GPU-based Asynchronous Particle Swarm Optimization

Luca Mussi
Henesis s.r.l. & Dept. of Information Engineering
University of Parma - Italy
luca.mussi@henesis.eu

Youssef S. G. Nashed
Dept. of Information Engineering
University of Parma - Italy
nashed@ce.unipr.it

Stefano Cagnoni
Dept. of Information Engineering
University of Parma - Italy
cagnoni@ce.unipr.it

ABSTRACT

This paper describes our latest implementation of Particle Swarm Optimization (PSO) with simple ring topology for modern Graphic Processing Units (GPUs). To achieve both the fastest execution time and the best performance, we designed a parallel version of the algorithm, as fine-grained as possible, without introducing explicit synchronization mechanisms among the particles' evolution processes. The results we obtained show a significant speed-up with respect to both the sequential version of the algorithm run on an up-to-date CPU and our previously developed parallel implementation within the nVIDIA™ CUDA™ architecture.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search—*Heuristic methods*

General Terms

Algorithms, Performance

Keywords

Parallelization, Speed-up technique, Particle Swarm Optimization, Implementation

1. INTRODUCTION

Particle Swarm Optimization (PSO) is a simple but powerful optimization algorithm, introduced by Kennedy and Eberhart in 1995 [8]. PSO searches the optimum of a function, termed *fitness function*, following rules inspired by the behavior of flocks of birds looking for food. As a population based meta-heuristic, PSO has recently gained more and more popularity due to its robustness, effectiveness, and simplicity. Whatever the choices of the algorithm structure, parameters, etc., and despite good convergence properties, PSO is still an iterative stochastic search process, which, depending on problem hardness, may require a large number of particle updates and fitness evaluations. Therefore, designing efficient PSO implementations is a problem of great practical relevance. It is even more critical if one considers real-time applications to dynamic environments in which, for example, the fast-convergence properties of PSO may be used to track moving points of interest (maxima or minima of a specific dynamically-changing fitness function). Among these, we have previously presented a number of computer vision applications in which PSO has been used to track moving objects [13] or to determine location and orientation of objects or posture of people [6, 14, 16]. Some of these applications rely on the use of GPU multi-core architectures for general-purpose high-performance parallel computing, which have recently attracted researchers' interest more and more, especially after handy programming environments, such as nVIDIA™ CUDA™ [18], have been introduced. Such environments or APIs take advantage of the computing capabilities of GPUs using parallel versions of high-level languages which require that only the highest-level details of parallel process management be explicitly encoded in the programs. The evolution both of GPUs and of the corresponding programming environments has been extremely fast and, up to now, far from any standardization. Recently, the first implementations of OpenCL [10], an environment which will allow the development of parallel programs for most commercial GPUs using a common coding language, have been released. However, at present, not only performance of implementations based on different architectures or compilers, but even the same programs run on different releases of software-compatible hardware, are very hard to compare. Execution time is therefore often the only direct objective quantitative parameter on which comparisons can be based.

In this paper we discuss our design of a parallel asynchronous PSO algorithm, and we compare it to the previous implementations in terms of execution time and numerical efficiency. The next section provides a brief overview of the standard PSO algorithm and of the problems related with its parallelization. Section 3 reviews the latest research developments in PSO parallelization. A description of our previous synchronous and proposed asynchronous versions of the algorithm, alongside advantages and disadvantages of each implementation, is provided in Section 4. Finally, the results obtained on classical benchmark functions are summarized and compared in Section 5, with the concluding remarks presented in Section 6.

GECCO'11, July 12–16, 2011, Dublin, Ireland.

2. PSO BASICS

The core of PSO is represented by the two functions which update a particle's position and velocity within the domain of the fitness function at time t, which can be computed using the following equations:

$$\begin{aligned} \boldsymbol{V}(t) &= w\ \boldsymbol{V}(t-1)\ + \\ &\quad C_1\ R_1\ [\boldsymbol{X}_{best}\ (t-1) - \boldsymbol{X}(t-1)]\ + \\ &\quad C_2\ R_2\ [\boldsymbol{X}_{gbest}(t-1) - \boldsymbol{X}(t-1)] \end{aligned} \quad (1)$$

$$\boldsymbol{X}(t) = \boldsymbol{X}(t-1) + \boldsymbol{V}(t) \quad (2)$$

where $\boldsymbol{V}$ is the velocity of the particle, C_1, C_2 are two positive constants, R_1, R_2 are two random numbers uniformly drawn between 0 and 1, w is the so-called 'inertia weight', $\boldsymbol{X}(t)$ is the position of the particle at time t, $\boldsymbol{X}_{best}(t-1)$ is the best-fitness position reached by the particle up to time $t-1$ (also termed personal attractor), $\boldsymbol{X}_{gbest}(t-1)$ is the best-fitness point ever found by the whole swarm (social attractor). Despite its simplicity, PSO is known to be quite sensitive to the choice of its parameters. Under certain conditions, though, it can be proved that the swarm reaches a state of equilibrium, where particles converge onto a weighted average of their personal best and global best positions.

