A Comparison of Many-threaded Differential Evolution and Genetic Algorithms on CUDA

Pavel Krömer, Jan Platoš, Václav Snášel, Ajith Abraham
Department of Computer Science
FEECS, VŠB Technical University of Ostrava
Ostrava, Czech Republic
Email: pavel.kromer@vsb.cz,jan.platos@vsb.cz,vaclav.snasel@vsb.cz,ajith.abraham@ieee.org

***Abstract*—The recent time has seen the rise of consumer grade massively parallel environments. Powerful GPUs and multi-core processors became widely available and easy to use programming APIs such as nVidia CUDA, OpenCL, and DirectCompute simplify the development of applications that can utilize them. In this environment, the nature inspired meta-heuristics can be in suitable cases implemented in parallel without additional costs. Backed by the power of modern GPGPUs, the meta-heuristics can be deployed to solve practical real world problems. In this paper, we compare differential evolution and genetic algorithms implemented on CUDA when solving the independent tasks scheduling problem.**

***Keywords*-genetic algorithms, differential evolution, CUDA, independent task scheduling**

I. Introduction

Modern Graphics Processing Units (GPUs) represent a budget environment for massively parallel computations. Hand in hand with the wide availability of the hardware, there are also APIs and modern development kits that enable rapid development of parallel applications. Naturally, many evolutionary algorithms including e.g. the genetic algorithms, genetic programming, and differential evolution, were implemented for the GPUs using different tools and approaches. Such a GPU implementations were shown to improve the performance of the algorithms dramatically and the speedup of evolutionary algorithms obtained by the use of the GPUs can contribute to the usage of evolutionary computation for practical problems.

In this study we compare the efficiency of genetic algorithms and differential evolution implemented on the nVidia Compute Unified Device Architecture (CUDA) platform. Both evolutionary algorithms were implemented from scratch following the same design principles and we have compared their performance when solving a benchmark problem. The chosen benchmark problem was the independent tasks scheduling problem, a well known combinatorial optimization problem that requires creating an efficient schedule of execution of a set of independent tasks on a set of resources (computing nodes). The problem was chosen because it is a real world problem, it was addressed by both, exact and meta-heuristic algorithms in the past, and there is a data set to execute the experiments readily available.

Modern graphics hardware has gained an important role in the area of parallel computing. Graphic cards have been used to accelerate gaming and 3D graphics applications, but recently, they have been used to perform general computations as well. The area of general purpose GPUs (GPGPUs) programming has become a hot topic in parallel computation.

The main advantage of the GPU is its structure. Standard CPUs (central processing units) contain usually 1-4 complex computational cores, memory registers and large cache memory. The GPUs contain up to several hundreds of simplified execution cores grouped into so-called multiprocessors. Every SIMD (Single Instruction Multiple Data) multiprocessor drives eight arithmetic logic units (ALU) which process data, thus each ALU of a multiprocessor executes the same operations on different data, stored in the registers or device memory. In contrast to standard CPUs which can re-schedule operations (out-of-order execution), current GPUs are an example of an in-order architecture. This drawback is overcome by their massive parallelism as described by Hager et al. [1]. Current general-purpose CPUs with clock rates of 3 GHz outperform a single ALU of the multiprocessors with its rather slow 1.3 GHz. The huge number of parallel processors on a single GPU chip compensates this drawback.

The GPGPU programming has offered a new platform for evolutionary computation [2]. The majority of the evolutionary algorithms including genetic algorithms [3]–[5], genetic programming [6], [7], and differential evolution [8]–[10] were implemented on the GPU. Most of the current implementations of said algorithms have two things in common: they struggle with random number generation and they map each candidate solution in the population to one GPU thread.

The nVidia CUDA-C language is an extension to C that allows development of GPU routines called kernels. Each kernel defines instructions that are executed on the GPU by many threads at the same time following the SIMD model. The threads can be organized into so called thread groups that can benefit from GPU features such as fast shared memory, atomic data manipulation, and synchronization. The CUDA runtime takes care of the scheduling and execution of the thread groups on available hardware. The set of thread groups requested to execute a kernel is called in CUDA terminology a grid. A kernel program can use several types of memory: fast local and shared memory, large but slow global memory, and fast read-only constant memory and texture memory.

