PARALLEL GENETIC ALGORITHM SOLVING 0/1 KNAPSACK PROBLEM RUNNING ON THE GPU

Petr Pospichal, Josef Schwarz and Jiri Jaros

Brno University of Technology Faculty of Information Technology
Department of Computer Systems
Bozetechova 2, 612 66 Brno
Czech Republic
phone: +420-54114 1364 fax: +420-541141270
{ipospichal,jarosjir,schwarz}@fit.vutbr.cz

Abstract: *In this work, we show that consumer-level $100 GPU can be used to significantly speed-up optimization of 0/1 Knapsack problem. We identify strong and weak points of GPU architecture and propose our parallel genetic algorithm model implemented in CUDA running entirely on the GPU. We show that GPU must be utilized for sufficiently long time in order to obtain reasonable program speedup. Then we compare results quality and speed of our model with single-threaded CPU code implemented using Galib. Peak speedup of GPU GA execution performance is 1340x resp. 134x for 4-bit resp. 40-bit problem instances while maintaining reasonable results quality.*

Keywords: *Parallel genetic algorithm, PGA, CUDA, GPGPU, 0-1 Knapsack problem*

1 Introduction

Genetic Algorithms (GA) [2] are powerful, domain-independent search techniques inspired by Darwinian theory. In general, GAs employ selection, mutation, and crossover to generate new search points in a state space. A genetic algorithm starts with a set of individuals that forms a population of the algorithm. On every iteration of the algorithm, each individual is evaluated using the fitness function and the termination function is invoked to determine whether the termination criteria have been satisfied. The algorithm ends if an acceptable solutions have been found or the computational resources have been spent.

Although GAs are very effective in solving many practical problems, their execution time can become a limiting factor for some huge problems, because a lot of candidate solutions must be evaluated.

There is variety of possibilities how to accelerate GAs. One of the most promising variant is an island model parallelization. The island models can fully explore the computing power of course grain parallel computers. The population is divided into a few subpopulations, and each of them evolves separately on different processor. Island populations are free to converge toward different sub-optima. The migration operator is supposed to mix good features that emerge locally in the different subpopulations.

Driven by ever increasing requirements from the video game industry, GPUs have evolved into a very powerful and flexible processors, while their price remained in the range of consumer market. They now offer floating-point calculation much faster than today's CPU and, beyond graphics applications; they are very well suited to address general problems that can be expressed as data-parallel computations (i.e. the same code is executed on many different data elements).

We have designed efficient parallel genetic algorithm model running entirely on the GPU and used it previously for optimization of simple numerical functions [9]. In this work, we would like to introduce its capabilities for accelerating 0/1 Knapsack problem solution as well.

1.1 0/1 Knapsack Problem

In Knapsack problem, we have n kinds of items, 1 through n. Each kind of item i has a value v_i and a weight w_i. All values and weights are nonnegative. The maximum weight that we can carry in the bag is W. The goal is to maximize value of carried items. Formally defined:

$$\text{maximize} \quad \sum_{i=1}^{n} v_i x_i \tag{1}$$

$$\text{while subject to} \quad \sum_{i=1}^{n} w_i x_i \leqslant W, \qquad x_i \in \{0,1\} \tag{2}$$

The problem is NP-complete and often arises in resource allocation with financial constraints. Similar problem also appears in combinatorics, complexity theory, cryptography and applied mathematics.

In case of genetic algorithms, we need to maintain population of acceptable solution. This can be done using various heuristics or by fitness penalization [3]. We have chosen simple linear penalization defined in fitness function as:

$$fitness(x, v, w, W) = \sum_{i=1}^{n} v_i x_i - penalization(x, w, W) \tag{3}$$

$$penalization(x, w, W) = \begin{cases} 0 & \text{if } (\sum_{i=1}^{n} w_i x_i - W) \leq 0 \\ -10 \cdot (\sum_{i=1}^{n} w_i x_i - W) & \text{otherwise} \end{cases} \tag{4}$$

Note that our goal was not to find preferentially globally optimal Knapshack GA solutions but rather explore performance of the GPU platform.

