Solving very large optimization problems (up to one billion variables) with a parallel evolutionary algorithm in CPU and GPU

Santiago Iturriaga, Sergio Nesmachnow
Centro de Cálculo, Facultad de Ingeniería
Universidad de la República, Montevideo, Uruguay
Email: {siturria,sergion}@fing.edu.uy

***Abstract*—This article presents the application of a parallel evolutionary algorithm implemented in both CPU and Graphic Processing Units (GPU), to solve large instances of the noisy OneMax problem with up to one billion variables. Actually, new GPU platforms provide the computing power needed to apply massively parallel strategies to solve large problems. We report here the experimental evaluation of both CPU and GPU implementations for a compact evolutionary algorithm. The proposed method demonstrates a high problem solving efficacy and shows a good scalability behavior when facing high dimension instances of the noisy OneMax problem, improving the computational efficiency and the results reported in previous similar approaches developed on CPU.**

***Keywords*-parallel evolutionary algorithms; GPU; noisy OneMax; one billion variables;**

I. Introduction

Evolutionary algorithms (EAs) are stochastic methods successfully used in the last twenty-five years to solve hard optimization, search, and machine learning problems, achieving a high level of problem solving efficacy in many application areas [1]. As long as the available processing power in modern computers has become larger, EAs have been applied to solve optimization problems with increasing difficulty and complexity.

In order to improve the efficiency of EAs, parallel implementations have been used to significantly enhance and speed up the search, allowing to reach high quality results in reasonable execution times even for hard-to-solve optimization problems [2].

The capability of scaling up to solve increasingly complex problem instances is a very desirable feature for modern optimization techniques, and EAs in particular. However, few articles has studied the applicability of EAs to solve very large—i.e. with a million or more variables—optimization problems (see a review of the related work in Section III). Nowadays, the new computing platforms—multicore CPUs and GPUs—provide an accessible infrastructure for applying massive parallelism with the goal of designing efficient EAs to solve large and complex problems.

Thus, there is still room to contribute in this line of research by studying highly efficient parallel EA implementations, able to deal with very large optimization problems by using the computational power of parallel systems.

In this line of work, the main contribution of this article are: i) to present a performance analysis for an efficient EA when solving very large optimization problems, ii) to introduce a new parallel EA implementation in order to take advantage of the massive computing power available in GPUs, and iii) to perform an experimental analysis demonstrating the capabilities the parallel EA implementations in CPU and GPU for solving very large optimization problems.

The reported research is a first step in the quest to devise specific guides for designing EAs to solve hard problems quicky, reliably, and accurately, by using cutting-edge hardware infrastructure available today in every research laboratory, and also in personal computers.

The manuscript is structured as follows. Section II presents the problem formulation. Section III introduces evolutionary computation and the compact EA applied in this work. After describing the computing infrastructures in Section IV, Section V describes the implementation details of the compact EA in CPU and GPU. The discussion of the experimental analysis and results are presented in Section VI, while the conclusions and possible lines for future work are formulated in Section VII.

II. The OneMax and the noisy OneMax problems

This section describes the OneMax and the noisy OneMax problems used to evaluate the EA proposed in this article.

A. OneMax problem

The OneMax Problem or *bitcounting problem* [3] is a simple optimization problem consisting in maximizing the number of ones of a bitstring. Formally, given a set of binary variables $\vec{x} = [x_1, x_2, \ldots, x_D]$, $x_i \in \{0, 1\}$, the OneMax problem is defined by the expression in Equation 1.

$$\max f(\vec{x}) = \sum_{i=1}^{D} x_i \tag{1}$$

The OneMax problem has a very simple formulation, but it is useful to perform empirical analysis to evaluate the computational efficiency of a given EA implementation. The linear-order OneMax fitness function models the computational complexity of the evaluation function used in

many traditional combinatorial optimization problems [5], including:

- *routing and path planning problems*, including Hamiltonian cycle and Traveling Salesman Problem (TSP), shortest path and spanning tree problems.
- *graph and set problems*, including covering and partitioning, coloring, matching, clique, etc.
- *scheduling problems*, including job sequencing, multiprocessor scheduling, open-shop and flow-shop.
- *propositional logic problems*, including the Satisfiability Problem (SAT) in all its flavors.

