An Implementation of Differential Evolution for Independent Tasks Scheduling on GPU

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

Department of Computer Science
Faculty of Electrical Engineering and Computer Science
VŠB – Technical University of Ostrava
17. listopadu 15, 708 33 Ostrava – Poruba, Czech Republic
{pavel.kromer,jan.platos,vaclav.snasel}@vsb.cz

Abstract. Differential evolution is an efficient meta-heuristic optimization method with solid record of real world applications. In this paper, we present a simple and efficient implementation of the differential evolution using the massively parallel CUDA architecture. We demonstrate the speedup and improvements obtained by the parallelization of this intelligent algorithm on the problem of scheduling of independent tasks in heterogeneous environments.

1 Introduction

Nowadays, many algorithms are redesigned and rewritten for the GPU since modern GPUs introduce massive parallelism for a budget price and new APIs simplify the development of parallel applications. Among others, many successful intelligent algorithms including genetic algorithms and genetic programming were implemented on GPU. In this paper we introduce a simple parallelization of the differential evolution using the nVidia CUDA technology and demonstrate its performance and quality on the problem of scheduling of independent tasks in heterogeneous environments.

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 [1,2]. 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 an HC suite is a NP-complete problem [3] and as such, it is a subject to various heuristic [2,4,5] and meta-heuristic [6,7,8,9] algorithms. This work fits into the framework of hybrid intelligent systems [10,11]. The implementation of the intelligent algorithm – the differetial evolution – is effectively split betweeen the CPU and GPU in order to achieve more efficient and robust tool.

1.1 GPU Computing

Modern graphics hardware plays an important role in the area of parallel computing. Graphics cards have been used to accelerate gaming and 3D graphics applications, but recently, they have been used to accelerate computations for various topics, e.g. remote sensing, environmental monitoring, business forecasting, medical applications or physical simulations etc. Architecture of GPUs (Graphics Processing Unit) is suitable for vector and matrix algebra operations, which leads to a wide use of GPUs in the area of information retrieval, data mining, image processing, data compression, etc. Nowadays, the programmer does not need to be an expert in graphics hardware because of existence of various APIs (Application Programming Interface), which help programmers to implement their software faster. Nevertheless, it will be always necessary to follow basic rules of GPU programming to write a more efficient code.

Four main APIs exists today. The first two are vendor specific, i.e. they were developed by two main GPU producers - AMD/ATI and nVidia. The API developed by AMD/ATI is called ATI Stream and the API developed by nVidia is called nVidia CUDA (Compute Unified Device Architecture). Both APIs are able to provide similar results. The remaining two APIs are universal. The first one was designed by Khronos Group and it is called OpenCL (Open Computing Language) and the second was designed by Microsoft as a part of DirectX and it is called Direct Compute. All APIs provide a general purpose parallel computing architectures that leverage the parallel computation engine in graphics processing units.

The main advantage of GPU is its structure. Standard CPUs (central processing units) contain usually 1-4 complex computational cores, registers and large cache memory. 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 the data, thus each ALU of a multiprocessor executes the same operations on different data, lying in the registers. In contrast to standard CPUs which can reschedule operations (out-of-order execution), the selected GPU is an in-order architecture. This drawback is overcome by using multiple threads as described by Hager et al. [12]. 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 chip compensates this drawback.

2 Differential Evolution

Differential evolution (DE) is a versatile and easy to use stochastic evolutionary optimization algorithm [13]. 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 [13].

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 [13].

Differential evolution is parametrized by two parameters [13]. 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 bits 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 traditional DE is illustrated in 1.

