Computing and Informatics, Vol. 34, 2015, 725–745

EFFECT OF DATA LAYOUT IN THE EVALUATION TIME OF NON-SEPARABLE FUNCTIONS ON GPU

Miguel Cárdenas-Montes

CIEMAT
Department of Fundamental Research
Avda. Complutense, 40, Madrid, Spain
e-mail: miguel.cardenas@ciemat.es

Miguel A. Vega-Rodríguez

University of Extremadura
ARCO Research Group
Department Technologies of Computers and Communications
Escuela Politécnica, Campus Universitario s/n, Cáceres, Spain
e-mail: mavega@unex.es

Abstract. GPUs are able to provide a tremendous computational power, but their optimal usage requires the optimization of memory access. The many threads available can mitigate the long memory access latencies, but this usually demands a reorganization of the data and algorithm to reach the performance peak. The addressed problem is to know which data layout produces a faster evaluation when dealing with population-based evolutionary algorithms optimizing non-separable functions. This knowledge will allow a more efficient design of evolutionary algorithms. Depending on the fitness function and the problem size, the most suitable layout can be implemented at the design phase of the algorithm, avoiding later costly code or data layout redesigns. In this paper, diverse non-separable functions, such as Rosenbrock and Rana functions, and data layouts are evaluated. The implemented layouts cover main techniques to maximize the performance: coalesced access to global memory, intensive use of on-chip memory: shared memory and registers, and variable reuse to minimize the global memory transactions. Conclusions about the optimum data layout related to the characteristics of the fitness function and the problem size are stated. Besides, the conclusions ease the decision-making process for future implementations of other non-separable functions.

Keywords: Non-separable function, GPU performance, parallel evaluation, Rosenbrock function, Rana function

1 INTRODUCTION

The GPU performance critically depends on the use of various types of memories and how the data are transferred between them. Besides, these data transfers are tightly bound to the data layout and how they are accessed in global memory. This work analyses the most suitable layout for efficient evaluation of non-separable functions.

For non-trivial problems, executing the reproductive cycle of an evolutionary algorithm with high-dimensional individuals and large population requires large computational resources. In many cases the evaluation of the fitness function is the most costly operation.

When dealing with population-based algorithms, a parallelism arises naturally, since each individual of the population is an independent unit. This allows accelerating the execution by evaluating the individuals in parallel.

Three major parallel models for evolutionary algorithms might be distinguished [1]: the island a/synchronous cooperation model, the parallel evaluation of the population, and the distributed evaluation of a single solution. The parallel evaluation of the population is recommended when the evaluation is the most time-consuming part of the algorithm, as it might occur when using non-separable functions.

Nowadays parallel evaluation on GPU has become a common practice in parallel evolutionary algorithms. A set of examples will be shown in the Related Work section. Because data layouts are not bound to any particular algorithm, the optimal choice can be applied to any population-based evolutionary algorithm. For this reason, the analysis of data layouts to state the most suitable one for a fast parallel evaluation of non-separable functions is widely applicable.

With regard to non-separable fitness functions, the well-known non-separable function has been firstly tested – the Rosenbrock function (Equation (1)) and four tailored fitness functions (Equations (3)–(6)) with different computational intensities. They were formulated following the recommendations of construction of non-separable functions [19]. Finally, predictions for other non-separable functions are compared by using the Rana function (Equation (2)).

Regarding the strategies or layouts proposed, they were constructed including the techniques recommended to improve the performance of GPU codes [14, 5], such as: coalesced access to global memory, intensive use of on-chip memory: shared memory and registers, and reuse of variables and data locality to minimize the global memory transactions.

The conclusions stated from this work allow to design more productive evolutionary algorithm, since a priori the most efficient layout for the corresponding problem size can be implemented.

The rest of this paper is organized as follows: Section 2 summarizes the related work and previous efforts done. In Section 3.1, the fitness functions used in this article are briefly described. In Sections 3.2 and 3.3, the layouts are detailed. An overview of Fermi architecture is presented in Section 3.4. The results are displayed and analysed in Section 4. And finally, the conclusions are presented in Section 5.

2 RELATED WORK

Evolutionary computing has taken the advantage of the GPU appearance. Several works are published every year describing how evolutionary problems are accelerated by transferring the execution partially or totally to GPU. However, all these works focus on the problem and how to accelerate it. This purpose is achieved through a wide variety of improvements, but in most of cases the layout is fixed at the beginning of the problem without testing alternative layouts.

Few examples of this kind of works are: implementations of genetic algorithm [11, 12], there are also examples in accelerating learning systems [6], examples of general-purpose parallel implementations of particle swarm optimizer in GPGPU [20, 21, 13, 8, 7]; and implementations of differential evolution [18, 3].

Generally, authors modify the parameters of the algorithm, such as population size, mutation or crossover rates in genetic algorithms, or maximum velocity in particle swarm optimizers; also GPU configuration parameters such as number of blocks or number of threads per block. However, the initial data layout remains mostly unaltered along the optimization process. For this reason, works addressing a similar scope have not been found.

3 METHODOLOGY

3.1 Benchmark Functions

In general, the difficulty to find high-quality solutions and the evaluation time of the fitness functions in the evolutionary computing increase with the dimensionality of the individuals and the population size. However, there are other relevant factors related with a *separability* of the fitness function. A function can be declared as *separable* if the variables are independent. This kind of problems are easier to solve by using evolutionary algorithms since a variable can be optimized while the rest are kept unchanged. By using this mechanism iteratively, all of the variables can be easily optimized. Even more, separable functions are often readily solved by local search methods. This is the main reason why some authors argue that they should not be incorporated to test suites [19].

On the contrary, the so-called *non-separable* fitness functions cannot implement this mechanism, since the variables are not independent. A non-separable function is called *m-non-separable* if at least m variables are not independent. In the extreme

case, where not any variable is independent of the others, the function is called *fully-non-separable*.

In evolutionary computing, specially in the benchmark functions used in continuous optimization contest [16, 17], there exist well-known fully-non-separable functions as Rosenbrock (Equation (1)) [15] or Rana function (Equation (2)).

$$f_{Rosenbrock} = \sum_{i=1}^{D-1} 100 \cdot \left[\left(x_i^2 - x_{i+1} \right)^2 + (x_i - 1)^2 \right], \tag{1}$$

$$f_{Rana} = \sum_{i=1}^{D-1} (x_{i+1} + 1.0) \cdot \cos(t_2) \cdot \sin(t_1) + \cos(t_1) \cdot \sin(t_2) \cdot x_i,$$
$$\text{where } t_1 = \sqrt{|x_{i+1} + x_i + 1.0|} \text{ and } t_2 = \sqrt{|x_{i+1} - x_i + 1.0|}. \tag{2}$$

On the other hand, functions specially designed for this work are also employed. These functions allow a finer-grained control over the computational intensity of the benchmark function. For example, expensive and non-expensive calculation functions (Equations (3), (4)) will allow an in-detail characterization of the behaviour of data layouts.