B. Noisy OneMax problem

In a noisy optimization problem [4], the function used to evaluate solutions is affected by exogenous noise. In practice, many real-world optimization problems concern objective functions which are perturbed by noise.

A noisy fitness function within an EA is modeled as $f_{NOISY} = f + n$, where f is the true fitness of a given solution and n is the external noise, a random variable usually defined according to a zero-mean Gaussian distribution.

The fitness function for the noisy OneMax problem is defined by Eq. 2, where $N(0, \sigma_N^2)$ is a Gaussian random variable with mean 0 and variance σ_N^2.

$$\max f(\vec{x}) = \sum_{i=1}^{D} x_i + N(0, \sigma_N^2) \tag{2}$$

By including exogenous noise in the fitness evaluation, the OneMax problem can be used to model complex optimization problems involving constraints and other features [5].

III. Evolutionary computation

This section introduces EAs and the compact genetic algorithm.

A. Evolutionary algorithms

EAs are non-deterministic methods that emulate the evolution of species in nature, which have been successfully applied for solving optimization problems underlying many complex real-life applications in the last twenty years [1].

An EA is an iterative technique that applies stochastic operators on a population of *individuals*, which encode tentative solutions of the problem, in order to improve their *fitness*. An evaluation function associates a fitness value to every individual, indicating its suitability to the problem. The initial population is generated at random or by using a specific heuristic for the problem. Iteratively, the probabilistic application of *recombinations* of individuals or random changes (*mutations*) in their contents, using a selection-of-the-best technique, guides the EA to better solutions.

The stopping criterion usually involves a fixed number of generations or execution time, a quality threshold on the best fitness value, or the detection of a stagnation situation. Specific policies are used for the *selection* of individuals to recombine and to determine which new individuals *replace* the older ones in each new generation. The EA returns the best solution found, regarding the fitness function values.

B. The compact genetic algorithm

The compact genetic algorithm (cGA, see Algorithm 1) [6] is an EA that represents the population as a probability distribution over the set of solutions, in order to allow an efficient use of the available memory. From the algorithmic point of view, the cGA is similar to a simple GA that uses the uniform crossover and the flip bit mutation operators. Each element in the probability vector represents the proportion of a given value (i.e. ones) in each gene position. Since the population is not explicitly stored and each gene is independently processed, a considerable amount of memory is saved when compared with a traditional GA.

Algorithm 1 Schema of the compact genetic algorithm.

1: **initialize probabilistic model**(p^0)
2: t $\leftarrow$ 0 {generation counter}
3: **while** not stop criteria **do**
4: $[x_1, x_2]$ = **generate** (p^t) {model sampling}
5: **evaluate** (x_1, x_2)
6: **selection** (x_1, x_2)
7: p^{t+1} = **probabilistic model update** (p^t)
8: **end while**
9: **return** best solution ever found

The phases in cGA are: i) *initialization*: the entries in the probability vector are initially set to 0.5; ii) *model sampling*: generate two candidate solutions by sampling the probability vector; iii) *evaluation*: the fitness of the selected individuals are computed; iv) *selection*: the tournament selection operator is applied in order to assure that the best individual propagates its characteristics; and v) *probabilistic model update*: After selection, the proportion of winning alleles is increase by $1/n$, according to the expression in Eq. 3.

$$p_i^{t+1} = \begin{cases} p_i^t + 1/n & \text{if } x_i^W \neq x_i^L \text{ and } x_i^W = 1, \\ p_i^t - 1/n & \text{if } x_i^W \neq x_i^L \text{ and } x_i^W = 0, \\ p_i^t & \text{otherwise.} \end{cases} \tag{3}$$

In Eq. 3, x_i^W and x_i^L are the i-th gene of the winner and loser individuals in the tournament, respectively, and p_i^t is the i-th element of the probability vector at generation t.

C. Parallel evolutionary algorithms

Parallel implementations became popular in the last decade as an effort to improve the efficiency of EAs. By splitting the population into several processing elements, parallel evolutionary algorithms (PEAs) allow reaching high quality results in a reasonable execution time even for hard-to-solve optimization problems [2].