II. Brief introduction of genetic algorithms and differential evolution

The genetic algorithms (GA) are based on the software implementation of genetic evolution [11]. Genetic algorithms evolve a population of chromosomes representing potential problem solutions encoded into suitable data structures (chromosomes). Candidate solutions are most often encoded as binary strings, integer vectors, or real vectors.

Artificial evolution consists of the iterative application of genetic operators, introducing to the algorithm evolutionary principles such as inheritance, the survival of the fittest, and random perturbations. Iteratively, the current population of candidate solutions is modified with the aim of forming a new and, it is hoped, better population to be used in the next generation. The evolution of problem solutions ends after specified termination criteria have been satisfied. After the termination of the search process, the evolution winner is decoded and presented as the most optimal solution found.

The differential evolution (DE) [12] is a population-based evolutionary optimizer that evolves real encoded vectors representing candidate solutions to given problem. The DE starts with an initial population of N real-valued vectors. During the optimization, DE generates new vectors that are perturbations of existing population vectors. The algorithm perturbs vectors with the scaled difference of two (or more) randomly selected population vectors and adds the scaled random vector difference to a third randomly selected population vector to produce so called trial vector (hence the name differential evolution). The trial vector competes with a member of the current population with the same index. If the trial vector represents a better solution than the population vector, it takes its place in the population [12].

Both algorithms are viable evolutionary meta-heuristics. The differential evolution represents an alternative to the concept of genetic algorithms. As well as genetic algorithms, it represents a highly parallel population based stochastic search meta-heuristic. In contrast to GA, differential evolution uses real encoding of chromosomes and different operations to evolve the population. It results in different search strategy and different directions found by DE when crawling a fitness landscape of the problem domain.

III. Related work

Ever since its inception, the GPGPUs were recognized as the devices that can leverage the use of (not only) evolutionary nature inspired meta-heuristics significantly. The raw power of up to several hundred cores on a single device can be utilized to accelerate various evolutionary algorithms. Usually, the most expensive step in the artificial evolution is the evaluation of candidate solutions, which can be in most cases done in parallel. Nevertheless, an efficient parallelization of the rest of the evolutionary meta-heuristics is also a key to the optimal use of the resources on the GPGPU devices. In this section, we provide brief summary of recent implementations of the GA and DE on the GPGPUs.

A. Genetic Algorithms on GPUs

The first attempts to run GAs on the GPUs predate the public availability of GPGPU APIs. At that time, the GA had to be translated to shader programs and the data structures to textures. For example, the work of Wong and Wong [13], [14], introduced a GPU parallelization of a modified GA extended with the Cauchy mutation operator used in evolutionary programming. Yu et al. [15] used the GPU to execute a real encoded parallel GA and discussed the different data structures mapped to GPU textures (population texture, fitness texture, random texture).

In [3], Maitre et al. presented an overview of GA implementation efforts on CUDA and concluded, that the GA is hard to implement in the SIMD environment. They presentd a custom language, EASEA, to help implementing the GA in parallel environments and achieved significant GA speedup orders of magnitude large.

Tsutsui and Fujimoto [4] developed a parallel GA to solve the quadratic assignment problem on CUDA and achieved a 3 - 28 times faster solution when compared to the CPU. Wong [5] proposed a CUDA implementation of a parallel multi-objective GA and improved the execution times of the algorithm 5 - 10 times.

B. Differential Evolution on GPUs

Due to the simplicity of its operations and fixed encoding of candidate solutions, DE is suitable for parallel implementation on the GPUs. In DE, each candidate solution is represented by a vector of real numbers and the population as a whole can be seen as a real matrix. Moreover, both mutation and crossover can be in DE implemented easily.

The first implementation of DE on the CUDA platform was introduced in the early 2010 by de Veronese and Krohling [9]. The DE algorithm was implemented using the CUDA-C language and it achieved on a set of benchmarking functions speedup between 19 and 34 times comparing to the CPU implementation. The generation of random numbers was implemented using the Mersenne Twister from the CUDA SDK and the selection of random trial vectors for mutation was done on the CPU.

