Many-threaded Implementation of Differential Evolution for the CUDA Platform

Pavel Krömer
VŠB - Technical University of Ostrava
17.listopadu 15
Ostrava, Czech Republic
pavel.kromer@vsb.cz

Václav Snášel
VŠB - Technical University of Ostrava
17.listopadu 15
Ostrava, Czech Republic
vaclav.snasel@vsb.cz

Jan Platoš
VŠB - Technical University of Ostrava
17.listopadu 15
Ostrava, Czech Republic
jan.platos@vsb.cz

Ajith Abraham
VŠB - Technical University of Ostrava
17.listopadu 15
Ostrava, Czech Republic
ajith.abraham@ieee.org

ABSTRACT

Differential evolution is an efficient populational meta – heuristic optimization algorithm successful in solving difficult real world problems. Due to the simplicity of its operations and data structures, it is suitable for a parallel implementation on multicore systems and on the GPU. In this paper, we design a simple yet highly parallel implementation of the differential evolution using the CUDA architecture. We demonstrate the speedup obtained by the proposed parallelization of the differential evolution on an NP hard combinatorial optimization problem and on a benchmark function of many variables.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Experimentation, Performance

Keywords

Differential Evolution, Parallelization, CUDA, Scheduling

1. INTRODUCTION

Many methods and algorithms are nowadays redesigned for the Graphics Processing Units (GPUs) because many modern GPUs offer massive parallelism for a budget price whereas new APIs simplify the development of parallel applications. Among others, many evolutionary algorithms including genetic algorithms, genetic programming, and differential evolution, were implemented for the GPU. Such GPU implementations were already shown to improve the performance of the algorithms dramatically. However, the advances in both, hardware resources and software tools for the GPU application development, motivate further development of the algorithms and their re-designing to utilize the latest features of the GPU computation. The speedup of evolutionary processing obtained by efficient use of the GPUs can contribute to the deployment of evolutionary computation for use cases that require short response times, e.g. on-line evolutionary optimization.

In this paper we propose a novel many-threaded implementation of the differential evolution (DE) for the nVidia Compute Unified Device Architecture (CUDA) platform. We demonstrate its performance and quality of obtained solutions on a real world combinatorial optimization problem, the scheduling of independent tasks in heterogeneous computing environments, and on a search for optimum of a benchmark function of many variables. The scheduling of independent tasks is an appealing combinatorial optimization problem consisting of a search for optimal mapping of a set of tasks to a set of available resources. This work compares the GPU implementation of DE for independent task scheduling to a simple and optimized single-thread CPU implementation of the same algorithm. The search for a minimum of a benchmark function of many variables is utilized to provide rough comparison of the proposed DE to existing GPU implementations. The CUDA implementation of the differential evolution uses the CUDA-C language and the recent nVidia CUDA Software Development Kit (SDK) 3.2.

This paper is organized as follows: in Section 2 we introduce the GPU computing, nVidia CUDA architecture and the nVidia CUDA SDK 3.2. In Section 3 are outlined the basic principles of the differential evolution and state of the art of the DE on GPUs. Next, the new DE implementation for the CUDA architecture is proposed. Finally, Section 4 provides a complex use case demonstrating the performance and quality of the proposed algorithm on a combinatorial optimization problem and when seeking for minimum of a test function of many variables.

2. GPU COMPUTING

Modern graphics hardware has gained an important role in the area of parallel computing. Graphic cards have been used to power gaming and 3D graphics applications, but recently, they have been used to accelerate general computations as well. The new area of general purpose GPU (GPGPU) programming has been flourishing since then.

Complex architecture of the GPUs is suitable for vector and matrix algebra operations, which leads to the wide use of GPUs in the area of scientific computing with applications in information retrieval, data mining, image processing, data compression and so on. Nowadays, the developer does not have to be an expert in graphics hardware because of the existence of various Application Programming Interfaces (APIs) that help to implement parallel applications rapidly. Nevertheless, it is still crucial to follow elementary rules of GPGPU programming to write efficient code. 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 reschedule 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. [8]. 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 GPUs were used to accelerate the implementation of many applications. Andrecut [2] described CUDA-based computing for two variants of Principal Component Analysis (PCA). The usage of parallel computing improved efficiency of the algorithm more than 12 times in comparison with CPU. Preis et al. [18] applied GPU on methods of fluctuation analysis, which includes determination of scaling behavior of a particular stochastic process and equilibrium autocorrelation function in financial markets. The calculation was more than 80 times faster than the previous version running on CPU. Patnaik et al. [16] used GPU in the area of temporal data mining in neuroscience. They analyzed spike train data with the aid of a novel frequent episode discovery algorithm, achieving a more than 430× speedup.

