Expert Systems with Applications 39 (2012) 10324–10333

Contents lists available at SciVerse ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

A co-evolutionary differential evolution algorithm for solving min–max optimization problems implemented on GPU using C-CUDA

Fábio Fabris *, Renato A. Krohling *

Department of Computer Science, Federal University of Espírito Santo, Av. Fernando Ferrari, 514, CEP 29075-910, Vitória, Espírito Santo, ES, Brazil

ARTICLE INFO

Keywords:
Optimization
Differential evolution
Co-evolutionary algorithms
Graphics processing unit (GPU)
Compute unified device architecture (CUDA)
Computational performance assessment

ABSTRACT

Several areas of knowledge are being benefited with the reduction of the computing time by using the technology of graphics processing units (GPU) and the compute unified device architecture (CUDA) platform. In case of evolutionary algorithms, which are inherently parallel, this technology may be advantageous for running experiments demanding high computing time. In this paper, we provide an implementation of a co-evolutionary differential evolution (DE) algorithm in C-CUDA for solving min–max problems. The algorithm was tested on a suite of well-known benchmark optimization problems and the computing time has been compared with the same algorithm implemented in C. Results demonstrate that the computing time can significantly be reduced and scalability is improved using C-CUDA. As far as we know, this is the first implementation of a co-evolutionary DE algorithm in C-CUDA.

1. Introduction

In many design tasks in engineering, the designer is interested in robust optimal solutions. In general, optimal robust design is formulated as min–max optimization problems. Min–max optimization problems appear in many areas of science and engineering (Du & Pardalos, 1995), for instance in game theory, robust optimal control and many others. Min–max problems are considered difficult to solve. Hillis (1990), in his pioneering work, proposed a method for building *sorting networks* inspired by the co-evolution of populations. Two independent genetic algorithms (GAs) were used, whereas one for building sorting networks (host) and the other for building test cases (parasites). Both GAs evolved simultaneously and were coupled through the fitness function.

Previous studies on *co-evolutionary algorithms* (CEA) have demonstrated the suitability of the approach to solve *constrained optimization problems* (Barbosa, 1996, 1999; Krohling & dos Santos Coelho, 2006; Shi & Krohling, 2002; Tahk & Sun, 2000). A constrained optimization problem is transformed into an unconstrained optimization problem by introducing Lagrange multipliers and solving the resultant min–max problem using a CEA.

CEAs usually operate in two or more populations of individuals. In most CEAs, the fitness of an individual depends not only on the individual itself but also the individuals of other population. In this case, the fitness of an individual is evaluated by means of a competition with the members of the other population (Hillis, 1990; Paredis, 1994; Rosin & Belew, 1995, 1997). Inspired by the work of Hillis (1990), the co-evolutionary approach has been extended to solve constrained optimization problems (Barbosa, 1996, 1999; Krohling & dos Santos Coelho, 2006; Shi & Krohling, 2002; Tahk & Sun, 2000). Barbosa (1996, 1999), presented a method to solve min–max problems by using two independent populations of GA coupled by a common fitness function. Also, Tahk and Sun (2000) used a *co-evolutionary augmented Lagrangian method* to solve min–max problems by means of two populations of Evolution Strategies with an annealing scheme. The first population was made up by the vector of variables and the second one is made up of the Lagrange multiplier vector. Laskari, Parsopoulos, and Vrahatis (2002) have also presented a method using *Particle Swarm Optimization* (PSO) for solving min–max problems, but not using a co-evolutionary approach.

Krohling and dos Santos Coelho (2006) proposed a Gaussian probability distribution to generate the accelerating coefficients of PSO. Two populations of PSO using Gaussian distribution were used to solve min–max optimization problems with promising results. Cramer, Sudhoff, and Zivi (2009) proposed a co-evolutionary approach to optimize the design of a ship vessel given a set of restrictions. The initial problem was transformed to a min–max and solved by a co-evolutionary GA. Liu, Fernández, Gielen, Castro-López, and Roca (2009) used a co-evolutionary differential evolution algorithm to solve the min–max associated with a constrained non-linear optimization problem of minimizing the cost of a analog circuit design given a set of restrictions specified by

* Corresponding authors. Tel.: +55 27 3335 2136; fax: +55 27 3335 2850 (F. Fabris).

E-mail addresses: ffabris@inf.ufes.br (F. Fabris), krohling.renato@gmail.com (R.A. Krohling).

the user. The authors reported that the design specifications were closely met, even with high constrained requirements, and the costs successfully reduced.

All these past works showed that co-evolutionary algorithms are being used with success to solve the min–max associated with constrained non-linear problems. However, the computational burden of this approach is considerable: considering two populations of n individuals each, the number of objective function calls required to evaluate one of the populations is n^2. Each individual in one population must be evaluated against each individual of the other. Evaluation of population needs to be performed many times during the optimization process and is the most relevant operation regarding computational time. So, even algorithms with fast convergence (like DE) will suffer from scalability problems if n is large or the objective function is complex. Previous works have been done in parallelizing co-evolutionary algorithms (Seredynski & Zomaya, 2002) but, as far as we know, never in context of co-evolutionary differential evolution algorithms.

The parallelization of the algorithm proposed in this work reduces the overhead of population evaluation from n^2 sequential unitary fitness evaluations to n evaluations of n functions at the same time. This is done by evaluating each individual of one population against every other in parallel. This reduces the response time and enlarges the set of tractable problems using this approach.

In this paper, two populations of independent DE are evolved: one for the variable vector, and the other for the Lagrange multiplier vector. At the end of the optimization process, the first DE provides the variable vector, and the second DE provides the Lagrange multiplier vector.

The rest of the paper is organized as follows: in Section 2, the formulation of the min–max problem is described. The standard DE is explained in Section 3. Section 4 shows the formulation of the co-evolutionary DE algorithm. In Section 5, the implementation of the co-evolutionary DE is presented to solve min–max problems. Section 6 provides simulation results and comparisons for a min–max problems and for optimization problems formulated as min–max problems followed by conclusions and directions for future research in Section 7.

2. Formulation of constrained optimization problems as min–max problems

Many problems in various scientific areas and real world applications can be formulated as constrained optimization problems. Generally, a constrained optimization problem is defined by:

$$\min_{\vec{x}\in\mathbb{R}^n} f(\vec{x}) \tag{1}$$

subject to:

$$g_i(\vec{x}) < 0, \quad i = 1,\dots,m,$$
$$h_i(\vec{x}) = 0, \quad i = 1,\dots,l,$$

where $f(\vec{x})$ is the objective function, $\vec{x} = [x_1, x_2, \dots, x_d]^T \in \mathbb{R}^d$ is the vector of variables, $\vec{g}(\vec{x})$ is the vector of inequality constraints, and $\vec{h}(\vec{x})$ is the vector of equality constraints.

The set $S \subseteq \mathbb{R}^d$ designates the search space, which is defined by the lower and upper bounds of the variables $\underline{x}_j$ and $\bar{x}_j$, respectively, with $j = 0,\dots,d$. Points in the search space, which satisfy the equality and inequality constraints, are feasible candidate solutions.