Zhu [8], and Zhu and Li [10] implemented the DE on CUDA as part of differential evolution-pattern search algorithm for bound constrained optimization problems and as a part of a differential evolutionary Markov chain Monte Carlo method (DE-MCMC) respectively. In both papers, performance of the algorithms was demonstrated on a set of continuous benchmarking functions.

IV. Implementation of GA and DE

In this section we introduce the details of the implementation of GA and DE on the CUDA platform. The goal of the implementation of both algorithms was achieving high parallelism while keeping the simplicity of the algorithms. Also, it is designed to avoid two common properties of up-to-date GA and DE implementations for CUDA: the candidate solution to single thread mapping (we use many threads to process each candidate solution) and the problems with random numbers generation (we use cuRAND to generate pseudorandom numbers on the GPU). The overall goal of the implementations is to to process each candidate solution by many threads during both, fitness function evaluation and evolutionary operations.

The implementations consists of a set of CUDA-C kernels for generation of initial population, generation of batches of pseudorandom numbers for decision making, merger of the old and new populations, the implementation of the operations specific for each meta-heuristic, and for evaluation of candidate solutions.

The kernels were implemented using the following principles:

i. Each candidate solution is processed by a thread block (thread group). The number of thread groups is in CUDA currently limited to $(2^{16} - 1)^2$ and hence the maximum population size is in this case the same.
ii. Each candidate solution gene (vector coordinate) is processed by a thread. The limit of threads per block depends in CUDA on the hardware compute capability and it is 512 for compute capability 1.x and 1024 for compute capability 2.x [16]. This limit enforces the maximum vector length. For the first use case considered in this paper, candidate vectors with length 512 are needed.
iii. Each kernel call aims to process the whole population in one step, e.g. it asks the CUDA runtime to launch M blocks with 512 threads in parallel. The runtime executes the kernel with respect to available resources.

The flowchart of used DE implementation is shown in Fig. 1 and the flowchart of used GA implementation is shown in Fig. 1. The DE is rather straightforward, but the GA contains additional steps such as pre-selection of parents and optional pre-computation of data for migration. They are performed on the CPU due its higher complexity. Parent selection is done on the CPU and for steady-state GA, the chromosomes to establish new population are selected by the CPU (pre-compute migration step).

This implementation brings several advantages. First, all the generic operations can be considered done in parallel and thus their complexity reduces from $M \times N$ (population size multiplied by vector length) to c (constant, duration of the operation plus CUDA overhead). Second, this implementation operates in a highly parallel way also on logical level. A population of offspring candidate solutions of the same size as the parent population is created in a single step and later merged with the parent population. Third, the evaluation of fitness function is accelerated by the GPU.

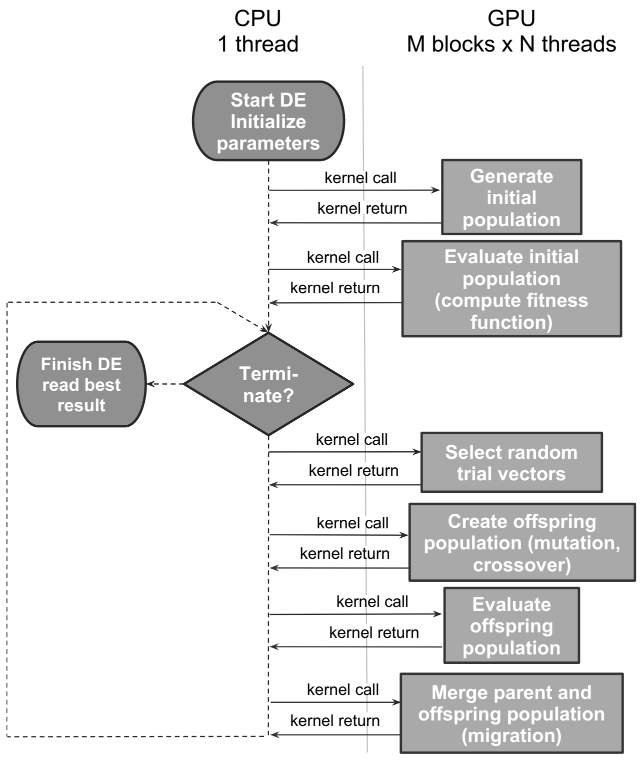
CPU
1 thread
GPU
M blocks x N threads
Start DE Initialize parameters
kernel call
kernel return
Generate initial population
kernel call
kernel return
Evaluate initial population (compute fitness function)
Finish DE read best result
Termi-nate?
kernel call
kernel return
Select random trial vectors
kernel call
kernel return
Create offspring population (mutation, crossover)
kernel call
kernel return
Evaluate offspring population
kernel call
kernel return
Merge parent and offspring population (migration)

Figure 1: The flowchart of the DE implementation on CUDA.