The GPGPU programming has offered a new platform for evolutionary computation [6]. The majority of the evolutionary algorithms including genetic algorithms [17], genetic programming [21, 11], and differential evolution [23, 5, 24] 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.

The structure of CUDA program and the relation of threads and thread groups to device memory is illustrated in Figure 1.

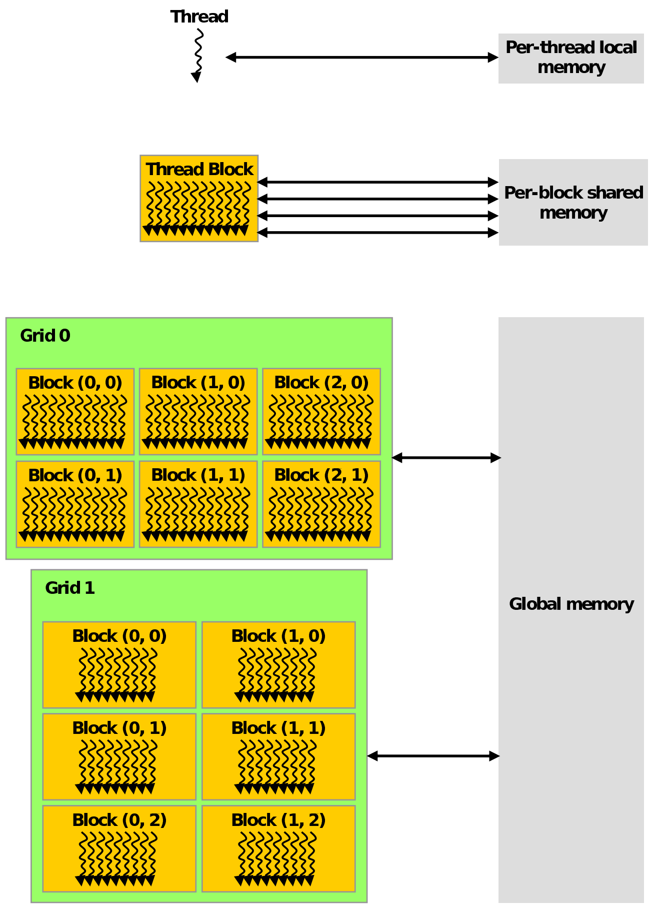


Figure 1: CUDA program structure and memory hierarchy [13].

The nVidia CUDA SDK 3.2 was released in November 2010 and one of its main features is the cuRAND library for generation of pseudorandom and quasirandom numbers on the GPU. The library can generate around 12 millions random floats per second and its methods can be easily used from within CUDA kernels and host programs [14].

3. DIFFERENTIAL EVOLUTION

Differential evolution (DE) is a versatile and easy to use stochastic evolutionary optimization algorithm [19]. DE is a population-based optimizer that evolves real encoded vectors representing the solutions to given problem. The DE starts with an initial population of N real-valued vectors. The vectors are initialized with real values either randomly or so, that they are evenly spread over the problem domain. The latter initialization leads to better results of the optimization process [19].

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. 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 [19].

Differential evolution is parametrized by two parameters [19]. Scale factor $F \in (0, 1+)$ controls the rate at which the population evolves and the crossover probability $C \in [0, 1]$ determines the ratio of coordinates that are transferred to the trial vector from its opponent. The number of vectors in the population is also an important parameter of the population. The outline of the classical DE is illustrated in Algorithm 1.

Algorithm 1: A summary of Differential Evolution

```
1  Initialize the population $P$ consisting of $M$ vectors;
2  Evaluate an objective function ranking the vectors in
   the population;
3  while Termination criteria not satisfied do
4      for $i \in \{1, \ldots, M\}$ do
5          Create trial vector $v_t^i = v_r^1 + F(v_r^2 - v_r^3)$, where
           $F \in [0, 1]$ is a parameter and $v_r^1$, $v_r^2$ and $v_r^3$ are
           three random vectors from the population $P$.
           This step is in DE called mutation;
6          Validate the range of coordinates of $v_t^i$.
           Optionally adjust coordinates of $v_t^i$ so, that $v_t^i$ is
           valid solution to given problem;
7          Perform uniform crossover. Select randomly one
           point (coordinate) $l$ in $v_t^i$. Let $v_t^i[l] = v^i[l]$. Let
           $v_t^i[m] = v^i[m]$ with probability $1 - C$ for each
           $m \in \{1, \ldots, N\}$ such that $m \neq l$;
8          Evaluate the trial vector. If the trial vector $v_t^i$
           represent a better solution than population
           vector $v^i$, replace $v^i$ in $P$ by $v_t^i$;
9      end
10 end
```

Differential evolution represents an alternative to the concept of genetic algorithms (GA). 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.