The Lagrange-based method (Du & Pardalos, 1995) is a classical approach to formulate constrained optimization problems. By introducing the Lagrangian formulation, the dual problem associated with the primal problem (1) can be written as:

$$\max_{\vec{\mu},\vec{\lambda}} L(\vec{x}, \vec{\mu}, \vec{\lambda}) \tag{2}$$

subject to:

$$\mu_i > 0, \quad i = 1,\dots,m,$$
$$\lambda_i > 0, \quad i = 1,\dots,l,$$

where:

$$L(\vec{x}, \vec{\mu}, \vec{\lambda}) = f(\vec{x}) + \vec{\mu}^T\vec{g}(\vec{x}) + \vec{\lambda}^T\vec{h}(\vec{x}) \tag{3}$$

and $\vec{\mu}$ is a $m * 1$ multiplier vector for the inequality constraints. The vector $\vec{\lambda}$ is a $l * 1$ multiplier vector for the equality constraints.

If the problem (1) satisfies the convexity conditions over S, then the solution of the primal problem (1) is the vector $\vec{x}^*$ of the saddle-point $\{\vec{x}^*, \vec{\mu}^*, \vec{\lambda}^*\}$ of $L(\vec{x}^*, \vec{\mu}^*, \vec{\lambda}^*)$ so that

$$L(\vec{x}^*, \vec{\mu}, \vec{\lambda}) \leqslant L(\vec{x}^*, \vec{\mu}^*, \vec{\lambda}^*) \leqslant L(\vec{x}, \vec{\mu}^*, \vec{\lambda}^*). \tag{4}$$

The saddle point can be obtained by minimizing $L(\vec{x}^*, \vec{\mu}, \vec{\lambda})$ with the optimal Lagrange multipliers $(\vec{\mu}^*, \vec{\lambda}^*)$ as a fixed vector of parameter. In general, the optimal values of the Lagrange multipliers are unknown a priori. According to the duality theorem (Du & Pardalos, 1995), the primal problem (1) subject to the inequality and equality constraints can be transformed into a dual or min–max problem.

Solving the min–max problem

$$\min_{\vec{x}} \max_{\vec{\mu},\vec{\lambda}} L(\vec{x}, \vec{\mu}, \vec{\lambda}) \tag{5}$$

provides the minimizer $\vec{x}^*$ as well as the Lagrange multipliers $(\vec{\mu}^*, \vec{\lambda}^*)$. However, for non-convex problems, the solution of the dual problem does not necessarily coincide with that of the primal problem. In that case, a penalty term associated with equality and inequality constraints is added to the Lagrangian function. The augmented Lagrangian is described as in Tahk and Sun (2000).

$$L_a(\vec{x}, \vec{\mu}, \vec{\lambda}, r) = f(\vec{x}) + \sum_{i=1}^{m} p_i(\vec{x}, \mu_i, r) + \vec{\lambda}^T\vec{h}(\vec{x}) + r\sum_{i=1}^{l} h_i^2(\vec{x}), \tag{6}$$

where the term p_i for the ith inequality constraint is given by

$$p_i(\vec{x}, \mu_i, r) = \begin{cases} \mu_i g_i(\vec{x}) + r g_i^2(\vec{x}), & \text{if } g_i(\vec{x}) \geqslant \frac{\mu_i}{2r}, \\ -\frac{\mu_i^2}{4r}, & \text{otherwise} \end{cases} \tag{7}$$

with r being a penalty factor. It can be shown that the solution of the primal problem and the augmented Lagrangian are identical. The goal is to find the saddle-point $(\vec{x}^*, \vec{\mu}^*, \vec{\lambda}^*)$. In Section 4, co-evolutionary differential algorithm (CDE) is presented to solve min–max problems. Next, we provide some background on DE.

3. Differential evolution

The optimization procedure differential evolution (DE) was introduced in 1997 by Storn and Price (1997). Similar to other evolutionary algorithms (EAs), it is based on the idea of evolution of populations of possible candidate solutions, which undergoes very simple operations of mutation, crossover and selection.

The candidate solutions of the optimization problem in DE are represented by vectors. The components of the vectors are the parameters of the optimization problem and the set of vectors forms the population. The basic idea is the operation to generate new candidate solutions by means of a weighted difference between two vectors of the population, to which is added a third vector. All the three vectors of the population are chosen randomly. The new created vector is the trial vector. If the fitness of the trial vector is better than the fitness of a pre-chosen vector, denominated target vector, then the target vector of the population is replaced by the trial vector. Unless stated otherwise, in our study, we are considering minimization problems. Therefore, the higher the value of the fitness, the smaller the values of the objective function.

The parameter vectors are denoted by $\overrightarrow{x_i} = [x_{i,1}, x_{i,2}, \ldots, x_{i,d}]^T$ with components $x_{i,j}$. The lower and upper bounds of x_i are given by $\underline{x}_i$ and $\bar{x}_i$, respectively. The index $i = 1,\ldots,n$ represents the individual's index in the population and $j = 1,\ldots,d$ is the position in d-dimensional individual. Thereafter we use t to indicate time (generation). During the initialization of the population, n vectors are generated randomly in the d-dimensional search space. After initialization, the fitness of each vector is calculated. The three operators: mutation, crossover and selection are described in the following (Storn & Price, 1997).

3.1. Mutation

By means of the mutation operator a new vector is created. For each possible solution vector $\overrightarrow{x_i}$, in generation t, denoted by $\overrightarrow{x_i^t}$, a mutant vector $\overrightarrow{v_i^t}$ is generated according to:

$$\vec{v}_i^t = \vec{x}_{r_1}^t + F_m\left(\vec{x}_{r_2}^t - \vec{x}_{r_3}^t\right), \tag{8}$$

where r_1, r_2 and r_3 are mutually different random integer indices selected from $1, 2, \ldots, n$. Further, r_1, r_2 and r_3 are different so that $n > 4$ is required. The real constant F_m is called mutation factor, which controls the amplification of the difference between two individuals so as to avoid search stagnation. The crossover rate usually lies in the interval [0.5; 1] as pointed by Storn and Price (1997).

3.2. Crossover

Following the mutation operation, crossover is applied to the population. For each mutant vector $\vec{v}_i^t$, a trial vector, $\vec{u}_i^t$ is generated according to:

$$u_{i,j}^t = \begin{cases} v_{i,j}^t, & \text{if } rand_j(0,1) < C_r \text{ or } j = k, \\ x_{i,j}^t, & \text{otherwise}, \end{cases} \tag{9}$$

where $rand_j(0,1)$ is a uniform random number generated into the interval [0; 1] for each variable j. The index $k \in [1,\ldots,d]$ is a random index, chosen once for each i to make sure that at least one variable is always selected from the mutated vector $\vec{v}_i^t$. C_r is the crossover rate, which controls the fraction of mutant values that are used. It usually takes values within the range [0.8; 1.0] as pointed out by Storn and Price (1997).

3.3. Selection

In the case of the selection operator, there is a competition between each individual $\vec{x}_i^t$ and its child $\vec{u}_i^t$. The winner of the competition is chosen deterministically based on the fitness values. The individual with best fitness is promoted to the next generation.

$$\vec{x}_i^{t+1} = \begin{cases} \vec{u}_i^t, & \text{if } fitness(\vec{u}^t) > fitness(\vec{x}^t), \\ \vec{x}_i^t, & \text{otherwise}. \end{cases} \tag{10}$$

Generally, the performance of a DE algorithm depends on the population size n, the mutation factor F_m, and the crossover rate C_r. In the last few years, different variants of DE have been developed and classified using the following notation: $DE/\alpha/\beta/\delta$, where α indicates the method for selecting the individual r_1 that will form the base of the mutated vector; β indicates the number of difference vectors used to perturb the base individual, and δ indicates the crossover mechanism used to create the child population. In our implementation, we used the standard *DE/rand/1/bin* (individual r_1 is take randomly from the population at iteration t) and *DE/best/1/bin* (individual r_1 is the best individual at the current iteration t), even though other DE variants are easily implemented with small code changes (Qin, Huang, & Suganthan, 2009).