V. Computational experiments

In this section we describe computational experiments conducted in order to find out which algorithm performs better for a given test problem.

A. Independent task scheduling

Independent task scheduling can be defined as a mapping of a set of tasks to a set of resources [17], [18]. Efficient scheduling is required to exploit the different capabilities of a set of heterogeneous resources but it is an NP-complete problem [19] and it cannot be solved by exact methods in reasonable time. Instead, it was a subject to various heuristic [18], [20], [21] and meta – heuristic [22]–[26] algorithms.

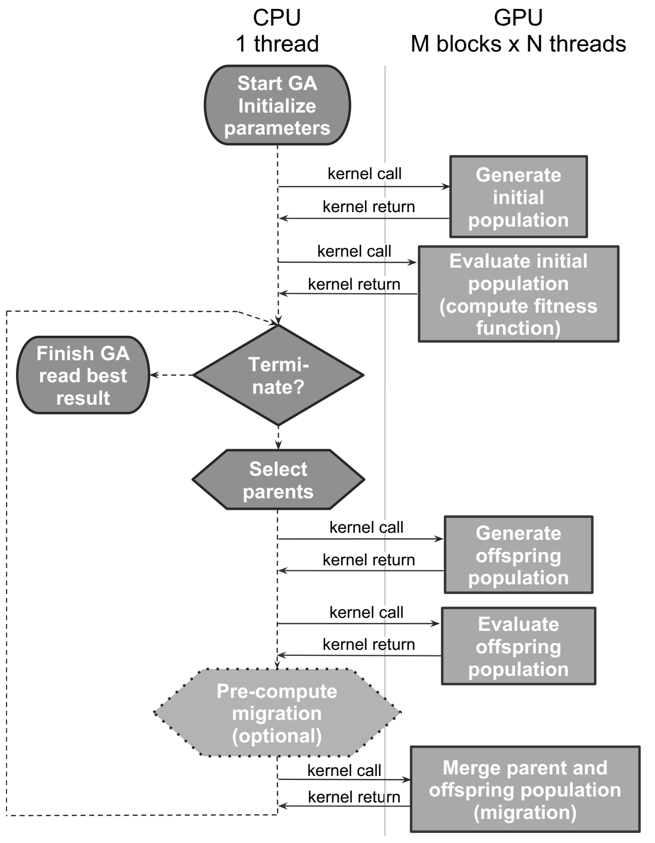

Figure 2: The flowchart of the GA implementation on CUDA.

Let $T = \{T_1, T_2, \dots, T_n\}$ denote the set of independent tasks with no inter-task dependencies that is in a specific time interval submitted to a resource management system (RMS). Assume at the time of receiving these tasks by RMS, m machines $M = \{M_1, M_2, \dots, M_m\}$ are available and no preemption is allowed (i.e. the tasks cannot change the resource they have been assigned to). Scheduling is done on machine level and it is assumed that each machine uses First-Come, First-Served (FCFS) method for performing the received tasks. We assume that each machine can estimate how much time it requires to perform each task. In [18] Expected Time to Compute (ETC) matrix is used to estimate the required time for executing a task in a machine. An ETC matrix is a $n \times m$ matrix in which n is the number of tasks and m is the number of machines. One row of the ETC matrix contains the estimated execution time for a given task on each machine. Similarly one column of the ETC matrix consists of the estimated execution time of a given machine for each task. Thus, for an arbitrary task T_j and an arbitrary machine M_i , $[ETC]_{j,i}$ is the estimated execution time of T_j on M_i. In the ETC model we take the usual assumption that we know the computing capacity of each resource, an estimation or prediction of the computational needs of each job, and the load of prior work of each resource.