3.1 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 [5]. 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 CUDA SDK and the selection of random trial vectors for mutation was done on the CPU.

Zhu [23], and Zhu and Li [24] 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.

The common properties of the above DE implementations are:

i. One GPU thread is used to process one candidate solution.

ii. The generation of random numbers is an issue. The Mersenne Twister from the CUDA SDK was used in both applications and some parts of the random number generation process were in the first case done on the CPU.

In this paper, we present a new implementation of DE for the CUDA architecture that uses many threads for each candidate solution and the efficient cuRAND library for random number generation on the GPU. Moreover, we demonstrate the performance of the algorithm on a combinatorial optimization problem and on a benchmark function.

3.2 Many-threaded Implementation of DE on the CUDA Platform

The goal of the implementation of differential evolution on the CUDA platform was achieving high parallelism while keeping the simplicity of the algorithm. The implementation consists of a set of CUDA-C kernels for generation of initial population, generation of batches of random numbers for the decision making, DE processing including generation of trial vectors, mutation and crossover, verification of the generated vectors, and the merger of parent and offspring populations. Besides these generic kernels implementing the DE, an implementation of the fitness function evaluation was done in a separate kernel. The overview of the presented DE implementation is shown in Figure 2.

The kernels were implemented using the following principles:

i. Each DE vector (i.e. 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 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 [13].

This limit enforces the maximum vector length. For the first use case considered in this paper, candidate vectors with length 512 are needed. The mapping of CUDA threads and thread blocks to DE candidate vectors is illustrated in Figure 3.

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.

Such an implementation brings several advantages. First, all the generic DE 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 DE operates in a highly parallel way also on logical level. A population of offspring vectors 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 vectors is accelerated by the implementation of the fitness function on GPU.

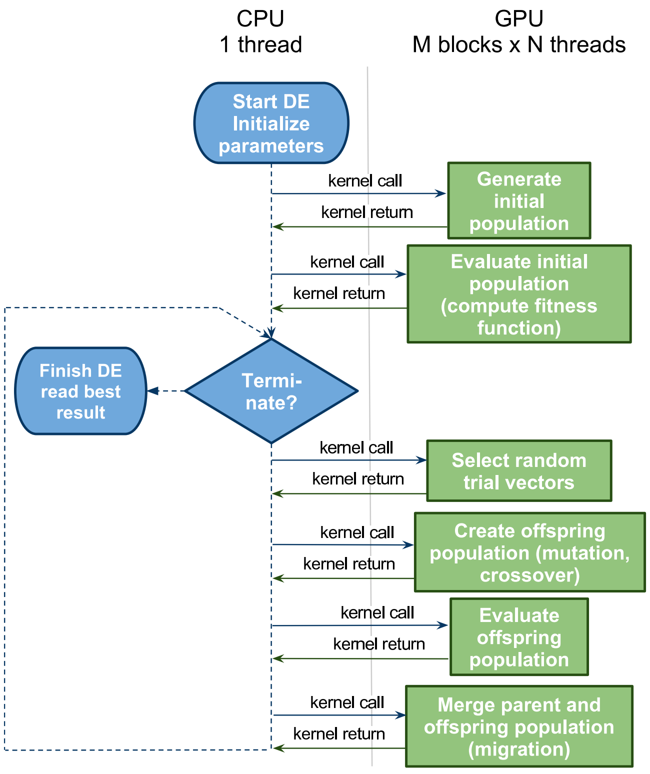

CPU
1 thread
GPU
M blocks x N threads
Start DE Initialize parameters
kernel call
kernel return
Generate initial population
Evaluate initial population (compute fitness function)
Finish DE read best result
Terminate?
Select random trial vectors
Create offspring population (mutation, crossover)
Evaluate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

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

4. EXAMPLES OF DIFFERENTIAL EVOLUTION ON CUDA

This section consists of two examples of the proposed DE on CUDA. First, a complex NP-hard combinatorial optimization problem is solved by the DE on CUDA and the performance and quality of obtained results is compared to the results of DE run on the CPU. Next, the proposed DE implementation is used to find minimum of a benchmark function and by that indirectly compared to a previous DE model for the CUDA architecture.

