libCudaOptimize: an Open Source Library of GPU-based Metaheuristics

Youssef S.G. Nashed, Roberto Ugolotti, Pablo Mesejo, Stefano Cagnoni

libCudaOptimize: an Open Source Library of GPU-based Metaheuristics

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

Roberto Ugolotti
Dept. of Information Engineering
University of Parma, Italy
rob_ugo@unipr.it

Pablo Mesejo
Dept. of Information Engineering
University of Parma, Italy
pmesejo@ce.unipr.it

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

ABSTRACT

Evolutionary Computation techniques and other metaheuristics have been increasingly used in the last years for solving many real-world tasks that can be formulated as optimization problems. Among their numerous strengths, a major one is their natural predisposition to parallelization.

In this paper, we introduce libCudaOptimize, an open source library which implements some metaheuristics for continuous optimization: presently Particle Swarm Optimization, Differential Evolution, Scatter Search, and Solis&Wets local search. This library allows users either to apply these metaheuristics directly to their own fitness function or to extend it by implementing their own parallel optimization techniques. The library is written in CUDA-C to make extensive use of parallelization, as allowed by Graphics Processing Units.

After describing the library, we consider two practical case studies: the optimization of a fitness function for the automatic localization of anatomical brain structures in histological images, and the parallel implementation of Simulated Annealing as a new module, which extends the library while keeping code compatibility with it, so that the new method can be readily available for future use within the library as an alternative optimization technique.

Categories and Subject Descriptors

I.2.5 [**Artificial Intelligence**]: Programming Languages and Software; D.2.13 [**Reusable Software**]: Reusable libraries

General Terms

Algorithms, Design

Keywords

Open Source Library, GPGPU, CUDA, Particle Swarm Optimization, Differential Evolution, Scatter Search, Solis and Wets local search

1. INTRODUCTION

In the last decades, several metaheuristics have been developed (among others, Particle Swarm Optimization (PSO) [6], Differential Evolution (DE) [16], Ant Colony Optimization (ACO) [3], Scatter Search (SS) [5], ...) and applied to a large variety of problems and fields. To facilitate the use of Evolutionary Computation (EC) methods in optimization problems, several software environments or libraries have been developed, like HeuristicLab [20], Matlab Optimization Toolbox [17], CILib [13], jMetal [4] or JCLEC [19]. Recently, these metaheuristics have also been developed on Graphics Processing Units (GPU) [8, 10], fully exploiting their intrinsic parallelism and obtaining significant speedups (up to 30 times) compared to single thread CPU implementations. However, no open-source software has been released to easily take advantage of this aspect.

The main idea behind our work is to offer a user the chance to apply metaheuristics as simply and fast as possible to his own problem of interest, exploiting the parallelization opportunities offered by modern GPUs as much as possible. To the best of our knowledge, there are no software tools in which the entire optimization process, from exploration operators to function evaluation, is completely developed on the GPU, and allows one to develop both local and global optimization methods. Only in the last years, some packages, like ParadisEO [9], have started to use GPUs to speed up their algorithms, however, as it stands, this is limited to parallel data access during fitness evaluation, while the method itself is still executed sequentially. Also, the GPU implementations so far in ParadisEO are strictly limited to local optimization/search methods.

We present libCudaOptimize, a GPU-based open source library that allows users to run their methods in parallel to optimize a fitness function, introduce a new optimization algorithm, or easily modify/extend existing ones. In the first case, the only thing one needs to do is to write the new fitness function in C++ or CUDA-C, while in the second

and third cases, one can take advantage of the framework offered by the library to avoid the need to go deep into basic implementation issues, especially regarding parallel code.

libCudaOptimize is expected to be used by users who have, at least, a basic knowledge of C++. Although no explicit understanding of CUDA-C or even of metaheuristics is required it is very useful anyway, nonetheless, one can use this library just by writing a C++ fitness function and launching one of the optimization techniques already implemented (to date PSO, DE, SS and Solis&Wets local search (SW) [15]). This allows one to:

- implement commonly successful techniques with limited efforts;
- easily compare the results obtained by running different techniques on different functions;
- analyze the effects of changing values of the parameters which regulate the behavior of the optimization techniques on user-defined problems;
- run high-dimensional optimization experiments on consumer level hardware, thanks to the efficient CUDA-C parallel implementation.