The PEAs proposed in this work are categorized within the *master-slave* model [7]: a master process guides the evolution while a group of slave processes perform the evolutionary search by executing the computationally expensive functions in parallel.

D. Related work

Deb et al. [8] presented the first attempt to solve an optimization problem with more than a million variables using EAs. They showed that a traditional GA is not useful to solve very large problems, since it requires an excessive execution time. The EA by Deb and Pal [9] computed near-optimum solutions for a problem with 100,000 variables in a few minutes, but took more than 10 hours when facing an instance with a million variables. Semet and Schoenauer [10] solved with an EA a large real-world instance of the train timetabling problem, with more than a million variables and two million constraints. The EA took 15 minutes to perform 20 generations before it prematurely converged, so it was hybridized with CPLEX in order to produce better results, but in that case four hours were required. The applicability of the previous approaches to solve generic optimization problems is limited since they use problem-specific operators and small population sizes, causing premature convergence.

Regarding more generic proposals, Kunasol et. al. [11] solved one-million bit problems by applying a GA and a search space reduction approach. The experimental evaluation solved the OneMax, the Royal Road and the Trap problems, but since a space reduction was applied, the EA does not really faced one-million-variable problems.

In 2007, Goldberg et al. [12] faced the OneMax problem by using a parallel implementation of cGA that "could solve very large scale problems with up to 32 million variables to full convergence". Several tweaks were applied to increase the efficiency, but the cGA still required a large parallel computing infrastructure and demanded hours to perform a single execution, mainly due to the very large population modeled—over one million individuals, sized according to a gambler ruin model [13]— When facing the one-billion problem, even using 256 processors in a large cluster, the authors needed to relax the stopping criterion to a very weak "relaxed convergence" (i.e. the probability for each variable is 0.501), which in reality means that cGA has an infinitesimal probability of sampling the optimal solution.

Suwannik and Chongstitvatana [14] applied an Estimation of Distribution Algorithm (EDA) with a compressed arithmetic coding to solve the one-billion Noisy OneMax problem. A population size of 3200 individuals was used, significantly smaller than the one used by Goldberg et al. [5]. The proposed EDA required about 16 hours to perform only 100 generations when solving the OneMax and almost 34 hours when solving the Noisy OneMax problem.

The related work indicates that few articles have tackled very large optimization problems using EAs, and that the proposed implementations are not computationally efficient. In this article, we adopt the cGA by Goldberg et al. to analyze how the new multithreading computing platforms (multicore CPUs and GPUs) allow extending the capabilities to solve very large optimization problems efficiently.

IV. New multicore platforms

This section briefly introduces the new multicore computing platforms using in this work.

A. Multicore CPUs

In a multicore processor, several independent computing units (i.e. cores) are available to execute program instructions. Executing in multiple cores at the same time by applying parallel computing techniques allows increasing the overall computational efficiency, specially for speeding high-demanding CPU applications. Nowadays, multi-core processors are widely used to solve complex problems in many application domains. Modern technologies such as Opteron Magny-Cours—used in the experimental analysis reported in this article— allow integrating up to 24 cores in the same processor/machine.

B. GPU computing

GPUs were originally designed to perform the graphic processing in computers, allowing the CPU to concentrate in the remaining computations. Nowadays, GPUs have a considerably large computing power, provided by hundreds of processing units with reasonable fast clock frequencies. In the last ten years, GPUs have been used as a powerful parallel hardware architecture to achieve efficiency in the execution of applications.

The CUDA [15] software architecture allows managing GPUs as a parallel computing device, able to execute a large number of threads in parallel. A specific procedure to be executed many times over different data can be isolated in a GPU-function using many execution threads. The function is compiled using a specific set of instructions and the resulting program (named *kernel*) is loaded in the GPU. The GPU has its own DRAM, and the data are copied from the DRAM of the GPU to the RAM of the host (and viceversa) using optimized calls to the CUDA API.

