A CUDA-Based Cooperative Evolutionary Multi-Swarm Optimization Applied to Engineering Problems

*Daniel Leal Souza[1,2] Otávio Noura Teixeira[1] Dionne Monteiro[2] Roberto Célio Limão de Oliveira[3]

[1]Laboratory of Natural Computing (LCN) – Area of Exact and Natural Sciences (ACET) – University Centre of Pará (CESUPA) – Belém - Brazil

[2]Laboratory of Applied Artificial Intelligence – Institute of Exact and Natural Sciences (ICEN) – Federal University of Para (UFPA) – Belém - Brazil

[3]Post-Graduate Program in Electrical Engineering (PPGEE), Institute of Technology (ITEC) – Federal University of Para (UFPA), Belém - Brazil

Abstract

This paper presents a variation of Evolutionary Particle Swarm Optimization applied to the concept of master/slave swarm with mechanism of sharing data for the acceleration of convergence. The implementation called Cooperative Evolutionary Multi-Swarm Optimization on Graphics Processing Units (CMEPSO-GPU) consists in using thousands of *threads* in various slave swarms on the CUDA parallel architecture, where each one works in a parallel and cooperative way in order to improve the search for best result and reduce the number of iterations. The use of CMEPSO-GPU applied to engineering problems showed superior results when compared to other implementations found in the scientific literature.

Keywords: PSO, EPSO, enxame, CUDA, multi-populações.

Authors' contact:
*daniel.leal.souza@gmail.com
{onoura,dionnecm}@gmail.com
limão@ufpa.br

1. Introduction

In recent years, the evolution in the computational resources becomes possible significant advances in the techniques of search and optimization. [Bastos Filho et al. 2008] affirms that these advances in the resources have brought success in the construction of more efficient solutions, also in studies of metaheuristics, which guide the heuristics to obtain optimal solutions applied in N-dimensional search space problems. These problems may be included in a practical range of applications in industry,in management of basic supplies such as electric energy, gas and petroleum refining [Lopes and Takahashi 2011].

One of the most well known metaheuristics with large use on engineering is Particle Swarm Optimization (PSO). Among its variants, Evolutionary Particle Swarm Optimization (EPSO) proposed in [Miranda and Fonseca 2006] stands out for including the set of operations of evolutionary strategies such as replication, mutation and selection applied in an PSO environment.

Another type of variation of PSO algorithm involves the concept of multi-population(swarm slaves). It is important to mention the work developed by [Van Den Bergh and Engelbrecht 2004], which uses the model of search partitioning in genetic algorithms to PSO, in order to reduce the deterioration of performance as the dimensionality of search space increases. In general, most of these metaheuristics use computing systems with parallel and distributed architecture under an inter-communicable environment between two or more swarms. The ultimate goal is a more efficient search and the exchange of information obtained by each of them with order to compare and refine the search based on the results already obtained by the neighboring clusters.

In the matter of advance of hardware resources, we can highlight the evolution of Graphics Processing Units (GPUs). Since 2003, the many-cores processors, strongly represented by GPU, has shown greater superiority in terms of speed, especially operations involving floating point data. [Kirk and Hwu 2010] Currently, many computing solutions, especially for applications in the scientific area, are developed by taking advantage of thousands of cores available in the multiprocessors found in a board video, thanks to technologies such as NVIDIA Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL).

The implementation of EPSO in a cooperative and multi-swarm model applied under GPGPU paradigm (general-purpose computing on graphics processing units) has a number of benefits, since each element can be treated by a *thread*, which tends to contribute to reducing the execution time, in addition to the potential increase in performance of search and optimization.

This paper presents an algorithm based on both EPSO and PSO mechanisms over a cooperative approach of master/slave swarm applied under CUDA architecture, called Cooperative Evolutionary Multi-Swarm Optimization on Graphics Processing Units (CMEPSO-GPU).

For comparison and validation, we used two engineering problems widespread in the scientific literature: Welded Design beam (WBD); Minimization of the Weight of a Tension/ Compression Spring (MWTCS).

2. Related Works

During the literature search done for this work, various publications involving the use of PSO on GPUs were found in the scientific literature. Some of the most relevant works to the context of this paper are briefly discussed below:

- ***Collaborative Multi-Swarm PSO for Task Matching Using Graphics Processing Units [Solomon et al. 2011]***: In this work, the authors expose an implementation of the cooperative and multi-swarm model of PSO executed under CUDA architecture for a task matching operation. This work uses a multi-swarm environment model similar to [Van Den Bergh and Engelbrecht 2004]. The algorithm uses data of discrete and continuous type, and the entire process is carried out continuously and in the end, the authors apply rounding to discrete values. The results obtained in this study show considerable improvements in processing time and in the values obtained at the end of execution when compared to a PSO of a swarm.
- ***GPU-based Asynchronous Particle Swarm Optimization [Mussi et al. 2011]***: This work presents an approach for asynchronous execution of PSO on the GPU and exposes the problems of a synchronous implementation. In this implementation, the authors propose a data processing where each *thread* represents a specific component of the particle, and particles are treated in a loop. The obtained results, in relation to the sequential version show an increase of performance of approximately 300-fold in tests using functions such as Rastrigin. No tests with restrictive functions were presented in this paper.