The remainder of this work is organized as follows: in section 2 libCudaOptimize is described, then the operations needed to start working with the library are presented in section 3, followed by two case studies in section 4 and by conclusions in section 5.

2. THE PACKAGE

2.1 Implemented Methods

In the present version, the library implements three different global optimization methods (PSO, DE and SS) and one local search technique (SW), which was added to demonstrate that the possibilities of this tool may go beyond the implementation of population-based metaheuristics.

2.1.1 Particle Swarm Optimization

Particle Swarm Optimization [6] is a bio-inspired optimization algorithm based on the simulation of the social behavior of bird flocks. In the last fifteen years PSO has been applied to a very large variety of problems [14] and numerous variants of the algorithm have been presented [1].

During the execution of PSO a set of particles moves within the function domain searching for the optimum of the function (best fitness value). The motion of each particle is driven by the best positions visited so far by the particle itself and by the entire swarm (gbest PSO) or by some pre-defined neighborhood of the particle (lbest PSO). Consequently, each particle relies both on "individual" and on "swarm" intelligence, and its motion can be described by the following two simple equations which regulate the position and the velocity updates:

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

where $P_n(t)$ and $v_n(t)$ are the position and velocity of the n^{th} particle in iteration t; c_1, c_2 and w (inertia factor) are positive constants, $rand()$ returns random values uniformly distributed in $[0, 1]$, BP_n is the best-fitness position visited so far by the particle and BLP_n is the best-fitness position visited so far by any particle of a neighborhood of the particle (which may be as large as the current swarm: in this case, this position would be the global best).

In particular, the PSO version implemented in this library is the same described in [10]: an lbest PSO relying on a ring topology with two neighbors and constant inertia factor.

2.1.2 Differential Evolution

Differential Evolution [16] recently gained credit as one of the most successful evolutionary algorithms. DE perturbs the current population members with the scaled differences of other individuals. Every element of the population acts as a parent vector and, for each of them, a donor vector is created. In the basic version of DE, the donor vector for the i^{th} parent (X_i) is generated by combining three random and distinct elements X_{r1}, X_{r2} and X_{r3}. The donor vector V_i is calculated as:

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

where F (scale factor) is a parameter that strongly influences DE's performances and typically lies in the interval $[0.4, 1]$. After mutation, every parent-donor pair generates an offspring (called the trial vector) by means of a crossover operation. Cr is called crossover rate and appears as one of the control parameters of DE, like F. This trial vector is then evaluated and, if its fitness is better than the parent's, it will eventually replace it.

The library offers the choice between the two most commonly used kinds of crossover (binomial, also called uniform, and exponential). With respect to mutation schemes [2], DE/rand/1 (explained above), DE/target-to-best/1, and DE/best/1 are available.

2.1.3 Scatter Search

Scatter Search [5] is a population-based algorithm in which a systematic combination between solutions (instead of a randomized one, as usually happens in EC) taken from a smaller pool of evolved solutions named the reference set R (usually around ten times lower than typical EC population sizes). R is drawn from a randomly initialized population, and it is composed of:

- a Best Set comprising the $|B_1|$ individuals of the initial population having the best fitness function;
- a Diverse Set with the $|B_2|$ farthest individuals from the Best Set (with $|R| = |B_1| + |B_2|$).

New offspring is generated by combining elements of the two sets. After this step, a local search is performed to improve the offspring and the two sets are eventually updated by including the best and the most diverse elements, respectively. If, after an iteration, no elements are replaced in R, a new population is created and the individuals that are the most distant from the Best Set are used to replace the Diverse Set. These operations are repeated until a stopping criterion is met.

The current implementation of SS in this library uses the BLX-α crossover as combination method, Solis&Wets as an improvement technique, while, for the update, two strategies are allowed: the former preserves only the best solutions, the

latter preserves both the best ones and the most diverse ones with respect to R.