These functions were constructed simulating the Rosenbrock function: incorporating a fixed number of dimensions to calculate each term of the fitness function. The recommendations for constructing this type of functions have been followed in order to compose the new non-separable functions [19].

$$f_{2-\text{light}} = \sum_{i=1}^{D-1} (x_i + x_{i+1})^2, \tag{3}$$

$$f_{2-\text{heavy}} = \sum_{i=1}^{D-1} \log \sqrt{\log\left(\frac{1}{\sqrt{x_i}}\right) + \log\left(\frac{1}{\sqrt{x_{i+1}}}\right)}, \tag{4}$$

$$f_{4-\text{light}} = \sum_{i=1}^{D-3} (x_i + x_{i+1} + x_{i+2} + x_{i+3})^2, \tag{5}$$

$$f_{4-\text{heavy}} = \sum_{i=1}^{D-3} \log \sqrt{\log\left(\frac{1}{\sqrt{x_i}}\right) + \log\left(\frac{1}{\sqrt{x_{i+1}}}\right) + \log\left(\frac{1}{\sqrt{x_{i+2}}}\right) + \log\left(\frac{1}{\sqrt{x_{i+3}}}\right)}. \tag{6}$$

3.2 Strategies

3.2.1 Strategy 1: Allocation of One Individual per Thread in Registers

In the two first presented strategies (S1 and S2) each individual is handled by a single thread. Both strategies exploit the fast access to on-chip memory. In S1, registers are used to accumulate the intermediate fitness values and coordinates of the individuals, whereas in S2 the shared memory is used instead of registers. Besides, in both strategies the input array is ranged as a sequence of individuals: firstly all the

coordinates of the first individual, then all the coordinates of the second individual, and so on.

In S1, each single thread executes a for-loop sequentially over the coordinates of one individual. At the beginning of the sequence, the first partial fitness values are necessary to be calculated. Next, one coordinate is read ahead to produce a new part of the fitness which is gathered over the previous value of the accumulated partial fitness.

In each step, only a new coordinate is downloaded from global memory to on-chip memory (register for S1 and shared memory for S2). The remaining values, necessary to calculate the partial fitness, were loaded and stored in the on-chip memory in previous steps. This schema saves global memory accesses, replacing them by faster on-chip memory accesses. The number of saved accesses per cycle and individual depends on the number of necessary values to calculate a partial fitness value.

A priori this strategy would be more suitable for the evaluation of large populations, because each individual evaluation is independent of the others, and therefore, a large parallelism degree is reached. For small populations of high-dimensional individuals, this strategy is penalized because few threads are mobilized (threads as individuals), and therefore a few streaming multiprocessors (SM) are active.

Regarding the limitations of the approach, it can be foreseen that this strategy is constrained by the total number of threads that the GPU can allocate. However, this number is high enough to allow fast evaluation of large size problems. A second limitation could be the consumption of on-chip memory when evaluating fitness functions requiring many coordinates of the individual. For example, Rosenbrock function only requires two coordinates, whereas the tailored functions require two and four respectively. In this case, the thread-block has to allocate a high number of registers to hold all of the necessary values to calculate the partial fitness.

Finally, an important drawback is envisaged for the S1 strategy. A conjunction between the data layout (a sequence of individuals) and the sequential access to the dimensions of each individual create stride pattern access to the global memory, which is pernicious for the performance.

Global memory is always read in chunks of 128 bytes (length of a cache line) by 32 consecutive threads[1]. If a part of a cache line reading involves not-necessary data for the calculations or simply not all necessary data for the calculation because the 32 threads fill the cache line, then global memory bandwidth is being wasted. As a consequence, the bus transactions are increased to complete the necessary data for the calculations.

In S1 and S2, when dimensionality is equal or higher than 32, due to the disposition of the individuals, only one float is valid (from the 128 bytes read) for the 32 threads involved. Therefore, to read the 32 dimensions for the 32 consecutive threads, 32 bus transactions are necessary. As can be evaluated, most of the memory bandwidth is misused.

[1] A particular case in Fermi architecture exists which modifies this feature (Section 3.4).

When dimensionality is lower than 32, more than one float is valid in each bus transaction, but still a part of the global memory bandwidth is underused. If dimensionality is 16, then only 2 data are valid in the transaction; and, if dimensionality is 8, then only 4 data, and so on. This cache trashing severely degrades the performance. In the following strategies S3 and S4, two alternatives to circumvent the stride access are presented[2].

In spite of the above mentioned disadvantages, it is important to emphasize the capacity of this strategy to cope with extremely large problem sizes. For this reason, and because it is one of the most intuitive layout when adapting to GPU evolutionary problems, this strategy is considered to be evaluated.

3.2.2 Strategy 2: Allocation of One Individual per Thread on Shared Memory

This second strategy is very similar to the previous one but using shared memory to support a stencil instead of registers. When implementing a stencil in shared memory, some accesses to global memory are already saved. Similarly to S1, some necessary coordinates to calculate the partial fitness were previously downloaded and stored from global memory into the shared memory. In order to hold these values on the shared memory, some arrays have to be defined. These arrays have the same role as the variables defined in registers in S1.

Both S1 and S2 handle the individuals in parallel, but sequentially their dimensions. A priori this is more suitable for large populations than for small populations of high-dimensional individuals.

This strategy benefits from the absence of divergence in the warps, as well as a significant reduction of global memory transactions. On the contrary, it suffers from limitations arising from the shared memory consumption, similarly to the register consumption in S1[3].

Finally, the most important drawback in S1 and S2 is the non-coalesced access to global memory. In the next strategies two alternative layouts are presented. They introduce the appropriate modifications in order to get coalesced access to global memory.

3.2.3 Strategy 3: Allocation of One Individual Per Thread-Block on Share Memory with Coalesced Access to Global Memory and Atomic Operations

In this third strategy, the main difference holds on how the individual is managed: each individual is handled by a single thread-block, instead of a single thread as in the previous strategies. This layout forces to select the number of threads per block

[2] In Fermi architecture L1 and L2 cache memory can mitigate partially this penalization.

[3] In the numerical experiments, the configuration with maximal shared memory (48 kB) is used.

equal to the dimensionality of the individual. This constraint indicates a weakness in the strategy: the maximum threads per block allowed in the GPU[4] is the maximum dimensionality that this strategy can evaluate. Even though the highest allowed dimension for the individuals is the maximum threads per block, a reduction of this constraint is expected by the progressive increment of this value in future GPU architectures.

Although there exist techniques to deal with individuals with a larger dimensionality than the maximum number of threads per block[5], these techniques are not considered in this work.