A study of particles' trajectory and guidelines for the choice of inertia and acceleration coefficients, in order to obtain convergence, can be found in [23] and work cited therein. Such work has been deepened and extended by Poli in [19]. Many variants of the basic algorithm have been developed [20], some of which have focused on the algorithm behavior when different topologies are defined for particles' neighborhoods [9]. A usual variant of PSO substitutes $\boldsymbol{X}_{gbest}(t-1)$ with $\boldsymbol{X}_{lbest}(t-1)$, which represents the 'local' best position ever found by all particles within a pre-set neighborhood of the particle under consideration. This formulation admits, in turn, several variants, depending on the topology of the neighborhoods. Among others, Kennedy and coworkers evaluated different kinds of topologies, finding that good performance is achieved using random and Von Neumann neighborhoods [9]. Nevertheless, the authors also indicated that selecting the most efficient neighborhood structure is, in general, a problem-dependent task. In the first implementations of PSO, particles were organized as a fully connected social network, best known as global-best topology. The PSO sequential implementation presently considered as Standard PSO [7] actually employs a stochastic star topology in which each particle informs a constant number K of random neighbors. The suggested default K value of 3 results in an algorithm which is not so different from the ones that rely on a classical and easily implementable ring topology, which is the topology we have chosen for our parallel implementation.

2.1 PSO parallelization

A main feature that affects the search performance of PSO is the strategy according to which the social attractor is updated. In 'synchronous' PSO, positions and velocities of all particles are updated one after another in turn during what, using evolutionary jargon somehow improperly, is usually called a 'generation'; this is actually a full algorithm iteration, which corresponds to one discrete time unit. Within the same generation, after velocity and position have been updated, each particle's fitness, corresponding to its new position, is evaluated. The value of the social attractor is only updated at the end of each generation, when the fitness values of all particles in the swarm are known.

The 'asynchronous' version of PSO, instead, allows the social attractors to be updated immediately after evaluating each particle's fitness, which causes the swarm to move more promptly towards newly-found optima. In asynchronous PSO, the velocity and position update equations can be applied to any particle at any time, in no specific order. Regarding the effect of changing the update order or allowing some particles to be updated more often than others, Oltean and coworkers [2] have published results of an approach by which they evolved the structure of an asynchronous PSO algorithm, designing an update strategy for the particles of the whole swarm using a genetic algorithm (GA) and showing empirically that the GA-evolved PSO algorithm performs similarly, and sometimes even better, than standard approaches for several benchmark problems. Regarding the structure of the algorithm, they also indicate that several features, such as particle quality, update frequency, and swarm size, affect the overall performance of PSO [3].

3. RELATED WORK

Recently, researchers have been extending the study of PSO in terms of new applications, new variants of the algorithm, and improvement of the overall performance and efficiency of the method.

PSO, like most population based optimization techniques, is inherently parallel. Parallel PSO seems to be the way to make practical use of this powerful search and optimization algorithm viable, in spite of its high computation cost. During the last decade, a considerable amount of literature about parallel PSO has been published. The first parallel PSO implementations relied on multiprocessor parallel machines or cluster computing systems [21, 24]. With the introduction of the GPUs, research shifted towards parallel PSO on the GPUs to alleviate multi-processor and cluster systems inefficiencies, such as network overhead, shared memory access, etc. Li et al. took advantage of GPU acceleration for developing parallel versions of PSO and GA through texture manipulation using shaders which are mainly used for graphics rendering purposes [12]. In 2009 de Veronese and Krohling developed the first implementation of PSO using nVIDIA™ CUDA™ [1].

Now that PSO can run efficiently on consumer-level graphics cards, researchers have experimented with new variants of the algorithm. Zhou and Tan extended the standard PSO to include the notion of 'unhealthiness' to describe swarms or sub-swarms stuck at local optima, then applying random mutations to the unhealthy particles' positions [26]. Also, Zhou and Curry created a hybrid between GPU PSO and pattern search to enhance the convergence of PSO [25].