We conducted an extensive search in literature regarding the publications involving the EPSO algorithm, however, no work with respect to implementation of the method created by [Miranda and Fonseca 2006] on GPUs under CUDA architecture and its use in an multi-swarm approach has been found.

3. Theoretical Basis: A Brief Description of Classic and Evolutionary Particle Swarm Optimization for CMEPSO

3.1 Particle Swarm Optimization (PSO)

PSO is a technique for stochastic optimization of nonlinear and continuous functions developed by James Kennedy and Russell Eberhart [Kennedy and Eberhart 1995]. It was based from studies related to the "social behavior" observed in some species

of birds. The PSO algorithm simulates a population of particles called "swarm", which operates inside of a search space predetermined [Shi and Eberhart 2000]. A particle is represented by two vectors storing values of position and speed, where the value of number of vector elements is equal to the amount of N variables corresponding to an N-dimensional search space. Based on a collective and cooperative exploration, the particles tend to progress in the search process as new and best results are found. The description of the movement of the particles is defined by equations for position (1), velocity (2) and inertia weight (3) proposed in [Eberhart and Shi 1998].

$$V_i^{(t+1)} = wV_i^{(t)} + Rnd_1C1(b_i - X_i^{(t)}) + Rnd_2C2(b_g - X_i^{(t)}) \quad (1)$$

$$X_i^{(t+1)} = X_i^{(t)} + V_i^{(t+1)} \quad (2)$$

$$w = \frac{(k-1.0)}{(I_{MAX}-1)(-W_{MAX}+W_{MIN})+W_{MAX}} \quad (3)$$

The variables of equations (1), (2) and (3) are: Time step (***t***); particle's index (***i***); inertia weight, where the value decreases with each iteration obeying a linear behavior (***w***); particle velocity (V); position of the particle (***X***); acceleration constants (***C1*** and ***C2***); particle's local best (b_i); swarm's global best (b_g); random numbers sampled from an uniform distribution between 0 and 1 (Rnd_1 *and* Rnd_2); limits for maximum and minimum value of inertia weight W_{MAX} and W_{MIN}); maximum number of iterations (I_{MAX}); iteration index (***k***).

In order to prevent particles from escaping from the search space, the position values of the particle are subjected to boundary conditions. This feature is used only if the value is outside the minimum or maximum limits. In order to prevent some particles may get trapped in local bests, the boundary condition includes a new constant called correction factor, which acts in the process of readjustment of position and velocity. It is important to note that if the values of position of the particles exceed the limits, the speed is changed to the negative value of its current value. The correction of values of variables of velocity and position are described by equations (4) and (5):

$$V_i^{(t+1)} = \begin{cases} -V_i^{(t+1)}, & \text{if } (X_i^{(t+1)} < X_{MIN}) \\ V_i^{(t+1)}, & \text{if } (X_{MIN} \le X_i^{(t+1)} \le_{XMAX}) \\ -V_i^{(t+1)}, & \text{if } (X_i^{(t+1)} > X_{MAX}) \end{cases} \quad (4)$$

$$X_i^{(t+1)} = \begin{cases} X_{MIN} + Rnd_1COR, & \text{if } (X_i^{(t+1)} < X_{MIN}) \\ X_i^{(t+1)}, & \text{if } (X_{MIN} \le X_i^{(t+1)} \le X_{MAX}) \\ X_{MAX} - Rnd_1COR, & \text{if } (X_i^{(t+1)} > X_{MAX}) \end{cases} \quad (5)$$

The variables of equations (4) and (5) are: Limits for maximum and minimum value of position (X_{MAX} and X_{MIN}); correction factor (***COR***); random numbers sampled from an uniform distribution between 0 and 1 (Rnd_1).

The PSO algorithm used as a basis for implementation of the CMEPSO-GPU algorithm is described in Algorithm 1:

Algorithm 1 Classic PSO

Initialize the population (velocity and positions);
Evaluate fitness for each particle in the swarm;
Initialize the local best (b_i);
Initialize the global global (b_g);
for $k = 1 \rightarrow I_{MAX}$ **do**
  Update inertia weight using equation *(3)*;
  Update velocity using equation *(1)*;
  Update position using equation *(2)*;
  Apply velocity correction using equation *(4)*;
  Apply position correction using equation *(5)*;
  Update fitness for each particle;
  Update particle's local best (b_i);
  Update swarm's global best (b_g);
end for

3.1 Evolutionary Particle Swarm Optimization (EPSO)

The evolutionary PSO (EPSO) developed by [Miranda and Fonseca 2006] is a metaheuristic that adds the mechanism of evolutionary strategies (EE) to the PSO algorithm, where the operators of classical recombination are replaced by particles' motion rules. According to [Miranda and Fonseca 2006], EPSO can be classified in two forms, by given the mechanisms of the algorithm, it can be interpreted as a variant of the PSO or as a variant of EE.

The mechanisms of evolutionary strategy found in EPSO of [Miranda and Fonseca 2006] and [11] are: ***Replication*** of particles; ***mutation*** of the following weights: inertia weight, acceleration constants (***C1/C2***) and the values of the global best; ***reproduction*** of new particles by the execution of equation (5) with weights mutated; ***evaluation*** of new individuals; ***selection*** of the best individuals.

