A Comparative Study of Three GPU-Based Metaheuristics

Youssef S.G. Nashed[1], Pablo Mesejo[1], Roberto Ugolotti[1], Jérémie Dubois-Lacoste[2], and Stefano Cagnoni[1]

[1] Department of Information Engineering, University of Parma, Italy
{nashed,pmesejo,rob_ugo,cagnoni}@ce.unipr.it
[2] IRIDIA, CoDE, Université Libre de Bruxelles, Belgium
jeremie.dubois-lacoste@ulb.ac.be

Abstract. In this paper we compare GPU-based implementations of three metaheuristics: Particle Swarm Optimization, Differential Evolution, and Scatter Search. A GPU-based implementation, obviously, does not change the general properties of the algorithms. As well, we give for granted that GPU-based implementation of both algorithm and fitness function produces a significant speed-up with respect to a sequential implementation. Accordingly, the main goal of this work has been to fairly assess the efficiency of the GPU-based implementations of the three metaheuristics, based on the statistical analysis of the results they obtain in optimizing a benchmark of twenty functions within a prefixed limited time.

Keywords: Global Continuous Optimization, Particle Swarm Optimization, Differential Evolution, Scatter Search, GPGPU.

1 Introduction

Modern graphics hardware has gained an important role in the area of parallel computing, since it has been used to accelerate general computations (General Purpose Graphics Processing Unit - GPGPU - programming), in addition to playing its natural role. CUDA^{TM} (Compute Unified Distributed Architecture) is a parallel computing environment by nVIDIA^{TM} which exploits the massively parallel computation capabilities of its massively parallel GPUs. In particular, CUDA-C [14] 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, following the Single Instruction Multiple Thread model.

Among the stochastic approaches to continuous optimization, Evolutionary Algorithms (EAs) [6] and Swarm Intelligence [1] algorithms offer a number of attractive features: 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.

In this paper, we compare the GPU implementations of three real-valued population-based optimization techniques: Particle Swarm Optimization (PSO)

C.A. Coello Coello et al. (Eds.): PPSN 2012, Part II, LNCS 7492, pp. 398–407, 2012.

[8], Differential Evolution (DE) [16], and Scatter Search (SS) [7] [1]. They have been tested on a benchmark of 20 numerical problems (also implemented on GPU), comprising unimodal/multimodal and separable/non separable functions (see Table 3). The main contributions of this paper are: i. the first implementation, to the best of our knowledge, of Scatter Search in CUDA^{TM} ; ii. a novel parallel version of DE that solves some of the problems of previous implementations; iii. the study of these three metaheuristics on a broader benchmark than those usually adopted to test GPU implementations; iv. an unbiased evaluation of the effectiveness of the GPU implementation, as each metaheuristic has been subjected to the same parameter optimization method and run for a pre-fixed limited amount of time.

2 Basics of the Three Metaheuristics

2.1 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a stochastic optimization algorithm which simulates the behavior of bird flocks. A set of particles (or solutions) move through a "fitness" function domain (search space) seeking the function optimum (best fitness value). Each particle's motion is described by two simple discrete-time equations which regulate the particles' velocity and position:

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

where $P_n(t)$ and $v_n(t)$ are the position and velocity of the n^{th} particle at time t; c_1, c_2 and w (inertia factor) are positive constants, $rand()$ returns random values uniformly distributed in $[0, 1]$, BP_n is the best-fitness location visited so far by the particle, and BLP_n is the best-fitness location visited so far by any particle in its neighborhood, which may include a limited set of particles or even coincide with the whole swarm.