Almost all recent GPU implementations assign one thread to each particle [1, 22, 25, 26] which, in turn, means that fitness evaluations have to be computed sequentially in a loop within each particle's thread. Since fitness calculation is often the most computation-intensive part of the algorithm, the execution time of such implementations is affected by the complexity of the fitness function and the dimensionality of the search domain. The speedup achieved by these

Listing 1: Sequential synchronous PSO

```
<Initialize positions/velocities of all particles>
<Set initial personal/global bests>
for(int i = 0; i < generationsNumber; i++)
{
        for(int j = 0; j < particlesNumber; j++)
        {
                <Evaluate the fitness particle j>
        }
        <Update the position of all particles>
        <Update all personal/global bests>

}

<Retrieve global best information to be returned as final result>
```

Listing 2: Sequential asynchronous PSO

```
<Initialize positions/velocities of all particles>
<Set initial personal bests>
for(int i = 0; i < generationsNumber; i++)
{
        for(int j = 0; j < particlesNumber; j++)
        {
                <Evaluate the fitness of particle j>
                <Update the position of particle j>
                <Update personal bests of particle j>
        }

}
<Calculate global best information to be returned as final result>
```

implementations is evaluated with respect to their sequential counterparts executing on the CPU.

In addition, state of the art research in GPU-based parallelization of PSO focuses on the synchronous version of the algorithm, while it was shown, on distributed or cluster systems, that asynchronous versions can achieve faster execution time without sacrificing numerical accuracy [11, 24]. The asynchronous GPU PSO we present in the following sections overcomes the shortcomings of asynchronous PSO enforced by the master-slave approach used in distributed systems implementations, while gaining good speedup when compared to our synchronous GPU implementation [15] as well as, obviously, to the standard sequential PSO implementation.

4. PARALLEL PSO FOR GPUS

As reported in Section 3, GPU implementations of PSO which assign one thread per particle, despite being the most natural way of parallelizing the algorithm, do not take full advantage of the GPU power in evaluating the fitness function in parallel. The parallelization only occurs on the number of particles of a swarm and ignores the dimensions of the function.

In our parallel implementations we designed the thread parallelization to be as fine-grained as possible; in other words, all independent sequential parts of the code are allowed to run simultaneously in separate threads. However, the performance of an implementation does not only depend on the design choices, but also on the GPU architecture, data access scheme and layout, and the programming model, which in this case is CUDA™. Therefore, it seems appropriate to outline the CUDA™ architecture and introduce some of its terminology.

CUDA™ is a programming model and instruction set architecture leveraging the parallel computing capabilities of nVIDIA™ GPUs to solve complex problems more efficiently than a CPU. At the abstract level, the programming model requires that the developer divide the problem into coarse sub-problems, namely *thread blocks*, that can be solved independently in parallel, and each sub-problem into finer pieces that can be solved cooperatively by all threads within the block [18]. From the software point of view, a *kernel* is equivalent to a high-level programming language function or method containing all the instructions to be executed by all threads of each thread block. Finally, at the hardware level, nVIDIA™ GPUs consist of a number of identical multi-threaded Streaming Multiprocessors (SM), each of which is made up of several cores that are able to run one thread block at a time. As the program invokes a kernel, a scheduler assigns thread blocks to SMs according to the number of available cores on each SM; the scheduler also ensures that delayed blocks are executed in an orderly fashion when more resources or cores are free. This makes a CUDA™ program automatically scalable on any number of SMs and cores.

The last thing to highlight is the memory hierarchy available to threads, and the performance associated with the read/write operations from/to each of the memory levels. Each thread has its own local *registers* and all threads belonging to the same thread-blocks can cooperate through *shared memory*. Registers and shared memory are physically embedded inside SMs and provide threads with the fastest possible memory access. Their lifetime is the same as the thread-block's. All the threads of a kernel can also access *global memory* whose content persists over all kernel launches [18]; however, read and write operations to global memory are orders of magnitude slower than those to shared memory and registers, therefore access to global memory should be minimized within a kernel.

The design and implementation issues of our algorithms are presented in the following sections.

4.1 Synchronous parallel PSO

The synchronous implementation [15] comprises three stages (kernels), namely: positions update, fitness evaluation, and bests update. Each kernel is parallelized to run a thread for each problem dimension. The function under consideration is optimized by iterating those kernels needed to perform one PSO generation. The three kernels must be executed sequentially and synchronization must occur at the end of each kernel run. Figure 1 better clarifies this structure. Since the algorithm is divided into three independent sequential kernels, each kernel must load all the data it needs initially and store the data back into global memory at the end of its execution. CUDA™ rules dictates that information sharing between different kernels is achievable only through the global memory.