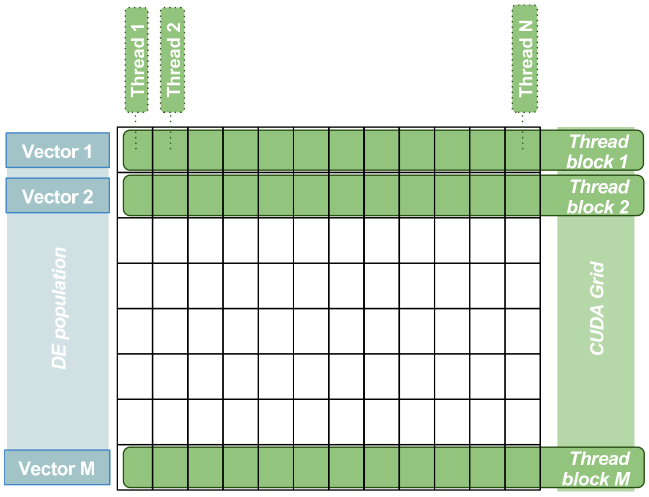


Figure 3: The mapping of CUDA threads and thread blocks to DE population elements.

4.1 Differential Evolution for Task Scheduling Optimization

In grid and distributed computing, mixed-machine heterogeneous computing (HC) environments utilize a distributed suite of different machines to perform different computationally intensive applications that have diverse requirements [1, 3]. Task scheduling, i.e. mapping of a set of tasks to a set of resources, is required to exploit the different capabilities of a set of heterogeneous resources. It is known, that an optimal mapping of computational tasks to available machines in a HC suite is a NP-complete problem [7] and as such, it is a subject to various heuristic [3, 12, 9] and meta – heuristic [20, 22, 15, 4] algorithms including the differential evolution [10].

An HC environment is a composite of computing resources (PCs, clusters, or supercomputers). Let $T = \{T_1, T_2, \dots, T_n\}$ denote the set of tasks that is in a specific time interval submitted to a resource management system (RMS). Assume the tasks are independent of each other with no inter-task data dependencies and preemption is not allowed (the tasks cannot change the resource they have been assigned to). Also assume at the time of receiving these tasks by RMS, m machines $M = \{M_1, M_2, \dots, M_m\}$ are within the HC environment. For our purpose, 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 in HC environment can estimate how much time is required to perform each task. In [3] 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. Assume that C_{ij} ($j = 1, 2, \dots, n, i = 1, 2, \dots, m$) is the completion time for performing j-th task in i-th machine and W_i ($i = 1, 2, \dots, m$) is the previous workload of M_i, then $\sum_{j \in S(i)} C_{ij} + W_i$ is the time required for M_i to complete the tasks included in it ($S(i)$ is the set of jobs scheduled for execution on M_i in schedule S). According to the aforementioned definition, makespan and flowtime can be evaluated using:

$$makespan = \max_{i \in \{1,2,\dots,m\}} \{ \sum_{j \in S(i)} C_{ij} + W_i \} \tag{3}$$

$$flowtime = \sum_{i=1}^{m} \sum_{j \in S(i)} C_{ij} \tag{4}$$

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. Since the differential evolution uses for problem encoding real vectors, real coordinates must be used instead of discrete machine numbers. The real-encoded DE vector is translated to schedule representation by simple truncation of its coordinates (e.g. $3.6 \rightarrow 3$, $1.2 \rightarrow 1$). Assume schedule S from the set of all possible schedules $Sched$. For the purpose of differential evolution, we define a fitness function $fit(S) : Sched \rightarrow \mathbb{R}$ that evaluates each schedule:

$$fit(S) = \lambda \cdot makespan(S) + (1 - \lambda) \cdot \frac{flowtime(S)}{m} \tag{5}$$

The function $fit(S)$ 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 $fit(S)$ is parameterized by the variable λ. The same schedule evaluation was used also in [4].

4.1.1 Scheduling Experiments

We have implemented the DE for scheduling of independent tasks on the CUDA platform to evaluate the performance and quality of proposed solution. The GPU implementation was compared to a simple CPU implementation (high level object oriented C++ code) and optimized CPU implementation (low level C code to achieve maximum performance). The optimized CPU implementation was created to provide a fair comparison of performance oriented implementations on the GPU and on the CPU. Optimized CPU and GPU implementations of the DE for scheduling optimization were identical with the exception of CUDA-C language constructions.