4. Co-evolutionary differential evolution algorithm for solving min–max problems

Two populations of DEs are involved in the co-evolutionary differential evolution (CDE) to solving the min–max problem formulated according to (6). The first DE focuses on evolving the variable vector $\vec{x}$ whilst the vector of Lagrangian multiplier $\vec{\theta} = [\vec{\mu}, \vec{\lambda}]$ is maintained "frozen". Only the variable vector $\vec{x}$ is represented in the population P_1. The second DE focuses on evolving the Lagrangian multiplier vector $\vec{\theta}$ whilst the Population P_1 is maintained "frozen". Only the multiplier vector $\vec{\theta}$ is represented in the population P_2. The two DEs interact with each other through a common fitness evaluation. For the first DE, the problem is a minimization problem and the fitness value of each individual $\vec{x}$ is evaluated according to:

$$f_1(\vec{x}) = \max_{\vec{\theta} \in P_2} L_a(\vec{x}, \vec{\theta}). \tag{11}$$

Each individual $\overrightarrow{x_i}$ of P_1 is evaluated against all individuals $\overrightarrow{\theta_i}$ of P_2 as illustrated in Fig. 1. The fitness for the individual $\overrightarrow{x_i}$ is the highest value of $f_i(\overrightarrow{x_i})$. This process is repeated for all n_{p1} individuals of P_1.

The second problem consists in a maximization problem and the fitness value of each individual is made up of the Lagrange multiplier $\vec{\theta}$ is evaluated according to:

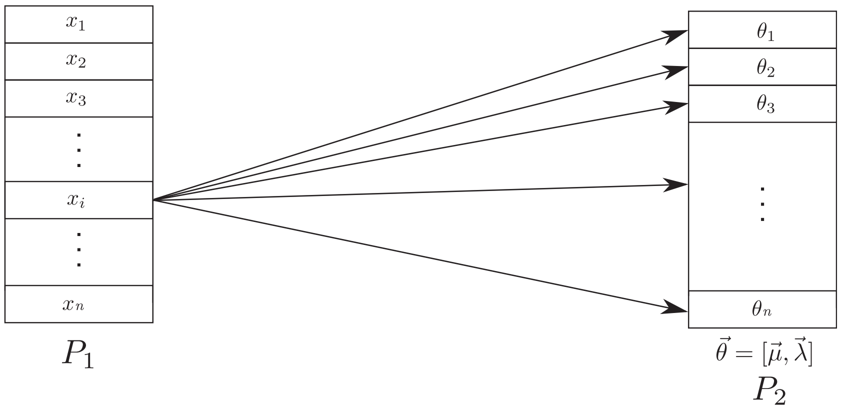

Fig. 1. Co-evolutionary approach to solve min–max problems (Barbosa, 1996, 1999).

$$f_2(\vec{\theta}) = max_{\vec{x} \in P_1} L_a(\vec{x}, \vec{\theta}). \tag{12}$$

Each individual of P_2 is evaluated against all individuals $\vec{x}_i$ of P_1. The fitness for individual $\vec{\theta}_j$ is the smallest value of $f_2(\vec{\theta})$. This process is repeated for all n_{p2} individuals of P_2.

The competition among individuals is updated in the next generation if better fitness values are obtained. The best individual in the population P_1 is the solution for the variable vector $\vec{x}$, and the best individual in the population P_2 is the solution for the Lagrangian multiplier vector $\vec{\theta}$. In the CDE algorithm, when one DE is running, the other serves as its environment, so each DE has a changing environment from generation to generation.

```
   Input: Number of Cycles (MaxCycles), Number of Iterations of Population
          A (MaxGenA), Number of Iterations of Population B (MaxGenB)
   Output: The solution of the non-linear problem
1  Initialize Population $P_1$
2  Initialize Population $P_2$
3  for $k = 1$ to $MaxCycles$ do
       /* For each individual of population $P_1$ */
4      for $j = 1$ to $MaxGen1$ do
5          Generate new Population $P_1$
           /* Evaluate new population $P_1$ against population $P_2$ */
6          Evaluate new Population $P_1$
       end
       /* For each individual of population $P_2$ */
8      for $j = 1$ to $MaxGen2$ do
9          Generate new Population $P_2$
           /* Evaluate new population $P_1$ against population $P_1$ */
10         Evaluate new Population $P_2$
       end
   end
13 Return the best individual that does not violates the constraints.
```

In the Algorithm 1 we present the pseudo-code of the co-evolutionary differential algorithm. The first inner loop (lines 4, 5 and 6) evolves $\vec{x}$ for *MaxGen*1 iterations and the second inner loop (lines 8, 9 and 10) evolves $\vec{\theta}$ for *MaxGen*2 generations. Those two loops are repeated *MaxIterations* times (line 3).

From the algorithm it is clear that the number of "evaluations of populations" of the algorithm is *MaxIterations*(*MaxGen*1 + *MaxGen*2). The number of evaluations of the objective function per population evaluation is equal to n^2 where n is the size of the population and assuming that both populations have n individuals. This formulation may be re-written as: $Evaluations_{seq} = MaxIterations\ (MaxGen1 * n^2 + MaxGen2 * n^2)$.

For instance, if we assume $MaxIterations = 200$, $MaxGen1 = 20$, $MaxGen2 = 20$, $n = 50$ we have: $Evaluations = 200(20 * 50^2 + 20 * 50^2) = 20000000$

This computation is unavoidable, but it is possible to evaluate the fitness of all n individuals in parallel. In this case, the evaluation number is equivalent to: $Evaluations_{par} = MaxIterations\ (MaxGen1 * n + MaxGen2 * n) = 200(20 * 50 + 20 * 50) = 400000$, a clear advantage. The *reduction ratio* of the proposed approach is defined as $Evaluations_{seq}/Evaluations_{par} = n$ that is: the larger the population, the greater is the the expected performance gain.

5. Implementation of a co-evolutionary DE using C-CUDA

5.1. Basic concepts of GPU and C-CUDA language

In recent years, the graphics processing unit (GPU) (GPGPU, 2010) has emerged as a powerful computation device and the main question that arises is why the GPU is much faster for computation than the central unit processing (CPU). The differences lie in the architecture. While the CPU is conceptualized for general purposes carrying out arbitrary operations like input/output access, processing, etc. the GPU is conceptualized for performance optimization for defined tasks (Díez, Moser, Schoenegger, Pruggnaller, & Frangakis, 2008). The central issue is that not all algorithms can be effectively implemented on the GPU. Only numerical problems that are inherently parallel may have profit of this technology.

