GPU-Based Automatic Configuration of Differential Evolution: A Case Study

Roberto Ugolotti, Pablo Mesejo, Youssef S.G. Nashed, Stefano Cagnoni

GPU-based Automatic Configuration of Differential Evolution: a case study

Roberto Ugolotti, Pablo Mesejo, Youssef S. G. Nashed, and Stefano Cagnoni

Department of Information Engineering, University of Parma, Italy
{rob_ugo,pmesejo,nashed,cagnoni}@ce.unipr.it

Abstract. The performance of an evolutionary algorithm strongly depends on the choice of the parameters which regulate its behavior. In this paper, two evolutionary algorithms (Particle Swarm Optimization and Differential Evolution) are used to find the optimal configuration of parameters for Differential Evolution. We tested our approach on four benchmark functions, and the comparison with an exhaustive search demonstrated its effectiveness. Then, the same method was used to tune the parameters of Differential Evolution in solving a real-world problem: the automatic localization of the hippocampus in histological brain images. The results obtained consistently outperformed the ones achieved using manually-tuned parameters. Thanks to a GPU-based implementation, our tuner is up to 8 times faster than the corresponding sequential version.

Keywords: Automatic Algorithm Configuration, Global Continuous Optimization, Particle Swarm Optimization, Differential Evolution, GPGPU

1 Introduction

In this paper we consider the problem of automatizing the configuration of an Evolutionary Algorithm (EA) [7]. This problem is particularly important because EAs are usually strongly influenced by the choice of the parameters which regulate their behavior [6].

Regarding automatic algorithm configuration, it is important to distinguish between parameter tuning and parameter control. In the former, parameter values are fixed in the initialization stage and do not change while the EA is running; in the latter, parameter values are adapted as evolution proceeds. This work is focused only on choosing parameter values, either numerical (integer or real) or nominal (representing different design options for the algorithm), which will then be kept constant during evolution.

Virtually, all EAs have free parameters that are partly set by the programmer and/or by the user. In this scenario, automatic parameter configuration is desirable, since *manual* parameter tuning is time consuming, error-prone, and may introduce a bias when comparing a new algorithm with a reference, caused by better knowledge of one with respect to the other or to possible differences in the amount of time spent tuning each of them. Nevertheless, it is important to

take the computational burden of these tuners into account, since they usually need to perform many tests to find the best parameter configuration for the algorithm under consideration.

In this regard, GPU implementations have arisen as a very promising way to speed-up EAs. Modern graphics hardware has gained an important role in parallel computing, since it has been used to accelerate general computations (General Purpose Graphics Processing Unit programming), as well as in computer graphics. In particular, CUDA^{TM} (Compute Unified Distributed Architecture) is a parallel computing environment by nVIDIA^{TM}, which exploits the massively parallel computation capabilities of its GPUs. CUDA-C [17] is an extension of the C language that allows development of GPU routines (termed *kernels*), that can be executed in parallel by several different CUDA^{TM} threads. We use GPU implementations of real-valued population-based optimization techniques (Particle Swarm Optimization (PSO) [9] and Differential Evolution (DE) [21]) to automatically configure DE parameters. The implementations of both metaheuristics on GPU are publicly available in a general-purpose library[1]. DE has been chosen as the method to be tuned mainly because it is the optimizer initially proposed and employed in the real-world problem under consideration [13], while PSO was chosen as tuner because of its easy parallelization and its ability to reach good results using a limited number of fitness evaluations.

The choice of using Evolutionary Algorithms or Swarm Intelligence [2] algorithms as tuners derives from the large number of attractive features they offer: robust and reliable performance, global search capability, virtually no need of specific information about the problem to solve, easy implementation, and, above all, implicit parallelism, which allows one to obtain a parallel implementation starting from a design which is also conceptually parallel.

We first demonstrate that our method works properly by optimizing DE parameters for some well-known benchmark functions and compare these results with the ones obtained by a systematic search of the parameter space. Then, we apply the same approach to the real-world problem of localizing the hippocampus in histological brain images, obtaining a significant improvement with respect to the manually-tuned algorithm. Finally, we prove that our GPU-based implementation significantly reduces computation time, compared to a sequential implementation.