According to [Miranda and Fonseca 2006] and [11] the addition of the evolutionary mechanisms to PSO provides a search system more robust, since the mutation can produce a better result at runtime. The changes in the equation of velocity proposed by [Miranda and Fonseca 2006] are described below. It is important to note that the equations of inertia weight (3) and position (2) are not changed.

$$V_i^{(t+1)} = mw^* V_i^{(t)} + mC1^*(b_i - X_i^{(t)}) + mC2^*(b_g^* - X_i^{(t)}) \quad (6)$$

The variables of equations (6) are: Time step (***t***); particle's index (***i***); inertia weight subjected to the process of mutation (***mw***); particle velocity (***V***); position of the particle (***X***); acceleration constants submitted to the mutation process (***mC1*** and ***mC2***); particle's local best (b_i); swarm's global best (b_g); a marker of the variables subjected to the mutation process (*);

The differences between the equations for velocity (1) and (5) are absence of random number generators, in addition to the mutation process of the the weights ***mw***, ***mC1***, ***mC2*** and the values of the global best (specified with the symbol '*'). The description of the mutation process are defined by equations (7), (8), (9) and (10) respectively.

$$mw_i^* = w + (1 + \sigma N(0,1)) \quad (7)$$

$$mC1_i^* = C1 + (1 + \sigma N(0,1)) \quad (8)$$

$$mC2_i^* = C2 + (1 + \sigma N(0,1)) \quad (9)$$

$$b_g^* = b_g + (1 + \sigma_g N(0,1)) \quad (10)$$

The variables of equations (7), (8), (9) and (10) are: Mutated inertia weight (***mw***); acceleration constants submitted to the mutation process (***mC1*** and ***mC2***); particle's index (***i***); swarm's global best (b_g); disturbance constant for the global best (σ_g); learning parameter for mutation of the inertia weight and acceleration constants (s); standard normal distribution (***N(0,1)***); marker of variables subjected to the mutation process (*).

Given the description above, the customized EPSO algorithm (based on the work developed by [Miranda and Fonseca 2006]) and used as the basis for the implementation of CMEPSO-GPU algorithm is described in the Algorithm 2:

Algorithm 2 Evolutionary PSO

Initialize the population (velocity and positions);
Evaluate fitness for each particle in the swarm;
Initialize the local best (b_i);
Initialize the global best (b_g);
for $k = 1 \rightarrow I_{MAX}$ **do**
  Update inertia weight using equation *(3)*;
  for all particles in the swarm, **do**
    Update velocity from the original particles using equation *(1)*;
    Update position from the original particles using equation *(2)*;
    Apply velocity correction to the original particles using equation *(4)*;
    Apply position correction to the original particles using equation *(5)*;
    Update fitness for the original particles;
    Update particle's local best (b_i);
    Update swarm's global best (b_g);
    Replicate particle N times;
    Apply mutation for weights (***w, C1, C2***) from all replicated particles;
    Update velocity from the replicated particles using equation *(6)*;
    Update position from the replicated particles using equation *(2)*;
    Apply velocity correction to the replicated particles using equation *(4)*;
    Apply position correction to the replicated particles using equation *(5)*;
    Update fitness for the replicated particles;
    Select the best particles;
    Update particle's local best (b_i);
    Update swarm's global best (b_g);
  end for all
end for

4. Compute Unified Device Architecture (CUDA)

CUDA architecture provides a set of extensions to the C language, which allow the implementation of parallel algorithms in video cards [Kirk and Hwu 2010] for general purpose. GPUs with CUDA architecture have many cores which allows to run collectively thousands of independent small parts of processing, called *threads* [Kirk and Hwu 2010].

The CUDA SDK (Software Development Kit) includes a compiler adapted for heterogeneous computing paradigm GPGPU (General Purpose Computing on Graphics Processing Units) and other tools capable of supporting heterogeneous applications, which have serial and parallel parts performed on the CPU (*host*) and GPU (*device*), respectively [Leite et al. 2010].

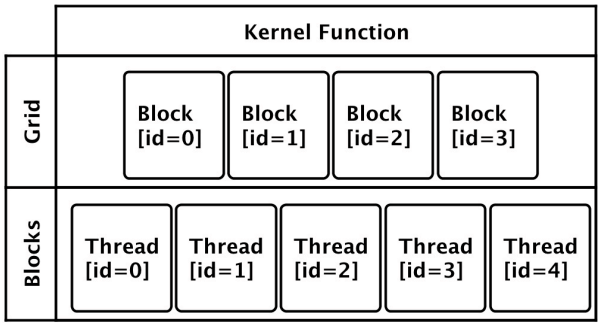
4.1 Thread Organization

To keep the organization in the executing data parallelism and data distribution effectively with control of memory access, CUDA uses three levels of organization: *thread*, *block* and *grid*. The explanation of each follows [Kirk and Hwu 2010]:

- ***Thread***: It is the basic unit of execution. Each *thread* is responsible for a copy of the program for a certain quantity of data. In some solutions, it is possible each *thread* stay responsible for an address of a vector or a matrix;
- ***Block*:** This structure stores a vector or a matrix (2D or 3D) of *threads*. All *blocks* have the same quantities of *threads*. Through this organization, it is possible to use synchronization of *threads* to *block* level;
- ***Grid***: It is the set of all *threads* that a *kernel* will use. A *Grid* is a vector or matrix of *blocks*. In multi-GPU environments, the *Grids* can not exchange information among themselves.