The two objectives to optimize during the task mapping are makespan and flowtime. Optimum makespan (meta-task execution time) and flowtime of a set of jobs can be defined as:

$$makespan = \min_{S \in Sched} \{ \max_{j \in Jobs} F_j \} \tag{1}$$

$$flowtime = \min_{S \in Sched} \{ \sum_{j \in Jobs} F_j \} \tag{2}$$

where $Sched$ is the set of all possible schedules, $Jobs$ stands for the set of all jobs to be scheduled, and F_j represents the time in which job j finalizes.

Minimizing makespan aims to execute the whole meta-task as fast as possible while minimizing flowtime aims to utilize the computing environment efficiently.

A schedule of n independent tasks executed on m machines can be naturally expressed as a string of n integers $S = (s_1, s_2, \dots, s_n)$ that are subject to $s_i \in 1, \dots, m$. The value at i-the position in S represents the machine on which is the i-the job scheduled in schedule S. This schedule encoding was used for the GA. The DE uses for problem encoding real vectors so real coordinates must be used instead of discrete machine numbers. The real-encoded DE vector is in this work translated to schedule representation by simple truncation of its coordinates (e.g. $3.6 \to 3$, $1.2 \to 1$).

Assume schedule S from the set of all possible schedules $Sched$. For the purpose of differential evolution, we define a fitness function $f(S) : Sched \to \mathbb{R}$ that evaluates each schedule:

$$f(S) = \lambda \cdot makespan(S) + (1 - \lambda) \cdot \frac{flowtime(S)}{m} \tag{3}$$

The function $f(S)$ is a sum of two objectives, the makespan of schedule S and flowtime of schedule S divided by number of machines m to keep both objectives in approximately the same magnitude. The influence of makespan and flowtime in $f(S)$ is parameterized by the variable λ. The same schedule evaluation was already used several times, see e.g. [25], [26].

B. Experiments

To evaluate the performance of the GA and DE for minimizing the makespan and flowtime, we have used the benchmark proposed in [18]. The simulation model is based on the ETC matrix for 512 jobs and 16 machines. The instances of the benchmark are classified into 12 different types of ETC matrices according to [18]:

- *task heterogeneity*, i.e. the amount of variance among the execution times of tasks for a given machine
- *machine heterogeneity*, i.e. the variation among the execution times for a given task across all the machines
- *consistency*. An ETC matrix is said to be consistent whenever a machine M_j executes any task T_i faster than machine M_k; in this case, machine M_j executes all tasks faster than machine M_k

Table I: GA and DE settings

GA		DE	
paremeter	value	parameter	value
population size	64	population size	64
mut. probability	0.01	F	0.9
cros. probability	0.8	C	0.9
selection	semi elitary		

Table II: Optimization results

ETC matrix	GA	DE
ThMhCc	2.34994e+07	9.55294e+06
ThMhCi	1.38284e+07	3.17031e+06
ThMhCs	1.45571e+07	4.28296e+06
ThMlCc	207687	189457
ThMlCi	182313	78844.7
ThMlCs	171689	104447
TlMhCc	755855	331510
TlMhCi	468615	104894
TlMhCs	493757	142368
TlMlCc	6834.15	6158.24
TlMlCi	5731.92	2550.79
TlMlCs	5873.45	3391.38

- *inconsistency* – machine M_j may be faster than machine M_k for some tasks and slower for others

GA and DE to solve the independent task scheduling problem were implemented as outlined in section IV with (3) with $T = 0.5$ as fitness function. The goal of the algorithm was to *minimize* the fitness. The parameters of GA and DE set on the basis of previous experience and after initial tuning are shown in Table I. The run of each algorithm was terminated after exactly one minute. The experiment was performed on a server with 2 dual core AMD Opteron processors at 2.6GHz and nVidia Tesla C2050 with 448 cores at 1.15GHz.