1.2 Graphics Processing Units (GPUs)

Historically, GPUs have been used exclusively for fast rasterization of graphics primitives such as line, polygon and ellipse. These chips had strictly fixed functionality. Over time, growing gaming market and more complex games won GPUs limited programmable functionality. This turned out to be very beneficial, so their capabilities quickly developed up to milestone, unified shader units. This hardware and software model gave birth to nVidia Compute Unified Device Architecture (CUDA) framework, which is now often used for General Purpose Computation on the GPUs (GPGPU). We use this framework for our PGA implementation.

Main advantage of the GPU is very high raw floating point performance resulting from moderate degree of parallelism. Proper usage of this hardware can lead to speedup up to several hundred times compared to common CPU. But in order to utilize such power, programmer must consider a variety of differences:

- GPU requires massive parallelism in order do be fully utilized. Application must be thereby decomposable to thousands of relatively independent tasks.
- GPU is optimised for SIMD type processing, meaning that target application must be data parallel or the performance is significantly decreased.
- Graphics card is connected to host system via PCI-express bus, which is, compared to GPU memory, very slow (80x for GTX285). OS driver transfer overhead is also very performance-choking for small tasks. So application should minimize number of data transfers between CPU and GPU. The GPU must also be utilized for sufficiently long time in order to obtain meaningful speedup.
- Properly designed application should take into account memory architecture as well. Main GPU memory is up to 500x slower compared to fast, but small onchip shared memory. This limitation is reduced in new, Fermi [6] GPU architecture.
- GPU is optimized for `float` data type, `double` is usually very slow. `short` offers often better performance than `int`.

Whole GPGPU concept is shown on figure 1. More details about the GPU architecture and CUDA application programming can be found in [1].

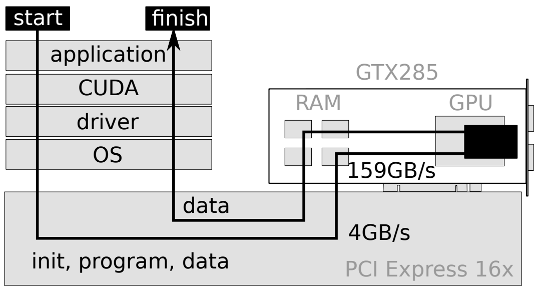


Figure 1: GPGPU datapath

2 Proposed Model: Parallel Genetic Algorithm Running Entirely on the GPU

NVidia GPU consists of multiple SIMD engines called multiprocessors. Each multiprocessor has number of FP processors with single instruction pointer, therefore it can perform one operation over multiple variables in parallel[1]. Additionally, these processors can be synchronized in order do maintain data consistency and they also contain small, but very fast shared memory. On the other hand, multiprocessors themselves operate independently and the only way how to exchange data between them is by using slow, main GPU memory. Therefore one multiprocessor can be effectively used for single parallel subtask such as island simulation in case of genetic algorithms.