Although efforts have been made to develop applications programming interface (API) to ease the programmer task, programming the GPU continues to be a hard task. NVidia (2010) developed a software platform named *compute unified device architecture*, for short, CUDA, which allows almost the direct translation of C code onto the GPU. In this case, the syntax consists of minimal extensions of the C language (NVidia, 2007). This feature provided an adequate environment to porting existing code from C language to CUDA. For programmers already familiar with C language, there exists a necessity to integrate the concept of thread before writing the first application in CUDA. The concept of thread embodies the GPU with the technique of SIMD (single instruction, multiple data) from vector computers. A thread is a sequence of instructions that can be executed on different data units in parallel (Díez et al., 2008). In this section, we briefly describe the building blocks necessary to write a basic program in C-CUDA.

In C-CUDA there are three types of functions: The host functions, which are called and executed only by the CPU. This kind of functions is the same as those functions implemented in C. The kernel functions, which are only executed by the device and callable only by the CPU. For this kind of function, the qualifier, *__global__*, must be inserted before the type of return of the function that is always void. Finally, the device functions, which are called and executed only by the device, whereas the qualifier, *__device__*, must be declared before the type of return of the function. In this case, it is allowed to return any type of value. Beyond the types of functions that are different of the conventional languages, C-CUDA presents in its structure the concepts of thread, block and grid (NVidia, 2007), which are depicted in Fig. 2.

The grid can have up to two dimensions of blocks, and the blocks, in turn, three dimensions of threads. These concepts allow perform all the parallelism of the program that will be executed onto the device. Each thread has an address that is represented by two vectors. The vector *threadIdx* of three position *x*, *y* and *z*, shows the address of the thread into the block. The other vector of two dimensions, *blockIdx*, addresses the block inside the grid and shows which block belongs to the thread (NVidia, 2007).

Two additional vectors which are very important are: *blockDim*, and *gridDim*, of three and of two positions, respectively. The first one contains the existing number of threads in each dimension of the block. The last one contains the number of blocks in each dimension of the grid (NVidia, 2007). It is also worthy to mention that threads in a same block share a memory of 16KB. This memory is called shared memory and is located close to the GPU core (streaming multiprocessor), functioning as cache of level 1 similar to hierarchies of memories existing for CPUs. The use of this kind of memory provides advantages due to its low latency (NVidia, 2007). All the sequential operations are executed onto the host computer, e.g., for memory allocation and freeing within the device is used the CUDA function *cudaMalloc* and *cudaFree*, respectively. To copy data to and from the device as well as to and from the host is used the CUDA function *cudaMemcpy*.

Recently, implementations of algorithms in C-CUDA to solve scientific problems have appeared with promising results (Che et al., 2008; Díez et al., 2008; Li, Wang, He, & Chi, 2007; Li, Zhang, & Liu, 2009; Luebke, 2008; Schenk, Christen, & Burkhart, 2008;

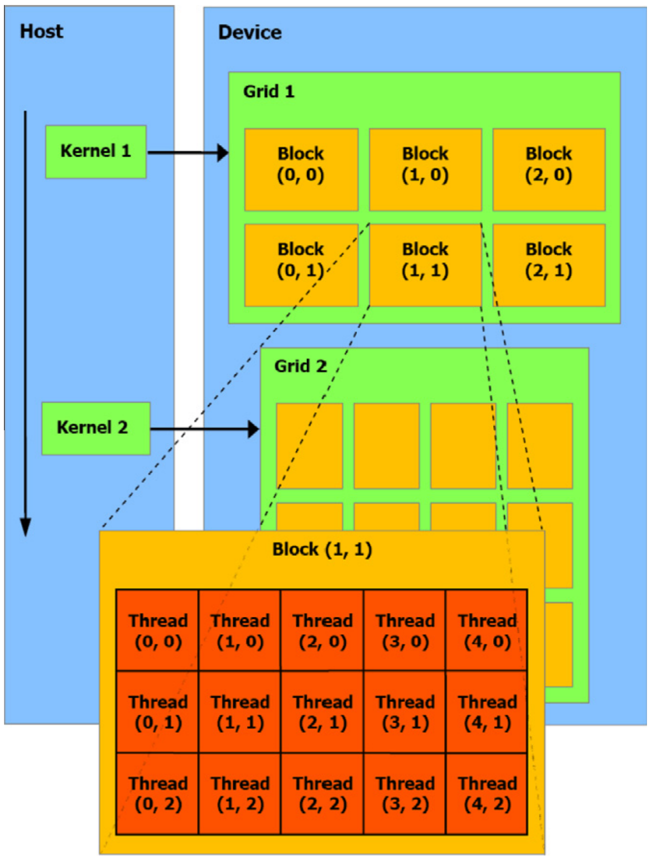

Fig. 2. CUDA architecture.

Seredynski & Zomaya, 2002; Veronese & Krohling, 2009, 2010; Zhou & Tan, 2009). This paper presents an implementation of a co-evolutionary DE algorithm in C-CUDA aiming at speedup the computing time for min–max optimization problems.

5.2. Implementation of the CDE algorithm

In this section, we comment the most relevant parts of the CDE-code developed in C-CUDA. In Fig. 1, we describe the code fragment regarding the main loop of the algorithm in C-CUDA. It begins by calculating the fitness of every individual in both randomly initialized populations. Next, the algorithm loops while the budget condition is not met. After that, for each outer iteration, an array of shuffled indexes is generated. In line 13 the fitness of each individual in the population A is updated. From lines 17–30, the population *A* is evolved: generation of new individuals (line 20) and evaluation (line 24) take place. The same steps are repeated for population *B*: the fitness of the population B is recalculated, then the population is evolved in the inner loop by generating, evaluating and selecting new individuals. The evaluation and generation function (the heavy computational burden) is performed in parallel.

The Fig. 2 shows the parallel computation of the fitness for each individual of populations *A* and *B*. Each block *indexed* by *blockIdx* calculates the fitness of the individual indexed by *blockIdx* from population *A* and *B* at the same time. Lines 3 and 4 calculate the fitness of the individual indexed by *blockIdx* for each individual in the opposite population. Notice that there is some redundancy in the calculations, each evaluation is done twice, this is necessary since shared memory is block-wise. One could use global memory to avoid this, but the bandwidth is smaller, yielding higher computational times.

The operations *refreshFitnessA* and *refreshFitnessB* are very similar to *refreshFitness*. The only difference is that these operations only update the fitness of population *A* or *B*, not *A* and *B* at the same time.

Fig. 3 shows the generation of the new individuals. First, three indexes are chosen from the shuffled array produced by the host. We guarantee the random picking by using the iteration number and thread *id* to index the vector in a non-linear fashion. The crossover needs an array of float random numbers, which are generated by the host and indexed using the *threadIdx* and a global iteration counter. The generation of an individual of population *B* is carried out in the same way.

Listing 4 shows the evaluation of the new individuals generated by the previous function. The idea is the same as the *refreshFitness* function. Next, the individual is compared by its counterpart and put in the new population if it has a higher fitness, otherwise the counterpart remains.

Since the focus of this application is function optimization using CDE, all individuals need to be evaluated according to an objective function. The part of the code for this purpose close to C-style does not need to be changed. In Section 6, all benchmark functions used in this work are described.