The CUDA architecture is built around a scalable array of multiprocessors, each one of them having eight scalar processors, one multithreading unit, and a shared memory chip. The multiprocessors are able to create, manage, and execute parallel threads, with reduced overhead. The threads are grouped in *blocks* (with up to 512 threads), which are executed in a single multiprocessor, and the blocks are grouped in *grids*. When a CUDA program calls a grid to be executed in the GPU, each one of the blocks in the grid is numbered and distributed to an available multiprocessor. When a multiprocessor receives a block to execute, it splits the threads in *warps*, a set of 32 consecutive threads. Each warp executes a single instruction at a time, so the best efficiency is achieved when the 32 threads in the warp executes the same instruction. Each time that a block finishes its execution, a new block is assigned to the available multiprocessor.

V. The parallel cGA in CPU and GPU

This section presents the implementation details of the parallel compact GA on CPU and GPU.

A. Performance analysis of cGA for the OneMax problem

The main goal of the experimental analysis is to study the ability to efficiently solve very large problems using cGA. So, initially, we carried out a performance analysis to know which operators contribute the most to the total execution time. The analysis was done using the standard `gprof` tool for C/C++ profiling in CPU, and using the CUDA Event API [16] in GPU. As an example, Figure 1 summarizes the contribution of the main functions in cGA when solving a one-million variables OneMax in CPU and GPU.

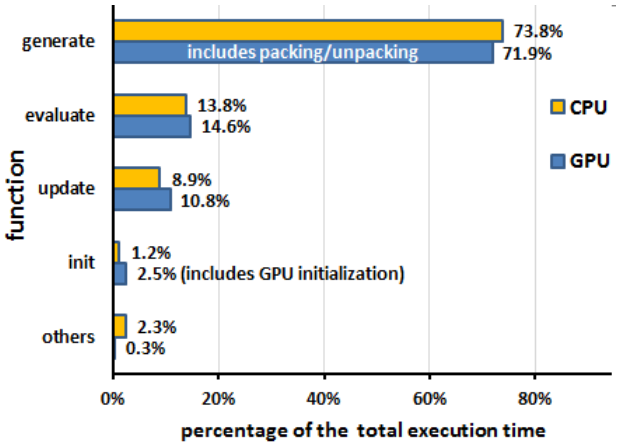

Figure 1. Performance analysis of cGA for the OneMax problem with one million variables in CPU and GPU.

The results in Figure 1 demonstrate that the random number generation is the function that contributes the most to the total execution time, by a significant margin.

cGA needs $2 \times n$ random numbers to generate the sampled individuals in each generation, thus for a representative run (about 50,000 generations) and n=1,000,000 variables, 100 billion calls to the random number generator are required.

B. Implementation details

In order to reduce the execution time required to perform the simulations, we have followed the suggestions for implementing efficient EAs given in [17]. In particular, we adopted the following decisions:

- since the generation of random numbers plays a major role in the computational requirements of cGA, we decided to use the Mersenne Twister generator [18], which provides both a very large period and significant efficiency improvements with respect to traditional random numbers generators.
- we avoided using dynamic memory to store the large variables in cGA (the probability vector and the two sampled individuals);
- we decided not to use the `char` type to represent binary variables;
- we reduced the number of function calls;

Several other performance-oriented modifications were applied in the Mersenne Twister implementation in order to avoid high-cost operations (e.g. floating point divisions).

C. Parallel model

Figure 2 presents a diagram for the parallel model used in the cGA implementation in both CPU and GPU. The parallel model follows a domain-decomposition strategy, splitting the problem domain (i.e. variables) in equally-sized chunks to be handled in parallel. We propose a multithreading implementation for cGA that takes advantage of the computing resources available to execute parallel threads in CPU and GPU to perform the costliest operations in cGA: i) the *evolution step*, which executes the probability model sample, the generation of candidate individuals x_1 and x_2, and the partial fitness evaluation for each chunk handled by each thread; and ii) the *probability vector update*, which computes the values of the probability vector p according to the winner individual in the tournament selection. The internal synchronization is needed in order to globally compute the fitness of complete solutions from partial solutions.

