Many-threaded Differential Evolution on the GPU

Pavel Krömer, Jan Platoš, Václav Snášel, and Ajith Abraham

Abstract Differential evolution (DE) is an efficient populational meta-heuristic optimization algorithm that has been applied to many difficult real world problems. Due to the relative simplicity of its operations and real encoded data structures, it is very suitable for a parallel implementation on multicore systems and on the GPUs that nowadays reach peak performance of hundreds and thousands of giga FLOPS (floating-point operations per second). In this chapter, we present a simple yet highly parallel implementation of the differential evolution on the GPU using the CUDA architecture and demonstrate its performance on selected test problems.

1 Introduction

The differential evolution (DE) is a popular meta-heuristic optimization algorithm belonging to wide family of evolutionary algorithms (EAs). As many other evolutionary algorithms, it aims to solve the optimization problems by a simulated evolution of a population of candidate solutions. The population of candidates evolved by the algorithm performs a massively parallel search through the problem domain towards globally optimal solutions. The candidate solutions can be seen as individual points on the fitness landscape of the solved problem that are iteratively and in many directions at once moved towards promising regions on the fitness landscape. The implicit parallelism of the algorithm makes it even more interesting for an implementation on a truly parallel platform.

In this chapter, we present a fine-grained implementation of the DE designed specifically for the super parallel SIMD (single-instruction multiple-data) devices such as the GPUs. The SIMD hardware found in the modern day GPUs supports parallel execution of hundreds of threads at the same time and the software drivers

Pavel Krömer, Jan Platoš, Václav Snášel, and Ajith Abraham
IT4Innovations, VŠB - Technical University of Ostrava, 17. listopadu 12, Ostrava, Czech Republic
e-mail: {pavel.kromer, jan.platos, vaclav.snasel}@vsb.cz, ajith.abraham@ieee.org

and runtime libraries allow efficient scheduling of tens of thousands of threads. The general-purpose computing on graphics processing units (GPGPU) can use either commodity hardware such as the GPUs used primarily for computer graphics and entertainment, or GPU co-processors designed to perform massively parallel computations in the first place.

This chapter is organized in the following way: in Section 2, the basic principles of the differential evolution and some of its applications are presented. Section 3 gives a brief overview of the GPU computing, the CUDA platform, and recent implementations of the differential evolution on the GPUs. The many-threaded differential evolution is presented in Section 4 and its performance on selected test problems, first reported in [16] and [17], is described in Section 5.

2 Differential Evolution

The differential evolution (DE) is a versatile and easy to use stochastic evolutionary optimization algorithm [30]. It is a population-based optimizer that evolves a population of real encoded vectors representing the solutions to given problem. The DE was introduced by Storn and Price in 1995 [40, 39] and it quickly became a popular alternative to the more traditional types of evolutionary algorithms. It evolves a population of candidate solutions by iterative modification of candidate solutions by the application of the differential mutation and crossover [30]. In each iteration, so called trial vectors are created from current population by the differential mutation and further modified by various types of crossover operator. At the end, the trial vectors compete with existing candidate solutions for survival in the population.

2.1 The DE Algorithm

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 space. The latter initialization leads to better results of the optimization [30].

During the optimization, the DE generates new vectors that are scaled perturbations of existing population vectors. The algorithm perturbs selected base vectors with the scaled difference of two (or more) other population vectors in order to produce the trial vectors. The trial vectors compete with members of the current population with the same index called the target vectors. If a trial vector represents a better solution than the corresponding target vector, it takes its place in the population [30].

There are two most significant parameters of the DE [30]. The scaling factor $F \in [0, \infty]$ controls the rate at which the population evolves and the crossover probability $C \in [0, 1]$ determines the ratio of bits that are transferred to the trial vector from its opponent. The size of the population and the choice of operators are another important parameters of the optimization process.

The basic operations of the classic DE can be summarized using the following formulas [30]: the random initialization of the ith vector with N parameters is defined by

$$x_i[j] = rand(b_j^L, b_j^U), \quad j \in \{0, \ldots, N-1\} \tag{1}$$

where b_j^L is the lower bound of jth parameter, b_j^U is the upper bound of jth parameter and $rand(a,b)$ is a function generating a random number from the range $[a,b]$. A simple form of the differential mutation is given by

$$v_i^t = v_{r1} + F(v_{r2} - v_{r3}) \tag{2}$$

where F is the scaling factor and v_r^1, v_r^2 and v_r^3 are three random vectors from the population. The vector v_{r1} is the base vector, v_{r2} and v_{r3} are the difference vectors, and the ith vector in the population is the target vector. It is required that $i \neq r1 \neq r2 \neq r3$. The differential mutation in 2D (i.e. for $N = 2$) is illustrated in Fig. 1. The uniform crossover that combines the target vector with the trial vector is given by

$$l = rand(0, N-1) \tag{3}$$

$$v_i^t[m] = \begin{cases} v_i^t[m] & \text{if } (rand(0,1) < C) \text{ or } m = l \\ x_i[m] & \end{cases} \tag{4}$$

for each $m \in \{1, \ldots, N\}$. The uniform crossover replaces with probability $1 - C$ the parameters in v_i^t by the parameters from the target vector x_i. The outline of the classic

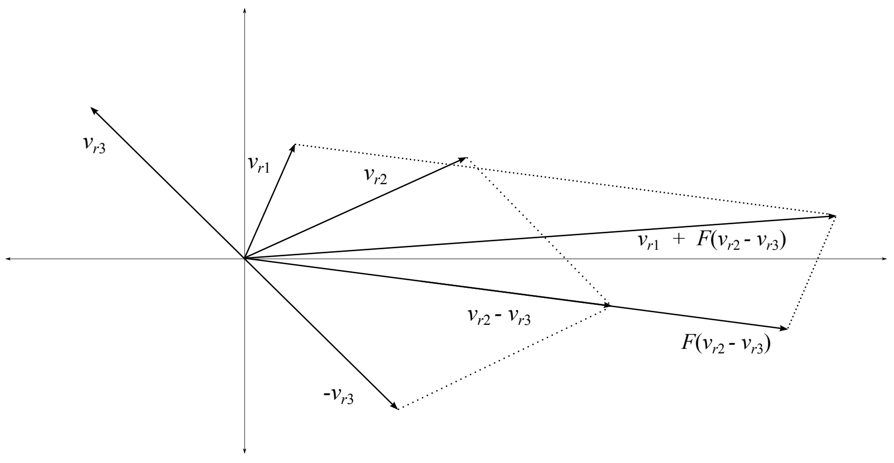


Fig. 1 The differential mutation.

DE according to [10] is summarized in Fig. 1. However, the monograph on DE by

```
1  Initialize the population $P$ consisting of $M$ vectors using eq. (1);
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          Differential mutation: Create trial vector $v_i^t$ according to eq. (2);
6          Validate the range of coordinates of $v_i^t$. Optionally adjust coordinates of $v_i^t$ so, that
           $v_i^t$ is valid solution to given problem;
7          Perform uniform crossover. Select randomly one parameter $l$ in $v_i^t$ and modify the
           trial vector using eq. (3);
8          Evaluate the trial vector. If the trial vector $v_i^t$ represent a better solution than
           population vector $v^i$, replace $v^i$ in $P$ by $v_i^t$;
9      end
10 end
```

Algorithm 1 A summary of classic Differential Evolution

Price, Storn, and Lampinen [30] lists a different version of the basic DE. They first form a whole new population of trial vectors P^t and subsequently merge P and P^t. It means that the newly created trial vectors do not enter the population of candidate solutions P immediately and therefore cannot participate in the creation of next trial vectors until the whole population was processed.