The DE algorithm needs two sets of random numbers: a set of distinct random numbers for selecting individuals for mutation and another set containing floats for deciding if a given position of an offspring will undergo the crossover. The first set of random numbers is generated by exchanging the positions of an ordered index vector. This set is copied to the device memory for each outer loop. The second set is generated only once, an array of *MAX_RAND* random numbers is generated by the function *rand* and stored in an array allocated on the device memory of size *MAX_RAND*. *MAX_-RAND* may be smaller than the total number of necessary random numbers. If this happens, the sequence will start to repeat itself.

The algorithm implemented in C-CUDA is avaiable from authors upon request.

6. Simulation results

6.1. Benchmark problems

Problem P_1 is a case study based on the *test function* 2 given by Barbosa (1996, 1999). The original function was a base problem to test the validity of the min–max approach. The *test function* 2 is one-dimensional, which is too simple for our purposes of analysing computational performance. So, we extended the formulation to a similar problem of 100 dimensions.

Problem P_1 consists in finding:

$$min_{\vec{x}}\ max_{\vec{y}}\ P_1(\vec{x},\vec{y}) = \sum_{i=1}^{100} x_i^2 - y_i^2 \tag{13}$$

with:

$$-1.0 < x_i < 1.0 \quad \text{and} \quad -1.0 < y_i < 1.0\ \forall i \in [1,\ldots,100],$$

The global solution is: $x_i = y_i = 0\ \forall i = 1,\ldots,100$.

In the following we describe constrained optimization problems.

The problem P_2 consists of minimizing:

$$P_2(\vec{x}) = 5x_1 + 5x_2 + 5x_3 + 5x_4 - 5\sum_{i=1}^{4} x_i^2 - \sum_{i=5}^{13} x_i, \tag{14}$$

subject to the constraints:

```
// calculate the fitness of the populations A and B
refreshFitness<<<nPop,nPop>>>(oldPopA,oldPopB,nPop,nPar,nRes,oldFit, oldRes
    );
   while(interation < maxRounds){

     // generate shuffled array of unique indexes for random operations
         int *shuffled=shuffle(nPop);
         cutilSafeCall(cudaMemcpy(cudaShuffle, shuffled, sizeof(int)*nPop,
             cudaMemcpyHostToDevice));
         free(shuffled);

     // recalculate the fitness of the population A
         refreshFitnessA<<<nPop,nPop>>>(oldPopA,oldPopB,nPop,nPar,nRes,
             oldFit, oldRes);
         cudaThreadSynchronize();

     // inner loop for population A
         for ( i = 0; i < itA; i++){
         // generate all new individuals
             generateA<<<1,nPop>>>(newPopA,oldPopA,oldPopB,cudaShuffle, nPop
                 , nPar, RAND, F, f_cross, boundsCuda, nRes, i+iteration);
             cudaThreadSynchronize();

         // evaluate and select individuals
             evaluateAndSelectA<<<nPop,nPop>>>(newPopA, oldPopA, oldPopB,
                 cudaShuffle, nPop, nPar, RAND, F, f_cross, bounds, nRes, i+
                 iteration, newFit, oldFit);
             cudaThreadSynchronize();

         // switch populations
         xChangePointers(newPopA, oldPopA); xChangePointers(newFit, oldFit);
         }

         // recalculate the fitness of the population B
         refreshFitnessB<<<nPop,nPop>>>(oldPopA, oldPopB,nPop,nPar,nRes,
             oldFit,oldRes);
         cudaThreadSynchronize();

         for ( i = 0; i < itB; i++){

             generateB<<<1,nPop>>>(newPopB, oldPopB, oldPopA, cudaShuffle,
                 nPop, nPar, RAND, F, f_cross, boundsCuda, nRes, i+iteration
                 );
             cudaThreadSynchronize();

         evaluateAndSelectB<<<nPop,nPop>>>(newPopB, oldPopB, oldPopA,
             cudaShuffle, nPop, nPar, RAND, F, f_cross, boundsCuda, nRes, i+
             iteration, newRes, oldRes);
             cudaThreadSynchronize();

         xChangePointers(newPopA, oldPopA); xChangePointers(newFit, oldFit);
         }
         iteration++;
     }
```

Listing 1. Main loop.

$$g_1(\vec{x}) = 2x_1 + 2x_2 + x_{10} + x_{11} \leqslant 10,$$
$$g_2(\vec{x}) = -8x_1 + x_{10} \leqslant 0,$$
$$g_3(\vec{x}) = -2x_4 - x_5 + x_{10} \leqslant 0,$$
$$g_4(\vec{x}) = 2x_1 + 2x_3 + x_{10} + x_{12} \leqslant 10,$$
$$g_5(\vec{x}) = -8x_2 + x_{11} \leqslant 0,$$
$$g_6(\vec{x}) = -2x_6 - x_7 + x_{11} \leqslant 0,$$
$$g_7(\vec{x}) = 3x_2 + 2x_3 + x_{11} + x_{12} \leqslant 10,$$
$$g_8(\vec{x}) = -8x_3 + x_{12} \leqslant 0,$$
$$g_9(\vec{x}) = -2x_8 - x_9 + x_{12} \leqslant 0,$$

the search space is defined in

$$0 \leqslant x_i \leqslant 1, \quad i = 1, \ldots, 9,$$
$$0 \leqslant x_i \leqslant 100, \quad i = 10, 11, 12,$$
$$0 \leqslant x_{13} \leqslant 1.$$

This problem is quadratic and its minimum is $P_2(\vec{x}^*) = -15$ located in $\vec{x}^* = (1,1,1,1,1,1,1,1,1,3,3,3,1)$. In this point only three constraints are not active: g_2, g_5 and g_8.

The problem P_3 consists of minimizing:

$$\begin{aligned} P_3(\vec{x}) = {} & x_1^2 + x_2^2 + x_1x_2 - 14x_1 \\ & -16x_2 + (x_3 - 10)^2 + 4(x_4 - 5)^2 + (x_5 - 3)^2 \\ & +2(x_6 - 1)^2 + 5x_7^2 + 7(x_8 - 11)^2 \\ & +2(x_9 - 10)^2 + (x_{10} - 7)^2 + 45, \end{aligned} \tag{15}$$

```
__global__ void refreshFitness(float* A, float* B, int nPop, int nPar, int
    nRes, float *fit, float *res){

    float fittA = target(&A[blockIdx.x*nPar], &B[threadIdx.x*nRes]);
    float fittB = target(&A[threadIdx.x*nPar], &B[blockIdx.x*nRes]);

    __shared__ float fitnessA[MAXPOP];
    __shared__ float fitnessB[MAXPOP];

    fitnessA[threadIdx.x] = fittA;
    fitnessB[threadIdx.x] = fittB;

    __syncthreads();
    int i;
    if(threadIdx.x == 0){
        float min = HUGE_VAL;
        for(i=0; i<nPop; i++){
            if(fitnessB[i] < min){
                min = fitnessB[i];
            }
        }
        res[blockIdx.x] = min;
    }else if(threadIdx.x == 1){
        float max = -HUGE_VAL;
        for(i=0; i<nPop; i++){
            if(fitnessA[i] > max){
                max = fitnessA[i];
            }
        }
        fit[blockIdx.x] = max;
    }
}
```

Listing 2. Refreshing the fitness of all individuals.