The remainder of this paper is structured as follows: in Section 2 we provide the theoretical foundations necessary to understand our work, including the metaheuristics and the problem used to test the system. In Section 3, a general overview of the method is presented, providing details about its GPU implementation. Finally, Section 4 presents the results of our approach followed, in Section 5, by some final remarks and a discussion about future developments.

[1] The code can be downloaded from http://sourceforge.net/p/libcudaoptimize [16]

2 Theoretical Background

In this section, we will briefly describe the metaheuristics used in this work (DE and PSO) and the hippocampus localization task used as test, and make a short review of the approaches in which metaheuristics have been used to tune EAs.

2.1 Particle Swarm Optimization

Particle Swarm Optimization is a well-established stochastic optimization algorithm which simulates the behavior of bird flocks. A set of N particles moves through a "fitness" function domain seeking the function optimum. Each particle's motion is described by two simple discrete-time equations which regulate the particles' velocity and position:

$$\begin{aligned}\boldsymbol{v_n}(t) &= w \cdot \boldsymbol{v_n}(t-1) \\ &\quad + c_1 \cdot \boldsymbol{rand()} \cdot (\boldsymbol{BLP_n} - \boldsymbol{P_n}(t-1)) \\ &\quad + c_2 \cdot \boldsymbol{rand()} \cdot (\boldsymbol{BNP_n} - \boldsymbol{P_n}(t-1)), \\ \boldsymbol{P_n}(t) &= \boldsymbol{P_n}(t-1) + \boldsymbol{v_n}(t),\end{aligned}$$

where $\boldsymbol{P_n}(t)$ and $\boldsymbol{v_n}(t)$ represent the n^{th} particle's position and velocity at time t; c_1, c_2 are positive constants which represent how strongly cognitive and social information affects the behavior of the particle; w is a positive inertia factor (possibly depending on time t) used to balance PSO's global and local search abilities; $\boldsymbol{rand()}$ returns a vector of random values uniformly distributed in $[0, 1]$; $\boldsymbol{BLP_n}$ and $\boldsymbol{BNP_n}$ are, respectively, the best-fitness location visited so far by particle n and by any particle in its neighborhood; this may include a limited set of particles (lbest PSO) or coincide with the whole population (gbest PSO). One of the most commonly used PSO versions is based on a "ring topology", i.e. each particle's neighborhood includes the particle itself, the K particles preceding it, and the K particles which immediately follow it.

2.2 Differential Evolution

Differential Evolution is one of the most successful EAs for global continuous optimization [3]. DE perturbs the current population members with the scaled difference of distinct individuals. At every generation, every individual $\boldsymbol{X_i}$ of the current population acts as a parent vector, for which a donor vector $\boldsymbol{D_i}$ is generated. In the original version of DE, the donor vector for the i^{th} parent ($\boldsymbol{X_i}$) is created by combining three distinct randomly-selected elements $\boldsymbol{X_{r1}}$, $\boldsymbol{X_{r2}}$ and $\boldsymbol{X_{r3}}$. The donor vector is then computed as:

$$\boldsymbol{D_i} = \boldsymbol{X_{r1}} + F \cdot (\boldsymbol{X_{r2}} - \boldsymbol{X_{r3}}),$$

where the scale factor F (usually $F \in [0, 1.5]$) is a parameter that strongly influences DE's performances. This mutation strategy is called *random*, but other

mutation strategies have been proposed. Two of the most successful ones are *best* (equation 1) and *target-to-best* (TTB) (equation 2):

$$\boldsymbol{D_i} = \boldsymbol{X_{best}} + F \cdot (\boldsymbol{X_{r1}} - \boldsymbol{X_{r2}}), \tag{1}$$

$$\boldsymbol{D_i} = \boldsymbol{X_i} + F \cdot (\boldsymbol{X_{best}} - \boldsymbol{X_i}) + F \cdot (\boldsymbol{X_{r1}} - \boldsymbol{X_{r2}}), \tag{2}$$

where $\boldsymbol{X_{best}}$ represents the best-fitness individual in the current population.