To better understand the difference between synchronous and asynchronous PSO the pseudo-code of the sequential versions of the algorithms is presented in Listing 1 and Listing 2. Our synchronous 3-kernel implementation of CUDA-PSO, while allowing for virtually any swarm size, required synchronization points where all the particles data had to be saved to global memory to be read by the next kernel. This

frequent access to global memory limited the performance of synchronous CUDA-PSO and was the main justification behind the asynchronous implementation.

4.2 Asynchronous parallel PSO

The design of the parallelization process for the asynchronous version is the same as for the synchronous one, that is: we allocate a thread block per particle, each of which executes a thread per problem dimension. This way every particle evaluates its fitness function and updates position, velocity, and personal best for each dimension in parallel.

The main effect of the removal of the synchronization constraint is to let each particle evolve independently of the others, which allows it to keep all its data in fast-access local and shared memory, effectively removing the need to store and maintain the global best in global memory. In practice, every particle checks its neighbours' personal best fitnesses, then updates its own personal best in global memory only if it is better than the previously found personal best fitness. This can speed up execution time dramatically, particularly when the fitness function itself is highly parallelizable. This is a feature which often characterizes fitness functions which are commonly used in several applications, such as the squared sum of errors over a data set in classification tasks, or other fitness functions which can be expressed as a vector dot product or matrix multiplication.

In contrast to the synchronous version, all particle thread blocks must be executing simultaneously, i.e., no sequential scheduling of thread blocks to processing cores is employed, as there is no explicit point of synchronization of all particles. Two diagrams representing the parallel execution for both versions are shown in Figure 1. Having the swarm particles evolve independently not only makes the algorithm more biologically plausible, as it better simulates a set of very loosely coordinated swarm agents, but it also does make the swarm more 'reactive' to newly discovered minima/maxima. The price to be paid is a limitation in the number of particles in a swarm which must match the maximum number of thread blocks that a certain GPU can maintain executing in parallel. This is not such a relevant shortcoming, as one of PSO's nicest features is its good search effectiveness; because of this, only a small number of particles (a few dozens) is usually enough for a swarm search to work, which compares very favorably to the number of individuals usually required by evolutionary algorithms to achieve good performance when high-dimensional problems are tackled. This consideration makes the availability of swarms of virtually unlimited size and the deriving potential in terms of search capabilities less appealing than it could seem at first sight, while increasing the relevance of the burden imposed, in terms of execution time, by the sequential execution of fitness evaluation. Also, currently, parallel system processing chips are scaling according to Moore's law, and GPUs are being equipped with more processing cores with the introduction of every new model.

5. RESULTS

We compared the performance of the different versions of our parallel PSO implementation and the sequential SPSO one on a 'classical' benchmark which comprised a set of functions which are often used to evaluate stochastic optimization algorithms. Our goal was to compare different parallel PSO implementations with one another and with a sequential implementation, in terms of speed, while checking that the quality of results was not badly affected by the sequential implementation. So we kept all algorithm parameters equal in all tests, setting them to the 'standard' values suggested in [7]: $w = 0.729844$ and $C_1 = C_2 = 1.49618$. Also, for the comparison to be as fair as possible, we adapted the SPSO by substituting its original stochastic-star topology with the same ring topology adopted in the parallel GPU-based versions and we downgraded it to 'float' precision to match the GPU-based algorithms' precision.

Our parallel algorithms were developed using CUDA™ version 3.2. Tests were performed on two different graphic cards (see Table 1 for detailed specifications). For the following experiments the sequential SPSO was run on a PC powered by a 64-bits Intel(R) Core(TM) i7 CPU running at 2.67GHz. For the GTS450, a swarm of 32 particles was run for 10000 generations. On the GTX260 GPU we ran the same experiments on a swarm of 27 particles, as that is the maximum number of particles running simultaneously such a card can support.

As for random number generation, the CURAND library available from CUDA™ provides fast uniform random number generation on the GPU [17]. This library is available since the introduction of CUDA™ version 3.2. In [15], in developing and testing the synchronous 3-kernel version we had to use an external kernel which implemented the Mersenne-Twister random number generator. Switching to the new generator presently provided by nVIDIA™ does not seem to affect the search performance at all, while it allowed us to reduce slow accesses to global memory, in which the random number sequences generated by the Mersenne-Twister kernel had to be stored for use by the other kernels.