Figure 1 shows the schematic organization of *threads* and blocks on CUDA architecture, where the parallel application (executed into a function called *kernel*) uses a quantity of four blocks with five *threads* each.

Figure 1. Scheme of *threads* and blocks. Adapted from [NVIDIA Corp. 2012]



5. Cooperative Multi-Swarm Approach for EPSO On The CUDA Architecture (CMEPSO-GPU)

The CMEPSO algorithm is a result of the integration between the mechanisms found on the EPSO and PSO, acting under a multi-swarm approach.

It consists in an environment where two or more slave swarms cooperate with the master swarm the values of the particles classified as being the global best found to current iteration. The optimization process that occurs in the master swarm tends to enhance the global best of the slave swarms, and contribute in finding the global best related to the master swarm.

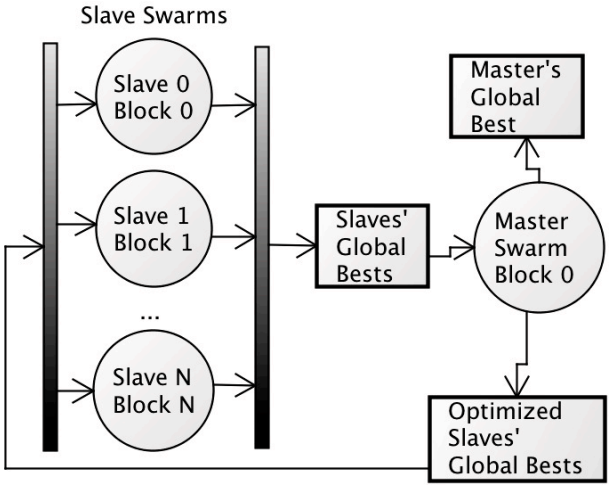
In relation to the parallelism applied to the slave swarms, the CUDA architecture provides a programming environment that makes possible to execute EPSO and PSO algorithms with high level of parallelism.

5.1 Multi-Swarm Structure With Slave/Master Approach

The basic structure of the multi-swarm environment in CMEPSO algorithm is based on the model found in the MCPSO algorithm (Multi-Populational Cooperative Particle Swarm Optimization). Proposed in [Niu et al. 2005], this approach is organized so that the slave swarms are executed in parallel, without exchanging information among themselves, where they provide their best solutions (global best) to the master swarm. The cooperativity of data is applied only in the relationship between the master swarm and the slave swarms.

The master swarm updates its particles, taking into consideration the information of the best solution found among all slave swarms. Note that this information is integrated as a third component in the velocity update equation (more details in section 5.1.2). Figure 2 shows the architecture of CMEPSO algorithm based on the model proposed by [Niu et al. 2005] and adapted for CUDA.

Figure 2. Schematic illustration of the master/slave model for the CMEPSO-GPU algorithm. Adapted from [Niu et al. 2005].



Based on the parallel implementation in CUDA, each *thread* is responsible for the manipulation of a particle (one *thread*, one particle (cardinality 1:1)), each block being responsible by a swarm (one *block*, one swarm (cardinality 1:1)). Figure 3 shows the arrangement of particles and slave swarms in the CMEPSO-GPU algorithm.

Figure 3. Scheme of the slave swarms and its particles in CUDA.

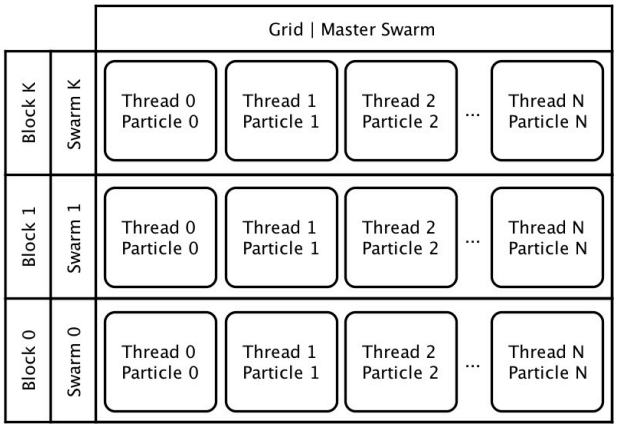
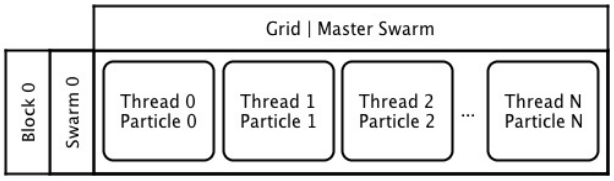


Figure 4 shows the arrangement of particles and master swarm in the CMEPSO-GPU algorithm.

Figure 4. Scheme of the master swarms and its particles in CUDA.



5.1.1 Equations for Slave Swarm

The slave swarms in CMEPSO-GPU algorithm operates in an parallel and independent way in the search and optimization of the problem and they all share the same amount of particles. Basically each swarm executes in parallel the process described in Algorithm 2 with some changes.