Since the individuals are disposed sequentially (as in S1 and S2), each thread accesses to some dimensions of an individual to calculate the partial fitness. These dimensions are accessed in a coalesced mode: consecutive threads access to consecutive global memory directions. After getting the necessary data, each thread calculates a chunk of the fitness values in parallel, and stores it in the shared memory for the later reduction.

Oppositely to the previous strategies, in S3 the calculation of the fitness function is executed in parallel, not only among the individuals, but also for a partial fitness of each individual. This is an advantage in relation to the previous strategies.

For the final reduction of the partial values of the fitness of the individual, two alternatives are considered:

- Folding of the array with a partial fitness by half successively up to accumulating the addition of all the values over the first element of the array. This reduction technique has a relevant advantage in a high-degree of parallelism reached; however it also has an important drawback: it is only valid for 2^n dimensional individuals. Individuals with different dimensionality from 2^n have to be filled in with null values to reach the next 2^n value. Whereas the objective is to present the most general implementation with the widest applicability, this reduction technique is dismissed.

- Use of atomic operations, concretely *atomicAdd()*. Although the atomic operations should produce a degradation of the performance, this can be mitigated by parallelizing the reduction with atomic operations for the individuals on shared memory.

[4] The maximum number of threads per block in pre-Fermi architecture is 512, whereas in Fermi architecture is 1 024.

[5] The techniques able to deal with larger individuals than the maximum number of threads per block are mainly two: to spread out the individual over more than a single thread-block including halo coordinates and later to use global memory to gather the partial fitness values of each individual, or to treat the individual in chunks of the maximum number of threads per block.

3.2.4 Strategy 4: Allocation of One Individual Per Thread on Registers with Coalesced Access to Global Memory

In this last strategy, the S1 strategy is modified to mitigate the penalty of the stride global memory access. In order to overcome the non-coalesced access, a transposition of the input data is performed previously to transfering them to global memory. This transposition modifies the disposition of the elements from a sequence of individuals to a sequence of the same coordinate of the individuals: firstly, the first coordinate of all individuals, then the second coordinate of all individuals, and so on.

By using the present disposition, 32 consecutive threads access to 32 consecutive floats (128 bytes) to obtain the value of a particular dimension of 32 individuals. As a result the access to global memory becomes coalesced and the length of the cache line is fully used.

Alike in S1, in S4 a single thread deals with an individual. Therefore, the individuals are handled in parallel but their dimensions are still sequentially managed. The other benefits and drawbacks of the S1 strategy are still valid for S4.

By comparing S3 and S4, two different ways to gain coalesced access to global memory are implemented. In S3 the calculations are modified to adapt them to the data layout; whereas, in S4 the modification is performed over the data layout.

3.3 Sequential Evaluation

For the sake of completeness, a purely sequential evaluation is also implemented, and its results are used in the comparisons. Considering the non-negligible cost of the data transfer between CPU and GPU, for reduced problem sizes, CPU evaluation is expected to be faster than the operations' set: transfer from CPU to GPU of population data, evaluation of individuals, and retrieval of the fitness data from GPU to CPU. Comparisons with sequential evaluation discern which problem sizes are more suitable for GPU or CPU evaluation.

All the numerical experiments have been executed in a computer with Intel Xeon E5520 processor at 2.27 GHz and 6 GB of memory, and NVIDIA C2075 (Fermi architecture).

3.4 Overview of Fermi Architecture

Although a CUDA kernel is ensured to be executed correctly on any CUDA device, its performance can vary depending on how the code is adapted to a particular architecture. There are several common optimization strategies in order to make an efficient use of the hardware although the details of these hardware-dependent strategies can change with each architecture.

This work focusses on data layout and how it affects the evaluation time of the population. However, the data layout is tightly bound to the global memory access pattern of threads, secondly, to the shared memory size which limits the maximum problem size allowed to be evaluated, and to the memory transaction

segment size which is a critical factor on how the bandwidth is used. Furthermore, L1 cache memory size has a relevant role when data access pattern is unpredictable or irregular.

All the previous features differ depending on the architecture of the CUDA device, and for this reason they have to be taken into account in this work.

First of all, in pre-Fermi architecture each SM had only 16 kB for registers, whereas in Fermi architecture this on-chip memory has grown up to 32 kB [9]. Other feature that has been incremented in the Fermi architecture is the maximum number of threads per block, from 512 to 1 024. These two features constitute a major limiting factor for the largest problem of allocatable size in the layouts.

Second, Fermi introduces a two-level transparent cache memory hierarchy. Each SM has 64 kB of on-chip memory, distributed between shared memory and L1 cache memory. Users can select diverse configurations of shared memory and L1. Through configuring shared memory size, limitations to the largest problem of allocatable size are introduced.

When implementing coalesced or non-coalesced access to global memory, the memory transaction segment size becomes an important factor in the final performance. In the pre-Fermi architecture the available memory transaction segment sizes are: 32, 64 and 128 bytes. A selected value depends on the amount of memory needed and the memory access pattern. The selection is automatic in order to avoid bandwidth wasting.

In the Fermi architecture, the memory transaction segment size follows a different rule. When L1 cache memory is enabled, the hardware always issues segment transactions of 128 bytes, the cache-line size; otherwise, 32 bytes segment transactions are issued. In our study, default configuration of L1 is enabled in all numerical experiments.

4 RESULTS AND ANALYSIS

4.1 Rosenbrock Function

In this section, we discuss the performance achieved by each strategy when evaluating the Rosenbrock function. The execution times (mean and standard deviation) of the strategies for several configurations are presented in Table 1. The problem sizes were selected for representing the state-of-the-art problem sizes, and mapping the transitions between the most suitable strategies. Most of the configurations were selected with a population equal to dimensionality. Where an unequal configuration is chosen, the objective is to map the transition between two best strategies, narrowing the uncertainty border.

For each problem size, diverse threads-per-block configurations are executed. From the execution times obtained, the best results for each strategy are retained and presented. For example, for 100×100 configuration, two configurations of threads per block are tested: 32 and 64; whereas, for $2\,000 \times 2\,000$, four configurations are tested: 128, 256, 512, and 1 024 threads per block. In order to fairly compare, for the

sequential strategy the equal number of executions are produced, and then retaining the best one. Furthermore, for each case a total of 20 tries are executed.

On the other hand, input data are randomly generated in the range $(0, 1)$. Identical input files are used for all the executions of a particular configuration.