The following implementations of PSO have been compared: (1) the sequential SPSO version modified to implement a two nearest-neighbors ring topology; (2) the synchronous three-kernel version of *CUDA-PSO*; (3) *CUDA-PSO* implemented asynchronously with only 1 kernel. Values were averaged over the 98 best results out of 100 runs.

Figures 2 compares average execution times and average final fitness values obtained for problem dimension D ranging from 2 to 128 in optimizing fitness functions from typical test-beds for function optimization [5]. They can be found in the SPSO package [7] and in the Black Box Optimization Benchmark suite [4]. We tested our code on the following functions: (a) the simple Sphere function within the domain $[-100, 100]^D$, (b) the High Conditioned Elliptic function within the domain $[-100, 100]^D$, (c) Rastrigin function, on which PSO is known to perform well, within the domain $[-5.12, 5.12]^D$, (d) the Rosenbrock function, which is non-separable and thus hard to solve by PSO, within the domain $[-30, 30]^D$, and (e) the Griewank function within the domain $[-600, 600]^D$.

In general, the asynchronous version was much faster than the synchronous version, at the price of being able to run swarms up to 27 or 32 depending on the graphics card. As can be observed in the figures which report average best fitness values, the two GPU-based versions produced very similar results. The sequential implementation appears to yield better results on the sphere function, as fitness values approach the precision limit of the floating-point representation and problem dimensions grow. However, this is probably an artefact caused by the limited precision of the

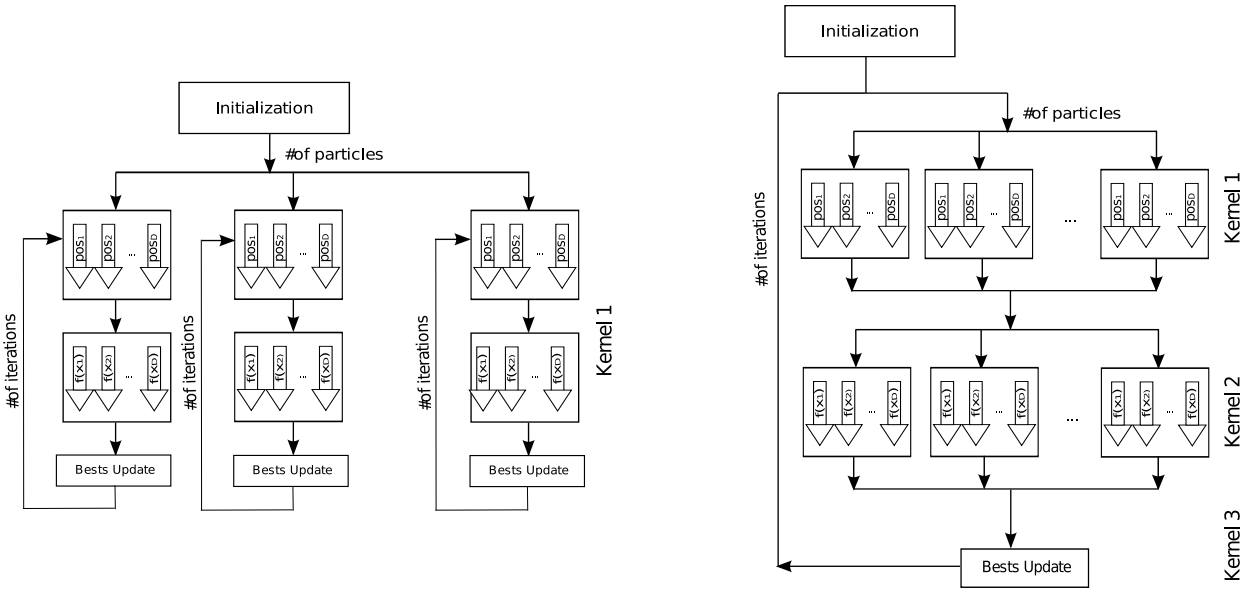

Figure 1: Asynchronous CUDA-PSO: particles run in parallel independently (left). Synchronous CUDA-PSO: particles evaluate fitness in parallel but have to wait the end of the generation before updating positions, velocities, and personal/global bests (right). Blocks represent particles and white arrows represent threads for each dimension of the search space.