2.1.4 Solis&Wets local search

Solis&Wets local search method [15] is a randomized hill-climber with adaptive step size. Each step starts at a point x. A perturbation p is randomly chosen from a Gaussian distribution with standard deviation ρ. If either $x + p$ or $x - p$ has a better fitness than x, a move to the best point is performed and a success is recorded, otherwise the position does not change and a failure is recorded. After N^+ consecutive successes ρ is increased, for getting faster to the local optima, while after N^- failures in a row, ρ is consequently decreased.

In the library, SW starts with a different ρ for every solution, that equals half the distance to the nearest neighbor. The user can set the parameters N^+ and N^-.

2.2 General-Purpose Computation on Graphics Hardware

General-purpose programming on GPU (GPGPU) is the way of using a graphic card, which typically handles computations only for computer graphics and gaming, to execute applications traditionally managed by the Central Processing Unit (CPU).

CUDA™ (Compute Unified Distributed Architecture) is a GPGPU environment, that includes a parallel computing architecture and programming model, developed by nVIDIA™. This programming model requires the problem under consideration be partitioned into sub-problems, that are solved independently in parallel by blocks of threads. In turn, each sub-problem is also partitioned into finer pieces, that can be solved cooperatively in parallel by all threads within the same block. Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks, as illustrated in Figure 1.

In particular, nVIDIA™ CUDA-C [11] is an extension of the C language for the development of GPUs routines (called kernels), which, when called, are executed N times in parallel by N different CUDA™ threads. Kernels are run on the device (GPU) while the rest of the code runs on the host (CPU). It is also important to notice that in CUDA™, host and devices have separate memory spaces and, in order to execute a kernel, the programmer needs to explicitly allocate memory on the device and, if needed, transfer data from and back to the host. This is the main bottleneck which is encountered: to optimize code for speed the programmer should reduce as much as possible the amount of these transfers.

Regarding the programming model, and in order to understand the implementation details and the case studies under consideration, there are four variables particularly useful for users who want to deeply dig into the library:

- `gridDim`, that stores the dimensions of the grid as specified during kernel invocation;
- `blockIdx`, that refers to the block index within the grid;
- `blockDim`, that contains the dimensions of the block, and
- `threadIdx`, that stores the thread index within the block.

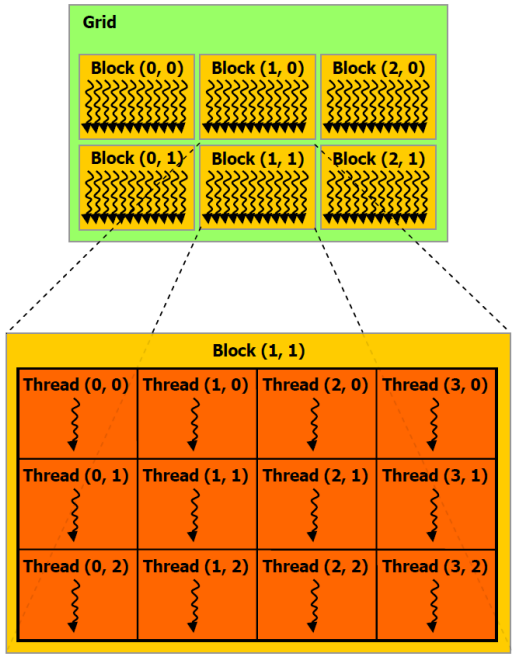

Figure 1: Grid of Thread Blocks [11].

Each of these variables is a `dim3`, a data type of CUDA-C that represents a three-dimensional vector, whose elements can be accessed as the elements `x`, `y` and `z`.