subject to:

$$\begin{aligned}
g_1(\vec{x}) &= 105 - 4x_1 - 5x_2 + 3x_7 - 9x_8 \geqslant 0,\\
g_2(\vec{x}) &= -10x_1 + 8x_2 + 17x_7 - 2x_8 \geqslant 0,\\
g_3(\vec{x}) &= 8x_1 - 2x_2 - 5x_9 + 2x_{10} + 12 \geqslant 0,\\
g_4(\vec{x}) &= 3x_1 - 6x_2 - 12(x_9 - 8)^2 + 7x_{10},\\
g_5(\vec{x}) &= -3(x_1 - 2)^2 - 4(x_2 - 3)^2 2x_3^2 7x_4 + 120 \geqslant 0,\\
g_6(\vec{x}) &= -x_1^2 - 2(x_2 - 2)^2 + 2x_1x_2 - 14x_5 + 6x_6 \geqslant 0,\\
g_7(\vec{x}) &= -5x_1 - 8x_2 - (x_3 - 6)^2 + 2x_4 + 40 \geqslant 0,\\
g_8(\vec{x}) &= -0.5(x_1 - 8)^2 - 2(x_2 - 4)^2 - 3x_5^2 + x_6 + 30 \geqslant 0,
\end{aligned}$$

the search space is defined by the following bounds:

$$-10 \leqslant x_i \leqslant 10, \quad i = 1, \ldots, 10.$$

The global minimum is $P_3(\vec{x}^*) = 24.3062091$ and the optimal point, $\vec{x}^* = (2.171996, 2.363683, 8.773926, 5.095984, 0.9906548, 1.430574, 1.321644, 9.828726, 8.280092, 8.375927)$. Only g_7 and g_8 are not active at the global optimum.

The problem P_4 consists of minimizing:

$$\begin{aligned}
P_4(\vec{x}) = {} & (x_1 - 10)^2 + 5(x_2 - 12)^2 + x_3^4 + 3(x_4 - 11)^2 + 10x_5^6\\
& + 7x_6^2 + x_7^4 - 4x_6x_7 - 10x_6 - 8x_7,
\end{aligned} \tag{16}$$

subject to:

$$\begin{aligned}
g_1(\vec{x}) &= 127 - 2x_1^2 - 3x_2^4 - x_3 - 4x_4^2 - 5x_5 \geqslant 0,\\
g_2(\vec{x}) &= 282 - 7x_1 - 3x_2 - 10x_3^2 - x_4 + x_5 \geqslant 0,\\
g_3(\vec{x}) &= 196 - 23x_1 - x_2^2 - 6x_6^2 + 8x_7 \geqslant 0,\\
g_4(\vec{x}) &= -4x_1^2 - x_2^2 + 3x_1x_2 - 2x_3^2 - 5x_6 + 11x_7 \geqslant 0,
\end{aligned}$$

the search space defined by the following bounds:

$$-10 \leqslant x_i \leqslant 10, \quad i = 1, \ldots, 7.$$

The global minimum is located at: $\vec{x}^*=(2.330499,, 1.951372, 0.4775414, 4.365726, 0.6244870, \quad 1.038131, 1.594227)$ and the optimal solution $P_4(\vec{x}^*) = 680.630\ 0573$. In this point the constraints g_1 and g_4 are active.

The problem P_5 consists of minimizing:

$$P_5(\vec{x}) = x_1 + x_2 + x_3 \tag{17}$$

subject to:

$$\begin{aligned}
g_1(\vec{x}) &= 1 - 0.0025(x_4 + x_6) \geqslant 0,\\
g_2(\vec{x}) &= 1 - 0.0025(x_5 + x_7 - x_4) \geqslant 0,\\
g_3(\vec{x}) &= 1 - 0.01(x_8 - x_5) \geqslant 0,\\
g_4(\vec{x}) &= x_1x_6 - 833.33252 - 100x_1 + 83333.333 \geqslant 0,\\
g_5(\vec{x}) &= x_2x_7 - 1250x_5 - x_2x_4 + 1250x_4 \geqslant 0,\\
g_6(\vec{x}) &= x_3x_8 - 1250000 - x_3x_5 + 25000x_5 \geqslant 0,
\end{aligned}$$

the search space defined by:

$$\begin{aligned}
&100 \leqslant x_1 \leqslant 10000,\\
&1000 \leqslant x_i \leqslant 10000, \quad i = 2, 3,\\
&10 \leqslant x_i \leqslant 1000, \quad i = 4, \ldots, 8.
\end{aligned}$$

The optimal solution is $\vec{x}^* = (579.31671359.943, 5110.071,, 182.0174, 295.5985, 217.9799, 286.4162, \ 395.5979)$ with optimal value $P_5(\vec{x}^*) = 7049.330923$.

Finally, P_6 is the optimization of a tensional compression string design problem taken from Huang, Wang, and He (2007). It consists in minimizing the weight $(\vec{x})$ of a tension/compression spring subject to various constraints like deflection, surge frequency, limitations on the outside diameter. The design variables are the wire

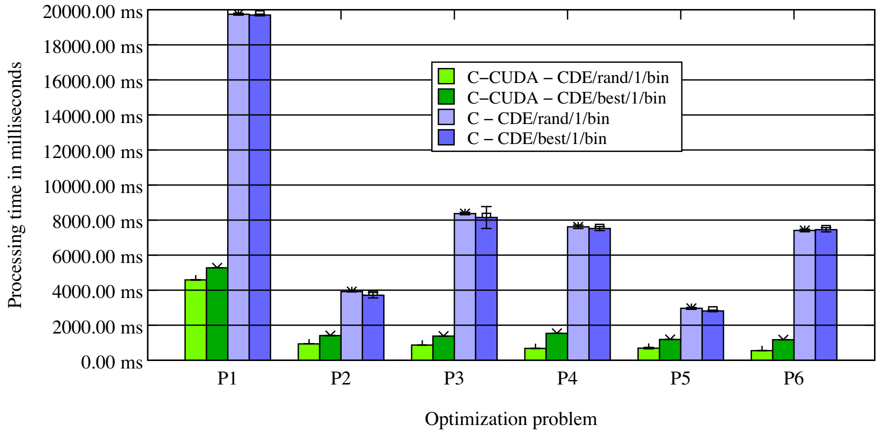

Fig. 3. Time required for computing solutions for P_1 up to P_6.

```
__global__ void generateA(float* newPopA, float* oldPopA, float* oldPopB,
    int* shuffled, int nPop, int nPar, float* RAND, float F, float f_cross,
     float *bounds, int nRes, int iteration){
    float trialVector[NPAR];
    int r1, r2, r3;
    int i = threadIdx.x;
    // mutation

    pick3(shuffled, nPop, &r1, &r2, &r3, iteration, i);
    makeTrial(oldPopA, trialVector, r1, r2, r3, F, nPar);
   __syncthreads();

    // crossover
   doCrossover(oldPopA, newPopA, trialVector, i, nPar, f_cross, RAND, iteration,
        i, 1);
   __syncthreads();
}
```

Listing 3. Generation individuals of population A.

diameter x_1, the mean coil diameter x_2 and the number of active coils x_3.