A few important points to be highlighted about the velocity equations are described below:

- The updating equations for velocity and position of original particles are not changed (equations (1) and (2) respectively);
- The update equation of the velocity applied to the replicas is modified in order to add variables wich generate random numbers uniformly in a range between zero and one. The addition of these variables allow a higher exploration in the search space by the replicas.

Changes in the equation (6) to the generated replicas in the slave swarms can be seen in equation (11).

$$V_i^{(t+1)} = mw^* V_i^{(t)} + Rnd_1 mC1^* (b_i - X_i^{(t)}) + Rnd_2 mC2^* (b_g^* - X_i^{(t)}) \quad (11)$$

The variables of equation (11) are: Time step (***t***); particle's index (***i***); mutated inertia weight (***mw***); velocity of the particle(***V***); position of the particle (***X***); random numbers sampled from an uniform distribution between 0 and 1 (***Rnd1*** and ***Rnd2***); acceleration constants submitted to the mutation process (***mC1*** and ***mC2***); particle's local best (***bi***); swarm's global best (***bg***); a marker of the variables subjected to the mutation process (*);

5.1.2 Equations for Master Swarm

The use of the master swarm under the context of the CMEPSO algorithm occurs after the parallel execution of slave swarms, where the values of the global best from every slave (b^S_g), as well as their values of velocity and fitness of each slave swarm are copied.

The update equations for velocity proposed in [Niu et al. 2005] are used in the master swarm, where is added the acceleration constant ***C3*** and the values of the best minimal/ maximum global found by the slave swarms (b^S_g). It is important to emphasize that the equation for velocity used by replicated particles (equation (11)) was also adapted to the equation, where the mutated factors were added. Changes in the equations for velocity proposed by [Niu et al. 2005] are described below.

$$V_i^{(t+1)} = mw^* V_i^{M(t)} + Rnd_1 C1^* (b_i^M - X_i^{M(t)}) + Rnd_2 C2^* (b_g^{M*} - X_i^{M(t)}) + Rnd_3 C3^* (b_g^S - X_i^{M(t)}) \quad (12)$$

$$V_i^{(t+1)} = wV_i^{M(t)} + Rnd_1 C1 (b_i^M - X_i^{M(t)}) + Rnd_2 C2 (b_g^M - X_i^{M(t)}) + Rnd_3 C3 (b_g^S - X_i^{M(t)}) \quad (13)$$

The variables of equations (12) and (13) are: Time step (***t***); particle's index (***i***); mutated inertia weight (***mw***); velocity of the particle (***V***); position of the particle (***X***); random numbers sampled from an uniform distribution between 0 and 1 (***Rnd1, Rnd2 and Rnd3***); mutated learning factor (***mC1, mC2 and mC3***); master swarm's local best (b^M_i); master swarm's global best (b^M_g); best global value obtained by the slave swarms (b^S_g); marker of variables subjected to the mutation process (*).

As well as the acceleration constants ***mC1*** and ***mC2***, the new constant ***mC3*** is also submitted to the mutation process. Equation (14) shows the mutation process for ***mC3***.

$$mC3_i^* = C3 + (1 + \sigma N(0,1)) \quad (14)$$

The variables of equation (14) are: Acceleration constants submitted to the mutation process (***mC3***); particle's index (***i***); learning parameter for mutation of the inertia weight and acceleration constants (σ); standard normal distribution (***N(0,1)***); marker of variables subjected to the mutation process (*).

5.1.3 Pseudo-code

Algorithm 3 describes the CMEPSO-GPU.

Algorithm 3 CMEPSO-GPU

```
Allocate memory space for reading/writing on the GPU;
run in parallel for each particle of the slave swarms
  Initialize the population (velocity and positions);
  Evaluate fitness for each particle in the swarm;
  Initialize the local best (bi);
  Initialize the global global (bg);
  Sync threads;
  if thread index = 0 do for each block
    Initialize global best (bg) of each slave swarm;
  end if
end run in parallel
Sync blocks;
run in parallel for each particle of the master swarm
  Initialize master's local best (bMi) with values of bg from each slave swarm;
  Sync threads;
  if ítthread index = 0 do
    Initialize master's global best (bMg);
  end if
end run in parallel
Sync blocks;
for k = 1 → IMAX do
  Update inertia weight using equation (3);
  run in parallel for each particle of the slave swarms
    Update velocity from the original particles using equation (1);
    Update position from the original particles using equation (2);
    Apply velocity correction to the original particles using equation (4);
    Apply position correction to the original particles using equation (5);
    Update fitness for the original particles;
    Update particle's local best (bi);
    Sync threads;
    if thread index = 0 do for each block
      Update global best (bg) of each slave swarm;
    end if
    Sync threads;
    Replicate particle N times;
    Apply mutation for each weights (w, C1, C2) from all replicated particles;
    Update velocity from the replicated particles using equation (11);
    Update position from the replicated particles using equation (2);
    Apply velocity correction to the replicated particles using equation (4);
    Apply position correction to the replicated particles using equation (5);
    Update fitness for the replicated particles;
    Select the best particles for the next iteration (slave swarm);
    Update particle's local best (bi);
    Sync threads;
    if thread index = 0 do for each block
      Update global best (bg) of each slave swarm;
    end if
  end run in parallel
  Sync blocks;
  Update the best global value found in the slave swarms (bSg);
  Send to the master swarm every slave swarms' global best (bg);
  run in parallel for each particle of the master swarm
    Update velocity from the original particles using equation (13);
    Update position from the original particles using equation (2);
    Apply velocity correction to the original particles using equation (4);
    Apply position correction to the original particles using equation (5);
    Update fitness for the original particles;
    Update particle's local best (bMi);
    Sync threads;
    if thread index = 0 do
      Update master's global best (bMg);
    end if
    Sync threads;
    Replicate particle N times;
    Apply mutation for weight (w, C1, C2, C3) from all replicated particles;
    Update velocity from the replicated particles using equation (12);
    Update position from the replicated particles using equation (2);
    Apply velocity correction to the replicated particles using equation (4);
    Apply position correction to the replicated particles using equation (5);
    Update fitness for the replicated particles;
    Select the best particles for the next iteration (master swarm)
    Update particle's local best (bMi);
    Sync threads;
    if thread index = 0 do
      Update master's global best (bMg);
    end if
  end run in parallel
  Sync blocks;
end for
```