2.3 Implementation details

libCudaOptimize is entirely written in C++ and CUDA-C and relies on two classes: `IOptimizer` and `SolutionSet` (see Figure 2). The former is an abstract class that includes all methods used for evolving a set of solutions (or population/swarm, where every particular solution is an individual/particle, depending on the used terminology), for setting evolution parameters and a reference to the set (it can evolve more than one set in parallel), represented by an instance of the class `SolutionSet`. Every different metaheuristic is implemented as a sub-class of `IOptimizer`. All these classes (see some examples at the bottom of Figure 2) have methods that allow a user to set the parameters of the metaheuristic. Moreover, most of the relevant parameters can be passed to the optimizer at the moment of its instantiation.

The class `SolutionSet` represents one or more sets of solutions and can be accessed in the user-defined fitness function, where it is used to access the elements of the population and to update their fitnesses after evaluation. There are methods that allow users to access the solutions, and their corresponding fitnesses, both on the device and the host. In this way, the user can employ these information both on C++ and CUDA-C function easily.

3. RUNNING THE LIBRARY

In this section we will briefly describe the operations needed to install the library and to start working with it.

3.1 How to install it

libCudaOptimize has been tested on Windows 7 and Ubuntu Linux, using graphics cards with compute capability (CC)

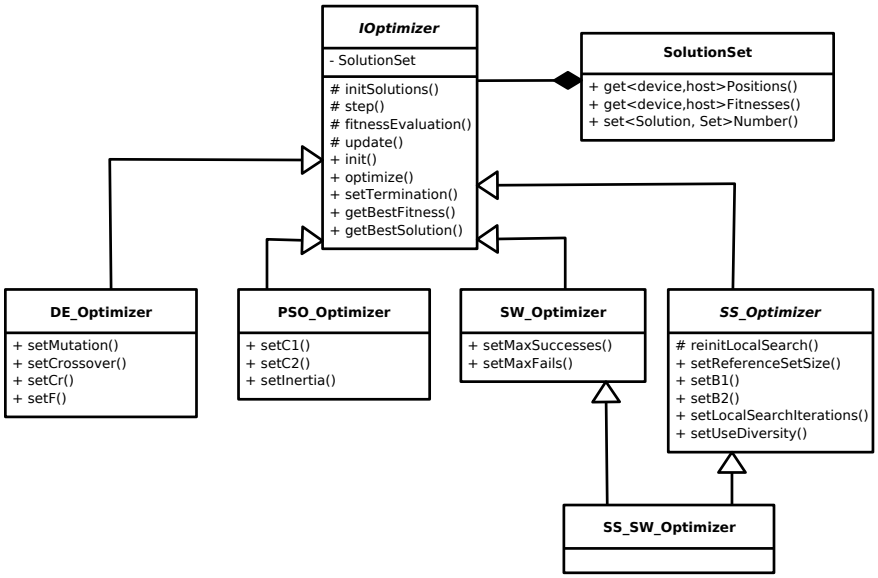

Figure 2: UML diagram. For every class, the most important methods are shown.

ranging from 1.3 to 2.1. GPUs with lower CC impose some limitations to the use of the library due to hardware limits: e.g. using a GPU with CC 2.0 or higher, the limit on the dimensions of a solution is 1024, while with a GPU with CC 1.3 this limit is lowered to 512.

In order to install libCudaOptimize, the following steps should be followed:

1. Download from the project webpage on Sourceforge (http://sourceforge.net/p/libcudaoptimize/) the packages: libCudaOptimize, the proper library, and testLibCudaOptimize, a small program which tests if the installation has been correctly performed. Both source code and binary installer are provided.

2. Build the library from source code: for doing so, the easiest way is to use CMake[1]. To build testLibCudaOptimize one has to link against the previously installed libCudaOptimize static library and the CUDA utilities library.

3.2 How to use the library

Basically, there are two ways to use this library. The first and most direct one is just to apply the included heuristics to optimize a user-defined fitness function. All one needs to do, in this simplest case, is to write a function in C++ or, to fully exploit the parallelization potentiality of the package, in CUDA-C. Then, one must select the heuristic, set its parameters, run it, and retrieve the solution(s) found. An example of this usage is shown in section 4.1.