There are also many other modifications to the classic DE. Mostly, they differ in the implementation of particular DE steps such as the initialization strategy, the vector selection, the type of differential mutation, the recombination operator, and control parameter selection and usage [30, 10].

The initialization strategy affects the way vectors in the initial population are placed in the problem space. In general, a better initial coverage of the problem space represents a better starting point for the optimization process because the vectors can explore various regions of the fitness landscape from the very beginning [30].

The selection strategy defines how are the target vector, the base vector, and the difference vectors selected. Moreover, it has an effect on the time each vector survives in the population, which can be given either by the age of the vector or by the fitness of the vector. Popular base vector selection strategies are the random selection and methods based on stochastic universal sampling [30, 3]. The random selection without restrictions allows the same vector to be used as a base vector more than once in each generation. The methods based on stochastic universal sampling ensure that all vectors are used as base vectors exactly once in each generation. The selection methods based on the stochastic universal sampling generate a permutation of vector indexes that defines which base vector will be coupled with which target vector [30]. An alternative base vector selection strategy is called the biased base vector selection. The biased base vector selection uses the information about the fitness value of each vector when selectiong base vectors. The biased base vector selection strategies include the best-so-far base vector selection (the best vector in the population is always selected as base vector), target-to-best base vector selection (the base vector is an arithmetic recombination of the target vector and the best-so-far vector) [30]. The biased base vector selection schemes introduce a more intensive selection pressure which can, as in other evolutionary techniques, result in faster convergence but it can also lead to a loss of diversity in the population.

The differential mutation is the key driver of the DE. It creates a trial vector as a recombination of base vector and scaled difference of selected difference vectors. The scaling factor F, which modifies vector differences, can be a firmly set constant, a random variable selected according to some probability distribution, or defined by some other function as e.g. in the self-adaptive DE variants. A variable scaling factor increases the number of vector differentials that can be generated given the population P [30]. In general, a smaller scaling factor causes a smaller steps in the fitness landscape traversal while a greater scaling factor causes larger steps. The former leads to longer time for the algorithm to converge and the latter can cause the algorithm to miss the optima [30].

The recombination (crossover) operator plays a special role in the DE. It achieves a similar goal as the mutation operator in other evolutionary algorithms, i.e. it controls the introduction of new material to the population using a mechanism similar to the $n-$point crossover in the traditional EAs. The crossover probability $C \in [0,1]$ defines the probability that a parameter will be inherited from the trial vector. Similarly, $C-1$ is the probability that the parameter will be taken from the target vector. Crossover probability has also a direct influence on the diversity of the population [10]. An increased diversity initiated by larger C means a more intensive exploration and faster convergence of the algorithm at the cost of the robustness of the algorithm [30]. Common DE crossover operators include the exponential crossover and the uniform crossover. Some other crossover operators are e.g. the arithmetic crossover and the either-or-crossover [30, 10].

2.2 DE Variants and Applications

Particular DE variants are often referred to using a simple naming scheme [30, 10] that uses the pattern DE/x/y/z. In the pattern, x describes the base vector selection type, y represents the number of differentials, and z is the type of crossover operator. For example, the classic DE is often called DE/rand/1/bin, which means that it uses random base vector selection, a single vector differential, and uniform (binominal) crossover. Some other variants of the DE described in the literature are [10]:

1. *DE/best/1/**, which always select the so far best vector as the base vector and its differential mutation can be described by:

$$v_i^t = v_{best} + F(v_{r2} - v_{r3}) \tag{5}$$

   This DE variant can use any type of the crossover operator.

2. *DE/*/n_v/** that uses arbitrary base vector selection strategy, n_v differential vectors, and type of the crossover operator. Its differential mutation can be described by:

$$v_i^t = v_{r1} + F \sum_{k=1}^{n_v} (v_{r2}^k - v_{r3}^k) \tag{6}$$

where $v_{r2}^k - v_{r3}^k$ is the *k*th vector differential.
3. *DE/rand-to-best/n_v/** combines random base vector selection with the *best* base vector selection strategy:

$$v_i^t = \gamma v_{best} + (1-\gamma) v_{r1} + F \sum_{k=1}^{n_v} (v_{r2}^k - v_{r3}^k) \tag{7}$$

It uses the parameter $\gamma \in [0,1]$ to controls the exploitation of the mutation. The larger γ the larger the exploitation. The parameter γ can be also adaptive.
4. *DE/current-to-best/1+n_v/** uses at least two difference vectors. The first differential participating in the mutation is computed between the best vector and the base vactor and all the other differentials are computed using randomly selected vectors:

$$v_i^t = v_{r1} + F(v_{best} - v_{r1}) + F \sum_{k=1}^{n_v} (v_{r2}^k - v_{r3}^k) \tag{8}$$

The DE is a very successful algorithm with a number of applications. The DE was used, among others, for merit analysis, for non-imaging optical design, for the optimization of industrial compressor supply systems, for multi-sensor fusion,for the determination of earthquake hypocenters, for 3D medical image registration, for the design of erasure codes, for digital filters, for the analysis of X-ray reflectivity data, to solve the inverse fractal problem, and for the compensation of RF-driven plasmas [30]. It was also used for evolutionary clustering [8] and to optimize the deployment of sensor nodes [35].

Despite its continuous nature, the DE was used also to solve the combinatorial optimization problems. The applications of the DE in this domain included the turbo code interleaver optimization [19], the scheduling of independent tasks in heterogeneous computing environments [15, 20], the search for optimal solutions to the linear ordering problem [38] and many others.

The DE is a successful evolutionary algorithm designed for continuous parameter optimization driven by the idea of scaled vector differentials. That makes it an interesting alternative to the wide spread genetic algorithms that are designed to work primarily with discrete encoding of the candidate solutions. As well as genetic algorithms, it represents a highly parallel population based stochastic search metaheuristic. In contrast to the GA, the differential evolution uses the real encoding of candidate solutions 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 GPU Computing

Modern graphics hardware has gained an important role in the area of parallel computing. The GPUs 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 computing on graphics processing units (GPGPU) has been flourishing since then. The data parallel 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.

To simplify the development of GPGPU programs, various vendors have introduced languages, libraries, and tools to create parallel code rapidly. The GPU platform and API developed by the nVidia is called CUDA (Compute Unified Device Architecture). It is based on the CUDA-C language, which 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 execution and the relation of threads and thread groups to device memory is illustrated in Fig. 2. The GPU programming has established a new platform for the evolutionary computation [9]. Majority of the evolutionary algorithms including the genetic algorithms (GA) [29], the genetic programming (GP) [34, 21], and the DE [43, 41, 44] were implemented on the GPUs. Most of the contemporary implementations of the evolutionary algorithms on the GPUs map each candidate solution in the population to a single GPU thread. However, recent work in the area of evolutionary computation on the GPUs has introduced further parallelization of GA by e.g. many-threaded implementation of the crossover operator and local search [12].