2.2 Differential Evolution

Differential Evolution has recently been shown to be one of the most successful EAs for global continuous optimization [2,18]. Unlike traditional EAs, DE perturbs the current population members with the scaled differences of randomly selected and distinct individuals. In the first iterations the elements are widely scattered in the search space and have a great exploration ability. As optimization proceeds, the individuals tend to concentrate in the regions of the search space with better fitness values, so the search automatically focuses on the most

[1] The code can be downloaded from `http://sourceforge.net/p/libcudaoptimize` [13].

promising areas. In DE, every element acts as a parent vector, for which a donor vector is created. In the original version of DE, the donor vector for the i^{th} parent (X_i) is generated by combining three random and distinct elements X_{r1}, X_{r2} and X_{r3}. The donor vector V_i is calculated as:

$$V_i = X_{r1} + F \cdot (X_{r2} - X_{r3})$$

where F (scale factor) is a parameter that strongly influences DE's performances and typically lies in the interval $[0.4, 1]$. After mutation, every parent-donor pair generates one child (trial vector) by means of a crossover operation. Two kinds of crossover are typically used: binomial (uniform) and exponential. Crossover is applied with a certain probability Cr (crossover rate) that, like F, is one of the control parameters of DE. Then, the trial vector is evaluated and its fitness is compared to its parent's: the better survives while the other is discarded.

2.3 Scatter Search

Scatter Search is based on a systematic combination between solutions (instead of randomized, as is usual in EAs) taken from a subset of the population, named the "reference set", that is usually significantly smaller than a typical EA population. SS is composed of five structural "blocks" or methods (see Figure 1):

1. Diversification Generation: a population of solutions P is generated, having a certain degree of quality and diversity. The reference set R is then drawn from P, and includes the $|R_1|$ solutions with best fitness, and the $|R_2|$ solutions from the reference set that are farthest from P according to the Euclidean distance (hence, $|R| = |R_1| + |R_2|$); the evolution process acts only on R;
2. Solution Combination: in most problems a specific method to combine solutions is needed, which can be applied to all solutions or only to selected ones (e.g., the best solutions, and/or randomly selected ones). In many cases an existing crossover operator, borrowed from other EAs, is employed;
3. Subset Generation: the procedure deterministically generates subsets of R, to which the combination method is applied.
4. Improvement: an improvement method (typically a local search) is applied to the original solutions and/or to combined solutions;
5. Reference Set Update: once a new solution is obtained it replaces the worst solution in R only if it improves the quality of the reference set in terms of fitness and/or diversity;

3 Parallel Implementation

The first parallel versions of PSO relied on multiprocessor parallel machines or cluster of computers. With the introduction of GPUs, research shifted towards GPU-based parallel PSO (GPU PSO) to alleviate multi-processor and cluster systems inefficiencies, such as network overhead, shared memory access, etc. In 2009 and 2010, respectively, the first implementations of PSO and DE

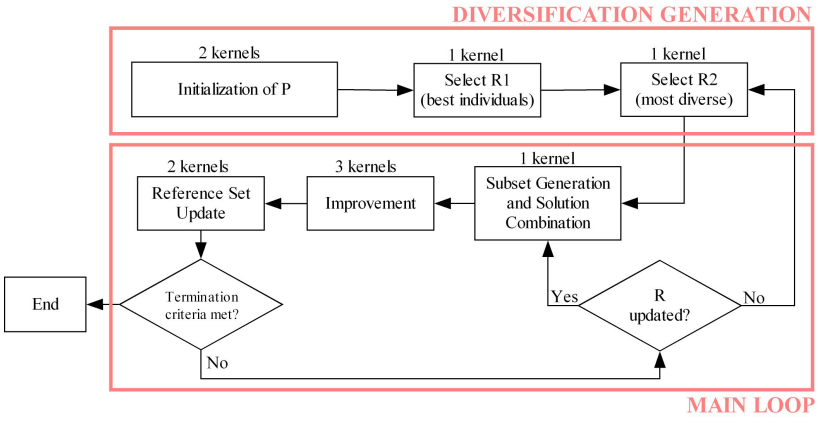

Fig. 1. Block diagram of the Scatter Search Algorithm

using nVIDIA CUDA^{TM} were developed [3,4]. Also in 2009, a hybrid between GPU PSO and pattern search, aimed at enhancing the convergence of PSO, was presented in [20]. After that, other implementations of DE have been developed [9,21], and fast versions of PSO have been implemented by relaxing the synchronicity constraints between particles' evaluation and update [12].