The second purpose of the library is to allow the user to design and implement an optimization technique, taking advantage of the structure of the algorithms implemented in libCudaOptimize. Since several EC methods share a similar structure, one can extend the superclass `IOptimizer` or one of its children in order to create a new optimizer. To do so, a mandatory step is to implement the four protected functions of `IOptimizer` shown in Figure 2:

- `initSolutions` initializes the candidate solutions within the search space, e.g. random initialization;
- `step` defines how the optimizer generates new potential solutions from the current population;
- `fitnessEvaluation` calls the user's fitness function;
- `update` is called after fitness evaluation and should update the population according to the results obtained: replace current individuals, update personal best locations, check constraints, ...

An example of this usage can be found in section 4.2.

It is important to notice that the user does not have to handle memory allocations and releases nor grid and kernels configuration, since these operations are taken care for by the library core.

4. CASE STUDIES

4.1 Optimization of a fitness function

The first case study examined is a method for automatic localization of the hippocampus in mouse brain histological images [18]. The localization uses two parametric models that are moved and deformed towards the image, in order to maximize a fitness function which is proportional to their overlap with two major structures of the hippocampus (see Figure 3) using PSO to generate candidate solutions. In the first implementation of this method, in which PSO was coded in Matlab and the fitness function in C++, these two parts were localized sequentially one after the other, but now this process can be run independently in parallel by two swarms.

The fitness function takes as inputs: (i) the image that contains the hippocampus, (ii) some vectors that represent the limits of the deformations, and (iii) another vector that represents the coordinates of the model. The values of this vector (usually having between 14 and 16 elements) are optimized by the methods provided by libCudaOptimize.

[1] www.cmake.org

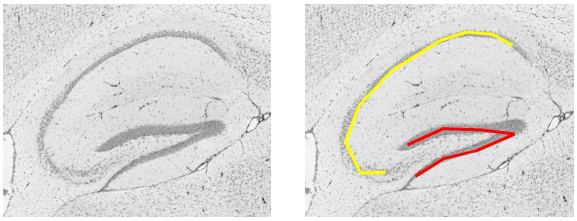

Figure 3: Example of a hippocampus and the result of the localization phase.

Thanks to the library features, we can evolve two swarms in parallel, each of which will be responsible of one model. The first step that was performed was the conversion of the fitness function from C++ to CUDA-C, which is quite straightforward and will not be explained here. It is worth noting that one needs to supply a C++ function pointer to the optimizer in order to call the CUDA-C kernel (fitness function). This C++ fitness function receives as input a pointer to the `SolutionSet` object, from which it can access the coordinates and the fitnesses of the swarm(s), which will be passed to CUDA-C kernel, along with any other data that is needed:

```
void localizeHippocampus(SolutionSet* s)
{
  localizeHippocampus_kernel<<<...,...>>>
    (s->getDevicePositions(),
     s->getDeviceFitnesses(),
     ...);
}
```

In the main function, all one has to do is choose the optimization technique, e.g. PSO, and create an instance of the correct extension of the class `IOptimizer`, setting, if necessary, some parameters (during the instantiation, or later using the functions listed in Figure 2). In this case, a pointer to the fitness function and the number of swarms (2), of solutions (64), and the dimension of the problem (16).

One can also set other parameters for the optimization, like the termination criteria (that can be total execution time, number of fitness evaluations, number of generations/iterations, or required fitness value). After all the parameters have been set, one has just to start the optimization process and retrieve the results:

```
void main() {
  PSO_Optimizer p(&localizeHippocampus,
    16, 2, 64);
  p.optimize();
  float* myResults = p.getBestSolution();
}
```

As one can see, a C++ (or CUDA-C) fitness function can be bound to libCudaOptimize by adding less than 10 lines of code.

Figure 4 and 5 show the execution time we obtained using libCudaOptimize changing the number of PSO generations (from 100 to 10000) and the number of particles of the swarms (from 16 to 512), respectively. Three different implementations have been compared:

1. both fitness function and PSO implemented in C++;
2. fitness function implemented in C++ using our parallel PSO;
3. both fitness function and PSO implemented in CUDA-C;

Tests were run on a 64-bit Intel(R) Core^{TM} i7 CPU running at 2.67GHz using CUDA v. 4.1 on a nVIDIA^{TM} GeForce GTS450 graphics card with 1GB of DDR memory and compute capability 2.1. For the first test, whose results are plotted in Figure 4, the optimization was run 10 times for each configuration, for a total of 1000×3 runs. As for the second, depicted in Figure 5, a total of 1250×3 experiments was performed.