Pop. × Dim.	S1	S2	S3	S4	Sequential
100 × 100	0.237±0.005	0.255±0.003	0.145±0.004	0.170±0.003	**0.026±0.001**
200 × 200	0.444±0.004	0.484±0.002	0.365±0.006	0.314±0.002	**0.098±0.003**
1 000 × 1 000	3.188±0.014	3.389±0.012	NAN	2.481±0.019	**2.432±0.041**
500 × 2 000	4.733±0.023	5.129±0.067	NAN	3.439±0.045	**2.422±0.029**
2 000 × 500	2.558±0.013	2.549±0.020	NAN	**1.962±0.071**	2.431±0.048
1 500 × 1 500	6.218±0.163	5.907±0.106	NAN	**4.651±0.018**	5.620±0.029
2 000 × 2 000	9.723±0.075	9.685±0.135	NAN	**7.533±0.110**	10.129±0.032
1 000 × 4 000	12.129±0.233	13.093±0.092	NAN	**9.539±0.253**	10.386±0.025
4 000 × 1 000	8.723±0.020	10.030±0.041	NAN	**6.633±0.067**	10.143±0.035
4 000 × 4 000	22.426±0.034	22.431±0.064	NAN	**22.425±0.049**	40.856±0.021
2 000 × 8 000	39.284±0.067	40.110±0.038	NAN	**30.369±0.101**	40.552±0.107
8 000 × 2 000	33.076±1.202	NAN	NAN	**26.601±0.069**	40.536±0.083

Table 1. Mean execution time and standard deviation for Rosenbrock function

The results in Table 1 provide information about which strategy is the most suitable. Mildly speaking, sequential evaluation is the fastest strategy for the evaluation of the Rosenbrock function if population and dimensionality are up to 1 000. For these configurations, the cost of data transfer between CPU and GPU penalizes the GPU implementations. This penalization is not balanced by a parallel, and therefore faster, evaluation of the population.

However, when increasing problem sizes, the evaluation time of the sequential strategy grows dramatically, no-longer being the best option. For problem sizes equal or larger than $1\,500 \times 1\,500$, the penalization due to the data transfer between CPU and GPU is counteracted by a faster evaluation.

When dealing with large problem sizes, the S4 strategy outperforms all of the other strategies. In this strategy, the coalesced access to data in global memory produces an efficient usage of the bandwidth. On the other hand, the a priori flaw of the sequential treatment of the coordinates of each individual in S4 becomes a robust feature when evaluating very high-dimensional individuals. In this case the on-chip memory consumption is still moderated, because few variables per individual have to be simultaneously allocated on registers.

On the contrary, for these very high-dimensional individuals, the maximum number of threads per block becomes a major limiting factor in S3. This does not allow to evaluate individuals larger than 1 024 in the Fermi architecture (512 for the pre-Fermi architecture). Even more, $1\,000 \times 1\,000$ and higher configurations are not evaluable by S3 due to the depletion of the shared memory.

This depletion of the shared memory is the reason why $8\,000 \times 2\,000$ configuration is not allocatable on S2 strategy. It demonstrates that strategies implementing

registers as a support for the intermediate data are more robust than strategies implementing shared memory.

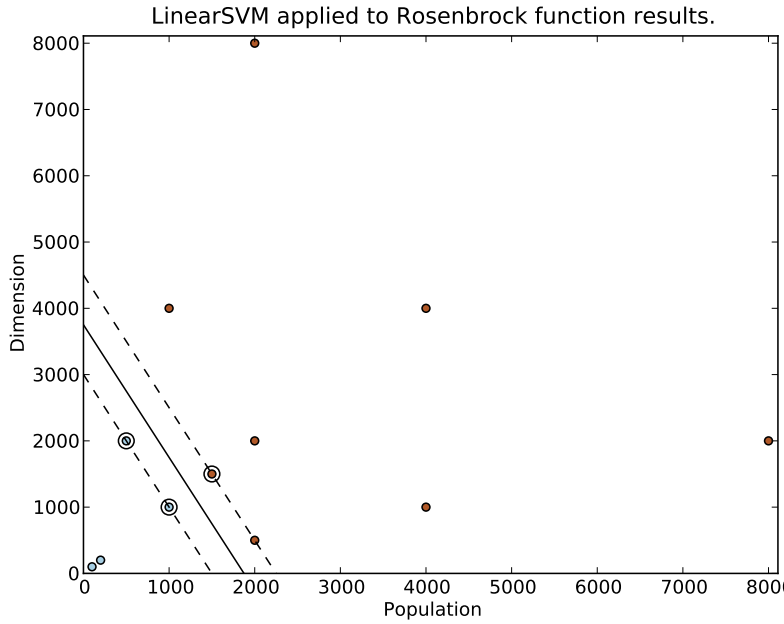

Figure 1. LinearSVM applied to the results of Rosenbrock function: the two data categories correspond to sequential evaluation for small and mid-size configurations, and S4 for large configurations. The support vectors for the first class (sequential evaluation) are $1\,000 \times 1\,000$ and $500 \times 2\,000$; whereas for the second class (S4) is $1\,500 \times 1\,500$.

In Figure 1, the results in Table 1 were analysed using Linear Support Vector Machine[6] (LinearSVM) [2]. The use of SVM allows the knowledge extraction from large numerical data sets. Through building a model with SVM, patterns in data can be inferred, in order the model is more comprehensible than the numerical output [4]. If data are linearly separable, then LinearSVM is the simplest SVM classification model. Particularly, the SVM models were produced by using scikit-learn API [10].

The application of LinearSVM to the numerical results allows showing the area (population size and dimensionality of individuals) where a particular data layout is the most suitable one, as well as the location of the borders between the areas.

Figure 1 shows a clear distinction between the suitable configurations for sequential evaluation and the suitable configurations for GPU evaluation. The maximum-

[6] It is assumed that the data are linearly separable.

margin hyperplanes provide a visual estimation for the borders between the two data classes: sequential evaluation for small and mid-sized configurations, and S4 for large configurations. The support vectors pinpoint the configurations limiting the data categories: $1\,000 \times 1\,000$ and $500 \times 2\,000$, for sequential strategy; and $1\,500 \times 1\,500$ for the S4 strategy.

Summarizing, decision-making about the most suitable evaluation strategy for the Rosenbrock function can be easily adopted by using Figure 1.

4.2 F2-Light and F2-Heavy

The Rosenbrock function constitutes an excellent example of a non-separable function. However more functions are needed if the target is to characterize the problem behaviour based on how the data layouts are presented. For this reason, four more functions are constructed and their results analysed in this section and the following one.

The first function, $f_{2-\text{light}}$, is a Rosenbrock-like function with a non-expensive calculation (Equation (3)), whereas the second one, $f_{2-\text{heavy}}$ (Equation (4)), is similar to the previous function, but replacing the very light inner calculations of the function by expensive operations from the computational point of view. Constructing these tailored functions the computational intensity assigned to each thread is under control. Similar to Rosenbrock, both functions involve only two consecutive coordinates of the individuals.