After mutation, every parent-donor pair generates one child (called trial vector, $\boldsymbol{T_i}$) by means of a crossover operation. Two kinds of crossover are typically used: *binomial* (uniform) and *exponential*; both are regulated by a parameter called *crossover rate* ($Cr \in [0, 1]$). In the former, a real number $r \in [0, 1]$ and an integer $M \in [1, S]$, where S is the size of the search space, are randomly generated for each element of the vector. The trial vector is then defined as follows:

$$T_{i,j} = \begin{cases} D_{i,j} & \texttt{if } r \leq Cr \texttt{ or } j = M \\ X_{i,j} & \texttt{otherwise} \end{cases}, j = 1 \ldots S$$

The exponential crossover, instead, generates an integer L according to this pseudo-code:

$L = 0$
`do`
$L = L + 1$
$r = rand()$
`while` $((r \leq Cr)$ `and` $(L \leq S))$

This value is then used to generate the trial vector:

$$T_{i,j} = \begin{cases} D_{i,j} & \texttt{for } j = \langle M \rangle_S \ldots \langle M + L - 1 \rangle_S \\ X_{i,j} & \texttt{otherwise} \end{cases}, j = 1 \ldots S$$

where $\langle \cdot \rangle_S$ represents the *modulo* S function.

After the crossover operation, the trial vector is evaluated and its fitness is compared to its parent's: only the one having the best fitness survives in the next generation, while the other is discarded from the population.

2.3 Hippocampus Localization in Histological Images

The hippocampus is a structure located in the mammalian brain that has long been known for its crucial role in learning and memory processes, as well as an early biomarker for Alzheimer disease and epilepsy. Thus, its automatic, robust and fast localization is of great interest for the scientific community.

As a real-valued benchmark for our method, we consider the problem of localizing the hippocampus in histological images (see Figure 1) extracted from the Allen Brain Atlas (ABA), a publicly available image database [1] which contains a genome-scale set of histological images (cellular-resolution gene-expression profiles) obtained by In Situ Hybridization of serial sections of mouse brains.

In particular, a localization method based on Active Shape Models has been recently presented to tackle this kind of images [13]. It uses a medial shape-based

representation of the hippocampus in polar coordinates, with the objective of creating simple models that can be managed easily and efficiently. Two parametric models (see Figure 1b) representing the two parts that compose the hippocampus (called SP and SG, see Figure 1a) are moved and deformed by DE according to an intensity-based similarity function between the model and the object itself.

Each model comprises two sets of points: the goal is to overlap the first one (Inner Set, $\boldsymbol{I}$) to the part of the hippocampus to be located, while placing the other one (Outer Set, $\boldsymbol{O}$), obtained by rigidly shifting the first one, immediately outside it. This model is subjected to external forces (driven by the image features) and internal forces (driven by the model itself). The target function H to be maximized (see [13] for a more detailed description) has three components: the external energy EE, the internal energy IE, and the contraction factor C.

$$H = EE - (IE + C)$$

In turn, EE can be divided in two components: PE, that depends on the control points of the model (denoted by black dots in Figure 1b) and CE, that is computed considering the points which belong to the segments connecting them:

$$EE = \gamma_P \cdot PE + \gamma_C \cdot CE$$
$$PE = \sum_{i=1}^{n} [T(N_3(I_i)) - T(N_3(O_i))]$$
$$CE = \sum_{i=2}^{n} \sum_{j=1}^{p} T(I_{i-1} + \frac{j}{p+1}(I_i - I_{i-1})) - \sum_{i=2}^{n} \sum_{j=1}^{p} T(O_{i-1} + \frac{j}{p+1}(O_i - O_{i-1}))$$

where I_i and O_i represent the points of the two sets, γ_P and γ_C are positive values that weigh the two components, $N_3(P)$ is a 3×3 neighborhood centered in P, $T(P)$ is the intensity of P if P is a point, or the average intensity if P is a neighborhood, and p is the number of points sampled in each segment.