floating-point representation; this may cause the 'accumulated' sequential sum of squares to start neglecting small terms as soon as it grows above a certain threshold, which is obviously more likely with high-dimensional problems. On the contrary, the parallel implementation, using a reduction to compute the sum of squares, is less affected by this problem, which causes the sequentially-computed fitness to be underestimated, since it gradually adds up terms pairwise: additions are therefore most often performed between terms of comparable magnitude. The same reason may also stand behind the unexpected behavior of the sequential algorithm regarding execution time, which appears to be non-monotonic with problem dimension, showing a surprising decrease as problem dimension becomes larger. In fact, code optimization (or the hardware itself) might lead several multiplications to be directly equaled to zero without even performing them, as soon as the sum of the exponents of the two factors is below the precision threshold; a similar though opposite consideration can be made for additions and the sum of the exponents. We actually verified that adding up terms all of comparable magnitude is much slower than adding the same number of terms on very different scales; what is more important, execution time linearly increases with the number of terms.

It is also worth noticing that the execution time graphs are virtually identical for the functions taken into consideration, which shows that GPUs are extremely effective at computing arithmetic-intensive functions, mostly independently of the set of operators used, and that memory allocation issues are prevalent in determining performance.

Taking speed-up values into consideration, one can also notice that the best performances were obtained on the Rastrigin and Griewank functions. This is probably due to the presence of complex math functions in their definition. In fact, GPUs have internal *fast math* functions which can provide good computation speed at the cost of slightly lower accuracy, which causes no problems in this case.

Table 1: Major technical features of the GPUs used for the experiments.

Model name	GeForce GTX260AMP[2]	GeForce GTS450
GPU clock (MHz)	650	783
Stream Multi Processors	27	4
CUDA cores	216	192
Bus width (bit)	448	128
Memory (MB)	896	1024
Memory clock (MHz)	2100	1804
Memory type	GDDR3	GDDR5
Memory bandwidth (GB/s)	117.6	57.7
CUDA compute capability	1.3	2.1

6. CONCLUSION

The new GPU-based version of our algorithm was able to significantly reduce execution time with respect to our previously-developed one, imposing limitations on the number of particles which seemed not to affect performances significantly, at least on the benchmark we used for tests. These, in any case, included high-dimensional versions of traditional benchmarks.

Depending on the degree of parallelization allowed by the fitness functions we considered, the asynchronous version of CUDA-PSO could reach speed-ups of up to about 300 (in the tests with the highest-dimensional Rastrigin functions) with respect to the sequential implementation, and often of more than one order of magnitude with respect to the corresponding GPU-based 3-kernel synchronous version, sometimes showing a limited, possibly only apparent, decrease of search performances.