The early GPU PSO implementations suffered from a coarse-grained parallelization (one thread per particle), that neglected 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 the PSO implementation evaluated in this work, first introduced in [11]. In it, a thread manages a single dimension of each particle, adding a further level of parallelism. Similar inefficiencies characterized the early implementations of DE, such as a partially sequential implementation of the fitness function and random number generation. These problems were addressed by [9] using four kernels executed sequentially. Our present implementation uses only one kernel for generating the trial vectors, and another for fitness evaluation and migration. In addition, we offer three different mutations and two kinds of crossovers, while early GPU-based DE considered only one mutation strategy (DE/rand/1) and one kind of crossover. Regarding SS, to the best of our knowledge, ours is the first parallel implementation of this metaheuristic.

The performance of CUDA^{TM} code depends chiefly on the thread configuration, number of kernels, and memory access schemes used. All methods presented here aim at exploiting fast-access local and shared memory instead of slower global memory to the greatest possible extent, considering the number of kernels as the main criterion to assess how well an algorithm can be parallelized.

PSO is divided into three kernels described in [11], while DE, as mentioned earlier, can be implemented as two kernels. Each thread of the first kernel performs the following instructions:

- generate two or three distinct random numbers on the GPU, according to the mutation strategy;
- calculate an element of the donor vector from the population members randomly selected in the previous step;
- decide whether to include the donor or the parent element in the trial vector, based on the type of crossover and the crossover rate, Cr.

The second DE kernel evaluates all trial vectors simultaneously in shared memory and, if the fitness has improved, it replaces the parent with the offspring.

Clearly, SS is not as inherently parallel as the two other metaheuristics (see Figure 1). In SS a diverse population is first initialized and evaluated; diversity is simulated by generating uniform random values for each dimension over the whole search space. Then, to build the reference set R, a parallel sort operation is required to find R_1, followed by another kernel that calculates pairwise Euclidean distances between solutions in $P - R$ and R, sequentially adding the solutions that are farthest from the reference set for $|R_2|$ iterations. As for selection and crossover, a kernel selects all solution pairs in the reference set for mating, and combines them through the BLX-α crossover , generating two distinct solutions chosen with α set to $(0.5 + \lambda)$ and $(0.5 - \lambda)$, respectively. The combined solutions make up the *pool*, to which a parallel implementation of the Solis & Wets search method [19] is then applied as improvement method. For the last step, we compared two methods for updating the reference set, one of which considers both quality and diversity as in [5], while the other updates the reference set with the best $|R|$ solutions in $(R \cup pool)$. The latter yielded better results in terms of both speed and accuracy, as proven by the automatic tuning process described in the following section.

4 Experiments

We give for granted that the parallel metaheuristics we consider are faster than the corresponding sequential versions on sufficiently large problems, while their accuracy is the same for identical configurations, because they implement the same algorithm. Many comparisons between the accuracy of the sequential versions of the algorithms [5,18] have already been made, but they gave insights on their intrinsic features rather than on the computational efficiency of possible implementations. In this work, we evaluate both quality and speed of their parallel versions, analyzing the accuracy they can achieve in a limited amount of time, to assess the degree of parallelization that each of them allows to reach.