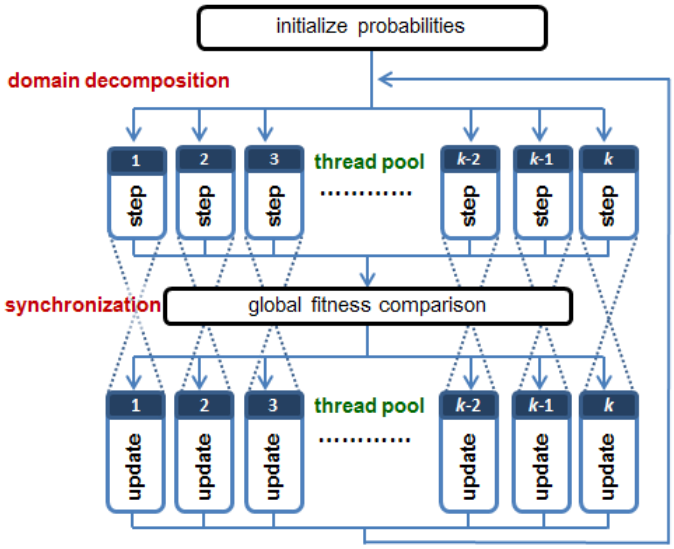

Figure 2. Schema for the parallel cGA in CPU and GPU.

D. CPU implementation

A traditional master-slave model for EAs is applied, where a master process is in charge of creating, managing, and synchronizing a pool of threads, which perform the model sampling and generate the sampled individuals. After the synchronization to globally compare the fitness of individuals, each thread in the pool performs the update of the probability vector. The thread support is implemented with the standard POSIX thread library (pthread) in C.

The CPU implementation does not apply a method for packing/unpacking individuals, since the available memory is enough to store the probability vector and the two sampled individuals for dimensions up to one billion.

E. GPU implementation

The GPU implementation of cGA is able to take advantage of multiple GPU devices installed on the host machine. So, the domain decomposition is applied at the GPU device level, splitting the domain in equally-sized sub-domains one for each GPU device. Each GPU device is managed by a CPU thread which executes the cGA skeleton and guides the GPU device code execution.

The first step of the algorithm is to initialize the GPU devices. This initialization takes some seconds, but it is performed only once during the initialization of the whole algorithm. After the devices are initialized, one CPU thread per GPU device is launched. Algorithm 2 presents the pseudocode of each CPU thread of the cGA for GPU.

Algorithm 2 Schema of each cGA thread in GPU.

1: **initialize probabilistic** $submodel_i(p^0)$
2: t $\leftarrow$ 0 {generation counter}
3: **while** not stop criteria **do**
4: $[x_1, x_2]$ = **generate** (p^t)
5: **evaluate thread** $submodel_i(x_1, x_2)$
6: synchronize()
7: **evaluate** $\sum_{k=0}^{n} submodel_k$ {the complete model}
8: synchronize()
9: **selection** (x_1, x_2)
10: p^{t+1} = **probabilistic model update** (p^t)
11: **end while**
12: **return** best solution ever found

The OneMax is a separable problem, so we also implemented an asynchronous version of cGA in GPU (cGA GPU-async), where the probabilistic model update is performed within each thread. This model executes significantly faster than the synchronous cGA, but it is only useful to solve separable optimization problems.

VI. Experimental analysis

This section describes the experimental analysis when solving the OneMax and Noisy OneMax (noise = 0.5%) with cGA in CPU and GPU.

A. Development and execution platform

The cGA for CPU was implemented in C using the standard pthread library, and evaluated in a Opteron 6172 Magny-Cours processor with 24 cores at 2.1 GHz, with 24 GB RAM and CentOS Linux. The cGA for GPU was implemented in CUDA v3.0 using OpenMP for thread management, and evaluated in a eight-cores Xeon processor at 2.33 GHz, with 8GB RAM and 4 Tesla C1060 GPUs. The computing infrastructure used in the experimental analysis is from Cluster FING, Universidad de la República, Uruguay (cluster website: `http://www.fing.edu.uy/cluster`).

B. Results and discussion

Table I reports the results when solving the OneMax and Noisy OneMax using cGA. In CPU, instances with 1, 8, and 24 million variables were solved (the multicore computer used does not allow scaling up to solve larger problems). In GPU, instances with 1, 8, and 32 million variables, and a very large instance with more than a billion variables—$2^{30} = 1\ 073\ 741\ 824$ variables, exactly—were solved.

