Time		Hardware Speedup
		Software PSO	*Hardware PSO*	
Sphere	1	2.07 sec.	200 µs	10350
	5	5.79 sec.	338 µs	17130
	10	10.99 sec	392 µs	28935
Rosenbrock	1	2.14 sec	344 µs	6220
	5	5.95 sec	444 µs	13400
	10	10.91 sec	800 µs	13637

The Virtex-II Pro FPGA resources required for the benchmark problems are listed in Table III. The required number of slice flip-flops increases linearly with the number of dimensions. In addition, the number of 4-input Lookup Tables (LUTs) required scales similarly. The maximum clock speed is decreased as the benchmark function complexity increases.

TABLE III. FPGA RESOURCES FOR HARDWARE PSO

Function	*n*	FPGA Resources		
		Slice Flip-Flops	*LUTs*	*Clock (MHz)*
Sphere	1	1118	1523	100
	5	4744	10631	59
	10	9249	20873	51
Rosenbrock	1	2244	5332	58
	5	5228	12513	45
	10	9940	25750	25

5 Conclusion and Future Work

A pipelined hardware implementation of PSO has been presented. The hardware implementation is shown to perform well on two standard benchmark problems when compared to a common software implementation of PSO in Matlab. When compared to the software implementation, the hardware implementation is between 6,220 - 28,935 times faster. The system is targeted for real-time applications where minimizing PSO execution time is critical. One such application is real-time neural network training.

A DSP implementation of a neural network for power system harmonics prediction has shown promising results [11]. However, each additional harmonic order that must be predicted requires additional processing power for training. Utilizing hardware PSO for real-time training of neural networks for higher order harmonics prediction is a future research area.

6 References

[1] L. Liu, W. Liu, D. A. Cartes, and N. Zhang. Real time implementation of particle swarm optimiation based model parameter identification and an application example. In Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intel ligence). IEEE Congress on, pages 3480–3485, Hong Kong, June 2008

[2] K. H. S. Hla, Y. Choi, and J. S. Park. Applying particle swarm optimization to prioritizing test cases for embedded real time software retesting. In Computer and Information Technology Workshops, 2008. CIT Workshops 2008. IEEE 8th International Conference on, pages 527–532, Sydney, QLD, July 2008.

[3] P. D. Reynolds, R. W. Duren, M. L. Trumbo, and I. Marks, R. J. FPGA implementation of particle swarm optimization for inversion of large neural networks. In Swarm Intel ligence Symposium, 2005. SIS 2005. Proceedings 2005 IEEE, pages 389–392, June 2005.

[4] G. Kokai, T. Christ, and H. H. Frhauf. Using hardware-based particle swarm method for dynamic optimization of adaptive array antennas. In Adaptive Hardware and Systems, 2006. AHS 2006. First NASA/ESA Conference on, pages 51–58, Istanbul, June 2006.

[5] A. Farmahini-Farahani, S. M. Fakhraie, and S. Safari. SOPC-based architecture for discrete particle swarm optimization. In Electronics, Circuits and Systems, 2007. ICECS 2007. 14th IEEE International Conference on, pages 1003–1006, Marrakech, Dec. 2007.

[6] R. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In Proceedings of the Sixth International Symposium on Micro Machine and Human Science, volume 1, pages 39–43, Nagoya, Japan, Oct. 1995.

[7] J. Kennedy and R. C. Eberhart. A discrete binary version of the particle swarm algorithm. In Systems, Man, and Cybernetics, 1997. 'Computational Cybernetics and Simulation'., 1997 IEEE International Conference on, volume 5, pages 4104–4108, Orlando, FL, October 1997.

[8] M. A. Khanesar, M. Teshnehlab, and M. A. Shoorehdeli. A novel binary particle swarm optimization. In Control & Automation, 2007. MED '07. Mediterranean Conference on, pages 1–6, Athens, June 2007.

[9] M. Rodgers, "Random numbers and their effect on particle swarm optimization," 2006. [Online]. Available: http://ncra.ucd.ie/COMP30290/crc2006/rodgers.pdf [Accessed: Jan. 21, 2009].

[10] B. Shackleford, M. Tanaka, R. J. Carter, and G. Snider. FPGA implementation of neighborhood-of-four cellular automata random number generators. In FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays, pages 106–112, New York, NY, USA, 2002. ACM.

[11] G.K. Venayagamoorthy J. Dai and R.G. Harley. Harmonic identification using an echo state network for adaptive control of an active filter in an electric ship. Accepted to Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on, June 2008.