Minimize:

$$P_6(x) = (x_3 + 2)x_2x_1^2, \tag{18}$$

subject to:

$$g_1(\vec{x}) = 1 - \frac{x_2^3x_3}{71785x_1^4} \leqslant 0,$$
$$g_2(\vec{x}) = \frac{4x_2^2 - x_1x_2}{12566(x_2x_1^3 - x_1^4)} + \frac{1}{5108x_1^2} - 1 \leqslant 0,$$
$$g_3(\vec{x}) = 1 - \frac{140.45x_1}{x_2^2x_3} \leqslant 0,$$
$$g_4(\vec{x}) = \frac{x_1 + x_2}{1.5} - 1 \leqslant 0$$

with:

$$0.5 \leqslant x_1 \leqslant 2.0,$$
$$0.25 \leqslant x_2 \leqslant 1.30,$$
$$2 \leqslant x_3 \leqslant 15.$$

The global solution is $P_6(\vec{x}^*) = 0.0127$, with $\vec{x}^* = (0.05, 0.315, 14.25)$.

All contrained optimization problems from P_2 up to P_6 described above are transformed according to formulation given in (6), resulting in min–max optimization problems.

6.2. CDE experimental settings

The experiments were run onto a CPU AMD Athlon x2 5200+ 2.7 GHz dual core with 512KB of cache per core, 3GB DDR2 800 MHz of main memory, and PCI-E 2.0. The GPU is a NVIDIA GTX 285 with 1GB of GDDR3 with Nvidia driver version 180.22 and CUDA version 2.1. It has 30 streaming multiprocessors (SMs), each SM with 8 streaming processors (SPs) resulting in a total of 240 SPs. In order to assess the computational performance we run the DE algorithm coded in C as well.

The DE algorithm requires some parameters to be set up a priori. Table 1 lists the parameters used on the experiments. Those parameters were set up based on the works by Storn and Price (1997) and previous experience.

6.3. Experimental results and discussions

6.3.1. Case study 1

In all experiments the population size was set to 50 and held constant. Each experiment was run until the maximum number of iterations, 200, was reached. In this experimental run we used the *CDE*/*rand*/1/*bin* variation of the algorithm. The computing time for C-CUDA, and C for all six optimization problems are given in Table 2. The best computational performance was achieved for $P_6(\vec{x})$, which corresponds to a speedup of circa 13 times over the C implementation. Each experiment was run 20 times for different initializations of the population.

```
__global__ void evaluateA(float* newPopA, float* oldPopA, float* oldPopB,
    int* shuffled, int nPop, int nPar, float* RAND, float F, float f_cross,
     float *bounds, int nRes,int iteration, float *newFit, float *oldFit){
    int i = threadIdx.x;
    int j = blockIdx.x;
    __shared__ float fitness[MAXPOP];

    fitness[i]=target(&newPopA[j*nPar], &oldPopB[i*nRes]);
    __syncthreads();

    if(i==0){
    int k;
    float max = -HUGE_VAL;
    for(k=0; k<nPop; k++){
        if(fitness[k]>max){
       max = fitness[k];
        }
    }
    newFit[j] = max;
       if (oldFit[j] < newFit[j]){
           copyIndividual(&oldPopA[j*nPar],&newPopA[j*nPar],nPar);
           newFit[j] = oldFit[j];
       }
    }
}
```

Listing 4. Evaluation of the new individuals.

Table 1
Parameters setup for the first round of experiments.

Number of global iterations(*MaxIterations*)	200
Maximization/Minimization iterations(*MaxGen*1/*MaxGen*2)	20
Mutation factor (F_m)	0.7
Population size (n)	50
Crossover probability (C_r)	0.7
Penalty factor (r)	10000000

Table 2
Running time results in milliseconds of *CDE*/*rand*/1/*bin* using C-CUDA and C for the benchmark optimization problems.

Problem	Language	Mean	Std. dev.	Speedup
P_1	C-CUDA	4584.80	12.70	4.32
	C	19800.50	55.60	
P_2	C-CUDA	934.00	2.70	4.19
	C	3910.00	20.80	
P_3	C-CUDA	866.20	1.6	9.66
	C	8355.00	70.30	
P_4	C-CUDA	674.70	4.50	11.31
	C	7620.00	95.70	
P_5	C-CUDA	683.30	37.50	4.27
	C	2955.00	55.10	
P_6	C-CUDA	550.80	4.80	13.45
	C	7395.00	77.50	

The speedup for each problem is shown is Table 2. From this table we can see that speed up was achieved for all problems. In terms of quality of the obtained solution for Case Study 1, we show the results in Table 3.

6.3.2. Case study 2

We carried out another set of simulations with the same problems, iteration number and number of individuals in the population, but using *CDE*/*best*/1/*bin* instead of *CD*/*rand*/1/*bin*. The computing time for the C-CUDA, and C for all six optimization problems are given in Table 4. The best computational performance was achieved for P_6, which corresponds to a speedup of circa 6 times over the C implementation. Each experiment was run 20 times with different initializations of the population.

Table 3
Convergence results using the *CDE*/*rand*/1/*bin*.

Problem	Optimal	Best	Median	Mean	Std. dev.	Worst
P_1	0.00	0.00	−0.00	−0.00	0.00	0.00
P_2	−15.00	−15.00	−14.87	−15.00	0.56	−12.45
P_3	24.31	24.47	24.48	24.48	0.00	24.51
P_4	680.63	680.63	680.63	680.63	0.00	680.63
P_5	7049.33	7047.83	7049.34	7048.03	2.10	7054.14
P_6	0.01	0.01	0.01	0.01	0.00	0.01

Table 4
Running time results in milliseconds of *CDE*/*best*/1/*bin* using C-CUDA and C for the benchmark optimization problems.

Problem	Language	Mean	Std. dev.	Speedup
P_1	C-CUDA	5277.00	3.00	3.73
	C	19690.00	45.60	
P_2	C-CUDA	1405.20	5.50	2.64
	C	3710.00	33.70	
P_3	C-CUDA	1376.80	2.40	5.92
	C	8145.00	241.00	
P_4	C-CUDA	1186.10	4.70	6.33
	C	7510.00	116.60	
P_5	C-CUDA	1187.70	2.70	2.37
	C	2800.00	45.70	
P_6	C-CUDA	1173.00	2.90	6.24
	C	7435.00	124.40	

In Table 4 we show the speed up comparing the C-CUDA and C version of the *CDE*/*best*/1/*bin* variant of the algorithm.

In terms of quality of the obtained solution for Case Study 2, we show the results in Table 5.

6.3.3. Discussions

Fig. 3 shows the computing time required for all the six optimization problems using the *CDE*/*rand*/1/*bin* and *CDE*/*best*/1/*bin*. Since the speedup depends on the complexity of the objective function and the constrains, it becomes larger as the objective functions get more complex. Moreover, the speedup was smaller in case study 2 due to the intrinsic sequential operation of finding the best individual of the current generation.

Table 5
Convergence results using the *CDE/best/1/bin*.

Problem	Optimal	Best	Median	Mean	Std. dev.	Worst
P_1	0.00	0.00	0.00	0.00	0.02	−0.09
P_2	−15.00	−15.00	−13.76	−13.83	3.74	−11.28
P_3	24.31	24.47	24.48	24.48	0.01	24.49
P_4	680.63	680.63	680.63	680.63	0.00	680.64
P_5	7049.33	7047.87	7066.34	7052.44	32.74	7184.93
P_6	0.01	0.01	0.01	0.01	0.00	0.01

Table 6
Running time results in milliseconds of *CDE/rand/1/bin* using C-CUDA and C for the benchmark optimization problems with population sizes P_1 and P_2 of 100 individuals.