3.1 Differential Evolution on the GPU

Due to the simplicity of its operations and real encoding of the candidate solutions, the DE is suitable for parallel implementation on the GPUs. In the DE, each candi-

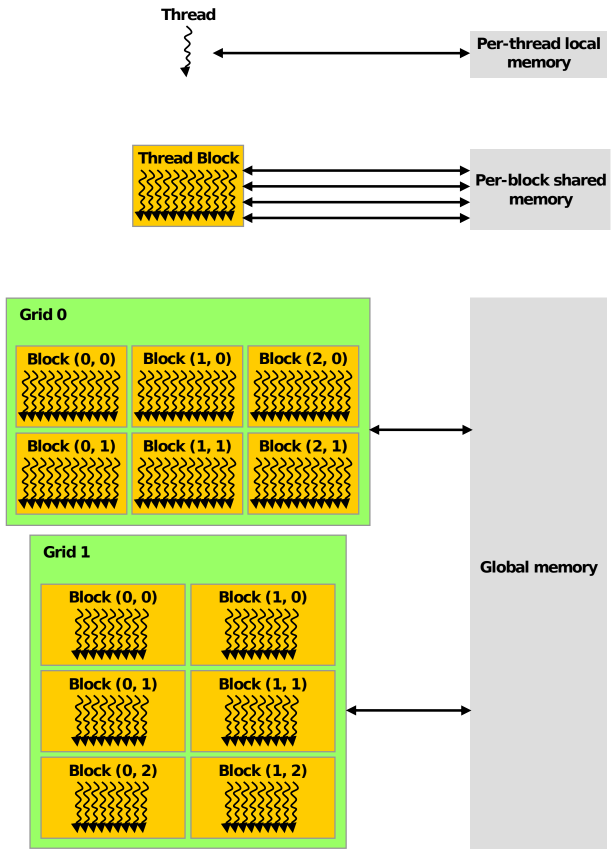


Fig. 2 CUDA-C program structure and memory hierarchy [26].

date solution is represented by a vector of real numbers (parameters) and the population as a whole can be seen as a real valued matrix. Moreover, both the mutation operator and the crossover operator can be implemented easily as straightforward vector operations.

The first implementation of the DE on the CUDA platform was introduced in the early 2010 by de Veronese and Krohling [41]. Their DE was implemented using the CUDA-C language and it achieved the speedup between 19 and 34 times comparing to the CPU implementation on a set of benchmarking functions. 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 [43], and Zhu and Li [44] implemented the DE on CUDA as a 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 cases, the performance of the algorithms was demonstrated on a set of continuous benchmarking functions.

The common property of the above DE implementations is the mapping of single GPU thread to one candidate solution and the usage of the Mersenne Twister from the CUDA SDK for random number generation on the GPU. Moreover, some parts of the random number generation process were [43] offloaded to the CPU. In this work, we use a new implementation of the DE on the CUDA platform using many threads to process each candidate solution and utilizing the GPU to generate random numbers needed for the optimization.

4 Many-threaded Differential Evolution on the GPU

The goal of the implementation of the DE on the CUDA platform was achieving high parallelism while retaining 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 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 Fig. 3.The kernels were implemented using the following principles:

1. Each candidate solution is processed by a thread block (thread group). The number of thread groups is in nVidia CUDA 4.0 limited to $(2^{16}-1)^3$ and in earlier versions to $(2^{16}-1)^2$. Hence, the maximum population size is in this case the same.
2. Each vector parameter 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 [26]. This limit enforces the maximum vector length. This is enough for the application area considered in this paper. The mapping of CUDA threads and thread blocks to the DE vectors is illustrated in Fig. 4.
3. Each kernel call aims to process the whole population in single step, e.g. it asks the CUDA runtime to launch M blocks with N threads in parallel. The CUDA 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 chromosomes 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.

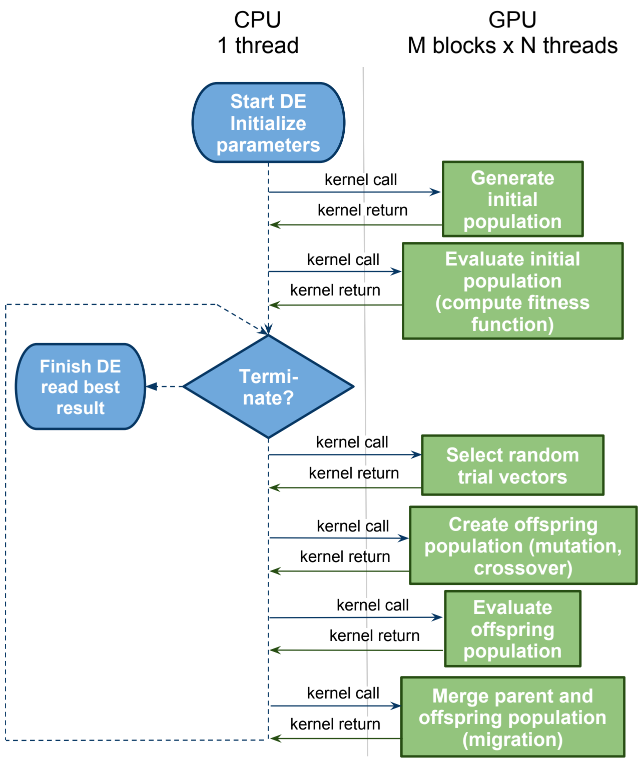

CPU
1 thread
GPU
M blocks x N threads
Start DE Initialize parameters
kernel call
kernel return
Generate initial population
kernel call
kernel return
Evaluate initial population (compute fitness function)
Finish DE read best result
Termi-nate?
kernel call
kernel return
Select random trial vectors
kernel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

Fig. 3 The flowchart of the DE implementation on CUDA.

5 The Performance of the Many-threaded DE on the GPU

The performance of the many-threaded DE was evaluated on several test problems. To perform the experiments, the *DE/rand/1/bin* type of the DE was implemented for both, the CUDA platform and sequential execution on the CPU. When not stated otherwise, the presented experiments were performed on a server with 2 dual core AMD Opteron processors at 2.6GHz and an nVidia Tesla C2050 with 448 cores at 1.15GHz.

In contrasts to the majority of DE applications, the many-threaded DE on the GPU was used to solve combinatorial optimization problems. However, to provide at least indirect comparison with previous DE implementations on CUDA, the optimization of the test function f_2 was performed.

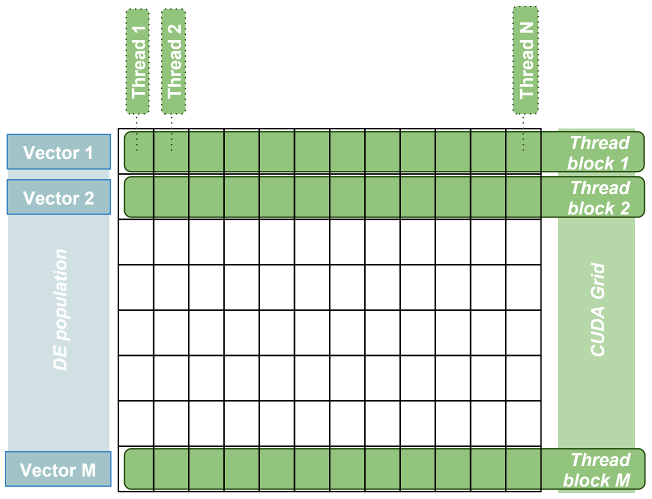


Fig. 4 The mapping of CUDA threads and thread blocks on DE population.