Algorithm 1. A summary of basic differential evolution

```
1  Initialize population $P$ consisting of $M$ vectors;
2  Evaluate chromosomes in initial population using objective function;
3  while Termination criteria not satisfied do
4      for $i \in \{1 \dots M\}$ do
5          Mutation. Create a trial vector $v_t^i = v_r^1 + F(v_r^2 - v_r^3)$, where $F \in [0, 1]$ is
           a parameter and $v_r^1, v_r^2, v_r^3$ are three random vectors from the population
           $P$.
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
           $v_t^i$. With probability $1 - C$ let $v_t^i[m] = v^i[m]$ for each $m \in \{1, \dots, N\}$
           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
```

2.1 Differential Evolution for Scheduling Optimization

An HC environment is 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 intertask 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 [2] 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 (metatask execution time) and flowtime of a set of jobs can be defined as shown in Equation 1.

$$makespan = \min_{S\in Sched}\{\max_{j\in Jobs} F_j\}, \quad flowtime = \min_{S\in Sched}\{\sum_{j\in Jobs} F_j\}\ . \tag{1}$$

where *Sched* is the set of all possible schedules, *Jobs* stands for the set of all jobs and F_j represents the time in which job j finalizes. Assume that $[C]_{j,i}$ $(j = 1, 2, \dots, n, i = 1, 2, \dots, m)$ 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 $\sum(C_i + W_i)$ is the time required for M_i to complete the tasks included in it. According to the aforementioned definition, makespan and flowtime can be evaluated according to Equation 2.

$$makespan = \min_{i\in\{1,\dots,m\}}\{\sum C_i + W_i\}, \quad flowtime = \sum_{i=1}^{m} C_i\ . \tag{2}$$

Minimizing makespan aims to execute the whole metatask 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 truncation of its elements. 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 in Equation 3.

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

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 [9].

3 DE for the Scheduling of Independent Tasks Implemented on the GPU

The goal of the implementation of differential evolution on the CUDA architecture was achieving high paralelism while keeping the simplicity of the algorithm. The implementation consists of kernels (i.e. parallel methods) for generation of initial population (that is generation of pseudo random numbers), selection of random competitors for each vector in the opulation, DE processing including generation of trial vectors and crossover, validation of the range of coordinates of generated vectors, and the merger of parent and offspring populations. Besides these generic kernels implementing the DE process, an implementation of the fitness function evaluation was done in separate kernel. The overview of our DE implementation is shown in Figure 1.

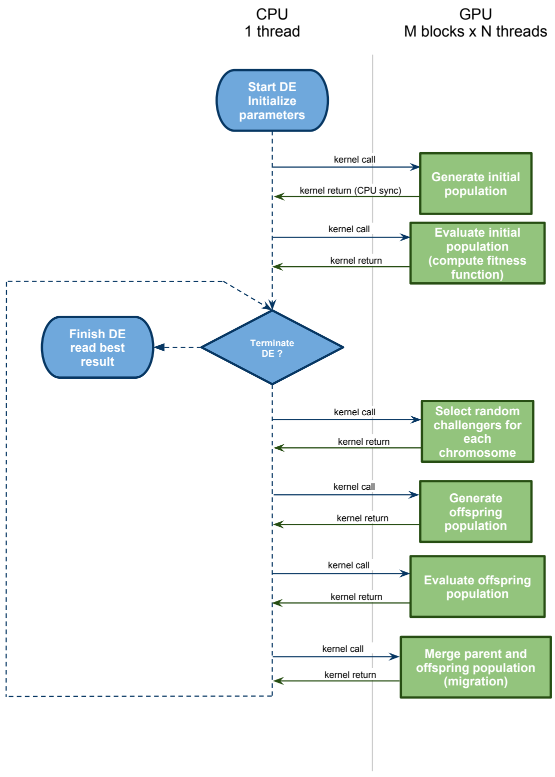

CPU
1 thread
GPU
M blocks x N threads
Start DE Initialize parameters
kernel call
kernel return (CPU sync)
Generate initial population
kernel call
kernel return
Evaluate initial population (compute fitness function)
Finish DE read best result
Terminate DE ?
kernel call
kernel return
Select random challengers for each chromosome
kernel call
kernel return
Generate offspring population
kernel call
kernel return
Evaluate offspring population
kernel call
kernel return
Merge parent and offspring population (migration)

