View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Cronfa at Swansea University

Parallel Implementation of Particle Swarm Optimization on FPGA

Alexandre L. X. Da Costa, Caroline A. D. Silva, Matheus F. Torquato and Marcelo A. C. Fernandes

***Abstract*—This work proposes a parallel implementation, with fixed point, of the Particle Swarm Optimization (PSO) algorithm on Field-Programmable Gate Array (FPGA). Results associated with the processing time and area occupancy on FPGA for several numbers of particles and dimensions were analyzed. Studies concerning the accuracy of the PSO response for the optimization problem using the Rastrigin function were also analyzed for the hardware implementation. The project was developed on the Virtex-6 xc6vcx240t 1ff1156 FPGA.**

***Index Terms*—Particle Swarm Optimization, Reconfigurable Hardware, FPGA, Parallel Computing.**

I. Introduction

PROBLEM solving using metaheuristics has been the object of study in literature and industry research. One of the most important aspects associated with metaheuristic algorithms is their capacity of providing a good approximate solution (sometimes the best solution) to a complex problem within a reasonable time [1], [2]. This characteristic ensured the widespread use of metaheuristics for solving real-time problems [3], [4]. The Particle Swarm Optimization (PSO) is a nature-inspired metaheuristic related to both Swarm Intelligence and Evolutionary Computation. It requires only primitive mathematical operators and is computationally inexpensive regarding both memory requirements and speed [5]. Although initially designed for solving nonlinear continuous functions, PSO is currently used to solve several other types of problems.

Parallel to the demands associated with metaheuristics, there are also the demands associated with processing large volumes of data, creating the novel lines of knowledge such as Big Data and Mining of Massive Datasets. This new demand shows that even simple algorithms may require a high computational effort when the volume of data grows exponentially [6]. One way to minimize this problem is the parallelization of algorithms. Parallelism comes naturally not only to reduce the search time but also to improve the quality of the solutions provided by [7]. The performance improvement provided by parallelization techniques can be intensified with an implementation of high-performance hardware platforms such as High-Performance Processors, Application-specific Integrated Circuits (ASIC) and Field-programmable Gate Arrays (FPGA). Among these platforms, FPGAs have been noteworthy as an alternative that combines high processing, low consumption and low cost when compared to other ones. FPGAs presents good rates when analyzing the power consumption and the silicon area efficiency [8]. In addition, as stated in [9], [10].

Thus, this paper aims to present a parallel implementation of the PSO algorithm in FPGA. It is important to note that the architecture proposed in this work seeks to optimize the performance of the PSO by reducing its processing time, making possible its use in real-time systems with large volume of data and severe processing time restrictions. All the details associated to the implementation, as well as the performance analyzes of the proposed system for different dimensions and number of particles are presented in the following sections.

Alexandre L. X. Da Costa, Caroline A. D. Silva and, Marcelo A. C. Fernandes are with Department of Computer Engineering and Automation, Federal University of Rio Grande do Norte, Natal, Brazil, e-mail: alexluz321@gmail.com, carolads@gmail.com, mfernandes@dca.ufrn.br. Matheus F. Torquato is with College of Engineering, Swansea University, Swansea, Wales, UK, e-mail: m.f.torquato@swansea.ac.uk

II. Particle Swarm Optimization

Particle Swarm Optimization is a stochastic population-based metaheuristic algorithm that mimics the social behavior of natural creatures like bird flocking and fish schooling. In these swarms, the coordinated behavior using local movements emerges without central control. A swarm consists of a population of N particles flying through a d-dimensional search space. Each particle is a candidate solution to the optimization problem.