5.1 Function Optimization

The previous GPU based DE implementations were most often tested using a set of continuous benchmarking functions. To provide at least a rough comparison of our approach to an another DE variant, we have implemented a DE searching for minimum of the test function f_2 from [41]:

$$f_2(x) = \sum_{i-1}^{n} \left(x_i^2 - 10cos(2\pi x_i) + 10\right) \tag{9}$$

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 many-threaded DE can find similar, better, or worse solution of a previously used benchmark function. From Table 1, it can be seen that the many-threaded DE has found in a very similar time frame a solution to f_2 with better (i.e. lower) fitness value. The many-threaded DE was executed on a more powerful GPU than the DE in [41], but it had to solve a function with 1.28 times larger 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.

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

Table 1 Comparison of the many-threaded DE with CUDA-C implementation from [41].

f_2	DE from [41]		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

5.2 Linear Ordering Problem

The linear ordering problem (LOP) is a well known NP-hard combinatorial optimization problem. It has been intensively studied and there are plenty of exact, heuristic and meta-heuristic algorithms for LOP. With its large collection of well described testing data sets, the LOP represents an interesting testbed for meta-heuristic algorithms for combinatorial optimization [22, 23].

The LOP can be formulated as a graph problem [22]. For a complete directed graph $D_n = (V_n, A_n)$ with weighted arcs c_{ij}, compute a spanning acyclic tournament T in A_n such that $\sum_{(i,j)\in T} c_{ij}$ is as large as possible. The LOP can be also defined as a search for an optimal column and row reordering of a weight matrix C [36, 37, 6, 22]. Consider a matrix $C^{n\times n}$, permutation Π and a cost function f:

$$f(\Pi) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} c_{\Pi(i)\Pi(j)} \tag{10}$$

The LOP is a search for permutation Π so that $f(\Pi)$ is maximized, i.e. the permutation restructures the matrix C so that the sum of its elements above the main diagonal is maximized. The LOP is an NP-hard problem with a number of applications in scheduling (scheduling with constraints), graph theory, economy, sociology (paired comparison ranking), tournaments and archaeology among others.

In economics, LOP algorithms are deployed to triangularize input-output matrices. The resulting permutation provides useful information on stability of the investigated economy. In archaeology, LOP algorithms are used to process the Harris Matrix, a matrix describing most probable chronological ordering of samples found in different archaeological sites [37]. Other applications of LOP include the equivalent graph problem, the related graph problem, the aggregation of individual preferences, ranking in sport tournaments and e.g. the minimization of crossing [22].

A variant of the LOP is the linear ordering problem with cumulative costs (LOPCC) that has applications in the optimization of the universal mobile telecommunication standard (UMTS) in mobile-phone telecommunication systems [31, 4, 22].

5.2.1 LOP Data Sets

There are several test libraries used for benchmarking LOP algorithms. They are well preprocessed, thoroughly described and the optimal (or so far best) solutions are available. Majority of the investigated algorithms were tested against the LOLIB library. The original LOLIB library contains 49 instances of input-output matrices describing European economies in the 70s. Optimal solutions of the LOLIB matrices are available. Although the LOLIB contains real world data, it is considered rather simple and easy to solve [32]. Mitchell and Bochers [24] have published an artificial LOP data library and a LOP instance generator. The data (MBLB) and code are available from Rensselaer Polytechnic Institute[1].

Schiavinotto and Stützle [36, 37] have shown that the LOLIB and MBLB instances are significantly different, having diverse high-level characteristics of the matrix entries such as sparsity or skewness. The search space analysis revealed that MBLB instances typically have higher correlation length and also a generally larger fitness-distance correlation than LOLIB instances. It suggests that MBLB instances should be easier to solve than LOLIB instances of the same dimension. Moreover, a new set of large artificial LOP instances (based on LOLIB) called XLOLIB was created and published. Another set of LOP instances is known as the Stanford Graph Base (SGB). The SGB is composed of larger input-output matrices describing the US economies.

Many LOP libraries are hosted by the Optsicom project[2]. The Optsicom archive contains LOLIB, SGB, MBLB, XLOLIB and other LOP instances. One important feature of the Optsicom LOP archive is that its LOP matrices are normalized [22], i.e. they were preprocessed so that:

- all matrix entries are integral
- $c_{ii} = 0$ for all $i = \{1, 2, \ldots, n\}$
- $min\{c_{ij}, c_{ji}\} = 0$ for all $1 \leq i < j \leq n$

5.2.2 LOP Algorithms

There are several exact and heuristic algorithms for the linear ordering problem. The exact algorithms are strongly limited by the fact that LOP is a NP-hard problem (i.e. there are no exact algorithms that could solve LOP in polynomial time). Among the exact algorithms, branch & bound approach based on LP-relaxation of the LOP for the lower bound, a branch & cut algorithm and interior point/cutting plane algorithm attracted attention [37]. Exact algorithms are able to solve rather small general instances of the LOP and bigger instances (with the dimension of few hundred rows and columns) of certain classes of LOP [37].

A number of heuristic algorithms including the greedy algorithms, local search, elite tabu search, scattered search and iterated local search were used to solve the

[1] http://www.rpi.edu/ mitchj/generators/linord/

[2] http://heur.uv.es/optsicom/LOLIB/

LOP instances [37, 13, 22]. In this work, we use the LOP as a testbed for the performance evaluation of a many-threaded DE implementation powered by the GPU.

The LOP candidate solutions were for the purpose of the DE represented using the random keys encoding [2]. With this encoding, the candidate solution consists of an array of real numbers. The change in the order of elements of the array after sorting corresponds to a permutation. Due to its simplicity, the random keys encoding is a natural choice for differential evolution of permutations.

Computational experiments were performed on a server described in the beginning of this section. For comparison, the LOP was also computed on a laptop with Intel Core i5 at 2.3GHz. Although the CPUs used in the experiments were multicore, the LOP evaluation was single-threaded. The comparison of fitness computation times for different population sizes is illustrated in Fig. 5 (note the log scale). In the benchmark, a population of candidate solutions of a LOP instance with the dimension 50 was evaluated on the GPU and on two CPUs.

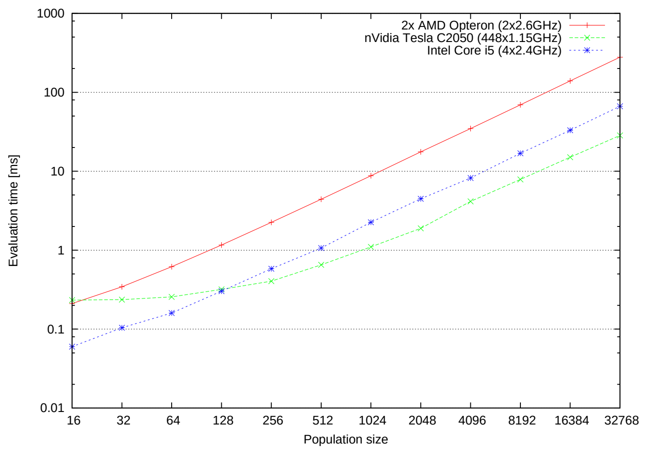


Fig. 5 The speed of LOP evaluation on the CPUs and on the GPU.

We can see that the Core i5 is always 3.3 - 4.2 times faster than the Opteron. The Tesla C2050 is is equally fast as the Opteron when evaluating 16 LOP candidates and 1.4 - 9.7 times faster for larger LOP candidate populations. The Core i5 is faster than GPU for population sizes 16, 32, and 64. The GPU performs similarly as Core i5 when evaluating 128 LOP candidates and 1.4 - 2.4 times faster for larger populations.