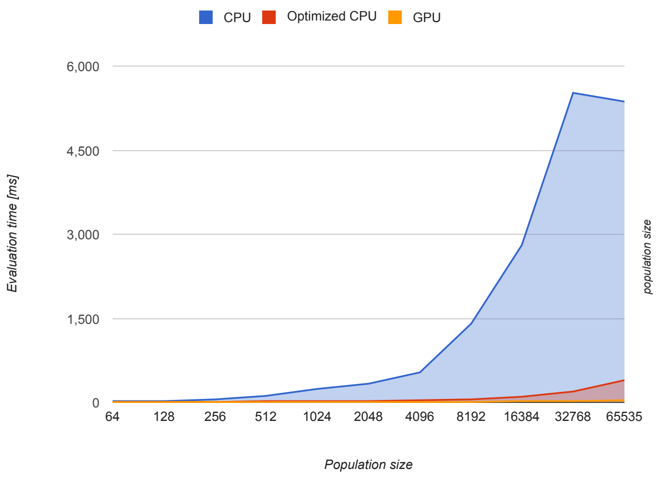
Fig. 1. The flowchart of the DE implementation on CUDA

4 Experiments

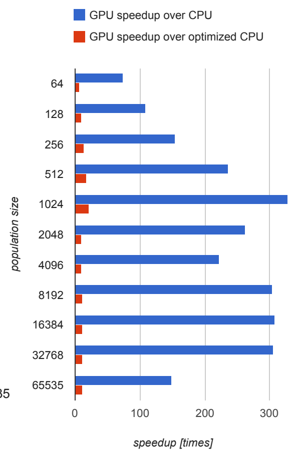
The kernels were implemented using the following principles: Each vector is processed by a thread group (set of threads sharing certain amount of memory, also known as block). The number of thread groups is in CUDA currently limited to $(2^{16} - 1)^2$ and hence the maximum population size is in this case the same. Each vector coordinate is processed by a thread. The limit of threads per block depends on the hardware and it can be 512 or 1024. This limit enforces the maximum vector length. For the scheduling with given ETC matrices, vectors with 512 coordinates are needed. 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 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.

We have implemented the DE for scheduling of independent tasks on the CUDA architecture to evaluate the performance and quality of such a solution. We have measured the speedup obtained by the implementation and compared it to a single threaded version of the algorithm. The comparison of fitness computation time and speedup on GPU for different population sizes is illustrated in 2a. We have compared a regular implementation on CPU (e.g. object oriented C++ code), optimized implementation on CPU (low level C code to achieve maximum performance) and GPU implementation that computed flowtime and makespan for the whole population of vectors. The experiment was conducted on a server



(a) Population evaluation time.



(b) GPU speedup.

Fig. 2. Population evaluation time on CPU and GPU for the scheduling problem

with 2 dual core AMD Opteron processors at 2.6GHz and nVidia Tesla C2050 with 448 cores at 1.15GHz

The GPU implementation was 73 - 327 times faster than CPU implementation and 6 - 21 times faster than optimized CPU implementation of the same algorithm. This, along with the speedup achieved by parallel implementation of the DE process contributes to the overall improvement of results.

Second, we have investigated the quality of results obtained by the DE and compared it to the results obtained by a single-thread CPU implementation. Average fitness value of optimized schedules for different configurations of the algorithm are shown in Table 1. The implementations are labeled as follows: type (CPU or GPU) / population size / limit of generations (thousands). We have used population sizes 1024 and 64 and the number of generations was selected so that the run time of all configurations was roughly the same. Other algorithm parameters were: $C = 0.8$, $F = 0.4$, and $T = 0.9$.