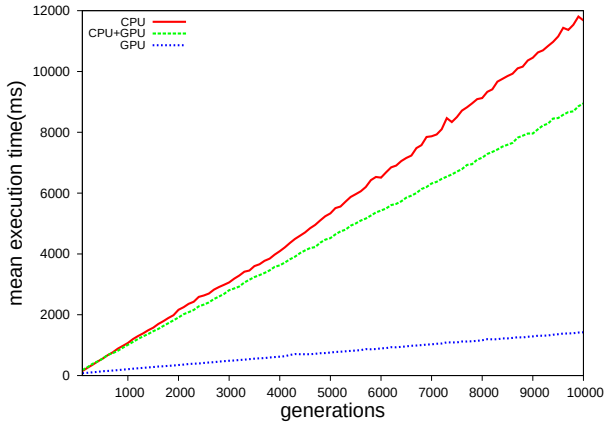

Figure 4: Execution time versus PSO generations, swarm size fixed at 64 particles.

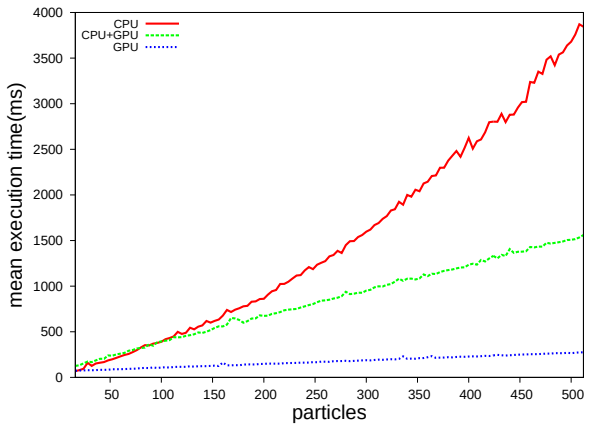

Figure 5: Execution time versus number of particles, termination criterion set as 200 generations.

The plots clearly show the GPU version outperforming the two other implementations. In Figure 5, the CPU version appears to have better performance than the CPU+GPU one with smaller swarms (less than 60 particles). This is due to the overhead introduced by the data transfer between host and device memory. However, when dealing with more complex configurations this overhead is overshadowed by the advantages of the parallel PSO execution.

4.2 Implementation of a new optimization method

As a second case study, we extended the library by implementing a metaheuristic which was not included in the basic set of optimizers provided by the library: Simulated Annealing (SA) [7]. This technique tries to mimic the physical annealing process, where a material is heated and slowly cooled into a uniform structure. SA may even perform bad moves (i.e. changes which leads to worse fitness) accordingly to a probability distribution dependent on the temperature of the system: a move is selected at random and, then, as the temperature decreases, the probability of accepting a bad move decreases (when temperature is zero, no bad moves are accepted, i.e. it behaves like hill climbing). The main building blocks of the algorithm are three: the candidate solution generation method (sometimes referred to as the neighborhood function), the acceptance probability function mentioned earlier, and the cooling schedule.

To make SA conform with the notions of a population-based technique, we adapted the method for considering more than one candidate solution at the same time. For the sake of simplicity, the neighborhood function simply generates random solutions from the current solution positions with a standard deviation of 0.1. The acceptance probability function is defined as follows:

$$P(x, x\prime, T) = \exp(\frac{f(x) - f(x\prime)}{T})$$

where P is the probability function, x the current solution, $x\prime$ the candidate solution, T the temperature of the system, and $f(x)$ the fitness of a solution. As for the cooling schedule, the temperature value is reduced by a factor of 0.98 after every iteration of the algorithm. Moving on to implementation details, code snippets are provided to explain the process. Firstly, we need to extend the `IOptimizer` implementing the required methods:

```
#include "IOptimizer.h"

class SA_Optimizer :
  public virtual IOptimizer
{
protected:
  SolutionSet m_neighbors;
  float m_temperature;

public:
  SA_Optimizer();
  virtual ~SA_Optimizer();

  virtual bool init();

  virtual void initSolutions();
  virtual void step();
  virtual void fitnessEvaluation();
  virtual void update();
};
```