5.2.3 Search for LOLIB Solutions on the GPU

The LOLIB solutions obtained by the DE on CUDA and by a sequential DE implementation are shown in Table 2. The table contains best and average error after 5s and 10s computed from 10 independent optimization runs for each LOP matrix. We can see that the largest average error after 5s is 0.232 for matrix t69r11xx, which is better than the best LOP solution found by the DE in [38]. The average error for all LOLIB matrices was 0.043 after 5s and 0.031 after 10s.

Table 2 The average error of LOLIB solutions (in percent) .

	after 5 seconds		after 10 seconds			after 5 seconds		after 10 seconds	
LOP	best	avg	best	avg	LOP	best	avg	best	avg
be75eec	1.13E-03	2.00E-03	0	8.83E-03	t70k11xx	4.30E-04	2.16E-03	1.07E-02	1.19E-02
be75np	2.53E-04	5.18E-04	1.01E-03	2.72E-03	t70l11xx	0	0	0	0
be75oi	0	0	0	0	t70n11xx	0	0	0	0
be75tot	1.11E-01	1.12E-01	1.11E-01	1.15E-01	t70u11xx	1.12E-01	1.12E-01	1.35E-01	1.42E-01
stabu1	5.69E-03	7.79E-03	2.30E-02	2.31E-02	t70w11xx	2.21E-02	1.23E-01	3.52E-02	4.50E-02
stabu2	6.47E-02	6.73E-02	6.16E-02	6.21E-02	t70x11xx	8.51E-02	8.57E-02	6.94E-02	8.85E-02
stabu3	1.38E-01	1.44E-01	1.50E-01	1.84E-01	t74d11xx	0	0	1.44E-02	1.44E-02
t59b11xx	0	0	0	0	t75d11xx	7.26E-03	9.38E-03	1.45E-04	8.42E-04
t59d11xx	0	0	0	0	t75e11xx	2.16E-02	8.47E-02	2.10E-02	2.72E-02
t59f11xx	0	0	0	0	t75i11xx	2.28E-04	1.96E-03	2.38E-02	6.43E-02
t59i11xx	2.19E-03	4.97E-02	3.39E-03	4.07E-03	t75k11xx	0	0	0	0
t59n11xx	0	0	0	0	t75n11xx	0	0	0	0
t65b11xx	2.60E-02	2.21E-01	7.46E-02	8.55E-02	t75u11xx	6.77E-02	6.83E-02	7.62E-02	7.79E-02
t65d11xx	2.89E-02	2.95E-02	2.96E-02	3.53E-02	tiw56n54	7.09E-03	1.03E-02	7.09E-03	7.89E-03
t65f11xx	8.52E-02	9.04E-02	1.37E-01	1.37E-01	tiw56n58	1.94E-03	2.01E-03	1.94E-03	2.53E-03
t65i11xx	2.23E-02	2.28E-02	1.01E-02	1.54E-02	tiw56n62	0	0	0	0
t65l11xx	0	0	0	0	tiw56n66	0	1.46E-02	0	0
t65n11xx	0	0	0	0	tiw56n67	0	3.24E-04	0	0
t65w11xx	1.00E-01	1.05E-01	1.12E-01	1.43E-01	tiw56n72	3.02E-03	3.05E-03	2.16E-04	4.97E-04
t69r11xx	1.77E-01	2.32E-01	4.60E-02	4.65E-02	tiw56r54	5.49E-03	1.19E-02	7.06E-03	7.14E-03
t70b11xx	0	0	0	0	tiw56r58	1.87E-03	9.70E-03	0	1.24E-04
t70d11xn	6.43E-02	6.51E-02	2.51E-02	2.90E-02	tiw56r66	0	3.55E-03	0	0
t70d11xx	0	3.89E-02	0	2.00E-04	tiw56r67	0	0	0	0
t70f11xx	2.08E-01	2.15E-01	1.11E-02	3.08E-02	tiw56r72	0	0	0	0
t70i11xx	1.11E-01	1.63E-01	7.97E-02	8.32E-02					

We have compared the progress of DE for LOP on the GPU and on the CPU. A typical example of the optimization is shown in Fig. 6. Apparently, the DE on the GPU delivers optimal or nearly optimal results very quickly compared to the CPU.

Interestingly, the results of the optimization after 10s were sometimes worse than the results of optimization after 5s (the results were obtained in separate program runs). It suggests that the DE on CUDA quickly converges to a suboptimal solution but sometimes fails to find global optimum. On the other hand, the algorithm has found global optimum in all test runs for 15 out of 49 LOP matrices, which is a good result for a pure meta-heuristic.

5.2.4 Search for N-LOLIB Solutions

The DE on CUDA was also used to find solutions to the normalized LOLIB (N-LOLIB) library. The results of DE for LOP implemented on the GPU are shown in Table 3. The average error for all N-LOLIB matrices was 0.034 after 5s and

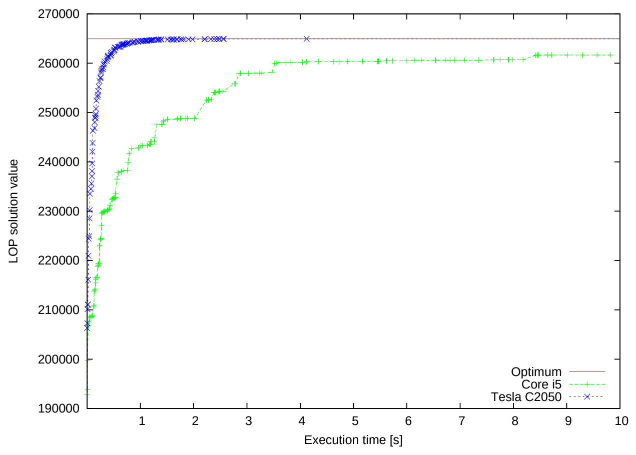


Fig. 6 Example of DE for LOLIB matrix be75eec on CPU and GPU.

0.037 after 10s. Moreover, the DE has found the optimal solution for 18 out of 50 LOP instances.

When we compare these results to N-LOLIB results obtained recently by several meta-heuristic methods in [22], we can see that the differential evolution performs better than pure genetic algorithms with average error 0.38 and optimal results found for 9 matrices. On the other hand, another meta-heuristics including tabu search, memetic algorithms, and simulated annealing performed better. However, we have to note that the DE used in this work is a pure meta-heuristic and uses no domain knowledge or local search to improve the solutions.

5.3 *Independent Task Scheduling*

In grid and distributed computing, the mixed-machine heterogeneous computing (HC) environments utilize a distributed suite of different machines to perform different computationally intensive applications that have diverse requirements [1, 5]. 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 an NP-complete problem [11] and as such, it is a subject to various heuristic [5, 25, 14] and meta-heuristic [33, 42, 28, 7] algorithms including the differential evolution [18].