Table 1. Automatically-tuned parameter values used to test different optimization techniques

DE	PSO	SS
$Cr = 0.879$	$c_1 = 1.862$	$\|P\| = 140$
$F = 0.520$	$c_2 = 1.881$	$\|R_1\| = 9, \|R_2\| = 1$
Exponential Crossover	$w = 0.494$	$\lambda = 0.220$
Random Mutation	Population Size = 125	Solis & Wets iterations = 85
Size = 48		

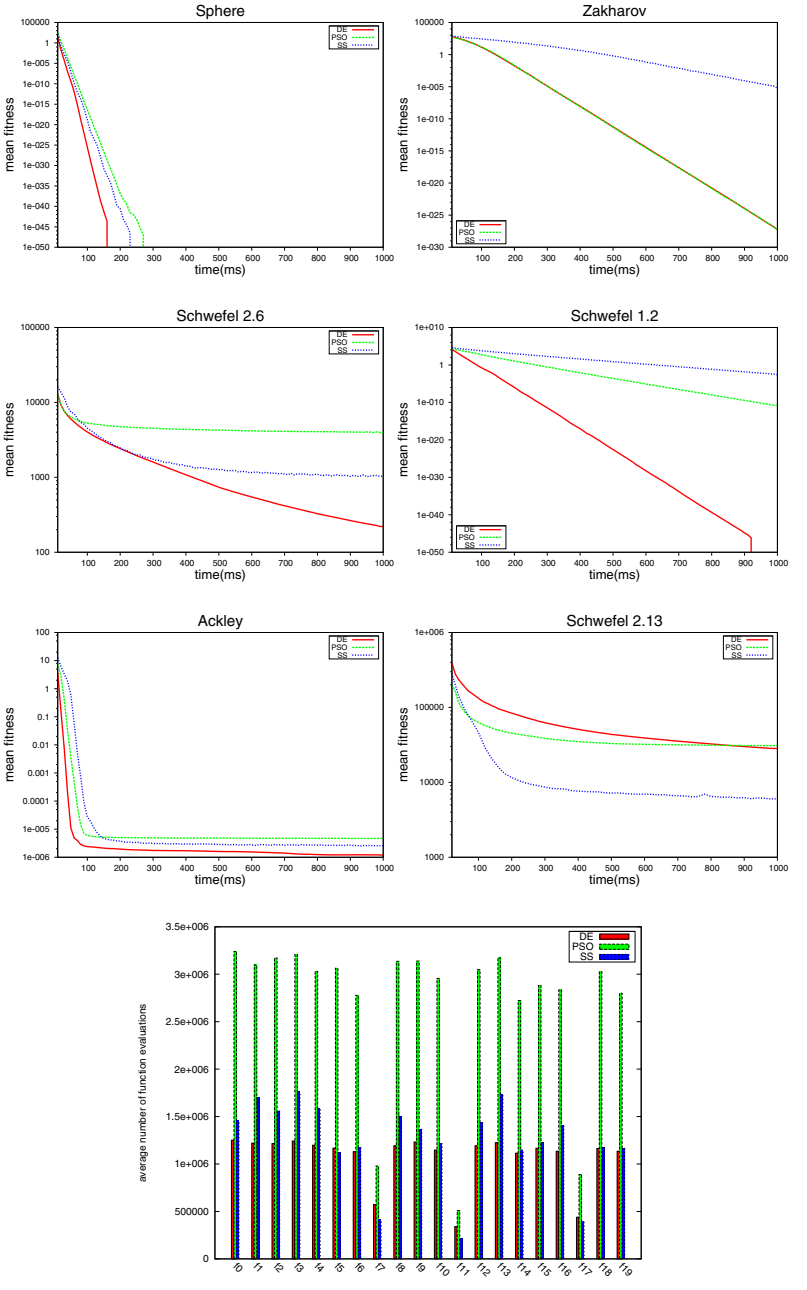

Fig. 2. Mean fitness vs time (up to 1 second) for six representative functions (unimodal separable, unimodal non-separable and multimodal non-separable), and number of function evaluations performed in 1 second by every method for each function on 30-dimensional problems.