First, the time needed to compute fitness for the population of DE vectors was measured for all three DE implementations. The experiment was conducted on a computing node with 2 dual core AMD Opteron processors at 2.6GHz and nVidia Tesla C2050 with 448 cores at 1.15GHz. The comparison of fitness computation times on for different population sizes is illustrated in 4 (note the log scale of the y-axis).

The GPU implementation was 25.2 - 216.5 times faster than CPU implementation and 2.2 - 12.5 times faster than optimized CPU implementation of the same algorithm. This, along with the speedup achieved by the parallel implementation of the DE process contributes to the overall improvement of the optimization results. To compare the performance of the DE for minimizing the makespan and flowtime of a schedule of independent tasks was used the benchmark proposed in [3]. The benchmarking 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 the following properties [3]:

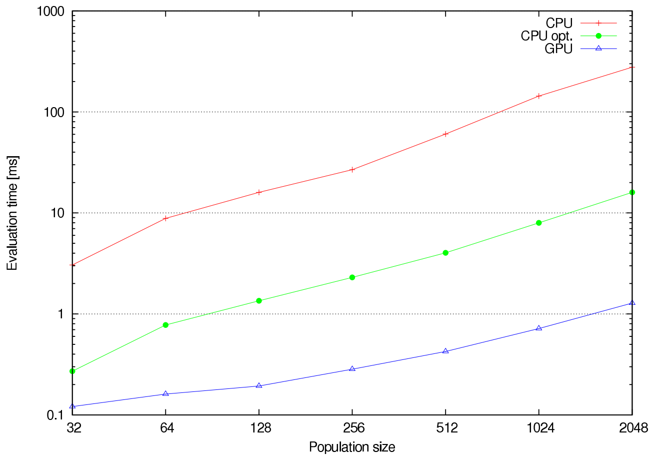


Figure 4: Comparison of schedule evaluation time on CPU and GPU.

- *task heterogeneity* – V_{task} represents the amount of variance among the execution times of tasks for a given machine
- *machine heterogeneity* – $V_{machine}$ represents 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
- *inconsistency* – machine M_j may be faster than machine M_k for some tasks and slower for others

Each ETC matrix was named using the pattern $TxMyCz$, where x describes task heterogeneity (*high* or *low*), y describes machine heterogeneity (*high* or *low*) and z describes the type of consistency (*i*ncosnsistent, *c*onsistent or *s*emi-consistent).

We have investigated speed and quality of the results obtained by the proposed DE implementation and compared it to the results obtained by CPU implementations. Average fitness value of the best schedules found by different DE variants after 30 seconds are listed in Table 1. The best results for each ETC matrix are shown in bold. We can see that the GPU implementation delivered the best results for population sizes 1024 and 512. However, the most successful population size was 64. Apparently, such a population size seems to be suitable for investigated scheduling problem with given dimensions (i.e. number of jobs and number of machines). When executing the differential evolution with population size 64, the optimized CPU implementation delivered best results for the consistent ETC matrices, i.e. ThMhCc, ThMlCc, TlMhCc and TlMlCc. In all other cases, the best result was found by the GPU powered differential evolution.

The progress of the DE with the most successful population size 64 for different ETC matrices is shown in Figure 5. The figure clearly illustrates the big difference in the DE on CPU and GPU. The DE executed on the GPU achieves the most significant fitness improvement during the first few seconds (roughly 5s) while the CPU implementations require much more time to deliver solution with similar quality, if they manage to do it at all. Needless to say, the optimized CPU implementation has always found better solution than the simple CPU optimization because it managed to process more candidate vectors in the same time frame.

4.2 Search for Function Minima

Previous GPU based DE implementations were most often benchmarked using a set of continuous benchmarking functions. To provide a rough comparison of the proposed approach with another DE implementation, we have implemented a DE searching for minimum of the test function f_2 from [5].

$$f_2(x) = \sum_{i-1}^{n} \left(x_i^2 - 10cos(2\pi x_i) + 10\right) \tag{6}$$

We note that the comparison is indirect and rather illustrative since the two algorithms were executed for the same test function but with different dimensions, on different hardware, and with different settings.