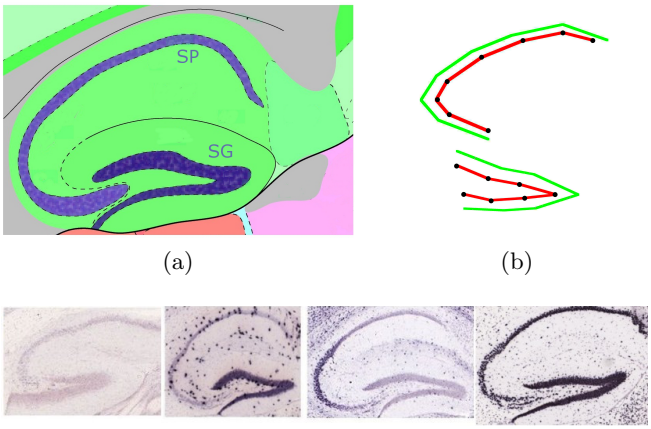
The internal energy IE is computed as:

$$IE = \xi_\rho \cdot \sqrt{\sum_{i=2}^{n} (\rho_i - \rho_{mi})^2} + \xi_\vartheta \cdot \sqrt{\sum_{i=2}^{n} (\vartheta_i - \vartheta_{mi})^2}$$

where ξ_ρ and ξ_ϑ are two positive weights that regulate the deformability of the model. (ρ_i, θ_i) represents a point of the model in polar coordinates, while (ρ_{mi}, θ_{mi}) represents a point of a reference model derived empirically from a set of "training" images.

Finally, the contraction factor C also regulates the model's deformability to avoid unfeasible situations that are not allowed by the nature of the hippocampus and is defined as follows:

$$C = \xi_c \cdot \|I_n - I_1\|$$



(a) (b)

(c)

Fig. 1: In the upper row, an anatomical atlas representation of the hippocampal region that highlights the two parts to be recognized (SP and SG) and an example of the parametric models used to localize the hippocampus; below, four examples of hippocampus images taken from the ABA.

2.4 Automatic Configuration of EAs

Parameter tuning for configuring and analyzing EAs is a topic frequently dealt with in the last years [6, 19]. In fact, the idea of a meta-EA was already introduced in 1978 by Mercer and Sampson [12], while Grefenstette, in 1986 [8], conducted more extensive experiments with a meta-GA and showed its effectiveness. However, the problem of tuning EAs by means of meta-EAs implies tackling two challenges: the noise in (meta)-fitness values and the very expensive (meta)-evaluations. Our GPU-based implementation tries to reduce the impact of the latter problem. Recently, REVAC, a specific type of meta-EA where the population approximates the probability density function of the most promising areas of the utility-landscape (similar to Iterative F-RACE), was shown to be able to find much more robust parameter values than one could set by hand, outperforming the winner of the competition on the CEC 2005 test-suite [20]. An attempt to find the optimal parameters for DE using a metaheuristic called Local Unimodal Sampling is described in [18]. As well, PSO parameter tuning has already been proposed based on other overlaying optimizers, like in [11], where PSO itself is used to this end.

3 Parameter Configuration using Metaheuristics

3.1 Parallel Implementation

The GPU-based implementation of the metaheuristics employed in this paper has been presented in [16].

The first implementations of PSO and DE based on nVIDIA CUDA™ were developed in 2009 and 2010, respectively [4, 5]. After that, other implementations of DE have been developed [10], and fast versions of PSO have been implemented by removing the constraint of synchronicity between particles [15]. The early GPU PSO implementations suffered from a coarse-grained parallelization (one thread per particle), that did not offer the opportunity to compute the fitness function, usually the most time-consuming process, in parallel over the problem dimensions. This aspect has been improved by letting each thread manage a single dimension of each particle, adding a further level of parallelism [14]. Similar inefficiencies characterized the early implementations of DE. These problems were subsequently addressed by [10] and [16].

Our GPU-based implementations of PSO and DE share a similar kernel configuration. Both are implemented as three distinct kernels: (i) the first kernel generates the solutions to be evaluated, (ii) the second kernel implements the fitness function, and (iii) the last kernel updates the population.