The `SA_Optimizer` class has only two more member variables: a `SolutionSet` instance representing the neighbors of the current state in an iteration, and a float variable as the temperature of the system. The method `initSolutions()` is implemented similarly to the initialization functions of `DE_Optimizer` and `PSO_Optimizer` classes, in which the positions of the population of solutions are set randomly over the search space. The only functions that had to be implemented from scratch were `step` and `update`, by specifying a neighborhood generating procedure and the acceptance probability function respectively. Each of these functions calls a CUDA-C kernel to execute the appropriate method in parallel, specifying the thread block configuration required. Every thread block performs the calculations for a solution in a set, while every thread in the block represents a dimension of the problem under optimization. Code snippets for the host and device functions for the `step` function, used for creating new candidate solutions that will be evaluated by `fitnessEvaluation`, are presented below:

```
void SA_Optimizer:: step()
{
  h_cudaSA_step(m_solutionSet.getDevicePositions(),
    m_neighbors.getDevicePositions(),... );
  m_temperature*=0.98f;
}

__global__ void sa_step(float* positions,
  float* neighborPositions,...)
{
  int tid=threadIdx.x;
  int solutionID=(blockIdx.x*gridDim.y)+blockIdx.y;
  int posID=(solutionID*blockDim.x)+tid;

  curandState localState=devStates[posID];

  if(tid<problemDimension)
  {
    float y=positions[posID]+
      (-0.1f+(curand_uniform(&localState)*0.2f));
    cropPosition(y);
    neighborPositions[posID]=y;
  }

  devStates[posID]=localState;
}
```

`h_cudaSA_step` is a host function whose only purpose is to call the kernel `sa_step`, which can not be called directly from a C++ class. This kernel is executed simultaneously by as many threads as the problem dimension. In practice, the number of threads for a given kernel should be a power of half the GPU warp size to achieve the coalesced access to global memory recommended by nVIDIA^{TM} [12]. The code above shows the most common and basic implementation of a neighborhood function, used by many derivative-free optimization methods. These algorithms usually test candidate positions sampled from a fixed or adaptive neighborhood around the current position. In our case, we generate candidate solutions by uniform sampling from a hypercube with side length of 0.2 and centered at the current solution. The CuRand library supplied with the CUDA SDK is used to generate random numbers, where, for every dimension of each solution a random number generator (`curandState`) is stored in a global memory array (`devStates`). First, the random number generator is loaded into a thread local variable for efficiency. Then, a new position `y` is computed for every dimension by adding a uniform random number in the range $[-0.1, 0.1]$ to the current position. Finally, the neigh-

bor position along with the random number generator are stored back in global memory at the end of the kernel.

It is worth noting the way the global memory is being accessed in the device code. `blockIdx.x` represents the set index, `blockIdx.y` represents the solution index within the set, and `threadIdx.x` is the dimension index in the search space. This SA code should be used as a guideline for the advanced user to extend the library with new optimization techniques.

5. CONCLUSIONS

We introduced libCudaOptimize, a free, fast and portable open source library for continuous optimization based on CUDA. Two practical examples of using and extending the library have been shown: the optimization of a fitness function which takes advantage of this framework, and the implementation of a new optimization technique that could be useful for solving more problems.

Regarding future work, several aspects can be improved or extended. Our next efforts will mainly be concerned about: the realization of some visualization and statistical tools in order to help behavioral analysis of EC techniques; more support for multiple solution sets, like allowing different sets to have independent termination criteria; the possibility to evolve solutions of data types other than floats; and the parallel implementation of other well-known optimization methods like Genetic Algorithms, Evolution Strategies or Evolutionary Programming as well as further expansions of the methods already present.