The purpose of this comparison is to show whether the proposed DE can find similar, better, or worse solution of a previously used benchmark function of many variables. From Table 2 can be seen that the proposed DE has found

Table 2: Comparison of proposed DE with CUDA-C implementation by de Veronese and Krohling.

f_2	DE from [5]		Proposed DE	
variables	value	time	value	time
100/128	278.18	0.64	232.8668	0.6604656
100/128	98.53	27.47	32.98331	27.00045
256	N/A	N/A	91.10914	27.00054
512	N/A	N/A	295.6335	27.00055

in a very similar time frame a solution to f_2 with better (i.e. lower) value. The proposed algorithm was executed on a more powerful GPU than the DE in [5], but it had to browse a search solution space with 1.28 times higher dimension. In 0.64 seconds, it delivered approx. 1.19 times better (in terms of fitness value) solution and in 27 seconds, it has found approximately 3 times better solution than the previous implementation. Moreover, the proposed DE has found in 27 seconds better solution even for the benchmark function of 256 variables.

This leads us to the conclusion that the proposed DE is able to find good minima of continuous functions and it appears to be at least competitive with previous CUDA-C implementations.

5. CONCLUSIONS

This paper introduces a new GPU implementation of the differential evolution. The algorithm was designed for the nVidia CUDA platform and in contrast to previous GPU implementations of the DE, it processes each candidate solution with many threads. It implements all DE related operations including random number generation on the GPU and it uses the latest features of the nVidia CUDA SDK 3.2.

The GPU implementation of differential evolution was used to find good schedules mapping a set of independent tasks to multiple machines. With the help of the GPU, the fitness of the schedules was evaluated 2.2 - 12.5 faster than on the CPU using C code and 25.2 - 216.5 times faster than on the CPU using object oriented C++ code. The implementation uses many threads to process each candidate solution and so a further performance increase is expected with more advanced GPUs.

In a direct comparison with CPU based implementations was shown that the differential evolution implemented on the CUDA platform can in most cases find schedules with better average fitness in long time (30 seconds). More importantly, the CUDA based DE can find significantly better solutions in a short time (below 5 seconds), which makes the algorithm and implementation prospective for real world usage.

In an indirect comparison with previously published GPU based DE implementation, the proposed DE delivered competitive results when seeking for an optimum of a test function, which suggests that the presented DE implementation on the CUDA platform can deliver good results also for problems with continuous solution spaces.

Acknowledgement

This work was supported by the Czech Science Foundation, under the grant no. GA102/09/1494, and by the Ministry of Industry and Trade of the Czech Republic, under the grant no. FR-TI1/420.

6. REFERENCES

[1] S. Ali, T. Braun, H. Siegel, and A. Maciejewski. Heterogeneous computing. In J. Urbana and P. Dasgupta, editors, *Encyclopedia of Distributed Computing*. Kluwer Academic Publishers, Norwell, MA, 2002.

[2] M. Andrecut. Fast gpu implementation of sparse signal recovery from random projections. *Engineering Letters*, 17(3):151–158, 2009.

[3] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D.

Table 1: The fitness of best schedule found in 30 sec using different population sizes (lower is better).

ETC	Population size = 64			Population size = 512			Population size = 1024		
matrix	CPU	CPU opt.	GPU	CPU	CPU opt.	GPU	CPU	CPU opt.	GPU
ThMhCc	1.07E+7	**9.03E+6**	9.57E+6	2.08E+7	1.69E+7	**9.46E+6**	2.42E+7	1.92E+7	**9.35E+6**
ThMhCi	6.60E+6	3.72E+6	**3.18E+6**	1.92E+7	1.77E+7	**3.19E+6**	2.18E+7	1.96E+7	**3.29E+6**
ThMhCs	7.48E+6	4.89E+6	**4.27E+6**	2.02E+7	1.73E+7	**4.24E+6**	2.37E+7	1.99E+7	**4.43E+6**
ThMlCc	194841	**180070**	186913	240260	206585	**188508**	269340	233054	**187939**
ThMlCi	118491	88383.6	**78645.4**	233159	213770	**78905.1**	251670	235113	**80649.6**
ThMlCs	141940	111729	**104012**	233885	205696	**104898**	257279	228244	**108694**
TlMhCc	361021	**322400**	334667	693637	564866	**328479**	787541	666462	**325734**
TlMhCi	219874	123442	**104475**	683699	579198	**104597**	728971	670957	**107532**
TlMhCs	243946	158307	**142704**	644251	567544	**143857**	769772	638295	**149150**
TlMlCc	6387.09	**5908.12**	6185.68	7647.84	6896.78	**6148.65**	9035.75	7804.94	**6155.41**
TlMlCi	3883.62	2813.56	**2540.56**	7882.03	7070.03	**2549.5**	8349	7685.84	**2619.71**
TlMlCs	4640.97	3697.94	**3388.26**	7657.22	6726.06	**3418.79**	8471.38	7669.97	**3581.62**

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.*, 61:810–837, June 2001.