The main difference between $f_{2-\text{light}}$ and $f_{2-\text{heavy}}$ is on the computational intensity supported by each thread to complete a partial fitness calculation. Because of this difference, each function results in a different relation between the access to data on global memory and the life-time of data on-chip memory. For the light version, the accesses to data in global memory per time unit are much higher than in the expensive version. For the expensive version, the calculation takes longer, and therefore, data should reside on the on-chip memory during a longer period. Consequently, the access to global memory per unit of time becomes more sparse.

The results of this study are presented using a LinearSVM plot: Figure 2 for $f_{2-\text{light}}$, and Figure 3 for $f_{2-\text{heavy}}$. The different results show the tendency for other non-separable functions when increasing its computational intensity.

The $f_{2-\text{light}}$ function has a lower computational intensity than Rosenbrock function, and for this reason, much larger problem sizes are the only suitable for GPU evaluation. The support vectors (Figure 2) indicate that the limiting configurations are: $4\,000 \times 1\,000$ and $1\,000 \times 4\,000$ for the largest suitable configuration for CPU evaluation, and $4\,000 \times 4\,000$ for the lowest suitable configuration for GPU evaluation. Similarly to the Rosenbrock function, S4 for the large-configurations case is the appropriate strategy.

By comparing the plots for $f_{2-\text{light}}$ and $f_{2-\text{heavy}}$, and both of them comparing with Rosenbrock one, the most significant deviation appears for $f_{2-\text{heavy}}$. In this function, due to its high-computational intensity a strong reduction of the config-

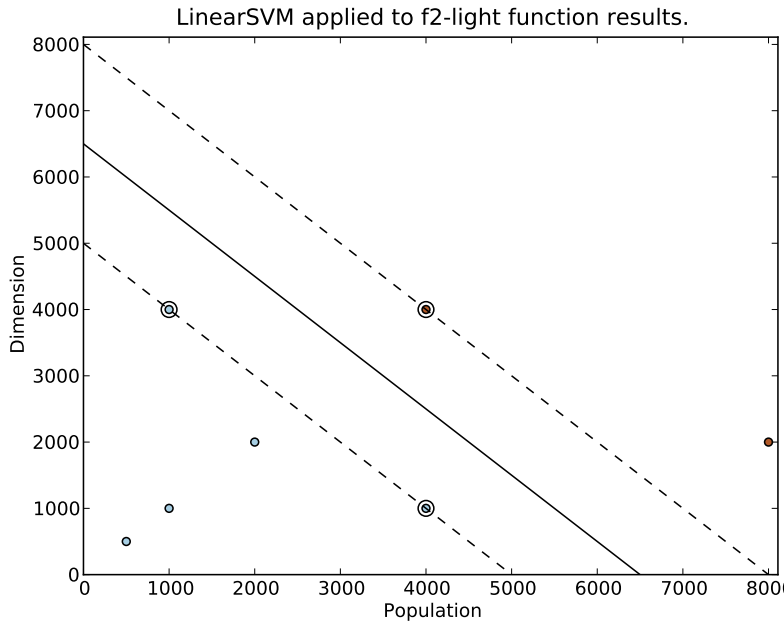

Figure 2. LinearSVM applied to the results of $f_{2-\text{light}}$ function: the two data categories correspond to sequential evaluation for small and mid-size configurations, and S4 for large configurations. The support vectors for the first class (sequential evaluation) are $4\,000 \times 1\,000$ and $1\,000 \times 4\,000$; whereas for the second class (S4) is $4\,000 \times 4\,000$.

urations where sequential evaluation is the most suitable option is observed. Only very small configurations, up to 16×16, are suitable for sequential evaluation. When increasing the problem size, the evaluation becomes more suitable for GPU in all cases. Moreover, since $f_{2-\text{heavy}}$ is more computational intensive than $f_{2-\text{light}}$, the penalization of data transfer between CPU and GPU is largely counteracted by a faster evaluation of the population.

Considering only the GPU implementations, two dominance areas appear in the plot[7] (Figure 3). In the inner one, the most suitable strategy is S3, whereas in the outer one it is S4. Therefore, LinearSVM for $f_{2-\text{heavy}}$ presents three optimal strategies depending on the configuration. For the very small problem sizes the most suitable strategy is sequential evaluation, being the support vector 16×16. When increasing the problem size, the most suitable strategy becomes S3. In this case the

[7] In the original implementation, SVM is only available for binary classification. Although there exist multiclass SVM implementations, in this work two simple and consecutive binary classifications are executed with two consecutive classes.

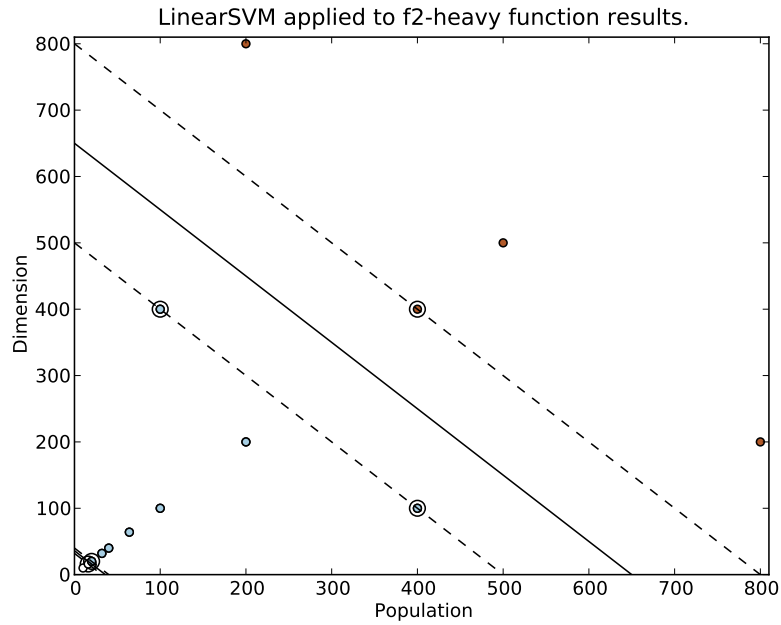

Figure 3. LinearSVM applied to the results of $f_{2-\text{heavy}}$ function: the three data categories correspond to sequential evaluation for smaller configurations, later when incrementing the problem size the best strategy becomes the S3 one, and finally S4 for larger configurations. The support vectors for the first class (sequential evaluation) are 16×16, and 20×20 for the lowest configuration of the second class (S3). This second class has support vectors: from 20×20 to 100×400 and 400×100 for the largest configurations. And finally, for the third class (S4) the support vector is 400×400.

support vector is 20×20 for the lowest configuration. On the other hand, S3 is the best strategy for a wide range of problem sizes. This class comprises from 20×20 to 100×400 and 400×100 as support vectors. For larger problem sizes (400×400 and larger ones) the most suitable strategy is S4.