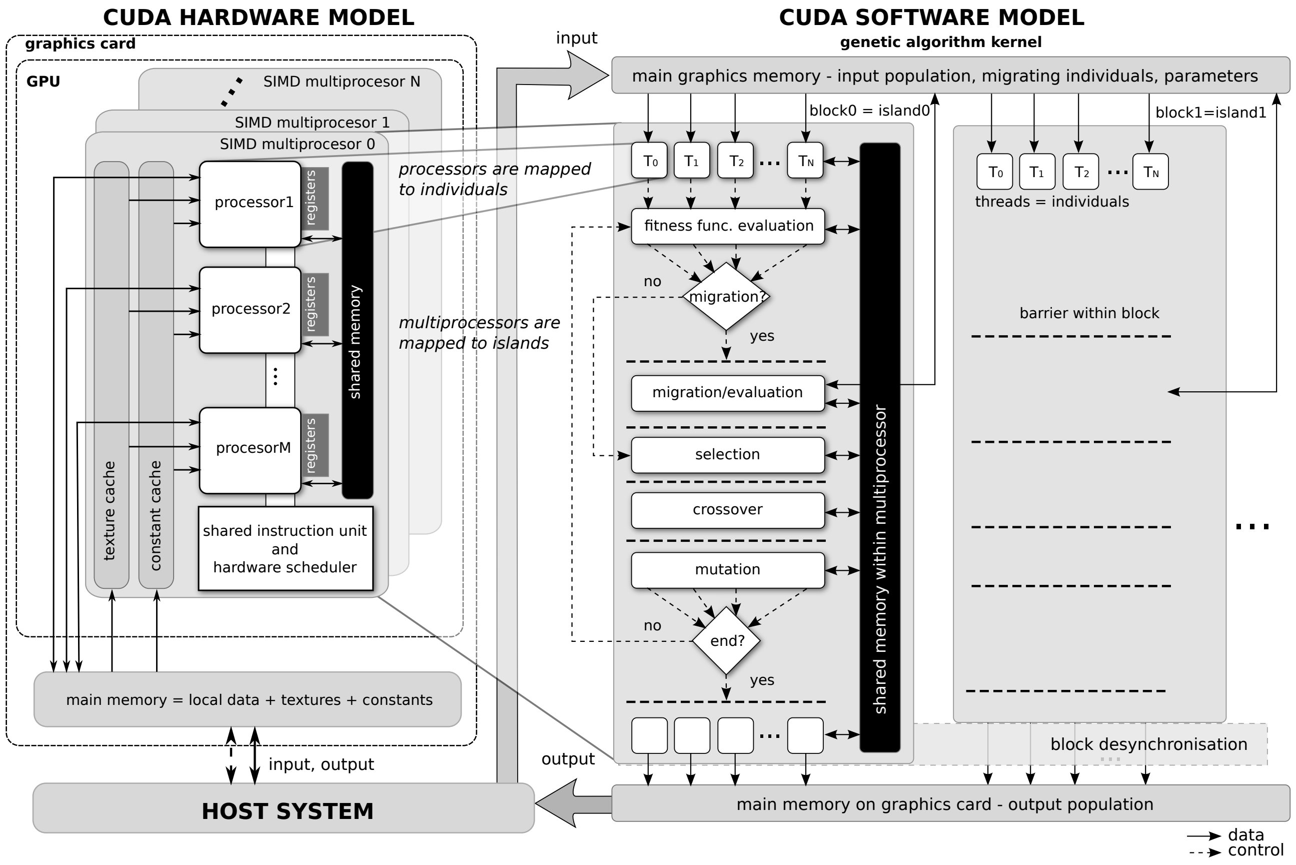


Figure 2: Mapping of the genetic algorithm to CUDA hardware and software model.

Figure 2 shows whole concept of fitting our parallel genetic algorithm model (right side) to the CUDA-capable GPU (left side). This model has following features:

- On the contrary to GPU local search methods [4], our model runs genetic algorithm entirely on the GPU. This is beneficial for avoiding system bus bottleneck.
- CUDA software threads are mapped to individuals, meaning that one FP processor simulates one individual. This ensures both SIMD-friendly execution and fine grained parallelism needed for the GPU to reach its full potential.
- Multiprocessors simulate islands which means that threads (individuals) can be synchronized easily in order to maintain data consistency and onchip hardware scheduler can swiftly swap existing islands between multiprocessors to hide memory latency.
- Fast, shared memory within the multiprocessor is used to maintain population of individuals. This is very beneficial for speedup, but forces us to limit population size to 16KB per island on most GPUs[2]. If the population was larger, slower main memory had to be used. Impact of this change depends on number of simulated islands as well as on population size – more simulated islands offer hardware scheduler room for hiding memory latency.