[4] J. Carretero, F. Xhafa, and A. Abraham. Genetic algorithm based schedulers for grid computing systems. *International Journal of Innovative Computing, Information and Control*, 3(7), 2007.

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

[6] 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*, pages 1–8. IEEE, 2010.

[7] D. Fernandez-Baca. Allocating modules to processors in a distributed system. *IEEE Trans. Softw. Eng.*, 15(11):1427–1436, 1989.

[8] 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*, pages 1 –7, 2008.

[9] 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*, volume 1, pages 8 –12, april 2009.

[10] P. Kromer, V. Snasel, J. Platos, A. Abraham, and H. Ezakian. Evolving schedules of independent tasks by differential evolution. In S. Caballé, F. Xhafa, and A. Abraham, editors, *Intelligent Networking, Collaborative Systems and Applications*, volume 329 of *Studies in Computational Intelligence*, pages 79–94. Springer Berlin / Heidelberg, 2011.

[11] W. Langdon and W. Banzhaf. A simd interpreter for genetic programming on gpu graphics cards. In M. O'Neill, L. Vanneschi, S. Gustafson, A. Esparcia Alcázar, I. De Falco, A. Della Cioppa, and E. Tarantino, editors, *Genetic Programming*, volume 4971 of *Lecture Notes in Computer Science*, pages 73–85. Springer Berlin / Heidelberg, 2008.

[12] 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*, volume 6, pages 3093–3098, aug. 2007.

[13] NVIDIA. *NVIDIA CUDA Programming Guide 3.2*. 2010.

[14] NVIDIA. *CUDA Toolkit 3.2 Math Library Performance*. 2011.

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

[16] D. Patnaik, S. P. Ponce, Y. Cao, and N. Ramakrishnan. Accelerator-oriented algorithm transformation for temporal data mining. *Network and Parallel Computing Workshops, IFIP International Conference on*, 0:93–100, 2009.

[17] P. Pospíchal, J. Jaroš, and J. Schwarz. Parallel genetic algorithm on the cuda architecture. In *Applications of Evolutionary Computation*, LNCS 6024, pages 442–451. Springer Verlag, 2010.

[18] T. Preis, P. Virnau, W. Paul, and J. J. Schneider. Accelerated fluctuation analysis by graphic cards and complex pattern formation in Econophysics. *New J. Phys.*, 11:093024, 2009.

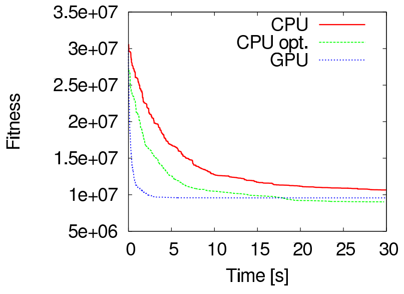
[19] K. V. Price, R. M. Storn, and J. A. Lampinen. *Differential Evolution A Practical Approach to Global Optimization*. Natural Computing Series. Springer-Verlag, Berlin, Germany, 2005.

[20] 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.

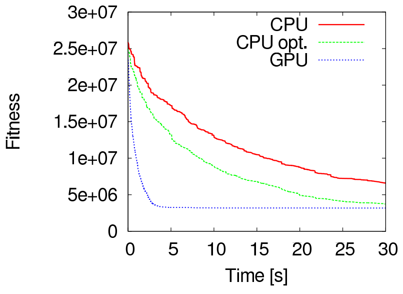
[21] D. Robilliard, V. Marion, and C. Fonlupt. High performance genetic programming on gpu. In *Proceedings of the 2009 workshop on Bio-inspired algorithms for distributed systems*, BADS '09, pages 85–94, New York, NY, USA, 2009. ACM.

[22] 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*, pages 232–242, London, UK, 2002. Springer-Verlag.