Table 2. Results on the 20 functions

	10 dimensions						30 dimensions					
	DE		PSO		SS		DE		PSO		SS	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std	Avg	Std	Avg	Std
f_0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
f_1	0.0	0.0	0.0	0.0	2.5e-03	9.2e-03	0.0	0.0	0.0	0.0	2.2e-06	6.3e-06
f_2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.1e-44	8.6e-44
f_3	0.0	0.0	0.0	0.0	7.0e-45	2.9e-44	0.0	0.0	0.0	0.0	6.7e-05	3.0e-04
f_4	0.0	0.0	0.0	0.0	1.1e-25	1.7e-25	0.0	0.0	0.0	0.0	5.1e-24	2.7e-24
f_5	0.0	0.0	0.0	0.0	0.0	0.0	2.5e-28	3.4e-28	1.8e-28	2.2e-28	1.2e-05	1.7e-05
f_6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	9.7e-012	9.9e-012	2.5e-03	3.3e-03
f_7	9.8e-06	6.9e-05	1.0e-03	4.0e-04	2.7e-04	3.5e-04	2.1e+02	3.3e+02	3.8e+03	1.1e+03	9.9e+02	3.4e+02
f_8	5.0e-01	5.0e-08	3.5e-02	1.3e-01	2.9e-01	2.5e-01	5.0e-01	0.0	4.2e-01	1.9e-01	5.0e-01	2.1e-05
f_9	0.0	0.0	5.2e-01	7.8e-01	6.9e-01	9.1e-01	0.0	0.0	7.2e+01	1.6e+01	3.5e+01	9.7e+00
f_{10}	5.9e+00	3.1e+01	1.2e+02	1.2e+02	8.1e+01	1.1e+02	1.7e+01	4.8e+01	2.9e+03	4.1e+02	2.4e+03	9.0e+02
f_{11}	1.2e-03	4.9e-06	1.2e-03	2.9e-05	1.2e-03	2.4e-18	1.2e-02	5.1e-05	1.2e-02	1.8e-04	1.2e-02	8.7e-18
f_{12}	0.0	0.0	1.1e-02	1.0e-02	1.1e-03	2.9e-03	7.4e-05	7.4e-04	6.3e-10	6.0e-09	1.5e-03	4.3e-03
f_{13}	0.0	0.0	3.9e-07	4.8e-07	5.9e-01	3.3e+00	0.0	0.0	2.1e-01	7.2e-01	2.2e+01	2.6e+01
f_{14}	0.0	0.0	6.7e-07	1.1e-06	0.0	0.0	1.1e-06	1.2e-06	4.5e-06	9.4e-07	9.3e-03	9.3e-02
f_{15}	0.0	0.0	1.2e-03	6.5e-03	6.3e-31	4.4e-30	0.0	0.0	1.5e-28	1.2e-27	1.1e-27	2.7e-27
f_{16}	3.3e-02	2.9e-02	1.0e-01	2.4e-02	7.3e-01	5.4e-01	3.2e-01	3.0e-02	8.5e+00	8.6e-01	8.7e+00	1.2e+00
f_{17}	4.5e+01	2.2e+02	1.3e+00	5.1e+00	4.7e+00	8.5e+00	2.8e+04	6.1e+03	3.1e+04	1.8e+04	5.2e+03	5.6e+03
f_{18}	1.0e-01	2.8e-17	1.0e-01	2.8e-17	9.8e-02	1.4e-02	1.9e-01	3.1e-02	2.0e-01	1.7e-02	2.4e-01	5.1e-02
f_{19}	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.3e-02	1.0e-01

The algorithms we compared have a number of parameters that affect both accuracy and parallelism. "Manual" parameter tuning is time consuming and may introduce a bias in comparing an algorithm with a reference, due to better knowledge of the algorithm under consideration and to possible different time spent tuning each of them. Therefore, the automatic *tuning* of all three algorithms was performed using the `irace` software package [10], to find the configurations that yielded the best results in a given time: we set this time to one second, since it is generally short enough to avoid reaching full convergence with all three methods, allowing one to compare their short-term performances.