The average final fitness obtained for each ETC matrix by the GA and DE is shown in Table II. We can clearly see that the DE was able to find significantly better schedules within the given minute. The differencies between final fitness for the DE and GA are also illustrated in Fig. 3. Indeed this is an interesting results that can be attributed to several reasons:

- the DE is a meta-heuristic that solves the independent task scheduling problem better than the GA. The no free lunch theorem [27] explains why some algorithms are better at solving certain problems and wors at solving another problems.
- the many-threaded implementation suits better to the DE than GA. However, we note that more GA operations have been performed on the CPU because their parallelization was impractical(e.g. the parent selection).

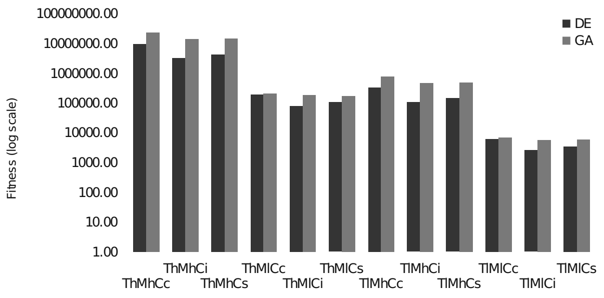

Figure 3: The flowchart of the GA implementation on CUDA.

VI. Conclusions

In this paper, we have compared a many-threaded implementation of two nature inspired meta-heuristics, the GA and the DE. In contrast to previous GPU implementations of the GA and DE, the presented implementation processes each candidate solution with many threads and generates the random numbers on the GPU. This approach seeks to utilize the resources of the GPGPU as much as possible.

The CUDA implementation of the DE was easier because the variant of the algorithm that was implemented can be almost entirely expressed using matrix vector operations. The parallelization of the GA was not so straightforward because some parts of the algorithm (parent selection, transition from one population to another) are not suitable for parallelization in a SIMD environment such as the CUDA. We have performed a direct comparison of the two algorithms on the same problem and the DE was clear winner in terms of finding better schedules than the GA. In the future, we will study whether the many-threaded GA implementation performs poorly also on other problems, which would mean that either our parallel model or its implementation is unsatisfactory.

Acknowledgement

This work was supported by the Ministry of Industry and Trade of the Czech Republic, under the grant no. FR-TI1/420.

References

[1] G. Hager, T. Zeiser, and G. Wellein, "Data access optimizations for highly threaded multi-core cpus with multiple memory controllers," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1 –7, 2008.

[2] T. J. Desell, D. P. Anderson, M. Magdon-Ismail, H. J. Newberg, B. K. Szymanski, and C. A. Varela, "An analysis of massively distributed evolutionary algorithms," in *IEEE Congress on Evolutionary Computation*, pp. 1–8, IEEE, 2010.

[3] O. Maitre, L. A. Baumes, N. Lachiche, A. Corma, and P. Collet, “Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA,” in *GECCO ’09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, (New York, NY, USA), pp. 1403–1410, ACM, 2009.

[4] S. Tsutsui and N. Fujimoto, “Solving quadratic assignment problems by genetic algorithms with gpu computation: a case study,” in *GECCO ’09: Proceedings of the 11th annual conference companion on Genetic and evolutionary computation conference*, (New York, NY, USA), pp. 2523–2530, ACM, 2009.

[5] M. L. Wong, “Parallel multi-objective evolutionary algorithms on graphics processing units,” in *GECCO (Companion)* (F. Rothlauf, ed.), pp. 2515–2522, ACM, 2009.

[6] D. Robilliard, V. Marion, and C. Fonlupt, “High performance genetic programming on gpu,” in *Proceedings of the 2009 workshop on Bio-inspired algorithms for distributed systems*, BADS ’09, (New York, NY, USA), pp. 85–94, ACM, 2009.

[7] W. Langdon and W. Banzhaf, “A simd interpreter for genetic programming on gpu graphics cards,” in *Genetic Programming* (M. O’Neill, L. Vanneschi, S. Gustafson, A. Esparcia Alcázar, I. De Falco, A. Della Cioppa, and E. Tarantino, eds.), vol. 4971 of *Lecture Notes in Computer Science*, pp. 73–85, Springer Berlin / Heidelberg, 2008.

[8] W. Zhu, “Massively parallel differential evolution - pattern search optimization with graphics hardware acceleration: an investigation on bound constrained optimization problems,” *Journal of Global Optimization*, pp. 1–21, 2010. 10.1007/s10898-010-9590-0.