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 pre-

Table 3 The average error of N-LOLIB solutions (in percent).

	after 5 seconds		after 10 seconds			after 5 seconds		after 10 seconds	
LOP	best	avg	best	avg	LOP	best	avg	best	avg
N-be75eec	0	1.10E-03	0	0	N-t70k11xx	3.79E-03	3.97E-03	6.26E-03	6.81E-03
N-be75np	3.49E-03	2.42E-02	2.79E-04	4.04E-04	N-t70l11xx	0	0	0	0
N-be75oi	0	1.80E-04	0	0	N-t70n11xx	0	0	0	0
N-be75tot	8.30E-02	1.33E-01	1.21E-01	1.22E-01	N-t70u11xx	1.39E-01	1.65E-01	1.40E-01	1.40E-01
N-stabu70	8.28E-04	1.60E-02	8.28E-04	3.56E-03	N-t70w11xx	1.96E-01	1.99E-01	1.59E-01	1.88E-01
N-stabu74	1.05E-01	1.06E-01	6.54E-02	1.02E-01	N-t70x11xx	6.76E-02	6.80E-02	3.13E-02	1.36E-01
N-stabu75	1.72E-01	2.27E-01	1.61E-01	1.61E-01	N-t74d11xx	0	0	0	0
N-t59b11xx	0	0	0	0	N-t75d11xx	0	1.73E-05	1.73E-04	3.54E-03
N-t59d11xx	0	0	0	0	N-t75e11xx	1.97E-02	2.30E-02	5.07E-03	1.36E-02
N-t59f11xx	0	0	0	0	N-t75i11xx	9.97E-03	1.09E-02	5.51E-05	3.86E-02
N-t59i11xx	6.05E-05	6.05E-05	6.05E-05	1.27E-03	N-t75k11xx	0	0	0	0
N-t59n11xx	0	0	0	0	N-t75n11xx	0	0	0	0
N-t65b11xx	3.08E-02	3.92E-02	1.34E-01	1.86E-01	N-t75u11xx	8.14E-02	8.16E-02	8.14E-02	8.78E-02
N-t65d11xx	7.99E-03	1.06E-02	3.45E-02	3.45E-02	N-tiw56n54	1.20E-02	1.69E-02	2.18E-03	7.10E-03
N-t65f11xx	1.61E-01	1.61E-01	1.61E-01	1.61E-01	N-tiw56n58	2.40E-03	2.48E-03	2.40E-03	2.40E-03
N-t65i11xx	2.51E-03	5.48E-03	1.82E-03	1.82E-03	N-tiw56n62	0	0	0	0
N-t65l11xx	0	0	0	0	N-tiw56n66	4.86E-03	6.31E-03	0	1.55E-02
N-t65n11xx	0	0	0	0	N-tiw56n67	0	3.58E-03	0	0
N-t65w11xx	3.04E-02	4.17E-02	1.60E-01	1.62E-01	N-tiw56n72	2.74E-04	6.30E-04	3.83E-03	3.83E-03
N-t69r11xx	6.90E-02	6.90E-02	5.16E-02	5.22E-02	N-tiw56r54	1.17E-02	1.69E-02	0	5.83E-04
N-t70b11xx	0	0	0	0	N-tiw56r58	2.32E-03	2.62E-03	0	2.08E-03
N-t70d11xxb	2.67E-02	3.17E-02	2.67E-02	6.97E-02	N-tiw56r66	1.24E-02	1.48E-02	0	8.59E-04
N-t70d11xx	0	1.55E-02	0	0	N-tiw56r67	0	0	0	0
N-t70f11xx	1.28E-02	1.28E-02	1.94E-03	1.17E-02	N-tiw56r72	0	0	0	0
N-t70i11xx	2.13E-02	8.01E-02	8.34E-02	8.66E-02	N-usa79	0.2.66E-01	2.86E-01	2.95E-02	3.42E-02

emption 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 [5], the expected time to compute (ETC) matrix was 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:

$$makespan = \min_{S \in Sched} \{ \max_{j \in Jobs} F_j \} \tag{11}$$

$$flowtime = \min_{S \in Sched} \{ \sum_{j \in Jobs} F_j \} \tag{12}$$

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:

$$makespan = \max_{i \in \{1,2,\dots,m\}} \{ \sum_{j \in S(i)} C_{ij} + W_i \} \tag{13}$$

$$flowtime = \sum_{i=1}^{m} \sum_{j \in S(i)} C_{ij} \tag{14}$$

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. $3.6 \to 3$, $1.2 \to 1$). Assume schedule S from the set of all possible schedules *Sched*. For the purpose of differential evolution, we define a fitness function $fit(S) : Sched \to \mathbb{R}$ that evaluates each schedule:

$$fit(S) = \lambda \cdot makespan(S) + (1 - \lambda) \cdot \frac{flowtime(S)}{m} \tag{15}$$

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 parametrized by the variable λ. The same schedule evaluation was used also in [7].

5.3.1 Search for Optimal Independent Task Schedules

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 the fitness for the population of DE vectors was measured for all three DE implementations. The comparison of fitness computation times on for different population sizes is illustrated in Fig. 7 (note the log scale of both axes).

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 operations contributes to the overall performance of the algorithm. To com-

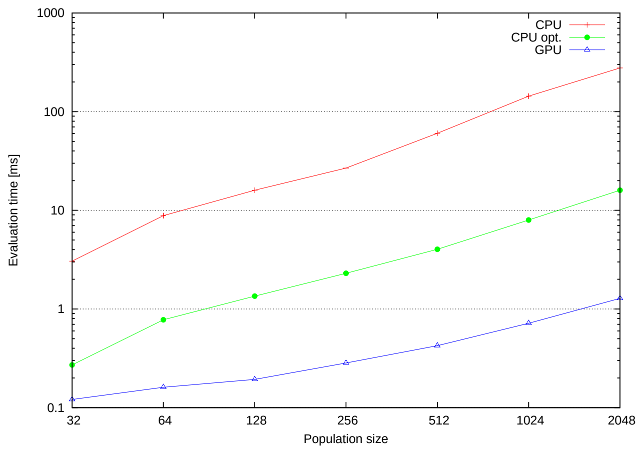


Fig. 7 Comparison of schedule evaluation time on CPU and GPU.

pare the GPU based and the CPU based DE implementations for the independent task scheduling, we have used the benchmark proposed in [5]. The benchmark contains several ETC matrices for 512 jobs and 16 machines. The matrices are labeled according to the following properties [5]:

- *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

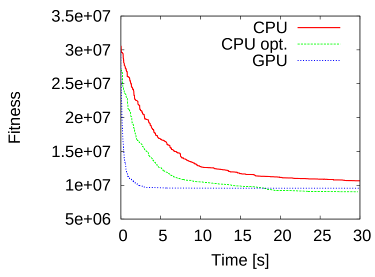
Each ETC matrix is named using the pattern TxMyCz, where x describes task heterogeneity (*h*igh or *l*ow), y describes machine heterogeneity (*h*igh or *l*ow) and z describes the type of consistency (*i*ncosnsistent, *c*onsistent or *s*emi-consistent).

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 4. 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.

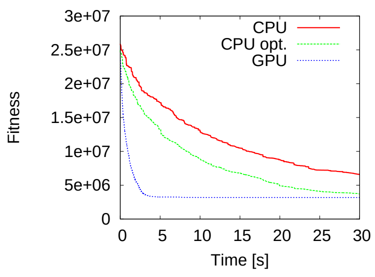
Table 4 The fitness of best schedule found in 30 sec using different population sizes (lower is better).