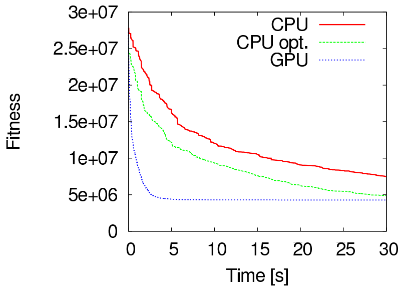
[23] W. Zhu. Massively parallel differential evolution -



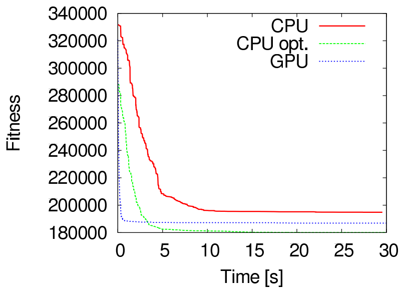
(a) ThMhCc



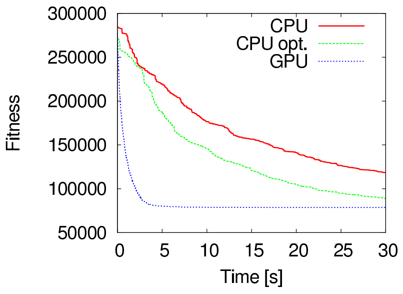
(b) ThMhCi



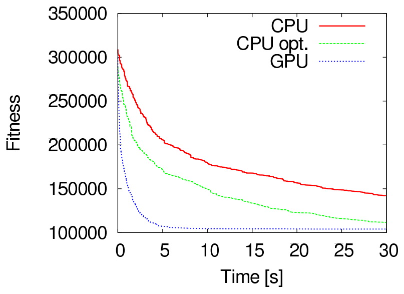
(c) ThMhCs



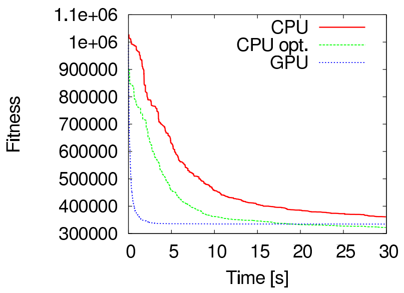
(d) ThMlCc



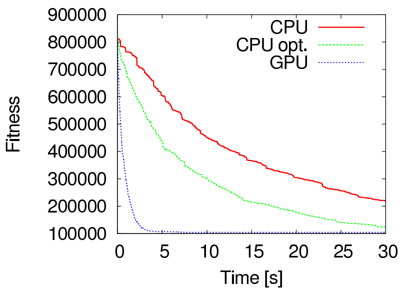
(e) ThMlCi



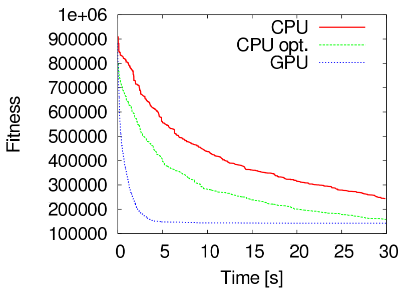
(f) ThMlCs



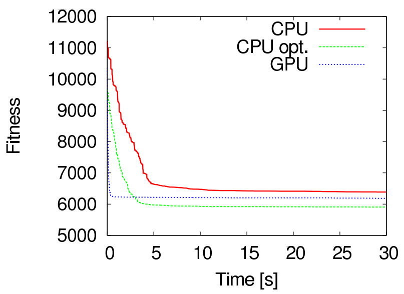
(g) TlMhCc



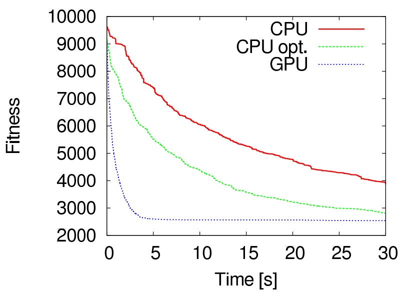
(h) TlMhCi



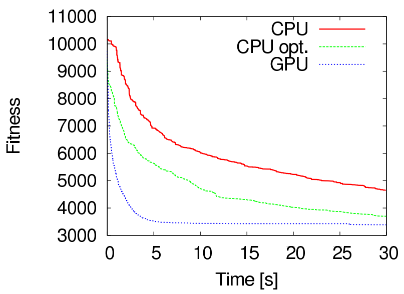
(i) TlMhCs



(j) TlMlCc



(k) TlMlCi



(l) TlMlCs

Figure 5: DE fitness improvement in time for different algorithms and different ETC matrices.

pattern search optimization with graphics hardware acceleration: an investigation on bound constrained optimization problems. *Journal of Global Optimization*, pages 1–21, 2010. 10.1007/s10898-010-9590-0.

[24] 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, pages 1–8, New York, NY, USA, 2010. ACM.