The PSO operation starts by generating an initial population of particles, and iteratively moving these particles over the search space, so the swarm reaches the optimal (or quasi-optimal) solution or any other stop criterion is met. Each particle's movement is guided toward its best position found (local best position) and toward the best position ever found by the swarm (global best position), which are updated as better positions are found. The success of a particle, i.e. the particle being in a better position than the other ones is measured by the fitness function, which often corresponds to the objective function of the addressed problem. Simple mathematical equations over the particle's position $\mathbf{p}_j(k+1) = \mathbf{p}_j(k) + \mathbf{v}_j(k+1)$, and particle's velocity $\mathbf{v}_j(k+1) = w\mathbf{v}_j(k) + c_1 r_1(\mathbf{pBest}_j - \mathbf{p}_j(k)) + c_2 r_2(\mathbf{gBest} - \mathbf{p}_j(k))$ are required to perform each particle's movement. Where $\mathbf{p}_j(k) = [p_{j1}(k), \ldots, p_{j,d}]$ and $\mathbf{v}_j(k) = [v_{j1}(k), \ldots, v_{j,d}]$ represent the position and velocity of the j-th particle, respectively; k indicates the current iteration; $\mathbf{pBest}_j = [p\mathrm{Best}_{j1}, \ldots, p\mathrm{Best}_{jd}]$ and $\mathbf{gBest} = [g\mathrm{Best}_1, \ldots, g\mathrm{Best}_d]$ represents the local and global best positions; r_1 and r_2 are random values in the range $[0, 1]$; and w, c_1 and c_2 are configurable parameters of the algorithm. Let N be the number of particles in the swarm $\mathbf{P}(k) = [\mathbf{p}_1(k), \ldots, \mathbf{p}_N(k)]$, each one having a length of m bits, a fitness value F_j, a j-th position $\mathbf{p}_j(k)$ in the search

space. The basic operation of PSO presented in the Algorithm 1.

Algorithm 1: Particle Swarm Optimization.

```
1  Initialize P[m](1) with random-valued particles;
2  for k ← 1 to K do
3      g̃BestF[m] ← Big-number;
4      for j ← 1 to N do
5          F_j(k)[m] ← FitnessEval(p_j[m](k));
6          if (F_j[m](k) < pBestF_j[m]) then
7              pBestF_j[m] ← F_j[m](k);
8              for i ← 1 to d do
9                  pBest_ji[m] ← p_ji[m](k);
10             end
11         end
12         if (F_j[m](k) < g̃BestF[m]) then
13             g̃BestF[m] ← F_j[m](k);
14             for i ← 1 to d do
15                 g̃Best_i[m] ← p_ji[m](k);
16             end
17         end
18     end
19     if (g̃BestF[m] < gBestF[m]) then
20         gBestF[m] ← g̃BestF[m];
21         for i ← 1 to d do
22             gBest_i[m] ← g̃Best_i[m];
23         end
24     end
25     if (stop criterion is met) then return(gBest[m]);
26     for j ← 1 to N do
27         for i ← 1 to d do
28             v_ji[m](k+1) ← w v_ji[m](k)
                   +c_1 r_1(pBest_ji[m] − p_ji[m](k))
                   +c_2 r_2(gBest_i[m] − p_ji[m](k));
29             p_ji[m](k+1) ← p_ji[m](k) + v_ji[m](k+1);
30         end
31     end
32 end
```

III. Parallel Metaheuristics

Although the use of metaheuristic algorithms significantly allows reducing the processing time of a complex problem, for nontrivial problems, executing the reproductive cycle of a simple population-based method on long particles and/or large populations usually requires high computational resources. In general, evaluating a fitness function for every particle is frequently the most costly operation of this algorithm [7], [11].

In this context, parallelization techniques can be used for an even greater processing time reduction. Among the several techniques of parallelization, the data decomposition arises intuitively for population-based metaheuristics. The most common data decomposition strategies for this class of metaheuristics are the parallelization of the fitness computation; and the concurrent execution of metaheuristics over multiple subpopulations. Indeed, the performance of population-based algorithms is often improved when running in parallel [7].

In the first case, the parallelization strategy does not modify the characteristics of convergence of the algorithm, since only the computation of fitness values of the particles is performed concurrently. In the second strategy, the population is split into different parts where several processes concurrently execute iterations of the corresponding metaheuristics, including the fitness evaluation computation, on different subpopulations. According to the modeling, every process either can exploit distinct search subspaces, or can use the entire search space. Although the first alternative may be more interesting from an optimization point of view, the second one is more often used for simpler and more comprehensive modeling. Thus, the convergence behavior could be different in sequential and parallel versions of the algorithms as the different subpopulations usually evolve concurrently and only exchange some information about their particles after completing some iterations [12]. In this work, a fine-grained concurrent execution of a Particle Swarm Optimization over a number of particles searching the entire search space is performed. This approach is particularly suitable for algorithms like PSO since the evaluation of the fitness function and the application of movement operators to the particles can be independently done.