3.2 General Design

A tuner (or meta-optimizer) is implemented exactly in the same way as any optimizer used to solve any other problem. The only significant difference consists in the evaluation of the fitness function F. In this case, each particle $\boldsymbol{X_i} = \{x_1, \ldots, x_n\}$ of a tuner represents an optimizer (or better, a set of $m \leq n$ parameters that describes an optimizer). An optimizer $O_i(\boldsymbol{X_i})$ is instantiated using the parameters $\boldsymbol{X_i}$ and the entire optimization process is repeated on the test function T times to compute fitness f_i as the average fitness over the T runs. This optimizer, used to tune DE, has been introduced in [22].

PSO and DE are real-valued optimization methods, so the problem of representing nominal parameters needs to be addressed. We chose to represent each nominal value by a vector which associates a number to each option available; the option associated with the highest number is then selected. Figure 2 shows the encoding of a particle representing a DE instance. Table 1 shows the range and possible values of the DE parameters in the representation we have used.

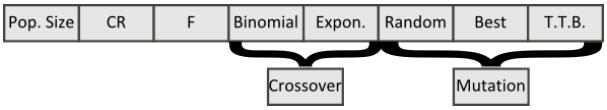


Fig. 2: Representation of a set of parameters used in the tuner. The first three elements represent numerical parameters (integer and real-valued numbers), while the last five encode the nominal parameters, i.e. crossover type and mutation strategy. In this case $n = 8$ values are needed to represent $m = 5$ parameters.

Table 1: Range of DE parameters allowed by the tuning algorithms. The last column shows the sampling step used in the systematic search used as reference.

Parameter	Values	Step
Crossover Rate	$[0.0, 1.0]$	0.1
Scale Factor (F)	$[0.0, 1.5]$	0.1
Population Size	[30,150]	10
Crossover	{binomial, exponential}	-
Mutation	{random, best, target-to-best}	-

Table 2: Benchmark functions. For every function, the table displays name, search space, formula, modality (multimodal, unimodal) and separability (separable, non separable). Optimum fitness values are 0 for all functions.

Name	Range	Formula	Modality	Separability
Zakharov	$[-10, 10]^n$	$\left(\sum_{i=1}^{n} {x_i}^2\right) + \left(\sum_{i=1}^{n} 0.5 \cdot i \cdot {x_i}^2\right)^2 + \left(\sum_{i=1}^{n} 0.5 \cdot i \cdot {x_i}^2\right)^4$	U	S
Schwefel 1.2	$[-100, 100]^n$	$\sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j\right)^2$	U	NS
Rastrigin	$[-5.12, 5.12]^n$	$\sum_{i=1}^{n} \left\{x_i^2 - 10 \cdot \cos(2\pi x_i) + 10\right\}$	M	S
Rosenbrock	$[-100, 100]^n$	$\sum_{i=2}^{n} 100(x_i - x_{i-1}^2)^2 + (1 - x_{i-1})^2$	M	NS

4 Experiments

In this section, we will describe the two sets of tests performed on benchmark functions and on the hippocampus localization problem. Tests were run on a 64-bit Intel Core i7 CPU running at 3.40GHz using CUDA v. 4.2 on an nVidia GeForce GTX680 graphics card with compute capability 3.0.

4.1 Benchmark Functions

Four classical benchmark functions (see Table 2) having different nature (unimodal/multimodal and separable/non separable) have been selected to test our method. For each function, we systematically sampled (with the sampling steps shown in Table 1) the parameter search space and ran 10 independent repetitions for each of the 13728 parameter combinations generated. This way, we created a *ground truth* used to assess the results of the meta-optimization. The termination criteria was set after 1000 generations and the search space size was set to 16.

Parameter tuning has been repeated 10 times (5 using DE as meta-optimizater, 5 using PSO) on each function. Table 3 shows the parameters used by the tuners based on DE and PSO. In order to choose them, we performed the tuning procedure described here over 8 benchmark functions using "standard" parameters for PSO and DE (see [22]). The best parameter configuration found, based on the number of times in which it obtained the best fitness value, was used in our

Table 3: Parameters of DE/PSO tuners.