[1]NVidia GPUs have 16 processors per multiprocessor
[2]New Fermi architecture has up to 48KB of shared onchip memory and 2 levels of data cache

Table 1: Hardware and software used for testing

hardware	
CPU	Core i7 920, 2.66 GHz
GPU	nVidia GeForce GTX260-SP216 (216 cores in 27 multiprocessors), 1.24Ghz
software	
OS and compilers	Ubuntu Linux 10.04 x64, GCC 4.3, CUDA toolkit 3.0
CPU GA	default Galib (single threaded)
GPU GA	presented custom GA

Table 2: Setting of genetic algorithm parameters

GA parameter	**value for quality test**	**value for speedup test**
number of individuals	256	2;4;8;16;32;64;128;256
problem instance size	4;20;25;40	4;40
number of islands (no migration)	1;128;1024	1;2;4;8;16;32;64;128;256;512;1024
crossover rate	0.7	
crossover type	uniform	
mutation rate	0.05	
mutation type	bitflip	
terminating crieria	1000 generations	
selection	Tournament (two random individuals)	

- Our model uses both Uniform and Gaussian pseudo-random generators (PRNG) mentioned in [10][5]. This ensures very quick execution but we think it has possibly negative effect on results quality.
- Current implementation uses parallel Tournament selection where two individuals are randomly chosen from population.

More details about our implementation can be found in [9].

Using `--use_fast_math` compiler parameter, CUDA programmer can choose between usage of less precise but faster mathematical function or more precise but slower ones. This can be beneficial for speedup in some cases.

3 Results

Table 1 summarizes our testing environment. We compare commonly used Galib[7] library (single-threaded, implemented in C++) with our GPU PGA model. This offers objective comparsion for results quality and offers idea of speedup that can be achieved if one makes effort in order to utilize the GPU. We also discuss potential speedup achivable by parallelization of CPU code.

Our hardware platform was 2-years old, mainstream 100$ GPU GTX260 and relatively new Core i7 CPU 920.

The used genetic algorithm parameters are shown in table 2. We measure GPU execution time using CUDA profiler, data transfer time is measured using `CuTimer` function and for whole program execution time we use Python `time.time()` function[3].

3.1 Problem Instances

Our benchmark 0/1 Knapsack problem instances can be found in [8] and are shown in table 3.

3.2 Testing Enviroment

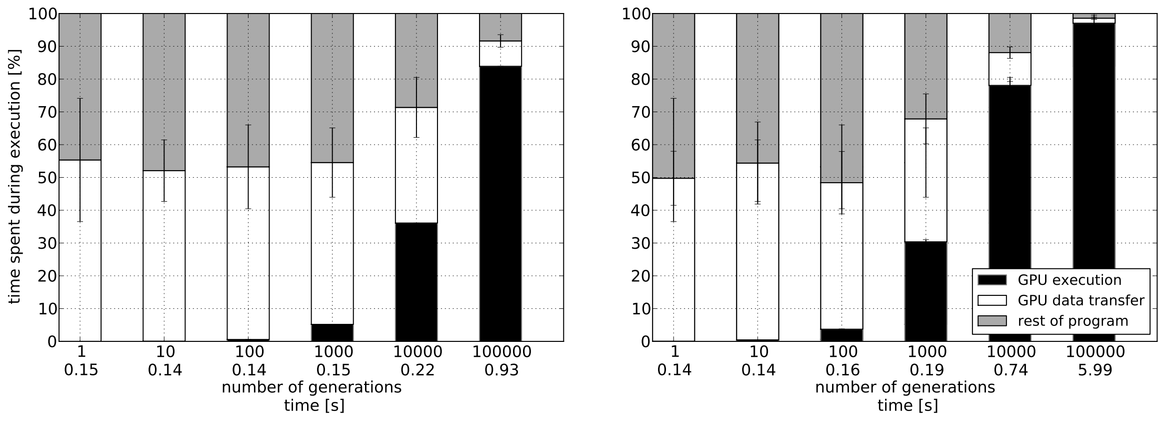
On the contrary to classical CPU computing, execution on the GPU involves variety of supplementary tasks: GPU initialization, allocation of GPU memory, data and program transfer along system bus and finally OS driver

[3]This function measures actual time spent by program startup, execution and deallocation. It offers more precise results than UNIX time utility.