Table I
Experimental results for cGA in CPU and GPU

variables	OneMax	time (m)	Noisy OneMax	time (m)
cGA-CPU				
1 million	97.9%	12	97.8%	14
8 millions	85.5%	250	82.5%	292
24 millions	71.3%	1018	71.1%	1197
cGA-GPU				
1 million	82.5%	10	81.3%	12
8 millions	79.1%	178	78.0%	189
32 millions	74.1%	819	72.9%	874
1074 millions	62.3%	5800	60.1%	5982
cGA-GPU-async				
1 million	91.0%	9	86.8%	10
8 millions	82.6%	126	79.8%	153
32 millions	77.8%	590	76.6%	622
1074 millions	66.8%	5251	64.0%	5402

The results in Table I indicate that cGA in CPU is able to efficiently solve up to 8 million variables, but the efficiency significantly reduces for 24 million variables. On the other hand, the GPU implementations (specially the asynchronous version) require less execution time to achieve similar results quality than the CPU version, and they also allow scaling up to solve very large instances. The three cGA implementations have a very robust behavior: the results do not vary significantly when solving the noisy OneMax.

The experimental analysis confirmed that cGA is not the best choice to efficiently solve large optimization problems, due to its very slow evolution pattern. cGA required 100.000 generations for reaching an average fitness of 66.8% for the largest problem instance tackled. However, the GPU implementations proposed in this work are able to perform that number of generations rather efficiently, allowing to perform about 20 one-billion-bits generations per minute.

Figures 3 and 4 summarize the comparative analysis of the parallel cGA implementations in CPU and GPU, plotting the results quality vs. the problem dimension (in logarithmic scale) and the execution time required.

VII. Conclusions and future work

This article presented a parallel EA applied to solve large instances of the OneMax and Noisy OneMax problems. The EA was conceived to efficiently scale up to solve problem instances with several millions and even more than one billion variables, by using the computing power available in new multicore infrastructures (CPU and GPU).

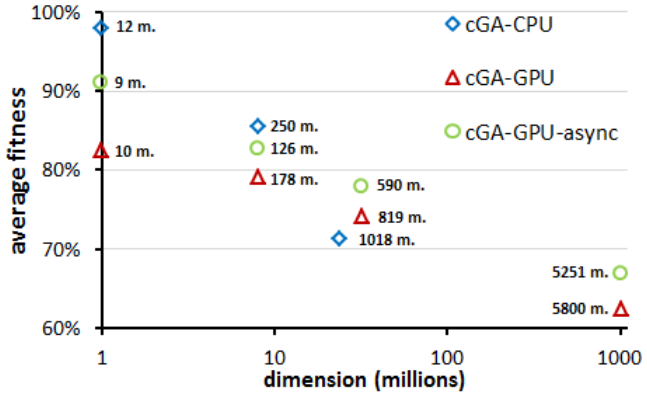

Figure 3. Comparison of parallel cGAs in CPU and GPU (OneMax).

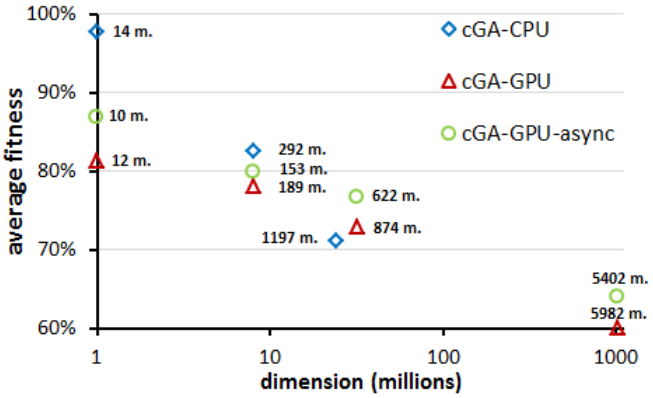

Figure 4. Comparison of parallel cGAs in CPU and GPU (Noisy OneMax).

Three versions of the proposed parallel EA were implemented: synchronous cGA in CPU and GPU, and an asynchronous version in GPU (only useful for separable optimization problems).

The experimental analysis showed that the GPU cGA implementations achieved the best results and efficiency values when solving problems with more than 8 million variables. This fact indicates that the new multicore GPU infrastructures provide a promising platform for implementing efficient EAs to solve very large optimization problems.