DE	PSO
TTB Mutation	$C_1 = 1.525$
Exponential Crossover	$C_2 = 1.881$
F = 0.52	$w = 0.443$
Cr = 0.91	Ring Topology ($K = 1$)
Population Size = 24, Generation = 64	

Table 4: Range of the parameters found by systematic search and meta-optimization. P is population, Mut is mutation strategy, $Cross$ is Crossover Type.

Zakharov	Schewfel 1.2	Rastrigin	Rosenbrock
Systematic Search			
$Cr \in [0.8, 0.9]$	$Cr \in [0.9, 1.0]$	$Cr \in [0.0, 0.4]$	$Cr \in [0.8, 0.9]$
$F \in [0.5, 0.6]$	$F \in [0.5, 0.7]$	$F \in [0.1, 0.6]$	$F \in [0.6, 0.7]$
$P \in [100, 150]$	$P \in [120, 150]$	$P \in [90, 150]$	$P \in [90, 150]$
$Mut \in \{Best, TTB\}$	$Mut \in \{Best, TTB\}$	$Mut \in \{Rand, Best\}$	$Mut \in \{Best, TTB\}$
$Cross \in \{Bin, Exp\}$	$Cross \in \{Bin, Exp\}$	$Cross \in \{Bin, Exp\}$	$Cross \in \{Bin, Exp\}$
Meta-Optimization			
$Cr \in [0.799, 1.0]$	$Cr \in [0.935, 1.0]$	$Cr \in [0.0, 0.344]$	$Cr \in [0.740, 0.939]$
$F \in [0.433, 0.714]$	$F \in [0.535, 0.667]$	$F \in [0.207, 0.596]$	$F \in [0.646, 0.686]$
$P \in [111, 146]$	$P \in [80, 150]$	$P \in [79, 149]$	$P \in [83, 150]$
$Mut \in \{Best, TTB\}$	$Mut \in \{Best, TTB\}$	$Mut \in \{Rand, Best\}$	$Mut \in \{Best, TTB\}$
$Cross \in \{Bin, Exp\}$	$Cross \in \{Bin, Exp\}$	$Cross \in \{Bin, Exp\}$	$Cross \in \{Bin, Exp\}$

tests. Notice that, with this parameter configuration, a maximum of 1536 sets of parameters (64 generations $\times$ 24 individuals) are evaluated in each run.

Table 4 shows, for each function, the ranges of the parameters for the best 10 settings found in the systematic search and for the 10 combinations obtained by the meta-optimization process. As can be observed, all parameter ranges virtually overlap, which proves that the meta-optimizer operates correctly. For every set of parameters, we performed 100 independent runs on the corresponding function. All optimizers were able to obtain a median value of 0 over the corresponding function. The Wilcoxon signed-rank test confirmed the absence of statistically significant differences between the results of the systematic search and the tuning, since the p-values obtained were always larger than 0.01.

To assess the speedup of the GPU implementation, we performed the same experiments adapting a sequential C++ DE implementation[2]. We compared the execution time of our method and of a sequential version of the tuner, using DE as meta-optimizer on Rastrigin function. Using the same parameters previously shown, the GPU version takes an average of 164.4 s for the entire meta-optimization process, while the sequential version takes 1007.5 s (6.1 times slower). Moreover, if we increase the problem dimension from 16 to 64, the two

[2] http://www1.icsi.berkeley.edu/~storn/code.html

Table 5: The ten sets of parameters generated by the meta-optimization processes. The three columns for SP and SG (see Figure 1a) indicate the average fitnesses and standard deviations and if there is a significant improvement against the original parameters (+), no difference (=) or a worse result (−). This is a maximization problem: a higher value represents a better solution.