Table 3: Used 0/1 Knapsack problem instances

id (size)	**limit**	**best**	**items weights** w
	(W)		**items values** v
9000 (4 bits)	100	473	$w = \{18,42,88,3\}$
			$v = \{114,136,192,223\}$
9050 (10 bits)	100	798	$w = \{27,2,41,1,25,1,34,3,50,12\}$
			$v = \{38,86,112,0,66,97,195,85,42,223\}$
9270 (25 bits)	300	3307	$w = \{40,1,10,24,11,26,4,2,34,6,5,9,20,19,42,18,37,21,3,31,43,45,35,30,7\}$
			$v = \{124,197,58,228,235,128,252,212,52,156,170,84,69,12,144,222,21,227,225,239,37,103,113,216,179\}$
9593 (40 bits)	600	4994	$w = \{31,4,27,1,18,8,5,7,21,3,2,20,15,10,9,23,12,6,39,34,17,28,14,32,25,42,16,19,37,49,11,47,44,26,33,43,36,35,13,29\}$
			$v = \{239,240,158,110,27,12,170,82,163,228,202,100,177,56,82,118,221,174,134,241,30,33,206,1,34,204,218,65,198,163,11,115,210,207,144,138,5,252,156,197\}$

overhead. This makes GPU unsuitable for simple tasks, as overhead is then too high relative to computation time. Charts 3 are showing percentage of time spent by actual GPU computation, data transfers and rest of program (mean values over 10 executions) for 4-bit and 40-bit Knapsack problem instances.



(a) 4-bit Knapsack problem (ID 9000)

(b) 40-bit Knapsack problem (ID 9594)

Figure 3: Relative time spent by subtasks

Charts show that time spent by GPU execution is insignificant for simple problems, where GPU is occupied for just 10s of miliseconds, but for sufficiently complex tasks it takes majority of time. Because there is no point in mastering CUDA and involving GPU in simple tasks that can be efficiently executed on CPU within fraction of second, we focus our measurement on time spend by actual GPU computation. This bypasses distortion caused by driver overhead and data transfers and offers accurate GPU performance estimation for sufficiently long executions.

3.3 Speedup Calculation

We chose performance unit as number of fitness function evaluations per second. 10 measurements were made as described in section 3.2 for CPU and GPU with `fastmath` turned on and off.

As can be seen in table 3, CPU performance doesnt change much for different problem instances wheres GPU execution performance higly varies. `FastMath=1` parameter leads to approx. 10% performance boost. GPU is fully saturated only when simulating multiple populations in parallel. This is not disadvantage as GA is stochastic method and runs are typically repeated many times in order to obtain reasonable results.

Put into charts 4 and 5 for `FastMath=1`, we can observe that GPU implementation is particularly powerful while given sufficient amount of work. Test were performed for 1000 generations, therefore time spent by GPU execution is relatively low compared to time spent by whole program. Charts (a) show values for whole program run times so they present more likely easy problems where GPU is utilized for just a short time period. On the

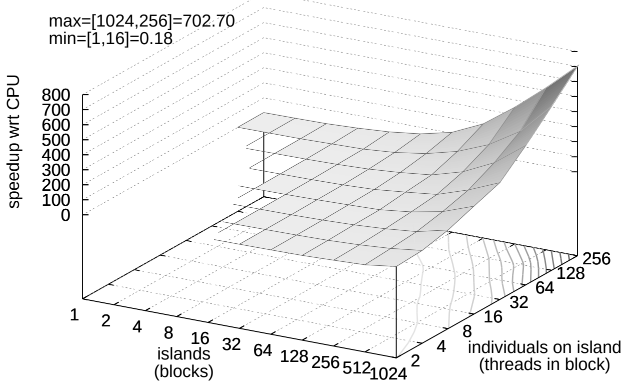
Table 4: Overall CPU vs GPU execution perfomance

architecture	**problem size**	**performance**	
		min	max
CPU	4	260 568 ±15%	905 386 ±2%
CPU	40	189 242 ±29%	398 922 ±1%
GPU fastmath=1	4	696 670 ±1%	1 120 131 805 ±0%
GPU fastmath=1	40	58 325 ±6%	111 234 460 ±1%
GPU fastmath=0	4	818 588 ±1%	953 053 908 ±0%
GPU fastmath=0	40	57 025 ±7%	102 174 403 ±0%