All the functions tested until this point coincide granting a relevant role to the computational charge in order to be a worthwhile GPU-based evaluation. An intensive computational charge can be reached through an increment of the problem size, or by using fitness functions with a high-computational intensity. The necessary threshold to be worthwhile a GPU-based evaluation can be reached with small configurations (20×20) for very expensive functions ($f_{2-\text{heavy}}$), or with very large problem sizes ($4\,000 \times 4\,000$) for inexpensive functions ($f_{2-\text{light}}$). In all the cases, S4 becomes the most suitable strategy when dealing with a high computational charge.

4.3 F4-Light and F4-Heavy

In order to characterize the behaviour of the evaluation time of non-separable functions, the $f_{2-\text{light}}$ and $f_{2-\text{heavy}}$ are modified by incrementing the number of necessary dimensions to calculate each partial fitness value. For this modification an increment of the computational intensity over the functions is applied.

In Equations (5) and (6) the new functions are presented. They are very similar to the functions used in the previous section but with more terms involved to calculate any partial fitness values. This modification puts an extra pressure over the on-chip memory consumption for the GPU-strategies. More resources have to be allocated in order to hold the extra variables.

With these new functions, the general tendency of the evaluation time of the non-separable functions is fully characterized. Decision-making about the most suitable layout for functions with similar morphology is eased with this broad study.

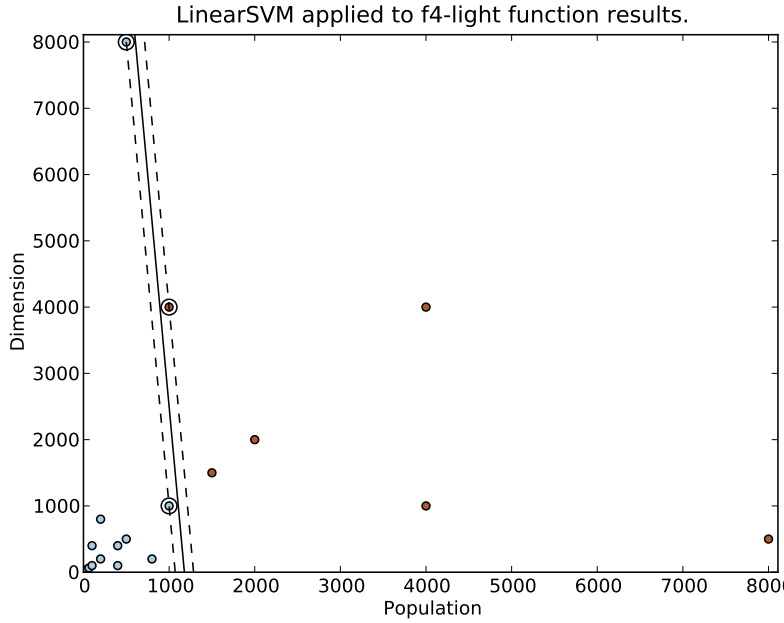

Figure 4. LinearSVM applied to the results of $f_{4-\text{light}}$ function: the two data categories correspond to the sequential evaluation for small and mid-size configurations, and S4 for large configurations. The support vectors for the first class (sequential evaluation) are: $500 \times 8\,000$ and $1\,000 \times 1\,000$; whereas for the second class (S4) it is $1\,000 \times 4\,000$.

Similarly to $f_{2-\text{light}}$, for $f_{4-\text{light}}$ (Figure 4) the most suitable strategy for small and mid-size problems is a sequential evaluation. Only for very large problem sizes

(larger than $1\,000 \times 1\,000$ or $500 \times 8\,000$) GPU-based evaluation (S4) becomes the most suitable election.

Due to the higher computational intensity of $f_{4-\text{light}}$ compared with $f_{2-\text{light}}$, the area dominated by sequential evaluation shrinks. Whereas the support vectors for sequential evaluation in $f_{2-\text{light}}$ are $1\,000 \times 4\,000$ and $4\,000 \times 1\,000$, for $f_{4-\text{light}}$ are $1\,000 \times 1\,000$ and $500 \times 8\,000$. In conclusion, for lower configurations in $f_{4-\text{light}}$ than in $f_{2-\text{light}}$, the penalization of data transfer between CPU and GPU is balanced by a faster evaluation of a more expensive function.

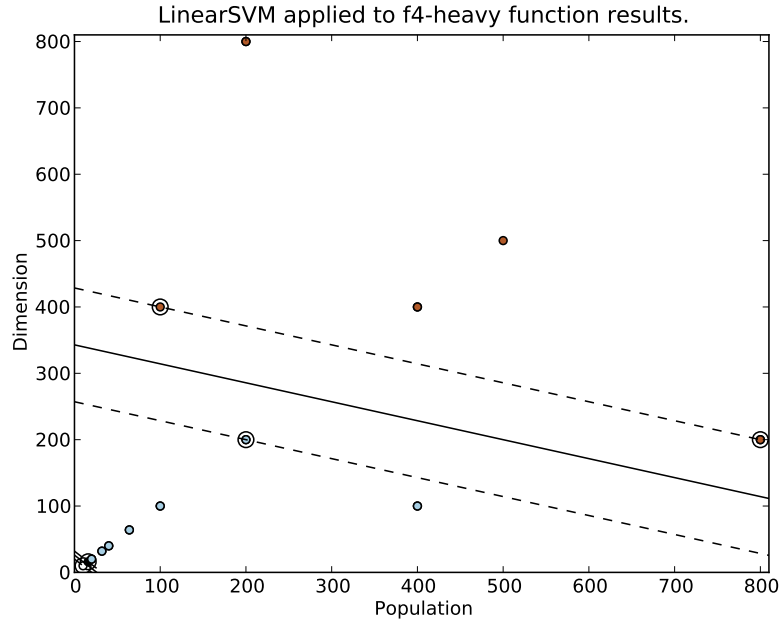

Figure 5. LinearSVM applied to the results of $f_{4-\text{heavy}}$ function: the three data categories correspond to sequential evaluation for tiny configurations, later, when incrementing the problem size, S3 is the best strategy and S4 for larger configurations. The support vector for the first class (sequential evaluation) is 10×10. For the second class (S3) it is 16×16 for the lowest configuration and 200×200 for the largest one. Finally, for the third class (S4): 100×400 and 800×200.

The observed trend of reduction of dominant CPU-based evaluation between $f_{2-\text{light}}$ and $f_{4-\text{light}}$ is accentuated when comparing $f_{2-\text{heavy}}$ and $f_{4-\text{heavy}}$ (Figure 5). When increasing the computational intensity of expensive versions of the fitness function by adding more terms, the GPU-based evaluation area is enlarged at the same time and the suitable problem sizes for sequential evaluation shrink.