IV. Parallel PSO Hardware Implementation

Figure 1 presents a general architecture of the hardware implementation of a parallel PSO algorithm. As discussed in Section III, the whole algorithm was developed using a parallel architecture in order to accelerate the processing speed, taking advantage of the resources available in hardware. The structure shown allows the visualization of four main blocks: the particles module (PM), the comparison module of **g**Best (CM_*g*Best), the register bank of **g**Best (RB_*g*Best) and, the register of the **g**Best fitness value (R_gBestF). In this implementation, a set of N particles of dimension d is optimized for K generations. The R_gBestF register is used to store the fitness value of the **g**Best coordinates, $g\text{BestF}[m]$. This value is updated with each generation, in case there is a better fitness value from PM, $\tilde{g}\text{BestF}[m]$ (Lines 19-24 of Algorithm 1). The modules previously mentioned and depicted in Figure 1 are going to be detailed in the following subsections.

A. Particles Module (PM)

The particles module is shown in Figure 2 and it is composed of N PM_P*j*, and $N-1$ CM_P*jv* submodules. The PM module input are the coordinates of $\mathbf{g}\text{Best}[m](k)$ with depth d. In Figure 2 the coordinates are shown by the bus size d, which is the input of the module and then it is distributed to all PM_P*j*. The set of N PM_P*j* implements the Lines 4-11 and Lines 26-31 of Algorithm 1, in parallel. Each j-th PM_P*j* is shown in details in Figure 3.

The CM_P*jv* implementation is shown in detail in Figure 4. They are used to compare which of the particles in the swarm has obtained the best fitness value, and thus pass the information on this particle forward. The CM_P*jv* compares the *j*-th with the *v*-th particle and pass to forward the best one. In each CM_P*jv*, $d+1$ two-input multiplexers are responsible

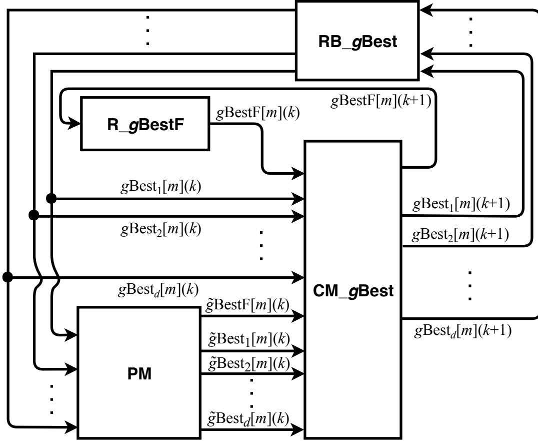

Fig. 1. Parallel PSO implementation.

for forwarding information about the particle with the lowest fitness value (see Figure 4). This includes the fitness value itself and the d components of the particle coordinates. The outputs of the multiplexers (CM_MUX0*jv* and CM_MUX1*jv* to CM_MUX*djv*) are selected by the CM_COMP*jv* comparator that selects the lowest fitness value. This notation will be adopted throughout the whole work. After the $N-1$ CM_P*jv* submodules processing (parallel implementation of the Lines 12-17 of the Algorithm 1) is generated the best particle of the k-th iteration, $\tilde{\text{g}}\text{Best}[m](k)$, and it is passed to the CM_*g*Best module.

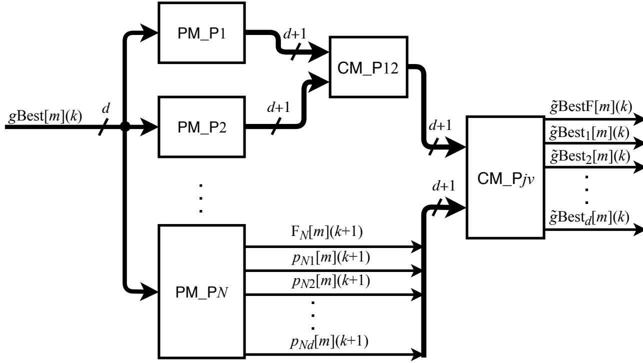

Fig. 2. PM implementation.