Based on our previous experience, for example in the real-

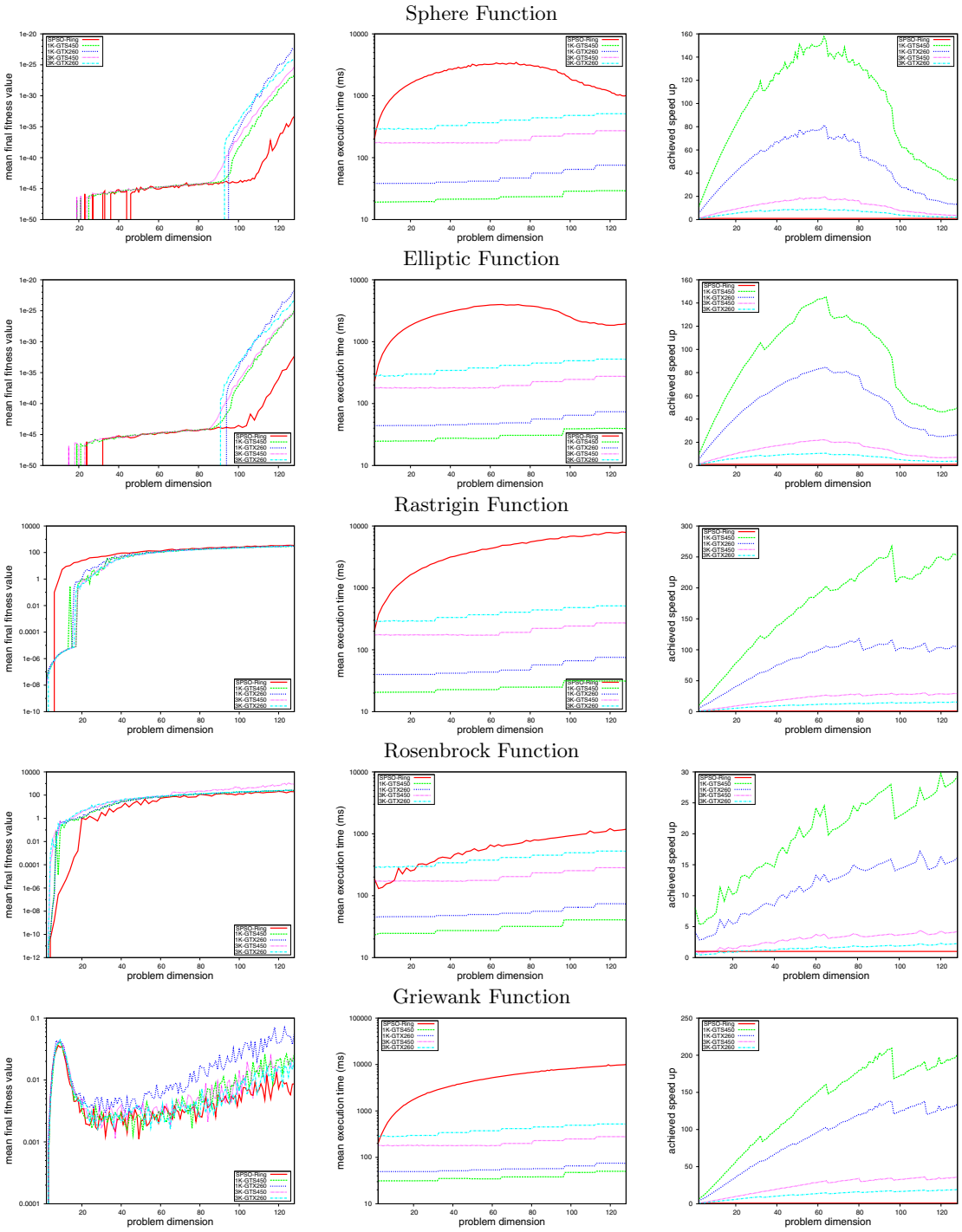

Figure 2: Average final fitness values (left column), average execution times (center) and speed-ups (right column) vs. problem dimension for the Sphere, High Conditioned Elliptic, Rastrigin, Rosenbrock and Griewank functions (top to bottom). Experiments were performed running one swarm of 32 particles (GTS-450) or 27 (GTX-260) for 10000 generations. Plotted values were averaged over the best 98 results out of 100 runs.

time computer vision application described in [14], the limitations that this asynchronous version imposes on swarm size do not prevent our GPU-based algorithms from being used to solve complex real-time problems. Real-time object detection applications usually require high detection accuracy, while often allowing for coarser localization. In Automatic Driving Assistance Systems, for instance, it is very important that most road signs be detected (especially, danger or obligation signs) rather than they be precisely localized in space. Therefore execution speed, which allows search to be repeated within the same frame, becomes more critical than accuracy in finding the optima. Algorithms like PSO, which exhibit good convergence properties while having more problems with search refinement, once the basin of attraction of an optimum is reached, are very well suited for these applications. In fact, the number of particles allowed by our most recent implementation is equal or larger than the swarm size which is normally used to perform such tasks. Future work will therefore include updating and extending the applications we have already developed using the synchronous version of CUDA-PSO, as well as developing new ones.

Other interesting developments may be offered by the availability of OpenCL, which will allow owners of different GPUs (as well as multi-core CPUs, which are also supported) than nVIDIA's to implement parallel algorithms on their own computing architectures. The availability of shared code which allows for optimized code parallelization even on more traditional multi-core CPUs will make the comparison between GPU-based and multi-core CPUs easier (and, possibly, fairer) besides allowing for a possible optimized hybrid use of computing resources in modern computers.

7. ACKNOWLEDGMENTS

Youssef S. G. Nashed is supported by the European Commission MIBISOC grant (Marie Curie Initial Training Network, FP7 PEOPLE-ITN-2008, GA n. 238819).

8. REFERENCES

[1] L. de P. Veronese and R. Krohling. Swarm's flight: Accelerating the particles using C-CUDA. In *IEEE Congress on Evolutionary Computation (CEC), 2009*, pages 3264–3270, May 2009.

[2] L. Dioşan and M. Oltean. Evolving the structure of the particle swarm optimization algorithms. In *European Conference on Evolutionary Computation in Combinatorial Optimization, EvoCOP'06*, pages 25–36. Springer Verlag, 2006.

[3] L. Dioşan and M. Oltean. What else is evolution of PSO telling us? *Journal of Artificial Evolution and Applications*, 1:1–12, 2008.