Acknowledgments

Youssef S. G. Nashed and Pablo Mesejo are funded by the European Commission (Marie Curie ITN MIBISOC, FP7 PEOPLE-ITN-2008, GA n. 238819). Roberto Ugolotti is funded by Compagnia di San Paolo (Programma Neuroscienze) and Fondazione Cariparma.

6. REFERENCES

[1] A. Banks, J. Vincent, and C. Anyakoha. A review of Particle Swarm Optimization. Part I: background and development. *Natural Computing*, 6:467–484, 2007.

[2] S. Das and P. Suganthan. Differential Evolution: A Survey of the State-of-the-Art. *IEEE Transactions on Evolutionary Computation*, 15(1):4–31, 2011.

[3] M. Dorigo and T. Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.

[4] J. J. Durillo, A. J. Nebro, F. Luna, B. Dorronsoro, and E. Alba. jMetal: A Java Framework for Developing Multi-Objective Optimization Metaheuristics. Technical Report ITI-2006-10, Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, December 2006.

[5] F. Glover, M. Laguna, and M. Rafael. Scatter search. In *Advances in Evolutionary Computation: Theory and Applications.*, pages 519–537. Springer-Verlag, 2003.

[6] J. Kennedy and R. Eberhart. Particle Swarm Optimization. In *Proc. of IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995.

[7] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

[8] P. Krömer, V. Snášel, J. Platoš, and A. Abraham. A comparison of many-threaded differential evolution and genetic algorithms on cuda. In *Third World Congress on Nature and Biologically Inspired Computing (NaBIC), 2011*, pages 509 –514, 2011.

[9] N. Melab, T. Luong, K. Boufaras, and E. Talbi. Towards paradiseo-mo-gpu: A framework for gpu-based local search metaheuristics. In J. Cabestany, I. Rojas, and G. Joya, editors, *Advances in Computational Intelligence*, volume 6691 of *Lecture Notes in Computer Science*, pages 401–408. 2011.

[10] L. Mussi, F. Daolio, and S. Cagnoni. Evaluation of parallel particle swarm optimization algorithms within the CUDA™ architecture. *Information Sciences*, 181(20):4642–4657, 2011.

[11] nVIDIA Corporation. *nVIDIA CUDA programming guide v. 4.1*, 2011.

[12] nVIDIA Corporation. *nVIDIA CUDA C programming - Best practices guide v. 4.1*, 2012.

[13] G. Pampara, A. Engelbrecht, and T. Cloete. CIlib: A collaborative framework for computational intelligence algorithms - part I. In *IEEE International Joint Conference on Neural Networks. IJCNN 2008.*, pages 1750 –1757, 2008.

[14] R. Poli. Analysis of the publications on the applications of Particle Swarm Optimisation. *J. Artificial Evolution and Applications*, pages 1–10, 2008.

[15] F. J. Solis and R. J. B. Wets. Minimization by Random Search Techniques. *Mathematics of Operations Research*, 6(1):19–30, 1981.

[16] R. Storn and K. Price. Differential Evolution- A Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces. Technical report, International Computer Science Institute, 1995.

[17] The Mathworks. Matlab Optimization Toolbox User's Guide.

[18] R. Ugolotti, P. Mesejo, S. Cagnoni, M. Giacobini, and F. Di Cunto. Automatic Hippocampus Localization in Histological Images using PSO-Based Deformable Models. In *Proc. Genetic and Evolutionary Computation Conference, GECCO '11*, 2011.

[19] S. Ventura, C. Romero, A. Zafra, J. Delgado, and C. Hervás-Martínez. JCLEC: A Java Framework for Evolutionary Computing. *Soft Computing*, 12(4):381–392, 2008.

[20] S. Wagner. *Heuristic Optimization Software Systems - Modeling of Heuristic Optimization Algorithms in the HeuristicLab Software Environment*. PhD thesis, Institute for Formal Models and Verification, Johannes Kepler University Linz, Austria, 2009.