6. Results

For comparison, the CMEPSO-GPU algorithm was subjected to tests with three engineering problems widely used in the scientific literature: Welded Beam Design (WBD) and Minimization of the Weight of a Tension / Compression Spring (MWTCS). The algorithm CMEPSO-GPU is written in the CUDA-C programming language into parallel code which is then executed on the GPU.

The parameter values are: Maximum number of iterations = 1000; quantity of particles for each slave swarm = 40; quantity of particles for master swarm = 10; number of replicates per particle = 5; quantity of slave swarms = 10; correction factor (***COR***) = 0.22; disturbance constant to the global best (σ_g) = 0.005; parameter of strategies for mutation of the inertia and acceleration factors (σ) = 0.22; factors ***C1*** and ***C2*** = 2.05; acceleration factor for the master swarm (***C3***) = 2.02; number of executed tests per function = 20.

The hardware description is shown in Table 1.

Table 1. Hardware Setup

Hardwar	Model	Specs
CPU	Intel Core i5	• **Clock**: 2.66 GHz; • **Cache**: 3 MB de Cache L3;
GPU	NVIDIA GeForce GT 330M	• **Number of CUDA cores**: 48; • **Clock**: 1.26 GHz; • **FLOPs/s**: 182 GFLOPs/s;

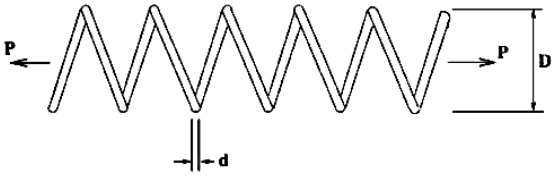
The results shown in Tables 2, 3 and 4 are the best values found in 20 executions for each problem.

6.1 Engineering Problems

6.1.1 Minimization of the Weight of a Tension/ Compression Spring (MWTCS)

Minimizing the weight of the tension / compression of a spring, subject to some restrictions, such as: Minimum deflection, shear stress, the wave frequency, limits on outside diameter and design variables [Teixeira et al. 2011]. The parameters for this problem are: Wire diameter (***d***); beam width (***D***); number of active coils (***P***). The constraints for MWTCS problem are represented by ***G*** variables (A total of four constraints). Figure 5 shows the layout of MWTCS problem.

Figure 5. Scheme of Minimization of the Weight of a Tension/ Compression Spring (MWTCS). Source: [Teixeira et al. 2011].



For the MWTCS problem, the CMEPSO-GPU algorithm was compared with the following pappers:

- [Teixeira et al. 2011]: GA with Social Interaction {1};
- [He and Wang 2007]: Co-evolutionary PSO {2};
- [Coello and Montes 2002]: Constraint-handling in GA {3};

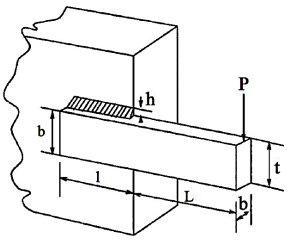
Table 2. Comparison of the best solutions for MWTCS

Variables	CMEPSO	{1}	{2}	{3}
X1(d)	0.050000	0.050180	0.051728	0.051989
X2 (D)	0.282023	0.279604	0.357644	0.363965
X3 (P)	2.000000	2.087959	11.244543	10.890522
G1	0.000000	-0.002840	-0.000845	-0.000013
G2	-0.235327	-0.249450	-0.000012	-0.000021
G3	-43.14613	-42.17600	-4.051300	-4.061338
G4	-0.778651	-0.780140	-0.727090	-0.722698
Violations	0	0	N/A	N/A
fitness	0.002820	0.002878	0.0126747	0.012681

6.1.2 Welded Beam Design (WBD)

Minimizing the manufacturing cost of a steel beam, subject to some restrictions, such as: Shear stress, efforts of beam in the bending, buckling load bar and deflection of the beam end and side constraints [Teixeira et al. 2011]. The parameters for this problem are: Thickness of the solder (***H***); beam width (***L***); thickness of the beam (***T***); length of the weld (***B***). The constraints for WBD problem are represented by ***G*** variables (A total of seven constraints). Figure 6 shows the layout of WBD problem.

Figure 6. Scheme of Welded Beam design (WBD). Source: [Teixeira et al. 2011].