The main lines for future work include to further analyze the computational efficiency of EAs when solving very large optimization problems, in order to design even more efficient EAs able to tackle large real-world problems. The implementations in GPU can be optimized to compute accurate results in reduced execution times. Other algorithmic approaches, different to cGA, and other parallel implementations (such as distributed memory versions) of EAs should be studied to better understand the contribution of parallel models when solving huge optimization problems.

VIII. Acknowledgments

The work of S. Iturriaga and S. Nesmachnow was partially funded by ANII and PEDECIBA, Uruguay.

References

[1] T. Bäck, D. Fogel, and Z. Michalewicz, Eds., *Handbook of evolutionary computation*. Oxford University Press, 1997.

[2] E. Alba, *Parallel Metaheuristics: A New Class of Algorithms*. Wiley-Interscience, 2005.

[3] J. Schaffer and L. Eshelman, "On Crossover as an Evolutionary Viable Strategy," in *Proc. of the 4th Int. Conf. on Genetic Algorithms*. Morgan Kaufmann, 1991, pp. 61–68.

[4] D. Goldberg, K. Deb, and J. Clark, "Genetic Algorithms, Noise and the Sizing of Populations," *Complex Systems*, vol. 6, pp. 333–362, 1992.

[5] K. Sastry, D. Goldberg, and X. Llorà, "Towards billion-bit optimization via a parallel estimation of distribution algorithm," in *Proc. of Genetic and Evolutionary Computation Conference*, London, England, UK, 2007, pp. 577–584.

[6] G. Harik, F. Lobo, and D. Goldberg, "The compact genetic algorithm," *IEEE Trans. Evol. Comput.*, vol. 3, no. 4, pp. 287–297, 1999.

[7] E. Alba and M. Tomassini, "Parallelism and evolutionary algorithms," *IEEE Trans. Evol. Comput.*, vol. 6, no. 5, pp. 443–462, 2002.

[8] K. Deb, A. Reddy, and G. Singh, "Optimal scheduling of casting sequence using genetic algorithms," *Materials and Manufacturing Processes*, vol. 18, no. 3, pp. 409–432, 2003.

[9] K. Deb and K. Pal, "Efficiently solving: A large-scale integer linear program using a customized genetic algorithm," in *Proc. of the Genetic and Evolutionary Computation Conference*, Seattle, WA, USA, 2004, pp. 1054–1065.

[10] Y. Semet and M. Schoenauer, "An efficient memetic, permutation-based evolutionary algorithm for real-world train timetabling," in *Proc. of the IEEE Congress on Evolutionary Computation*, Edinburgh, UK, 2005, pp. 2752–2759.

[11] N. Kunasol, W. Suwannik, and P. Chongstitvatana, "Solving One-Million-Bit Problems using LZWGA," in *Int. Symposium on Information and Communication Technologies*, 2006.

[12] D. Goldberg, K. Sastry, and X. Llorà, "Toward routine billion-variable optimization using genetic algorithms," *Complexity*, vol. 12, no. 3, pp. 27–29, 2007.

[13] G. Harik, E. Cantú-Paz, D. Goldberg, and B. Miller, "The gambler's ruin problem, genetic algorithms, and the sizing of populations," *Evol. Comput.*, vol. 7, no. 3, pp. 231–253, 1999.

[14] W. Suwannik and P. Chongstitvatana, "Solving one-billion-bit noisy onemax problem using estimation distribution algorithm with arithmetic coding," in *Proc. of the IEEE Congress on Evolutionary Computation*, 2008, pp. 1203–1206.

[15] nVidia, "CUDA website," Available online http://www.nvidia.com/object/cuda_home.html, 2010, accessed on May 2012.

[16] A. Malony, S. Biersdorff, W. Spear, and S. Mayanglambam, "An experimental approach to performance measurement of heterogeneous parallel applications using CUDA," in *Proc. of the 24th ACM Int. Conf. on Supercomputing*. New York, NY, USA: ACM, 2010, pp. 127–136.

[17] S. Nesmachnow, F. Luna, and E. Alba, "Time analysis of standard evolutionary algorithms as software programs," in *Proc. of the 11th Int. Conf. on Intelligent Systems Design and Applications*, Cordoba, Spain, 2011, pp. 271–276.

[18] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3–30, 1998.