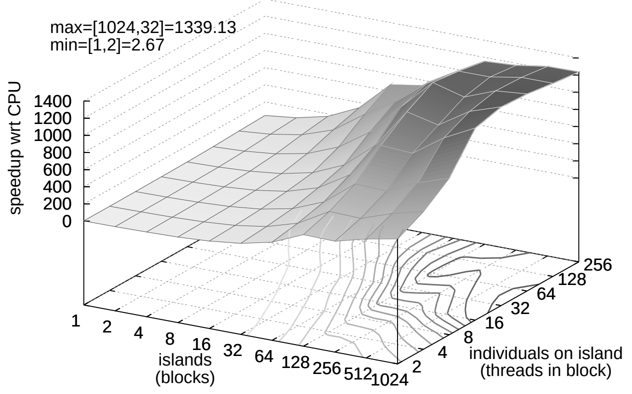
other hand, charts (b) show values for actual GPU execution performance without including data transfers and OS driver overhead and therefore present values for problems where GPU is utilized for at least several seconds.

As we can see, problem instance size also highly affects GPU performance. 10 times larger problem cuts CPU performance to half, but to 10% in case of the GPU. We think that this is because of mostly uncached and less tolerant memory hierarchy used on the GPU.

Average speedup is 462.2 resp. 44.4 for 4bit resp. 40bit problem and actual GPU execution performance. For 1000 generations including data transfers and OS driver overhead, average speedup is 61.0 resp. 22.0. Note that used library is single-threaded, therefore ideally parallelized CPU code could run up to 4-times faster on used CPU.

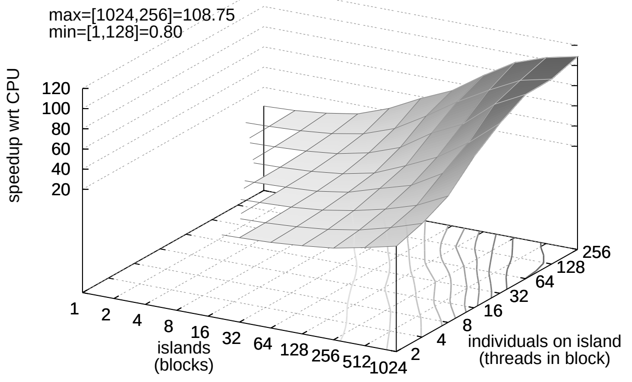


(a) with GPU supplementary tasks

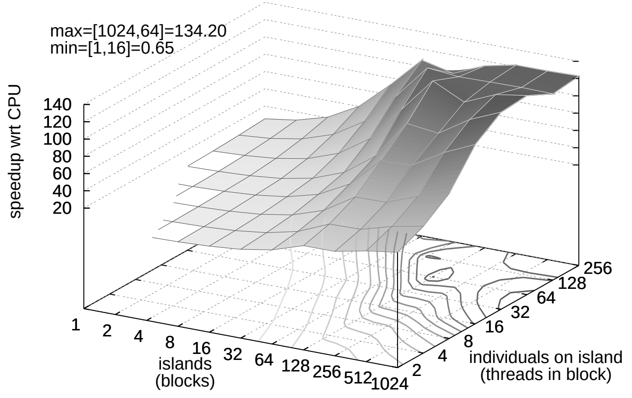


(b) without GPU supplementary tasks

Figure 4: Speedup against CPU for 4-bit Knapsack problem (values below 1 are hidden)