For the WBD problem, the CMEPSO-GPU algorithm was compared with the following pappers:

- [Souza et al. 2011]: PSO with correcton factor and CUDA {4};
- [Teixeira et al. 2011]: GA with Social Interaction {1};
- [He and Wang 2007]: Co-evolutionary PSO {2};

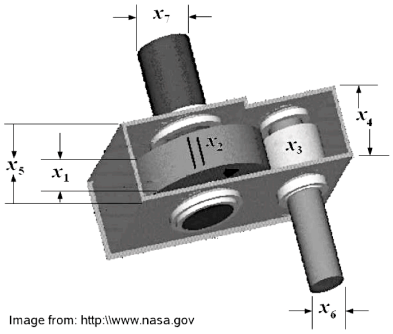
Table 3. Comparison of the best solutions for WBD

Variables	CMEPSO	{4}	{1}	{2}
X1(h)	0.205730	0.205735	0.171937	0.202369
X2 (l)	1.517675	1.517678	4.122129	3.544214
X3 (t)	9.036624	9.036624	9.587429	9.048210
X4 (b)	0.205730	0.205730	0.183010	0.205723
G1	-0.001953	-0.001953	-8.067400	-12.83979
G2	-0.001953	-0.001953	-39.33680	-1.247467
G3	0,000000	0,000000	-0.011070	-0.001498
G4	-3.607646	-3.607646	-3.467150	-3.429347
G5	-0.080730	-0.080730	-0.236390	-0.079381
G6	-0.235540	-0.235540	-16.02430	-0.235536
G7	-0.000488	-0.000977	-0.046940	-11.68135
Violations	0	0	0	N/A
fitness	1.458883	1.458889	1.664373	1.728024

6.1.3 Speed Reducer Design (SRD)

Speed reduction system by minimizing the weight subject to certain restrictions, such as: Bending of stress of gear teeth, surface tension, transverse deviations of the stems and tensions on the axis [Teixeira et al. 2011]. The variables of the problem are: width face (***X1***); teeth module(***X2***); number of teeth of the pinion (***X3***); length of the first axis between the bearings (***X4***); length of the second axis between the bearings (***X5***); diameter of the first axis (***X6***); diameter of the second axis (***X7***). The constraints for SRD problem are represented by ***G*** variables (A total of eleven constraints). Figure 7 shows the layout of the SRD problem.

Figure 7. Scheme of Speed Reducer Design (SRD). Source: [Teixeira et al. 2011]



For the SRD problem, the CMEPSO-GPU algorithm was compared with the following pappers:

- [Teixeira et al. 2011]: GA with Social Interaction {1};
- [Brajevic et al. 2010]: Modified Bee Colony Algorithm {5};
- [Canigna et al. 2010]: Simple Constrained PSO {6};

Table 4. Comparison of the best solutions for SRD

Variable	CMEPSO	{1}	{5}	{6}
X1	3.500000	3.500459	3.500000	3.500000
X2	0.700000	0.700020	0.700000	0.700000
X3	17.000000	17.00503	17.000000	17.000000
X4	7.300000	7.300251	7.300000	7.300000
X5	7.800000	7.800195	7.800000	7.800000
X6	2.900000	2.900041	3.350215	3.350214
X7	5.286684	5.286863	5.286683	5.286683
G1	-0.073915	-0.074364	-0.073915	-0.073915
G2	-0.197999	-0.198624	-0.197996	-0.197998
G3	-0.107955	-0.108202	-0.499172	-0.499172
G4	-0.901472	-0.901443	-0.901471	-0.901471
G5	-1.000000	-1.000000	-2.22E-16	0.000000
G6	0.000000	-0.000102	-3.331E-16	-5.00E-13
G7	-0.702500	-0.702403	-0.702500	-0.702500
G8	0.000000	-0.000103	0.000000	-1.00E-16
G9	-0.795833	-0.795801	-0.583333	-0.583333
G10	-0.143836	-0.143857	-0.051326	-0.051325
G11	-0.010852	-0.011074	-0.010852	-0.010852
Violations	0	0	N/A	N/A
fitness	2896.25927	2897.5314	2996.3481	2996.34816

7. Conclusion and Future Works

Based on the results obtained from the experiments evolving the engineering problems, the CMEPSO-GPU algorithm showed the best results compared to the values obtained by other implementations. Although the focus of this work has been the implementation of the algorithm under CUDA, CMEPSO can be easily ported to other parallel architectures such as Beowulf with MPI. Some key points about the experiments can be seen below:

- The improvement noted in results can be verified and attributed by following factors: 1) The use of boundary conditions with correction factor applied to a random variable with the rate of correction, where the particle that escapes of the limits of search returns to inner search space; 2) The execution of the master swarm in the particles tends to produce optimized values for each best global of the slave swarms; 3) The replicates produced by each particle allowed a better exploration of the search space.
- From a simplified viewpoint, the algorithm CMEPSO can be classified as a hybrid metaheuristic between PSO and EPSO applied under a multi-swarm approach.
- The use of CUDA had a significant contribution to improvements on performance and feasibility of execution for multi-populations algorithms with a small time processing (average of 2.1472 seconds for WBD, 1.9833 seconds for MWTCS and 2.4109 seconds for SRD among the twenty executions of each problem). In a final analysis, the GPGPU platform of massive parallelism opens possibility for testing involving large loads of processing data in a feasible runtime.