	Parameters					SP			SG		
	Cr	F	P	Mut	$Cross$	Avg	Std	Cmp	Avg	Std	Cmp
Original	0.9	0.7	150	TTB	Exp	142.6	14.7		136.2	17.8	
DE 1	0.859	0.41	121	$Rand$	Bin	144.5	12.1	+	141.0	13.9	+
DE 2	0.952	0.427	150	$Rand$	Exp	145.0	11.3	+	141.9	13.2	+
DE 3	0.952	0.419	150	$Rand$	Exp	145.0	11.5	+	141.8	13.1	+
DE 4	0.9	0.431	144	$Rand$	Bin	144.8	11.7	+	141.3	13.8	+
DE 5	0.954	0.44	115	$Rand$	Exp	144.9	11.7	+	141.4	13.7	+
PSO 1	0.949	0.448	149	$Rand$	Exp	145.1	11.2	+	141.8	13.1	+
PSO 2	0.953	0.471	143	$Rand$	Exp	145.0	11.3	+	141.7	13.2	+
PSO 3	0.974	0.473	150	$Rand$	Exp	145.0	11.5	+	142.0	13.0	+
PSO 4	0.783	0.347	150	$Rand$	Bin	144.6	11.8	+	141.0	13.7	+
PSO 5	0.922	0.437	139	$Rand$	Exp	145.0	11.2	+	141.4	13.2	+

versions take 458.2 s and 3683.7 s, respectively and, consequently, the speedup increases to 8. Please notice that these speedups are only indicative of the problem, since they depend on a very high number of parameters, like population size, number of evaluations, degree of parallelism and complexity of the fitness function and, obviously, on the hardware on which the optimizer is run.

4.2 Hippocampus Localization

After checking the correctness of our approach, we repeated the same procedure to solve the hippocampus localization task described in Section 2.3. The parameters of the meta-optimizer are the same used in the previous sections.

Table 5 shows the parameters obtained by our meta-optimizer and by the manually-tuned ones used in [13]. To exclude any possible bias in favor of the former, we used a slightly different (and better working) setting for the reference, increasing its population to the maximum allowed, 150. This choice has been made because the implementation in [13] was sequential and this led us to use a lower number of particles for speed issues, while in our GPU implementation population size has no impact on computation time.

Each row in the table presents the method used in the meta-optimization process and the parameters found. Each set of parameters have been tested 10 times over 320 different hippocampus images.

In order to check the statistical significance of the results obtained, a Kruskal-Wallis test was performed with a level of confidence of 0.01. Since the normality and homoscedasticity assumptions were not met, as checked through the application of Kolmogorov-Smirnov and Bartlett's tests, non-parametric tests were used. The p-value was close to zero, suggesting that at least one sample median

is significantly different from the others. The columns termed "Cmp" indicate the results of the Wilcoxon signed-rank test used to assess the statistical significance of the difference between the generated optimizer and the reference manually-tuned one (Original) for the two parts that compose the model (SP and SG). This value is a "+" if the results of the automatically tuned DE are significantly better than the original ones (significance of $p = 0.01$, using the Bonferroni-Holm correction), a "−" when the original method is better, and a "=" when there are no statistically significant differences.

As can be seen, meta-optimization always leads to similar results, and the automatically selected parameters always lead to higher mean and lower standard deviation than the ones set after a time-consuming manual tuning.

5 Discussion

In this paper we proposed a method to find the best parameters for an Evolutionary Algorithm (in this case, Differential Evolution) to solve a real-world problem. This method uses an evolutionary process to search the space of the algorithm parameters to find the best possible configuration for it.

We tested this method on four benchmark functions and the comparison with a systematic search proved that our method is actual effective. Afterwards, we repeated the same procedure optimizing the DE algorithm parameters in a real-world application, where Differential Evolution is used to localize hippocampi in brain histological images. The results obtained by the automatically-tuned optimizers outperform the manually-tuned ones almost systematically.

As future work, we will try to increase the degree of parallelization in order to take even more advantage of the parallel nature of the algorithms.

We evaluate the goodness of our tuning procedure by the quality of the solution obtained by the tuned optimizer. In the same way, we could add another step to empirically find a good set of parameters for the meta-optimizer. Of course, this could lead to an infinite repetition of the same procedure and to an exponential increase in optimization time. However, our practical aim is just to improve the performance of manually-tuned optimizers without increasing complexity too much, when dealing with problems in which the optimizer's performance is particularly sensitive to the parameters.