The tuner was run on all 20 functions with a budget of 30000 experiments, each being one run of one configuration on one function with a termination criterion of one second. Since the functions have different fitness ranges, a rank-based test is prferable to a test based on the solutions' mean values. Accordingly, the Friedman test was used to discard significantly worse configurations. We tuned the parameters for 30-dimensional problems, and assumed that such configurations are good also for lower-sized ones. Table 1 displays the parameters that have been tuned for each algorithm, and the best corresponding values.

We compared our results to the values that are most commonly used in literature. For instance, the authors in [2] suggest $F \in (0.4, 0.95)$ and $Cr \in (0.9, 1)$ for multimodal separable functions (the most common ones in our benchmark); we obtained similar results. Regarding PSO, in most papers, $c_1 = c_2 = 2.0$ [15], while our automatic tuning set them to slightly smaller values.

To evaluate both the effectiveness and the efficiency of the three parallel implementations, tests on 20 numerical benchmark functions (see Table 3) were run on a 64-bit Intel(R) Core^{TM} i7 CPU running at 2.67GHz using CUDA^{TM} v. 4.1 on a nVidia GeForce GTS450 graphics card with 1GB of DDR memory and compute capability 2.1 [14]. Table 2 reports the results obtained executing 100 runs per function (6000 independent runs) and setting 1 second as the only

Table 3. Benchmark functions. For every function, the table shows the name, the range of the search space, the formula, the multimodality (multimodal, unimodal) and the separability (separable, non separable). All minima are in $\{0\}^n$.

	Name	Range	Formula		
f_0	Sphere	$[-100, 100]^n$	$\sum_{i=0}^{n-1} x_i^2$	U	S
f_1	Elliptic	$[-100, 100]^n$	$\sum_{i=0}^{n-1} (10^6)^{\frac{i-1}{D-1}} x_i^2$	U	S
f_2	Sum of Squares	$[-1, 1]^n$	$\sum_{i=0}^{n-1} i x_i^2$	U	S
f_3	HyperEllipsoid	$[-1, 1]^n$	$\sum_{i=0}^{n-1} i^2 \cdot x_i^2$	U	S
f_4	Schwefel 2.22	$[-10, 10]^n$	$\sum_{i=0}^{n-1} \lvert x_i \rvert + \prod_{i=0}^{n-1} \lvert x_i \rvert$	U	S
f_5	Zakharov	$[-10, 10]^n$	$\left(\sum_{i=0}^{n-1} {x_i}^2\right) + \left(\sum_{i=0}^{n-1} 0.5 \cdot i \cdot {x_i}^2\right)^2 + + \left(\sum_{i=0}^{n-1} 0.5 \cdot i \cdot {x_i}^2\right)^4$	U	S
f_6	Schwefel 1.2	$[-100, 100]^n$	$\sum_{i=0}^{n-1} \left(\sum_{j=0}^{i} x_j\right)^2$	U	NS
f_7	Schwefel 2.6	$[-100, 100]^n$	$max\{\mathbf{A}_i\mathbf{x} - \mathbf{B}\}$, $i = 0, \ldots, n-1, \mathbf{x} = [x_0, \ldots, x_{n-1}]$, $\mathbf{A}_i, \mathbf{B}$ defined in [17].	U	NS
f_8	Dixon-Price	$[-10, 10]^n$	$(x_0 - 1)^2 + \sum_{i=1}^{n-1} \left(i \cdot \left(2{x_i}^2 - x_{i-1}\right)^2\right)$	U	NS
f_9	Rastrigin	$[-5.12, 5.12]^n$	$\sum_{i=0}^{n-1} \{x_i^2 - 10 \cdot \cos(2\pi x_i) + 10\}$	M	S
f_{10}	Schwefel 2.26	$[-500, 500]^n$	$418.9829 \cdot n + \sum_{i=0}^{n-1} \left(x_i \cdot \sin\sqrt{\lvert x_i \rvert}\right)$	M	S
f_{11}	Katsuura	$[-1000, 1000]^n$	$\prod_{i=0}^{n-1} \left(1 + (i+1) \sum_{k=1}^{d} round(2^k x_i) 2^{-k}\right) - 1$	M	S
f_{12}	Griewank	$[-600, 600]^n$	$\sum_{i=0}^{n-1} \frac{x_i^2}{4000} - \prod_{i=0}^{n-1} \cos(\frac{x_i}{\sqrt{i}}) + 1$	M	NS
f_{13}	Rosenbrock	$[-100, 100]^n$	$\sum_{i=0}^{n-1} 100(x_i - x_{i-1}^2)^2 + (1 - x_{i-1})^2$	M	NS
f_{14}	Ackley	$[-32, 32]^n$	$-20e^{-0.2\sqrt{\frac{1}{n}\sum_{i=0}^{n-1} x_i^2}} - e^{\frac{1}{n}\sum_{i=0}^{n-1} \cos(2\pi x_i)} + 20 + e$	M	NS
f_{15}	Griewank + Rosenbrock	$[-5.12, 5.12]^n$	$f_{griewank}(f_{rosenbrock})$	M	NS
f_{16}	Scaffer	$[-100, 100]^n$	$\sum_{i=0}^{n-1} F(x_i, x_{i+1}), x_n = x_0$ where $F(x, y) = 0.5 + \frac{\sin^2\left(\sqrt{x^2+y^2}\right) - 0.5}{1 + 0.0001(x^2+y^2)}$	M	NS
f_{17}	Schwefel 2.13	$[-\pi, \pi]^n$	$\sum_{i=0}^{n-1} \left(\mathbf{A}_i - \mathbf{B}_i(\mathbf{x})\right)^2, \mathbf{x} = [x_0, \ldots, x_{n-1}]$ $\mathbf{A}_i, \mathbf{B}_i(x)$ defined as in [17].	M	NS
f_{18}	Salomon	$[-10, 10]^n$	$-\cos\left(2\pi\sqrt{\sum_{i=0}^{n-1} {x_i}^2}\right) + 0.1\sqrt{\sum_{i=0}^{n-1} {x_i}^2} + 1$	M	NS
f_{19}	Levy	$[-10, 10]^n$	$\sin^2(\pi y_0) + \sum_{i=0}^{n-2} \left[(y_i - 1)^2 \left(10\sin^2(\pi y_i + 1)\right)\right] + (y_{n-1} - 1)^2 \left(1 + 10\sin^2(2\pi y_{n-1})\right)$ where $y_i = 1 + \frac{x_i - 1}{4}, i = 0, \ldots, n-1$	M	NS