As some future works, we can highlight:

- Inclusion of mechanisms for social interaction based on games theory, inspired on the work developed by [Teixeira et al. 2011].
- More research related to other methods involving boundary correction.
- Study and implementation of mechanisms from other metaheuristics (GA, AISO, ACO) to CMEPSO.
- Comparison with other parallel methods and architectures (MPI, OpenMP, pThreads, etc.).

Acknowledgements

This work is supported financially by Research Support Foundation of Pará (FAPESPA) and Federal University of Pará (UFPA).

References

BASTOS FILHO, C. J. A., CARACIOLO, M. P., MIRANDA, P. B. C. AND CARVALHO, D. F., 2008. "Multi Ring PSO". *In: SBRN'2008 (the 10th Brazilian Symposium on Neural Networks),* 111-116.

LOPES, H. S. AND TAKAHASHI, R. H. C., 2011. Computação Evolucionária em Problemas de Engenharia (in portuguese). Ed. OMNIPAX, 1st edition.

MIRANDA, V. AND FONSECA, N., 2006. "Evolutionary Particle Swarm Optimization, a New Algorithm with Applications in Power Systems". *In: Transmission and Distribution Conference and Exhibition 2002: Asia Pacific. IEEE/PES,* 407-416.

VAN DEN BERGH, H. AND ENGELBRECHT, A. P., 2004. "A Cooperative Approach to Particle Swarm Optimization" *IEEE Transactions on Evolutionary Computation*, 225-239.

KIRK, D. B. AND HWU, W. W., 2010. Programming Massively Parallel Processors: A Hands-on Approach. Elsevier & Morgan Kaufman, 1st edition.

SOLOMON, S., THULASIRAMAN, P. AND THULASIRAMAN, R., 2011. "Collaborative Multi-swarm PSO for Task Matching Using Graphics Processing Units". *In: GECCO '11 Proceedings of the 13th annual conference on Genetic and evolutionary computation,* 1563-1570.

MUSSI, L., NASHED, Y. S. G. AND CAGNONI, S., 2011. "GPU-based Asynchronous Particle Swarm Optimization". *In: GECCO '11 Proceedings of the 13th annual conference on Genetic and evolutionary computation*, 1555-1562.

KENNEDY, J. AND EBERHART, R., 1995. "Particle Swarm Optimization". *In: Proc. of the IEEE Int. Conf. on Neural Networks*, 1942-1948.

SHI, Y. AND EBERHART, R., 2000. "Comparing Inertia Weights and Constriction Factors". *In: Proc. of the IEEE Int. Conf. on Neural Networks*, 1942-1948.

EBERHART, R., AND SHI, Y., 1998. "A Modified Particle Swarm Optimizer". *In: IEEE International Conference of Evolutionary Computation*, 69-73.

LEITE, H., BARROS, J. AND MIRANDA, V., 2010. "The Evolutionary Algorithm EPSO to Coordinate Directional Overcurrent Relays". *In: Developments in Power System Protection (DPSP 2010). Managing the Change, 10th IET International Conference*, 1-5.

NVIDIA CORP, 2012. *NVIDIA CUDA-C Programming Guide* Available from: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, [Accessed 17 may 2012].

NIU, B., ZHU, Y. AND HE, X., 2005. "Multi-population Cooperative Particle Swarm Optimization". *In: Proc. European Conference on Artificial Life*, 874-883.

SOUZA, D. L., MONTEIRO, G. D., MARTINS, T. C., TEIXEIRA, O. N. AND DMITRIEV, V. A., 2011. "PSO-GPU: Accelerating Particle Swarm Optimization In CUDA-Based Graphics Processing Units". *In: GECCO 2011*, ACM Digital Library, 837-838.

TEIXEIRA, O. N., LOBATO, W. A. L. L., YANAGUIBASHI, H. S., CAVALCANTE, R. V., SILVA, D. J. A. AND OLIVEIRA, R. C. L., 2011. "Algoritmo Genético com Interação Social na Resolução de Problemas de Otimização Global com Restrições." (in portuguese). *In: Computação Evolucionária em Problemas de Engenharia* (in portuguese), Ed. OMNIPAX, 1st edition, 197-223.

HE, Q. AND WANG, L., 2007. "An Effective Co-evolutionary Particle Swarm Optimization for Constrained Engineering Design Problems". *In: Engineering Applications of Artificial Intelligence*, 89-99.

COELLO, C. AND MONTES, E., 2002. "Constraint-handling in Genetic Algorithms Through the Use of Dominance-based Tournament Selection". *In: Advanced Engineering Informatics*, 193-203.

BRAJEVIC, I., TUBA, M. AND SUBOTIC, M., 2010. "Improved artificial bee colony algorithm for constrained problems". *In: Proceedings of the 11th WSEAS International Conference on Neural Networks, Fuzzy Systems and Evolutionary Computing*, Stevens Point, USA: WSEAS, 185-190.

CAGNINA, L., ESQUIVEL, S. AND COELLO, C., 2008. "Solving Engineering Optimization Problems With the Simple Constrained Particle Swarm Optimizer". *In: Informatica Magazine*, 32(3), 319-326.