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**

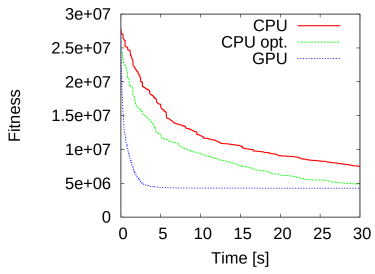
The progress of the DE with the most successful population size 64 for different ETC matrices is shown in Figs. 8 and 9. The figures clearly illustrate the big difference between the DE on CPU and GPU. The DE executed on the GPU achieves the most significant fitness improvement during the first few seconds (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.



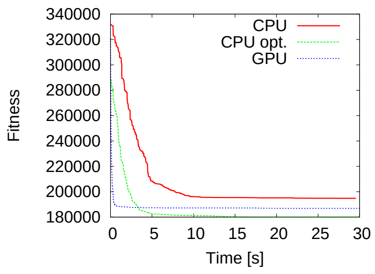
(a) ThMhCc



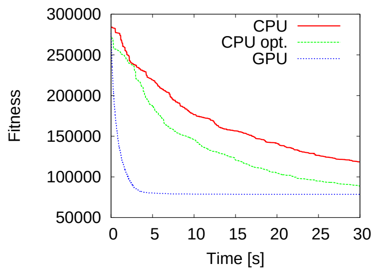
(b) ThMhCi



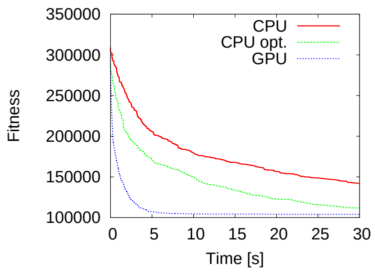
(c) ThMhCs



(d) ThMlCc



(e) ThMlCi



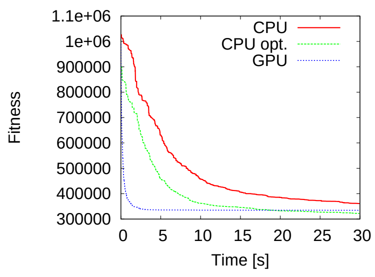
(f) ThMlCs

Fig. 8 Fitness improvement of different DE implementations for ThM*C* matrices.

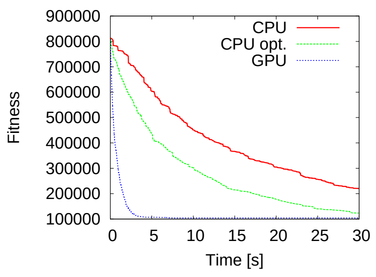
6 Conclusions

This chapter described the design and implementation of a fine grained DE on the GPUs. The basic steps of the algorithm were implemented with respect to the super parallel SIMD architecture of the GPUs allowing efficient parallel execution of hundreds of threads. The fine grained many-threaded DE design was chosen in order to maximize the utilization of the resources provided by the GPUs.

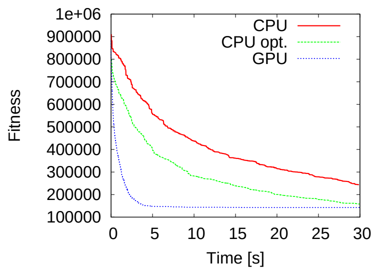
The performance of the solution was demonstrated on a series of computational experiments. The experimental evaluation involved continuous function optimiza-



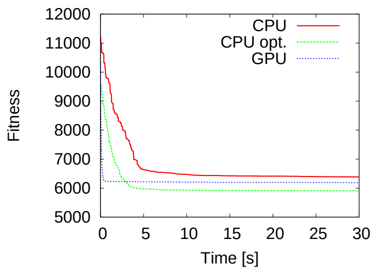
(a) TlMhCc



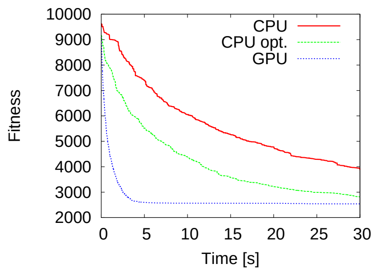
(b) TlMhCi



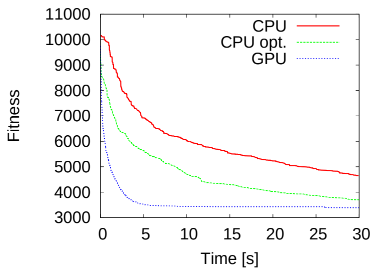
(c) TlMhCs



(d) TlMlCc



(e) TlMlCi



(f) TlMlCs

Fig. 9 Fitness improvement of different DE implementations for TlM*C* matrices.

tion to provide a rough comparison with a previous DE design for the GPU, and two popular combinatorial optimization problems with real world applications. The many-threaded DE implemented on the CUDA platform has shown good performance and good results in all test cases.

Acknowledgement

This work was supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070) and by the Bio-Inspired Methods: research, development and knowledge transfer project, reg. no. CZ.1.07/2.3.00/20.0073 funded by Operational Programme Education for Competitiveness, co-financed by ESF and state budget of the Czech Republic.

References

1. Ali, S., Braun, T., Siegel, H., Maciejewski, A.: Heterogeneous computing. In: J. Urbana, P. Dasgupta (eds.) Encyclopedia of Distributed Computing. Kluwer Academic Publishers, Norwell, MA (2002).
2. Ashlock, D.: Evolutionary computation for modeling and optimization. Springer (2005).
3. Baker, J.E.: Reducing bias and inefficiency in the selection algorithm. In: Genetic algorithms and their applications : proceedings of the second International Conference on Genetic Algorithms, pp. 14–21. L. Erlbaum Associates Inc., Hillsdale, NJ, USA (1987).
4. Bertacco, L., Brunetta, L., Fischetti, M.: The linear ordering problem with cumulative costs. European Journal of Operational Research **127**(3), 1345–1357 (2008).
5. Braun, T.D., Siegel, H.J., Beck, N., Bölöni, L.L., Maheswaran, M., Reuther, A.I., Robertson, J.P., Theys, M.D., Yao, B., Hensgen, D., Freund, R.F.: 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 (2001).
6. Campos, V., Glover, F., Laguna, M., Martí, R.: An experimental evaluation of a scatter search for the linear ordering problem. J. of Global Optimization **21**(4), 397–414 (2001).
7. Carretero, J., Xhafa, F., Abraham, A.: Genetic algorithm based schedulers for grid computing systems. International Journal of Innovative Computing, Information and Control **3**(7) (2007)
8. Das, S., Abraham, A., Konar, A.: Automatic hard clustering using improved differential evolution algorithm. In: Metaheuristic Clustering, *Studies in Computational Intelligence*, vol. 178, pp. 137–174. Springer Berlin / Heidelberg (2009).
9. Desell, T.J., Anderson, D.P., Magdon-Ismail, M., Newberg, H.J., Szymanski, B.K., Varela, C.A.: An analysis of massively distributed evolutionary algorithms. In: IEEE Congress on Evolutionary Computation, pp. 1–8. IEEE (2010)
10. Engelbrecht, A.: Computational Intelligence: An Introduction, 2nd Edition. Wiley, New York, NY, USA (2007)
11. Fernandez-Baca, D.: Allocating modules to processors in a distributed system. IEEE Trans. Softw. Eng. **15**(11), 1427–1436 (1989).
12. Fujimoto, N., Tsutsui, S.: A highly-parallel tsp solver for a gpu computing platform. In: I. Dimov, S. Dimova, N. Kolkovska (eds.) Numerical Methods and Applications, *Lecture Notes in Computer Science*, vol. 6046, pp. 264–271. Springer Berlin / Heidelberg (2011).
13. Huang, G., Lim, A.: Designing a hybrid genetic algorithm for the linear ordering problem. In: GECCO, pp. 1053–1064 (2003)
14. Izakian, H., Abraham, A., Snásel, V.: 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 (2009).
15. Krömer, P., Abraham, A., Snášel, V., Platoš, J., Izakian, H.: Differential evolution for scheduling independent tasks on heterogeneous distributed environments. In: Advances in Intelligent Web Mastering - 2, *Advances in Soft Computing*, vol. 67, pp. 127–134. Springer Berlin / Heidelberg (2010).