[4] N. Hansen, A. Auger, R. Ros, S. Finck, and P. Pošík. Comparing results of 31 algorithms from the Black-Box Optimization Benchmarking BBOB-2009. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*, pages 1689–1696, New York, NY, USA, 2010. ACM.

[5] N. Hansen, R. Ros, N. Mauny, M. Schoenauer, and A. Auger. PSO facing non-separable and ill-conditioned problems. Research Report RR-6447, INRIA, 2008.

[6] Š. Ivekovic, E. Trucco, and Y. Petillot. Human body pose estimation with particle swarm optimisation. *Evolutionary Computation*, 16(4):509–528, 2008.

[7] J. Kennedy and M. Clerc, 2 2006. http://www.particleswarm.info/Standard_PSO_2006.c.

[8] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proc. IEEE Int. conf. on Neural Networks*, volume IV, pages 1942–1948, Washington, DC, USA, 1995. IEEE CS Press.

[9] J. Kennedy and R. Mendes. Population structure and particle swarm performance. In *Proc. of the Congress on Evolutionary Computation - CEC*, pages 1671–1676, Washington, DC, USA, 2002. IEEE CS Press.

[10] Khronos Group. *The OpenCL Specification*, 1.0 edition, October 2009.

[11] B.-I. Koh, A. D. George, R. T. Haftka, and B. J. Fregly. Parallel asynchronous particle swarm optimization. *International Journal for Numerical Methods in Engineering*, 67:578–595, 2006.

[12] J. Li, D. Wan, Z. Chi, and X. Hu. An efficient fine-grained parallel particle swarm optimization method based on GPU-acceleration. *International Journal of Innovative Computing, Information and Control*, 3(6 B):1707–1714, 2007.

[13] L. Mussi and S. Cagnoni. Particle swarm for pattern matching in image analysis. In I. Serra, R.and Poli and M. Villani, editors, *Artificial life and evolutionary computation*, pages 89–98. World Scientific, Singapore, 2010.

[14] L. Mussi, S. Cagnoni, E. Cardarelli, F. Daolio, P. Medici, and P. Porta. GPU implementation of a road sign detector based on particle swarm optimization. *Evolutionary Intelligence*, 3(3-4):155–169, 2010.

[15] L. Mussi, F. Daolio, and S. Cagnoni. Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture. *Information Sciences*, 2010. In press.

[16] L. Mussi, S. Ivekovic, and S. Cagnoni. Markerless articulated human body tracking from multi-view video with GPU-PSO. In *9th International Conference on Evolvable Systems: From Biology to Hardware (ICES), 2010*, volume 6274 LNCS, pages 97–108, 2010.

[17] nVIDIA. *CUDA CURAND Library*. nVIDIA Corporation, August 2010.

[18] nVIDIA Corporation. *nVIDIA CUDA programming guide v. 3.2*, October 2010.

[19] R. Poli. Mean and variance of the sampling distribution of particle swarm optimizers during stagnation. *IEEE Trans. Evolutionary Computation*, 13(4):712–721, 2009.

[20] R. Poli, J. Kennedy, and T. Blackwell. Particle swarm optimization: an overview. *Swarm Intelligence*, 1(1):33–57, 2007.

[21] J. F. Schutte, J. A. Reinbolt, B. J. Fregly, R. T. Haftka, and A. D. George. Parallel global optimization with the particle swarm algorithm. *International Journal for Numerical Methods in Engineering*, 61:2296–2315, 2004.

[22] J. St.Charles, T. Potok, R. Patton, and X. Cui.

Flocking-based document clustering on the graphics processing unit. *Studies in Computational Intelligence*, 129:27–37, 2008.

[23] F. Van den Bergh and A. Engelbrecht. A study of particle swarm optimization particle trajectories. *Information Sciences*, 176(8):937–971, 2006.

[24] G. Venter and J. Sobieszczanski-Sobieski. A parallel particle swarm optimization algorithm accelerated by asynchronous evaluations. In *6th World Congresses of Structural and Multidisciplinary Optimization*, 2005.

[25] Y. Zhou and Y. Tan. GPU-based parallel particle swarm optimization. In *IEEE Congress on Evolutionary Computation (CEC), 2009*, pages 1493–1500, May 2009.

[26] Y. Zhou and Y. Tan. Particle swarm optimization with triggered mutation and its implementation based on GPU. In *Proceedings of the 12th Annual Genetic and Evolutionary Computation Conference (GECCO), 2010*, pages 1007–1014, 2010.