Table 1. Average fitness values of optimized schedules on CPU and GPU

ETC	Implementation				
Matrix	CPU/1024/300	CPU/64/300	GPU/1024/30	GPU/64/100	GPU/64/90
l-l-s	4026.055	4057.75	3964.43	**3720.595**	3746.435
l-l-i	2845.72	2832.415	2819.075	**2689.105**	2715.63
l-l-c	6807.95	6820.44	**6691.08**	6753.845	6760.52
l-h-s	180229	178137	174380.5	161464.5	**159097.5**
l-h-i	118625	116126.5	*116943.5*	**110959.5**	110964
l-h-c	412743	396145.5	389817.5	392084.5	**388914.5**
h-l-s	121705	124647	*122083.5*	112245	**112203.5**
h-l-i	86917.6	86305.25	86814.9	83255	**82760.8**
h-l-c	204818.5	205140.5	201239.5	**200608.5**	200650
h-h-s	5226555	5293475	5186865	4806145	**4794115**
h-h-i	3597320	3575925	*3584020*	**3370300**	3389225
h-h-c	11530150	11759800	11404550	11422600	**11357100**
time [sec]	31	31	29	34	31

In the table, the best average fitness is typeset in bold. In all cases, the schedule with best fitness was found by the GPU variant of the algorithm. Only in two cases, the fitness of schedules found by one of the CPU implementations was better than the GPU/1024/30 variant of the algorithm. Nevertheless, also in these cases the overall best schedule was found by an GPU implementation of the algorithm.

5 Conclusions

This paper introduces a GPU implementation of the differential evolution. The algorithm was described and the implementation on the nVidia CUDA platform was presented. The GPU implementation of differential evolution was used to find good schedules for the independent tasks scheduling problem. With the help of the GPU, the fitness function for this problem is evaluated 6 - 21 faster in one case and 73 - 327 times faster in another case. In a direct comparison with CPU based implementation was shown that the differential evolution on GPU can find schedules with better average fitness.

Acknowledgement

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

References

1. Ali, S., Braun, T., Siegel, H., Maciejewski, A.: Heterogeneous computing (2002)
2. Braun, T.D., Siegel, H.J., Beck, N., Boloni, 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 (2001)
3. Fernandez-Baca, D.: Allocating modules to processors in a distributed system. IEEE Trans. Softw. Eng. 15, 1427–1436 (1989)
4. Munir, E., Li, J.Z., Shi, S.F., Rasool, Q.: Performance analysis of task scheduling heuristics in grid. In: 2007 International Conference on Machine Learning and Cybernetics, vol. 6, pp. 3093–3098 (2007)
5. Izakian, H., Abraham, A., Snasel, V.: Comparison of heuristics for scheduling independent tasks on heterogeneous distributed environments. In: International Joint Conference on Computational Sciences and Optimization, CSO 2009, vol. 1, pp. 8–12 (2009)
6. 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)
7. YarKhan, A., Dongarra, J.: Experiments with scheduling using simulated annealing in a grid environment. In: Parashar, M. (ed.) GRID 2002. LNCS, vol. 2536, pp. 232–242. Springer, Heidelberg (2002)
8. Page, A.J., Naughton, T.J.: Framework for task scheduling in heterogeneous distributed computing using genetic algorithms. Artificial Intelligence Review 24, 137–146 (2004)
9. Carretero, J., Xhafa, F., Abraham, A.: Genetic algorithm based schedulers for grid computing systems. Int. Journal of Innovative Computing, Information and Control 3 (2007)
10. Abraham, A., Corchado, E., Corchado, J.M.: Editorial: Hybrid learning machines. Neurocomput. 72, 2729–2730 (2009)
11. Corchado, E., Abraham, A., de Carvalho, A.: Editorial: Hybrid intelligent algorithms and applications. Inf. Sci. 180, 2633–2634 (2010)
12. Hager, G., Zeiser, T., Wellein, G.: Data access optimizations for highly threaded multi-core cpus with multiple memory controllers. In: IEEE Int. Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–7 (2008)
13. Price, K.V., Storn, R.M., Lampinen, J.A.: Differential Evolution A Practical Approach to Global Optimization. Natural Computing Series. Springer, Berlin (2005)