termination criterion. The first column is the function under consideration. The following ones are divided into two blocks according to the number of dimensions (10 and 30). Within each block, the mean best fitness and the standard deviation over all runs are reported for each method. Results reported on a grey background highlight those case in which the median over 100 runs obtained by the method is significantly better than the other methods, according to the Kruskal-Wallis test, with a confidence level of 0.01.

5 Discussion

The results reported in Table 2 and Figure 2 allow one to draw some conclusions about the behaviour of the three parallel metaheuristics. Conforming with

previous results obtained by sequential implementations, DE obtained the best results, sometimes tied with some other method, in 35 out of the 40 experiments performed, while PSO was the best method, sometimes tied with some other method, in 20 out of 40 functions, its main drawback being its tendency to stagnate and find sub-optimal solutions more often than DE, even if a higher number of function evaluations is run. Regarding SS, whose first parallel implementation is presented here, it obtained the best result in 12 out of 40 problems; however, this metaheuristic, which is not as parallelizable as the other methods, as reflected by the number of kernels, has achieved better performance over multimodal non-separable problems and time-consuming fitness functions, like Katsuura.

All tests were run with a temporal limit of one second, a short time in which all three methods can generally obtain results close to the optima without reaching full convergence. Figure 2 shows that PSO requires almost three times as many fitness function evaluations as DE to converge on 30-dimensional problems. It is important to notice that the population size in PSO is also almost three times as large as in DE, which justifies the larger number of fitness evaluations. However, this may represent a shortcoming only if applied to larger-dimensional functions than those considered in this work.