In $f_{4-\text{heavy}}$, sequential evaluation the fastest choice is only for 10×10 configuration, whereas for a slightly larger configuration as 16×16 it becomes faster when evaluating on GPU with S3 strategy. In $f_{2-\text{heavy}}$, for 16×16 the best choice is still CPU-based evaluation.

More comparisons between $f_{2-\text{heavy}}$ and $f_{4-\text{heavy}}$ show a similar pattern in both cases (Figures 3 and 5). For low problem sizes, sequential evaluation is the fastest choice. However, due to the high-computational intensity of the function, any increment in the problem size carries out a switch from this choice to GPU-based evaluation choice.

When the problem reaches a size of 16×16, S3 becomes the most suitable strategy up to the size of 200×200. For larger problem sizes, the most suitable strategy is S4. This pattern reproduces the schema of $f_{2-\text{heavy}}$ with a constriction towards the lower problem sizes of the dominance areas of sequential and S3 strategy. S4 strategy is the fastest choice as much as the computational intensity of function grows. For configurations such as 100×400 or 800×200, and larger ones, the best option is S4.

The tendency provides clues about the behaviour of the problem when progressively reducing the computational intensity of the fitness function: S3 dominance area disappears, while sequential and S4 strategies fill the previous S3 area.

Other remarkable result is the critical role of the coalesced access in the performance in all functions analysed. In the strategies where it is implemented (S3 and S4) they outperform the strategies where the coalesced access is not implemented. The results suggest that coalesced access to global memory should be a major requirement in the design of evolutionary algorithms on GPU. Therefore, it can be stated that this option is mandatory to obtain an efficient implementation.

Furthermore, comparing with $f_{2-\text{heavy}}$, the pressure exerted in $f_{4-\text{heavy}}$ by the higher on-chip memory consumption produces a shared memory depletion for lower problem sizes in S2 strategy, as well as it happened in S3. Therefore, the experiments demonstrated that implementations based on registers are more robust for very large problem sizes than others based on shared memory.

4.4 Rana Function

Up to this point the proposed strategies have been tested against diverse non-separable functions. The SVM plots presented have a predictable capacity for other configurations of these particular fitness functions. However, this information is valuable not only for the functions already analysed, but also to predict the behaviour of the best strategies for other non-previously tested functions. To test the capacity of forecasting the best strategy to evaluate an arbitrary non-separable fitness function, the Rana function (Equation (2)) is used as the benchmark.

Rana function has a computational intensity closer to the previous expensive functions than to the light ones. Therefore, depending on the problem size, three best strategies are expected: for the lowest range of problem sizes, sequential evaluation will be the most suitable strategy, for mid-range it will be S3 strategy, and

finally, S4 strategy will be the most suitable for the upper-range of problem sizes. In Figure 6 the LinearSVM digesting the numerical results of the Rana Function are shown. The foreseen schema is roughly reproduced.

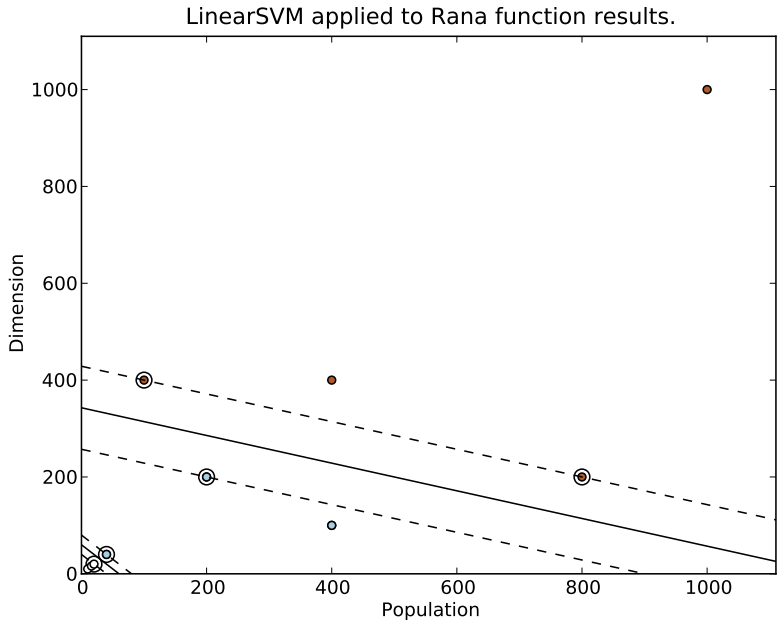

Figure 6. LinearSVM applied to the results of the Rana function: the three data categories correspond to sequential evaluation for smaller configurations, later when incrementing the problem size the best strategy becomes S3 one, and finally S4 for larger configurations. The support vectors for the first class (sequential evaluation) are 20×20, and 40×40 for the lowest configuration of the second class (S3). This second class has as support vectors: from 40×40 to 200×200 for the largest configurations. And finally, for the third class (S4) the support vector is 100×400 and 800×200.

The results presented for f_{Rana} (Figure 6) are reproduced in the schema of $f_{4-\text{heavy}}$ rather than $f_{2-\text{heavy}}$. Therefore, from the computational point of view f_{Rana} is close to $f_{4-\text{heavy}}$. There are no differences for the support vectors of the largest suitable configuration for S3 (200×200), or for the lowest suitable configuration for S4 (100×400 and 800×200). However, there exist some differences in the largest suitable configuration for sequential evaluation: 20×20 for the Rana function and 10×10 for $f_{4-\text{heavy}}$.

The adjustment between predictions and numerical experiments is good enough as a predictive rule for other non-separable functions.

5 CONCLUSION

This paper has presented a study analysing the impact of the data layout over the fitness calculation time when evaluating non-separable functions on GPU. The work has proved that a correct choice of the data layout reduces the fitness calculation time. This knowledge of the optimal evaluation strategy contributes to create faster algorithms, specially when dealing with high-computational intensity non-separable functions.

Besides, since the analysed layouts are not bound to any particular algorithm, the conclusions can be extrapolated to any population-based evolutionary algorithm which evaluates similar fitness functions.

In more detail, it can be stated that a sequential evaluation is the most suitable choice when dealing with non-expensive functions, except for very large problem sizes where GPU-based evaluation is the optimal option. On the contrary, GPU-based evaluation is recommended for most of the problem sizes of high-computational intensity non-separable functions. In this case, two strategies dominate the problem size landscape: S3 (each thread-block handles one individual on shared memory with coalesced access to global memory) for mid-size configurations and S4 (each thread handles one individual on registers with coalesced access to global memory) for larger configurations. However, due to the tight restrictions in maximum problem size allocatable in S3 and the relatively small difference in the evaluation time, we strongly recommended implementation of S4 for all configurations.