[9] L. de Veronese and R. Krohling, “Differential evolution algorithm on the gpu with c-cuda,” in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pp. 1 –7, 2010.

[10] W. Zhu and Y. Li, “Gpu-accelerated differential evolutionary markov chain monte carlo method for multi-objective optimization over continuous space,” in *Proceeding of the 2nd workshop on Bio-inspired algorithms for distributed systems*, BADS ’10, (New York, NY, USA), pp. 1–8, ACM, 2010.

[11] M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press, 1996.

[12] K. V. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution A Practical Approach to Global Optimization*. Natural Computing Series, Berlin, Germany: Springer-Verlag, 2005.

[13] T. Wong and M. Wong, “Parallel evolutionary algorithms on Consumer-Level graphics processing unit,” in *Parallel Evolutionary Computations*, pp. 133–155, Springer, 2006.

[14] M. Wong and T. Wong, “Implementation of parallel genetic algorithms on graphics processing units,” in *Intelligent and Evolutionary Systems*, pp. 197–216, ACM, 2009.

[15] Q. Yu, C. Chen, and Z. Pan, “Parallel genetic algorithms on programmable graphics hardware,” in *ICNC (3)* (L. Wang, K. C. 0001, and Y.-S. Ong, eds.), vol. 3612 of *Lecture Notes in Computer Science*, pp. 1051–1059, Springer, 2005.

[16] NVIDIA, *NVIDIA CUDA Programming Guide 3.2*. 2010.

[17] S. Ali, T. Braun, H. Siegel, and A. Maciejewski, “Heterogeneous computing,” in *Encyclopedia of Distributed Computing* (J. Urbana and P. Dasgupta, eds.), Kluwer Academic Publishers, Norwell, MA, 2002.

[18] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, “A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems,” *J. Parallel Distrib. Comput.*, vol. 61, pp. 810–837, June 2001.

[19] D. Fernandez-Baca, “Allocating modules to processors in a distributed system,” *IEEE Trans. Softw. Eng.*, vol. 15, no. 11, pp. 1427–1436, 1989.

[20] E. Munir, J.-Z. Li, S.-F. Shi, and Q. Rasool, “Performance analysis of task scheduling heuristics in grid,” in *Machine Learning and Cybernetics, 2007 International Conference on*, vol. 6, pp. 3093–3098, aug. 2007.

[21] H. Izakian, A. Abraham, and V. Snasel, “Comparison of heuristics for scheduling independent tasks on heterogeneous distributed environments,” in *Computational Sciences and Optimization, 2009. CSO 2009. International Joint Conference on*, vol. 1, pp. 8 –12, april 2009.

[22] G. Ritchie and J. Levine, “A hybrid ant algorithm for scheduling independent jobs in heterogeneous computing environments,” in *Proceedings of the 23rd Workshop of the UK Planning and Scheduling Special Interest Group*, December 2004.

[23] A. YarKhan and J. Dongarra, “Experiments with scheduling using simulated annealing in a grid environment,” in *GRID ’02: Proceedings of the Third International Workshop on Grid Computing*, (London, UK), pp. 232–242, Springer-Verlag, 2002.

[24] A. J. Page and T. J. Naughton, “Framework for task scheduling in heterogeneous distributed computing using genetic algorithms,” *Artificial Intelligence Review*, vol. 24, pp. 137–146, 2004.

[25] J. Carretero, F. Xhafa, and A. Abraham, “Genetic algorithm based schedulers for grid computing systems,” *International Journal of Innovative Computing, Information and Control*, vol. 3, no. 7, 2007.

[26] P. Kromer, V. Snasel, J. Platos, A. Abraham, and H. Ezakian, “Evolving schedules of independent tasks by differential evolution,” in *Intelligent Networking, Collaborative Systems and Applications* (S. Caballé, F. Xhafa, and A. Abraham, eds.), vol. 329 of *Studies in Computational Intelligence*, pp. 79–94, Springer Berlin / Heidelberg, 2011.

[27] D. H. Wolpert and W. G. Macready, “No free lunch theorems for optimization,” *Evolutionary Computation, IEEE Transactions on*, vol. 1, pp. 67–82, August 2002.