B. Comparison Module of g*Best (CM_*g*Best) and Register Bank of* g*Best (RB_*g*Best)*

This module is similar to the comparator module, CM_P*jv*, showed in Figure 4. Therefore, in this case the inputs are the $\tilde{g}$BestF and gBestF) plus d pairs of coordinates (a set of coordinates for each particle to be compared) where d is the particle dimension. The objective of this module, it is updated the gBest particle if necessary. This module implements the Lines 19-24, in parallel.

The RB_*g*Best module is designed entirely with registers only. Each one of the d registers holds one of the dimensions of the best particle position in the swarm, called g$Best$. The values stored in these registers, in turn, will be forwarded to the CM_gBest in order to be compared with the new position found in the swarm, and thus, to detect if g$Best$ has been exceeded or not.

V. Results and Discussion

In order to validate the PSO implementation proposal on FPGA, the algorithm was analyzed by optimizing the Rastrigin function [13], [14], defined by the expression $f(x) = Ad + \sum_{i=1}^{d}\left(x_i^2 - A\,cos(2\pi x_i)\right)$, where $A = 10$ is a constant and d is the number of dimensions of the function. This function is widely used to validate optimization techniques due to its high complexity which is a result of its numerous local minimums. The presented work used the Rastrigin function with $d = 3$, $d = 6$, and $d = 10$ dimensions. For each dimension, different implementations of the PSO were tested, whose swarms were composed of $n = 5$, $n = 10$ or $n = 15$ particles. In addition, the m size (in bits) of the particles had also undergone variations. The target FPGA was the Virtex 6 xc6vcx240t 1ff1156. This Virtex 6 FPGA has 301440 registers, 150720 logical cells (LUT) that can be used to implement logical functions or memories and 768 DSP cells with multipliers and accumulators.

In all tests carried out, the PSO operations were performed at a sampling rate (or throughput) $R_s = \frac{1}{T_s}$ (Sample per second - Sps), where T_s is the time for each k-th iteration. Thus, R_s also can represent iterations per second (Ips). How the design uses the full parallel technique, it spends one clock cycle per iteration, in other words, T_s is the length of the one clock cycle.

The Tables I, II and III present the synthesis results in the target FPGA for PSO implementations with swarms of $n = 5$, $n = 10$, and $n = 15$ particles, respectively. It was observed that, in general, the sampling rate R_s and the occupation of logic cells were very sensitive both to the increase in the number of particles in the swarm and to the increase in the number of bits of the particles that make up the particles. The columns RN, LN and MN show the number of registers, number of LUTs and number of multipliers already embedded in hardware, respectively. The columns RF show the occupation rate (in percent) of the fitness function (PM_FitnessEvalP*j* blocks) with regard to the total of logic cells occupation, columns LN. The estimate of the dynamic power consumption in watts is presented in the columns called of DP.

The logical cells (LUTs) occupation, LN columns, was crescent and approximately linear according to the increase of the number of particles, as for the increase of the size of the particles and as to the increase of the size of the optimization function. In the most critical implementations ($n = 15$ particles, $d = 10$ dimensions and $m = 20$ bits, and $n = 15$ particles, $d = 6$ dimensions and $m = 32$ bits), the LUTs occupancy rate did not even reach 30%. This fact is important for implementations that demand larger populations or particles carrying more information, that is, more bits to represent them. The most critical situation regarding occupation was observed in the use of the FPGA embedded multipliers (DSP blocks), column MN. It was used 79% of