16. Krömer, P., Platoš, J., Snásel, V.: Differential evolution for the linear ordering problem implemented on cuda. In: A.E. Smith (ed.) Proceedings of the 2011 IEEE Congress on Evolutionary Computation, pp. 790–796. IEEE Computational Intelligence Society, IEEE Press, New Orleans, USA (2011)
17. Krömer, P., Snásel, V., Platoš, J., Abraham, A.: Many-threaded implementation of differential evolution for the cuda platform. In: N. Krasnogor, P.L. Lanzi (eds.) GECCO, pp. 1595–1602. ACM (2011)
18. Krömer, P., Snásel, V., Platoš, J., Abraham, A., Ezakian, H.: Evolving schedules of independent tasks by differential evolution. In: S. Caballé, F. Xhafa, A. Abraham (eds.) Intelligent Networking, Collaborative Systems and Applications, *Studies in Computational Intelligence*, vol. 329, pp. 79–94. Springer Berlin / Heidelberg (2011).
19. Krömer, P., Snášel, V., Platoš, J., Abraham, A.: Optimization of turbo codes by differential evolution and genetic algorithms. In: HIS '09: Proceedings of the 2009 Ninth International Conference on Hybrid Intelligent Systems, pp. 376–381. IEEE Computer Society, Washington, DC, USA (2009).
20. Krömer, P., Snášel, V., Platoš, J., Abraham, A., Izakian, H.: Scheduling independent tasks on heterogeneous distributed environments by differential evolution. In: Proceedings of the International Conference on Intelligent Networking and Collaborative Systems, INCOS '09. Barcelona, Spain, pp. 170–174. IEEE Computer Society (2009).
21. Langdon, W., Banzhaf, W.: 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, E. Tarantino (eds.) Genetic Programming, *Lecture Notes in Computer Science*, vol. 4971, pp. 73–85. Springer Berlin / Heidelberg (2008).
22. Martí, R., Reinelt, G.: The Linear Ordering Problem - Exact and Heuristic Methods in Combinatorial Optimization, *Applied Mathematical Sciences*, vol. 175. Springer Heidelberg Dordrecht London New York (2011).
23. Martí, R., Reinelt, G., Duarte, A.: A benchmark library and a comparison of heuristic methods for the linear ordering problem. Computational Optimization and Applications pp. 1–21 (2011).
24. Mitchell, J.E., Borchers, B.: Solving linear ordering problems with a combined interior point/simplex cutting plane algorithm. Tech. rep., Mathematical Sciences, Rensselaer Polytechnic Institute, Troy, NY 12180–3590 (1997).
25. Munir, E., Li, J.Z., Shi, S.F., Rasool, Q.: Performance analysis of task scheduling heuristics in grid. In: Machine Learning and Cybernetics, 2007 International Conference on, vol. 6, pp. 3093–3098 (2007).
26. NVIDIA: NVIDIA CUDA Programming Guide 3.2 (2010). URL http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf
27. NVIDIA: CUDA Toolkit 3.2 Math Library Performance (2011). URL http://developer.download.nvidia.com/compute/cuda/3_2/docs/CUDA_3.2_Math_Libraries_Performance.pdf
28. Page, A.J., Naughton, T.J.: Framework for task scheduling in heterogeneous distributed computing using genetic algorithms. Artificial Intelligence Review **24**, 137–146 (2004)
29. Pospíchal, P., Jaroš, J., Schwarz, J.: Parallel genetic algorithm on the cuda architecture. In: Applications of Evolutionary Computation, LNCS 6024, pp. 442–451. Springer Verlag (2010).
30. Price, K.V., Storn, R.M., Lampinen, J.A.: Differential Evolution A Practical Approach to Global Optimization. Natural Computing Series. Springer-Verlag, Berlin, Germany (2005).
31. Proakis, J.G.: Digital Communications, 4th edn. McGraw-Hill, New York (2001)
32. Reinelt, G.: The Linear Ordering Problem : Algorithms and Applications, *Research and Exposition in Mathematics*, vol. 8. Heldermann Verlag Berlin (1985)
33. Ritchie, G., Levine, J.: 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 (2004)

34. Robilliard, D., Marion, V., Fonlupt, C.: High performance genetic programming on gpu. In: Proceedings of the 2009 workshop on Bio-inspired algorithms for distributed systems, BADS '09, pp. 85–94. ACM, New York, NY, USA (2009).
35. Roy, S., Izlam, S.M., Ghosh, S., Das, S., Abraham, A., Krömer, P.: A modified differential evolution for autonomous deployment and localization of sensor nodes. In: N. Krasnogor, P.L. Lanzi (eds.) GECCO (Companion), pp. 235–236. ACM (2011)
36. Schiavinotto, T., Stützle, T.: Search space analysis for the linear ordering problem. In: G.R. Raidl, J.A. Meyer, M. Middendorf, S. Cagnoni, J.J.R. Cardalda, D. Corne, J. Gottlieb, A. Guillot, E. Hart, C.G. Johnson, E. Marchiori (eds.) Applications of Evolutionary Computing, *Lecture Notes in Computer Science*, vol. 2611, pp. 322–333. Springer-Verlag, Berlin, Germany (2003)
37. Schiavinotto, T., Stützle, T.: The linear ordering problem: Instances, search space analysis and algorithms. Journal of Mathematical Modelling and Algorithms **3**(4), 367–402 (2004)
38. Snášel, V., Krömer, P., Platoš, J.: Differential evolution and genetic algorithms for the linear ordering problem. In: J.D. Velásquez, S.A. Ríos, R.J. Howlett, L.C. Jain (eds.) KES (1), *Lecture Notes in Computer Science*, vol. 5711, pp. 139–146. Springer (2009)
39. Storn, R.: Differential evolution design of an IIR-filter. In: Proceeding of the IEEE Conference on Evolutionary Computation ICEC, pp. 268–273. IEEE Press (1996)
40. Storn, R., Price, K.: Differential Evolution- A Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces. Tech. rep. (1995).
41. de Veronese, L., Krohling, R.: Differential evolution algorithm on the gpu with c-cuda. In: Evolutionary Computation (CEC), 2010 IEEE Congress on, pp. 1 –7 (2010).
42. YarKhan, A., Dongarra, J.: Experiments with scheduling using simulated annealing in a grid environment. In: GRID '02: Proceedings of the Third International Workshop on Grid Computing, pp. 232–242. Springer-Verlag, London, UK (2002)
43. Zhu, W.: 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).
44. Zhu, W., Li, Y.: 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, pp. 1–8. ACM, New York, NY, USA (2010).