Acknowledgments

Roberto Ugolotti is funded by Fondazione Cariparma. Pablo Mesejo and Youssef S. G. Nashed are funded by the European Commission (Marie Curie ITN MIBISOC, FP7 PEOPLE-ITN-2008, GA n. 238819).

References

1. Allen Institute for Brain Science: Allen Reference Atlases. http://mouse.brain-map.org (2004-2006)

2. Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm Intelligence: From Natural to Artificial Systems. Oxford (1999)
3. Das, S., Suganthan, P.: Differential Evolution: A Survey of the State-of-the-Art. IEEE Transactions on Evolutionary Computation **15**(1) (2011) 4–31
4. de Veronese, L., Krohling, R.: Swarm's flight: Accelerating the particles using C-CUDA. In: Proc. IEEE Congress on Evolutionary Computation. (2009) 3264–3270
5. de Veronese, L., Krohling, R.: Differential evolution algorithm on the GPU with C-CUDA. In: Proc. IEEE Congress on Evolutionary Computation. (2010) 1–7
6. Eiben, A.E., Smit, S.K.: Parameter tuning for configuring and analyzing evolutionary algorithms. Swarm and Evolutionary Computation **1**(1) (2011) 19 – 31
7. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer Verlag (2003)
8. Grefenstette, J.: Optimization of control parameters for genetic algorithms. IEEE Trans. Syst. Man Cybern. **16**(1) (1986) 122–128
9. Kennedy, J., Eberhart, R.: Particle Swarm Optimization. In: Proc. IEEE International Conference on Neural Networks. Volume 4. (1995) 1942–1948
10. Krömer, P., Snášel, V., Platoš, J., Abraham, A.: Many-threaded implementation of differential evolution for the CUDA platform. In: Proc. of Genetic and Evolutionary Computation Conference (GECCO), ACM (2011) 1595–1602
11. Meissner, M., Schmuker, M., Schneider, G.: Optimized Particle Swarm Optimization (OPSO) and its application to artificial neural network training. BMC Bioinformatics **7** (2006)
12. Mercer, R., Sampson, J.: Adaptive search using a reproductive metaplan. Kybernetes **7** (1978) 215–228
13. Mesejo, P., Ugolotti, R., Di Cunto, F., Giacobini, M., Cagnoni, S.: Automatic hippocampus localization in histological images using differential evolution-based deformable models. Pattern Recognition Letters **34**(3) (2013) 299 – 307
14. Mussi, L., Daolio, F., Cagnoni, S.: Evaluation of parallel Particle Swarm Optimization algorithms within the CUDA architecture. Information Sciences **181**(20) (2011) 4642–4657
15. Mussi, L., Nashed, Y.S.G., Cagnoni, S.: GPU-based asynchronous particle swarm optimization. In: Proc. of the Genetic and Evolutionary Computation Conference (GECCO), ACM (2011) 1555–1562
16. Nashed, Y.S.G., Ugolotti, R., Mesejo, P., Cagnoni, S.: libCudaOptimize: an open source library of GPU-based metaheuristics. In: Proc. of the Genetic and Evolutionary Computation Conference (GECCO) companion, ACM (2012) 117–124
17. nVIDIA Corporation: nVIDIA CUDA Programming Guide v. 4.0. (2011)
18. Pedersen, M.E.H.: Tuning and Simplifying Heuristical Optimization. Master's thesis, University of Southampton (2010)
19. Smit, S.K., Eiben, A.E.: Comparing parameter tuning methods for evolutionary algorithms. In: Proc. of the IEEE Congress on Evolutionary Computation. (2009) 399–406
20. Smit, S.K., Eiben, A.E.: Beating the 'world champion' evolutionary algorithm via REVAC tuning. In: Proc. of IEEE Congress on Evolutionary Computation. (2010) 1–8
21. Storn, R., Price, K.: Differential Evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces. Technical report, International Computer Science Institute (1995)
22. Ugolotti, R., Nashed, Y.S.G., Mesejo, P., Cagnoni, S.: Algorithm Configuration using GPU-based Metaheuristics. In: Proc. of the Genetic and Evolutionary Computation Conference (GECCO) companion. (2013)