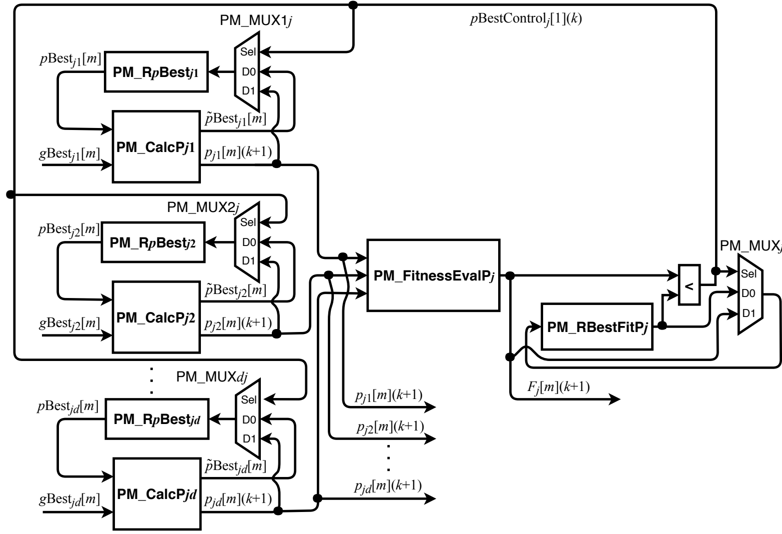

Fig. 3. PM_Pj submodule.

TABLE I
FPGA SYNTHESIS RESULTS FOR PSO WITH $N = 5$ PARTICLES.

m	$d=3$						$d=6$						$d=10$					
	RN	LN	MN	RF (%)	R_s (MSps)	DP (W)	RN	LN	MN	RF (%)	R_s (MSps)	DP (W)	RN	LN	MN	RF (%)	R_s (MSps)	DP (W)
12	723	2936	45	52	66.94	0.160	1366	5774	90	55	61.56	0.308	2224	9301	150	58	53.75	0.436
16	928	3808	45	45	61.67	0.187	1790	7380	90	48	55.99	0.332	2895	12239	150	49	50.64	0.499
20	1192	4808	60	38	59.66	0.239	2259	9350	120	40	51.83	0.396	3680	15327	200	42	46.00	0.617
32	1903	7924	120	24	45.54	0.311	3609	15134	240	27	42.29	0.596	5882	24765	400	28	34.89	0.738

TABLE II
FPGA SYNTHESIS RESULTS FOR PSO WITH $N = 10$ PARTICLES.

m	$d=3$						$d=6$						$d=10$					
	RN	LN	MN	RF (%)	R_s (MSps)	DP (W)	RN	LN	MN	RF (%)	R_s (MSps)	DP (W)	RN	LN	MN	RF (%)	R_s (MSps)	DP (W)
12	1436	5933	90	52	62.13	0.223	2674	11239	180	56	44.00	0.346	4386	18590	300	58	42.44	0.530
16	1826	7673	90	45	52.84	0.244	3499	14704	180	48	42.12	0.390	5716	24980	300	48	37.96	0.567
20	2308	9767	120	37	50.72	0.309	4414	18743	240	40	41.70	0.487	7187	31084	400	41	33.34	0.621
32	3693	15717	240	24	39.07	0.410	7010	30215	480	27	32.68	0.642	–	–	–	–	–	–

TABLE III
FPGA SYNTHESIS RESULTS FOR PSO WITH $N = 15$ PARTICLES.

m	$d=3$						$d=6$						$d=10$					
	RN	LN	MN	RF (%)	R_s (MSps)	DP (W)	RN	LN	MN	RF (%)	R_s (MSps)	DP (W)	RN	LN	MN	RF (%)	R_s (MSps)	DP (W)
12	2128	8634	135	53	53.83	0.244	4029	17394	270	54	45.06	0.453	6496	28007	450	58	39.52	0.648
16	2710	11179	135	46	49.50	0.296	5229	22018	270	48	38.52	0.444	8462	36574	450	49	33.52	0.663
20	3452	14609	180	37	41.52	0.325	6541	28906	360	39	35.03	0.523	10706	46683	600	41	33.10	0.693
32	5469	23357	360	25	33.36	0.440	10425	46293	720	26	28.75	0.535	–	–	–	–	–	–

these resources to a configuration of $n = 15$ particles with $d = 10$ dimensions. Alternatively, this problem can be solved by implementing part of the multipliers with logical cells.