Acknowledgments. Youssef S. G. Nashed, Pablo Mesejo and Jérémie Dubois-Lacoste are funded by the European Commission (Marie Curie ITN MIBISOC, FP7 PEOPLE-ITN-2008, GA n. 238819). Roberto Ugolotti is funded by Compagnia di S.Paolo and Fondazione Cariparma. The authors want to thank Thomas Stützle for his comments and Enrico Viappiani for the help in implementing CUDA^{TM} code.

References

1. Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm Intelligence: From Natural to Artificial Systems. Oxford (1999)
2. Das, S., Suganthan, P.: Differential Evolution: A Survey of the State-of-the-Art. IEEE Transactions on Evolutionary Computation 15(1), 4–31 (2011)
3. de Veronese, L., Krohling, R.: Swarm's flight: Accelerating the particles using C-CUDA. In: Proc. IEEE Congress on Evolutionary Computation, pp. 3264–3270 (2009)
4. de Veronese, L., Krohling, R.: Differential evolution algorithm on the GPU with C-CUDA. In: Proc. IEEE Congress on Evolutionary Computation, pp. 1–7 (2010)
5. Duarte, A., Martí, R., Glover, F., Gortázar, F.: Hybrid scatter tabu search for unconstrained global optimization. Annals of Operations Research 183(1), 95–123 (2011)
6. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer (2003)
7. Glover, F.: Heuristics for integer programming using surrogate constraints. Decision Sciences 8(1), 156–166 (1977)
8. Kennedy, J., Eberhart, R.: Particle Swarm Optimization. In: Proc. IEEE International Conference on Neural Networks, vol. 4, pp. 1942–1948 (1995)

9. Krömer, P., Snášel, V., Platoš, J., Abraham, A.: Many-threaded implementation of differential evolution for the CUDA platform. In: Proc. 13th Annual Conference on Genetic and Evolutionary Computation, GECCO 2011, pp. 1595–1602. ACM (2011)
10. López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T., Birattari, M.: The irace package, iterated race for automatic algorithm configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium (2011)
11. Mussi, L., Daolio, F., Cagnoni, S.: Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture. Information Sciences 181(20), 4642–4657 (2011)
12. Mussi, L., Nashed, Y.S.G., Cagnoni, S.: GPU-based asynchronous particle swarm optimization. In: Proc. 13th Annual Conference on Genetic and Evolutionary Computation, GECCO 2011, pp. 1555–1562. ACM (2011)
13. Nashed, Y.S.G., Ugolotti, R., Mesejo, P., Cagnoni, S.: libCudaOptimize: an Open Source Library of GPU-based Metaheuristics. In: Proc. Genetic and Evolutionary Computation Conference, GECCO 2012 (in press, 2012)
14. nVIDIA Corporation: nVIDIA CUDA Programming Guide v. 4.0. (2011)
15. Poli, R., Kennedy, J., Blackwell, T.: Particle swarm optimization. Swarm Intelligence 1(1), 33–57 (2007)
16. 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)
17. Suganthan, P.N., Hansen, N., Liang, J.J., Deb, K., Chen, Y., Auger, A., Tiwari, S.: Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization. Natural Computing, 1–50 (2005)
18. Vesterstrom, J., Thomsen, R.: A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In: Proc. IEEE Congress on Evolutionary Computation, pp. 1980–1987 (2004)
19. Wets, F.J., Solis, R.J.: Minimization by random search techniques. Mathematics of Operations Research 6(1), 19–30 (1981)
20. Zhou, Y., Tan, Y.: GPU-based parallel particle swarm optimization. In: Proc. IEEE Congress on Evolutionary Computation, pp. 1493–1500 (2009)
21. Zhu, W.: Massively parallel differential evolution–pattern search optimization with graphics hardware acceleration: an investigation on bound constrained optimization problems. Journal of Global Optimization 50(3), 417–437 (2011)