By considering only the GPU implementations, the importance of the coalesced access to global memory has to be underlined in order to reduce the evaluation time. Regardless of the problem size, all cases S3 and S4 (strategies with coalesced access) outperformed S1 and S2 (strategies without coalesced access).

Finally, it can be concluded that the knowledge of the most suitable evaluation strategy when evaluating non-separable functions allows to implement more efficient evolutionary algorithms avoiding later expensive redesign processes.

Acknowledgement

The authors would like to thank Antonio Gómez-Iglesias for his interest and support of this work. The research leading to these results has received funding by the Spanish Ministry of Economy and Competitiveness (MINECO) for funding support through the grant FPA2013-47804-C2-1-R.

REFERENCES

[1] Alba, E.—Tomassini, M.: Parallelism and Evolutionary Algorithms. IEEE Trans. Evolutionary Computation, Vol. 6, 2002, No. 5, pp. 443–462.

[2] Cortes, C.—Vapnik, V.: Support-Vector Networks. Machine Learning, Vol. 20, 1995, No. 3, pp. 273–297.

[3] Fabris, F.—Krohling, R. A.: A Co-Evolutionary Differential Evolution Algorithm for Solving Min-Max Optimization Problems Implemented on GPU Using C-CUDA. Expert Syst. Appl., Vol. 39, 2012, No. 12, pp. 10324–10333.

[4] Han, J.—Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers Inc., 2006.

[5] Kirk, D. B.—Hwu, W.-M. W.: Programming Massively Parallel Processors: A Hands-On Approach. Morgan Kaufmann, February 2010.

[6] Luong, T. V.—Melab, N.—Talbi, E.-G.: GPU-Based Island Model for Evolutionary Algorithms. Proceedings of Genetic and Evolutionary Computation Conference (GECCO 2010), ACM, 2010, pp. 1089–1096.

[7] Mussi, L.—Daolio, F.—Cagnoni, S.: Evaluation of Parallel Particle Swarm Optimization Algorithms within the CUDA Architecture. Inf. Sci., Vol. 181, 2011, No. 20, pp. 4642–4657.

[8] Mussi, L.—Nashed, Y. S. G.—Cagnoni, S.: GPU-Based Asynchronous Particle Swarm Optimization. In: Krasnogor, N., Lanzi, P. L. (Eds.): Proceedings of Genetic and Evolutionary Computation Conference (GECCO 2011), ACM, 2011, pp. 1555–1562.

[9] Nvidia: Fermi Compute Architecture Whitepaper, 2009.

[10] Pedregosa, F.—Varoquaux, G.—Gramfort, A.—Michel, V.—Thirion, B.—Grisel, O.—Blondel, M.—Prettenhofer, P.—Weiss, R.—Dubourg, V.—Vanderplas, J.—Passos, A.—Cournapeau, D.—Brucher, M.—Perrot, M.—Duchesnay, E.: Scikit-Learn: Machine Learning in Python. Journal of Machine Learning Research, Vol. 12, 2011, pp. 2825–2830.

[11] Pospíchal, P.—Schwarz, J.—Jaros, J.: Parallel Genetic Algorithm Solving 0/1 Knapsack Problem Running on the GPU. 16th International Conference on Soft Computing MENDEL 2010, Brno, University of Technology, 2010, pp. 64–70.

[12] Pospíchal, P.—Jaros, J.—Schwarz, J.: Parallel Genetic Algorithm on the CUDA Architecture. 2010 International Conference on Applications of Evolutionary Computation, LNCS, Vol. 6024, Part I, 2010, pp. 442–451.

[13] Rabinovich, M.—Kainga, P.—Johnson, D.—Shafer, B.—Lee, J. J.—Eberhart, R.: Particle Swarm Optimization on a GPU. IEEE International Conference on Electro/Information Technology (EIT), IEEE, 2012, pp. 1–6.

[14] Sanders, J.—Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional, July 2010.

[15] Shang, Y.-W.—Qiu, Y.-H.: A Note on the Extended Rosenbrock Function. Evolutionary Computation Vol. 14, 2006, No. 1, pp. 119–126.

[16] Tang, K.—Li, X.—Suganthan, P. N.—Yang, Z.—Weise, T.: Benchmark Functions for the CEC 2010 Special Session and Competition on Large-Scale Global Optimization. Nature Inspired Computation and Applications Laboratory (NICAL), School of Computer Science and Technology, University of Science and Technology of China (USTC), 2009.

[17] Tang, K.—Yao, X.—Suganthan, P. N.—MacNish, C.—Chen, Y. P.—Chen, C. M.—Yang, Z.: Benchmark Functions for the CEC 2008 Special Session

and Competition on Large Scale Global Optimization. Nature Inspired Computation and Applications Laboratory, USTC, 2007.

[18] de Veronese, L. P.—Krohling, R. A.: Differential Evolution Algorithm on the GPU with C-CUDA. 2010 IEEE Congress on Evolutionary Computation (CEC), 2010, Vol. 1–7.

[19] Whitley, D.—Mathias, K.—Rana, S.—Dzubera, J.: Building Better Test Functions. Proceedings of the 6th International Conference on Genetic Algorithms, Morgan Kaufmann, 1995, pp. 239–247.

[20] Zhou, Y.—Tan, Y.: GPU-Based Parallel Particle Swarm Optimization. Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2009), Trondheim, Norway, May 18–21, 2009, pp. 1493–1500.

[21] Zhou, Y.—Tan, Y.: Particle Swarm Optimization with Triggered Mutation and Its Implementation Based on GPU. Proceedings of Genetic and Evolutionary Computation Conference (GECCO 2010), ACM, 2010, pp. 1–8.

Miguel Cárdenas-Montes has been working since 2004 as research scientist in CIEMAT (Madrid, Spain). He graduated in Physics at the University of Seville in 1992 and he received the Ph.D. degree in computer science from the University of Extremadura, in 2014. His research activities include GPU computing and evolutionary algorithms, with special focus on astrophysics and other complex problems. He is a member of projects: EELA, EGEE-I, EGEE-II, EDGeS, EUFORIA, and EGI-InSPIRE (6th and 7th Framework Program of European Union). He is Professor at the University of Huelva and University Camilo José Cela.

Miguel A. Vega-Rodríguez received his Ph.D. in computer science from the University of Extremadura, Cáceres, Spain, in 2003. He is currently Professor of computer architecture in the Department of Computer and Communications Technologies, University of Extremadura. He has authored or co-authored more than 450 publications including journal papers, book chapters, and peer-reviewed conference proceedings. His main research interests include parallel and distributed computing, reconfigurable computing, and evolutionary computation. He is an editor and reviewer of several international JCR impact factor journals.