Problem	Language	Mean	Std. Dev.	Relative increase
P_1	C-CUDA	5006.80	11.60	1.09
	C	90251.00	80.90	4.56
P_2	C-CUDA	1300.10	2.10	1.39
	C	14852.90	100.10	3.79
P_3	C-CUDA	1162.00	2.70	1.34
	C	32673.00	192.70	3.90
P_4	C-CUDA	858.20	2.80	1.27
	C	29993.00	250.70	3.94
P_5	C-CUDA	907.00	3.50	1.31
	C	11384.00	223.30	3.85
P_6	C-CUDA	486.50	3.10	1.13
	C	29210.00	629.40	3.94

We consider that both variants of the algorithm successfully found the minimum of the functions. The *best* variant performed slightly worse. Specially if one considers the standard deviation and worst values. This is to the fact that this version is more prone to get stuck into local minima.

To better evaluate the gain of the parallelization we performed another set of experiments doubling the size of the population. We observed an smaller relative increase in the C-CUDA version compared to the serial version. Table 6 shows the obtained times. This means that the parallel is both faster and more scalable compared to the serial version. When we double the population the computation time of parallel version increases by a factor of less than two. On the other hand, the computation time of the serial version increases by a factor of more than two. It is also clear that the standard deviation of the parallel version is much smaller compared to the serial version.

Since many researchers in evolutionary computation implement their versions of EA in C, the implementation in C-CUDA offers a great advantage for accelerating the computing time for challenging optimization problems. We have shown that in the two most common variants of the algorithm, implementation in C-CUDA presents a considerably improvement in terms of computational performance.

7. Conclusions

In this paper, we present an implementation of a co-evolutionary differential evolution algorithm in C-CUDA. We noticed a reduction of the computing time as compared to a C implementation and at the same time good convergence results. We also noticed that the scalability of the algorithm considering the population size, the computation time of the parallel version is increased by a much smaller factor than the computation time of the serial version. Therefore, this kind of technology is very encouraging for solving challenging real-world optimization problems, since the CDE algorithm has inherently parallel nature. The area of evolutionary computation may take profits of this technology regarding speedup. Work in progress is investigating the implementation of enhanced versions of the DE algorithm and a way to generate random numbers during the execution of the algorithm. This is a very important issue in evolutionary algorithms.

References

Barbosa, H. J. C. (1996). A genetic algorithm for min–max problems. In *Proccedings of the first international conference on evolutionary computation and its applications* (pp. 99–109).

Barbosa, H. J. C. (1999). A coevolutionary genetic algorithm for constrained optimization. CEC 99. In *Proceedings of the IEEE Congress on Evolutionary Computation* (vol. 3, pp. 1605–1611).

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., & Skadron, K. (2008). A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing, 68*(10), 1370–1380.

Cramer, A., Sudhoff, S., & Zivi, E. (2009). Evolutionary algorithms for minimax problems in robust design. *IEEE Transactions on Evolutionary Computation, 13*(2), 444–453.

Díez, D. C., Moser, D., Schoenegger, A., Pruggnaller, S., & Frangakis, A. S. (2008). Performance evaluation of image processing algorithms on the GPU. *Journal of Structural Biology, 164*(1), 153–160.

Du, D., & Pardalos, P. M. (1995). *Minimax and applications*. Norwell, MA: Kluwer.

GPGPU, 2010. Gpgpu homepage. URL http://www.gpgpu.org.

Hillis, W. D. (1990). Co-evolving parasites improve simulated evolution as an optimization procedure. *Phys. D, 42*(1-3), 228–234.

Huang, F. Z., Wang, L., & He, Q. (2007). An effective co-evolutionary differential evolution for constrained optimization. *Applied Mathematics and Computation, 186*(1), 340–356.

Krohling, R. A., & dos Santos Coelho, L. (2006). Coevolutionary particle swarm optimization using gaussian distribution for solving constrained optimization problems. *IEEE Transactions on Systems, Man, and Cybernetics, Part B, 36*(6), 1407–1416.

Laskari, E. C., Parsopoulos, K. E., & Vrahatis, M. N. (2002). Particle swarm optimization for minimax problems. In *Proceedings of the IEEE 2002 congress on evolutionary computation* (pp. 1582–1587).

Li, J., Zhang, L., & Liu, L. (2009). A parallel immune algorithm based on fine-grained model with GPU-acceleration. *International Conference on Innovative Computing, Information and Control*, 683–686.

Li, J.-M., Wang, X.-J., He, R.-S., & Chi, Z.-X. (2007). An efficient fine-grained parallel genetic algorithm based on. *GPU-accelerated*, 855–862.

Liu, B., Fernández, F. V., Gielen, G., Castro-López, R., & Roca, E. (2009). A memetic approach to the automatic design of high-performance analog integrated circuits. *ACM Transactions on Design Automation Electronic Systems, 14*(3), 1–24.

Luebke, D. (2008). Cuda: Scalable parallel programming for high-performance scientific computing. In *5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro* (pp. 836–838).

NVidia. (2007). CUDA compute unified device architecture programming. URL http://www.developer.download.nvidia.com/compute/cuda/1_1%NVIDIA_CUDA_Programming_Guide_1.1.pdf.

NVidia. (2010). Nvidia homepage. URL <http://www.nvidia.com>.

Paredis, J. (1994). Steps towards coevolutionary classification neural networks. *Artificial Life IV*, 359–365.

Qin, A., Huang, V., & Suganthan, P. (2009). Differential evolution algorithm with strategy adaptation for global numerical optimization. *IEEE Transactions on Evolutionary Computation, 13*(2), 398–417.

Rosin, C. D., Belew, R. K. (1995). Methods for competitive co-evolution: Finding opponents worth beating. In *Proceedings of the sixth international conference on genetic algorithms* (pp. 373–380).

Rosin, C. D., & Belew, R. K. (1997). New methods for competitive coevolution. ****Evol. Comput., 5*(1), 1–29.

Schenk, O., Christen, M., & Burkhart, H. (2008). Algorithmic performance studies on graphics processing units. *Journal of Parallel and Distributed Computing, 68*(10), 1360–1369.

Seredynski, F., & Zomaya, A. Y. (2002). Parallel and distributed computing with coevolutionary algorithms. In *International parallel and distributed processing symposium* (pp. 202–209).

Shi, Y., Krohling, R. A. (2002). Co-evolutionary particle swarm optimization to solve min–max problems (pp. 1682–1687).

Storn, R., & Price, K. (1997). Differential evolution – A simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization, 11*(4), 341–359.

Tahk, M.-J., & Sun, B.-C. (2000). Coevolutionary augmented lagrangian methods for constrained optimization. *IEEE Transactions on Evolutionary Computation, 4*(2), 114–124.

Veronese, L. P., Krohling, R. A. (2009). Swarm's flight: Accelerating the particles using C-CUDA. In *Proccedings of 2009 IEEE congress on evolutionary computation (CEC)* (pp. 3264–3270).

Veronese, L. P., Krohling, R. A. (2010). Differential evolution algorithm on the GPU with C-CUDA. In *Proccedings of 2010 IEEE congress on evolutionary computation (CEC)* (pp. 1–7).

Zhou, Y., & Tan, Y. (2009). GPU-based parallel particle swarm optimization. *Proceedings of the IEEE Congress on Evolutionary Computation*, 1493–1500.