The results showed significant gains in comparison to the studies in the literature presented in [15]–[19] where, for the Rastrigin function with $n = 10$ particles and $d = 6$ dimensions, the throughput obtained was between $44\,$MSps (or 44 mega iterations per second) and $32\,$MSps (or 32 mega iterations per second) for 12 and 32 bits, respectively. These values are equivalent to a speedup of $\approx 212\times$ ($\frac{44\,\text{MSps}}{207\,\text{KSps}}$) for 12 bits and of $\approx 154\times$ ($\frac{32\,\text{MSps}}{207\,\text{KSps}}$), for 32 bits [18].

The architecture proposed in [15], [19] used a semi-parallel approach where there is a general purpose processor to for computing the velocity and position of all particle (Lines 26-31 of Algorithm 1), and this creates a serialization process (a bottleneck). In the scheme proposed in this paper, each particle has a specific processor for computing their velocity and position avoiding the bottleneck. The works [15], [19] need $N \times d$ clocks for computing the velocity and position of all particle, and in this paper, the Lines 26-31 of Algorithm 1 were parallelized, and all information is computed in one clock. Similar to works presented in [15] and [19], the papers [16]–[18] shared five specific processor for computing the

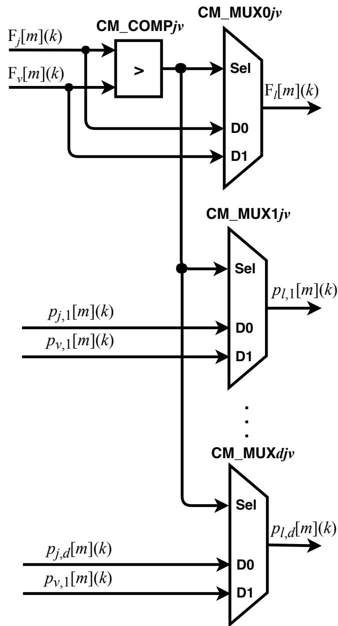

Fig. 4. CM_Pjv submodule.

velocity and position of all particles. Another bottleneck found in [16]–[18] is about the information storage. The velocity, position, and best position are store on, and this way creates serial access. The architecture proposed in this paper, each particle has specific registers for storage the velocity, position, and best position, in other words, there is parallel access for all information.

Regarding the hardware occupation, it was observed that the implementation proposed in this work also obtained gains when compared to other similar works from the literature. In [15], ≈ 145000 LUTs were used from a Virtex FPGA 6 for $n = 10$ particles and $d = 6$, by contrast, in the proposal here presented only ≈ 30000 were used, that represents, a reduction of $4.8\times$. For the same PSO configurations, the work [18] used 24025 registers and 73881 LUTs, already in work here presented were used 7010 registers and 30215 LUTs, a reduction about $3.4\times$ and $2.4\times$ for the registers and LUTs, respectively.

In order to compare with multi-core platform the PSO, based on Algorithm 1, was implemented on Intel(R) Core(TM) i7-7820HQ CPU 2.90 GHz 16 GB 2133 MHz LPDDR3 500 GB SSD. For the same PSO parameter used to FPGA, the multi-core platform had a throughput about the 9.5 Kps (or 9500 iterations per second). The FPGA speedup was of $\approx 4631\times$ ($\frac{44\,\text{MSps}}{9.5\,\text{KSps}}$) for 12 bits and of $\approx 3368\times$ ($\frac{32\,\text{MSps}}{9.5\,\text{KSps}}$), for 32 bits.

It is important to note that in none of the observed works a hardware with $n = 15$ particles and dimension $d = 10$ was mentioned. These results show that this proposed implementation can be used as a reference for several other associated works.

VI. Conclusion

This paper presented a proposal for a parallel implementation of the Particle Swarm Optimization algorithm (PSO) at fixed point on FPGA. All details regarding the proposal implementation were presented and analyzed in terms of occupation area and processing time. The proposed architecture was submitted to tests with the Rastrigin function obtaining the expected results. The results obtained were quite significant and point to new possibilities of using embedded PSO in hardware for real time applications with large data volume.