(a) with GPU supplementary tasks



(b) without GPU supplementary tasks

Figure 5: Speedup against CPU for 40-bit Knapsack problem (values below 1 are hidden)

Table 5: Quality of results

platform	parameters	quality [%]	GPU (program) time [s]
CPU	256 individuals, single island	94.83	– (190.72)
GPU	256 individuals, single island	86.67	1.02 (67.11)
	256 individuals, 128 islands (no migration)	96.64	6.43 (127.22)
	256 individuals, 1024 islands (no migration)	98.39	52.20 (623.23)

3.4 Results Quality

We measured quality of results given by GA as follows:

$$\text{score} = 100 \cdot \frac{\sum_1^n F_a}{n \cdot F_b} [\%] \tag{5}$$

where $n = 100$ is number of repetitions, F_a is the best acceptable fitness (not overweighted) found in whole final population of the executed GA and finally, F_b is fitness of best solution of the optimized problem instance (defined by problem instance in table 3, column best). Scores shown in table 3 are mean values for all 4 problem instances.

As it is evident from table, GPU can optimise Knapsack problem better in the same amount of time. Again, 1000 generations is not enough to utilize GPU for reasonable amount of time so GPU and program times differ significantly.

GPU solution runs slightly worse for same parameters. We think that this is caused by fast pseudo random number generator running on the GPU. Overall, GPU is able to optimise Knapsack problem faster. `FastMath=1` doesnt affect solution quality, which is beneficial as its usage leads to 10% better performance.

4 Conclusion

We have analyzed strong and weak points of utilizing GPU for general purpose computation and proposed our parallel genetic algorithm model running entirely on the GPU. We have shown that our model is able to optimise 4 different 0/1 Knapsack problem instances faster than CPU Galib code. Average speedup of GPU GA execution is 462x resp. 44x for 4-bit resp. 40 bit problem instance, peak speedup for 1024 simulated populations is 1340x resp. 134x. We have also presented that GPU must be utilized for sufficiently long evolutionary epoch in order to obtain reasonable program speedups.

Acknowledgement: This research has been carried out under the financial support of the research grants Natural Computing on Unconventional Platforms, GP103/10/1517 (2010-2013) of Grant Agency of Czech Republic, Security-Oriented Research in Information Technology, MSM 0021630528 (2007-13), the BUT FIT grant FIT-S-10-1 and the research plan MSM0021630528.

References

[1] nVidia Corporation: *Nvidia CUDA Programming Guide 2.0.*

[2] Holland, J. H.: *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[3] Olsen, A.: Penalty functions and the knapsack problem. *The 1st IEEE Conference on Evolutionary Computation.* Part 2 (of 2); Orlando, FL; USA; 27 June-29 June 1994. pp. 554-558. 1994

[4] Quan, D., Yang, L: Solving 0/1 Knapsack Problem for Light Communication SLA-Based Workflow Mapping Using CUDA, cse, vol. 1, pp.194-200, *2009 International Conference on Computational Science and Engineering*, 2009

[5] Pharr, M. and Fernando, R.: *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison-Wesley Professional, 2005, ISBN 0-321-33559-7.

[6] nVidia Corporation: *NVIDA's Next Generation CUDA Compute Architecture: Fermi.* nVidia Corporation, 2009.

[7] Matthew, W.: GAlib: *A C++ Library of Genetic Algorithm Components.* Massachusetts Institute of Technology, 1996.

[8] Czech Technical University In Prague: *Knapsack problem benchmark instances.* http://service.felk.cvut.cz/courses/36PAA/knapsolv.html#bench

[9] Pospichal, P., Jaros, J., Schwarz, J.: Parallel Genetic Algorithm on the CUDA Architecture, In: *Applications of Evolutionary Computation*, Berlin Heidelberg, DE, Springer, 2010, s. 442-451

[10] Nguyen, H.: GPU Gems 3. Addison-Wesley Professional, 2007, ISBN 978-0321515261