References

[1] M.-C. R. Rojas-Morales, N.; Rojas and E. M. Ureta, "A survey and classification of opposition-based metaheuristics," *Computers & Industrial Engineering*, 2017.

[2] A. A. Soler-Dominguez, A.; Juan and R. Kizys, "A survey on financial applications of metaheuristics," *ACM Comput. Surv.*, 2017.

[3] M. A. Kamel, X. Yu, and Y. Zhang, "Real-time optimal formation reconfiguration of multiple wheeled mobile robots based on particle swarm optimization," in *2016 12th World Congress on Intelligent Control and Automation (WCICA)*, June 2016, pp. 703–708.

[4] V. Fathi and G. A. Montazer, "An improvement in rbf learning algorithm based on pso for real time applications," *Neurocomputing*, vol. 111, pp. 169 – 176, 2013.

[5] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Particle swarm optimization*, vol. 4, 1995, pp. 1942–1948.

[6] A. Leskovec, J.; Rajaraman and J. D. Ullman, "Mining of massive datasets." *Cambridge university press*, 2014.

[7] E. Alba, G. Luque, and S. Nesmachnow, "Parallel metaheuristics: Recent advances and new trends," *International Transactions in Operational Research*, vol. 20, pp. 1–48, 01 2012.

[8] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation (Systems on Silicon)*. Morgan Kaufmann, 2010.

[9] A. C. D. de Souza and M. A. C. Fernandes, "Parallel fixed point implementation of a radial basis function network in an fpga," *Sensors*, vol. 14, no. 10, pp. 18 223–18 243, 2014.

[10] M. F. Torquato and M. A. C. Fernandes, "High-performance parallel implementation of genetic algorithm on fpga," *Circuits, Systems, and Signal Processing*, Jan 2019. [Online]. Available: https://doi.org/10.1007/s00034-019-01037-w

[11] M. Cárdenas-Montes, M. A. Vega-Rodríguez, J. J. Rodríguez-Vázquez, and A. Gómez-Iglesias, "Gpu-based evaluation to accelerate particle swarm algorithm," in *Computer Aided Systems Theory – EUROCAST 2011*, R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 272–279.

[12] M. Cámara, J. Ortega, and F. Toro, "Comparison of frameworks for parallel multiobjective evolutionary optimization in dynamic problems," in *Studies in Computational Intelligence*, 01 2012, vol. 415, pp. 101–123.

[13] L. A. Rastrigin, "Extremal control systems," in *Theoretical Foundations of Engineering Cybernetics Series*, Moscow, 1974.

[14] H. Mühlenbein, M. Schomisch, and J. Born, "The parallel genetic algorithm as function optimizer," *Parallel Comput.*, vol. 17, no. 6-7, pp. 619–632, Sep. 1991.

[15] R. M. Calazan, N. Nedjah, and L. M. Mourelle, "A hardware accelerator for particle swarm optimization," *Applied Soft Computing*, vol. 14, pp. 347 – 356, 2014.

[16] A. Rathod and R. A. Thakker, "Fpga realization of particle swarm optimization algorithm using floating point arithmetic," in *2014 International Conference on High Performance Computing and Applications (ICHPCA)*, Dec 2014, pp. 1–6.

[17] D. M. M. Arboleda, C. H. Llanos, L. d. S., and M. Ayala-Rincon, "Hardware architecture for particle swarm optimization using floating-point arithmetic," in *2009 Ninth International Conference on Intelligent Systems Design and Applications*, Nov 2009, pp. 243–248.

[18] D. M. Munoz, C. H. Llanos, L. d. S. Coelho, and M. Ayala-Rincon, "Comparison between two fpga implementations of the particle swarm optimization algorithm for high-performance embedded applications," in *2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA)*, Sept 2010, pp. 1637–1645.

[19] S.-A. Li, C.-C. Hsu, C.-C. Wong, and C.-J. Yu, "Hardware/software co-design for particle swarm optimization algorithm," *Information Sciences*, vol. 181, no. 20, pp. 4582 – 4596, 2011, special Issue on Interpretable Fuzzy Systems.