Self-organizing Migration Algorithm on GPU with CUDA

Michal Pavlech

Tomas Bata University in Zlin, Faculty of Applied Informatics
nám. T.G.Masaryka 5555, 760 01 Zlín
Czech Republic
pavlech@fai.utb.cz

Abstract. A modification of Self-organizing migration algorithm for general-purpose computing on graphics processing units is proposed in this paper. The algorithm is implemented in C++ with its core parts in c-CUDA. Its implementation details and performance are evaluated and compared to previous, pure C++ version of algorithm. 6 commonly used artificial test functions are used to test the performance. The test results clearly show significant speed gains without a compromise in convergence quality.

Keywords: SOMA, CUDA, GPGPU, evolutionary algorithm.

1 Introduction

Modern graphics cards (GPUs) are capable of much more than processing and displaying of visual data. Their multiprocessor architecture is suitable for parallel algorithms which can benefit from the SIMD architecture. GPUs have generally lower clock frequency than modern processors but rely on parallel execution of instructions over large blocks of data.

Evolutionary algorithms (EAs) are in their very essence parallel processes and as such are suitable for implementation on such multiprocessor devices with little or no modifications to their functionality. First attempts on utilizing the processing power of GPUs for EAs were done before the release of general purpose computing APIs. Researchers had to modify algorithms to fit them into specialized GPU processing units – the shaders and data structures had to be translated into textures. An example of this approach is the work of Wong , Wong and Fok who used GPU to increase the performance of genetic algorithm and reported speedup of up to 4.42 times [1, 2].

With the emergence of general purpose computing frameworks for GPUs, like Compute Unified Device Architecture (CUDA), Open Computing Language (OpenCL) and DirectCompute, new possibilities have risen and general purpose computing on graphics processing units (GPGPUs) became available to a broader audience.

A large number of evolutionary algorithms has been ported to GPUs, with CUDA being arguably the most common API used. Some of these algorithms include: genetic algorithm [3, 4], genetic programming [5], differential evolution [6], ant colony optimization [7] and particle swarm optimization [8, 9].

V. Snasel et al. (Eds.): SOCO Models in Industrial & Environmental Appl., AISC 188, pp. 173–182.
springerlink.com

Self-organizing migration algorithm (SOMA) was chosen for a subset of its characteristics which are not common in other EAs and make it suitable for parallel architecture with limited communication between processing units.

The paper is divided as follows: first part describes SOMA, modifications and steps necessary for its implementation using CUDA. Next part describes methodology used in testing the algorithms performance followed by results of the tests and conclusion.

2 Methods

2.1 SOMA

SOMA was first introduced by Zelinka [10]. It is modeled after behavior of intelligent individuals working cooperatively to achieve common goal, for example pack of animals working together to find food source. This behavior is mimicked by individuals moving towards another individual, known as the leader, which generally has the best fitness value. Efficiency and convergence ability of SOMA was proved in numerous applications [11, 12].

SOMA uses slightly different nomenclature in comparison to other evolutionary algorithms, one round of the algorithm is called migration. There is a set of three control parameters which control the algorithm's behavior and significantly affect its performance:

- *PathLength* ∈ [1; 5]: Specifies how far from the leader will the active individual stop its movement.
- *Step* ∈ [0.11; *PathLength*]: Defines the size of discreet steps in solution space.
- *PRT* ∈ [0; 1]: Perturbation controls creation of perturbation vectors which influence the movement of active individual.

The movement of individuals through error space is altered by a random perturbation. In order to perturb movement of individuals, boolean vector **PRTVector** is generated according to equation:

$$\begin{aligned} \mathbf{PRTVector}_j &= 1 \text{ if } \text{rand}_j(0,1) < PRT \\ &= 0 \text{ otherwise} \end{aligned} \tag{1}$$

where $j = 0,1,...,$ *Dimension*-1. The **PRTVector** is generated before movement of active individual, and is generated for each individual separately. Value 0 in **PRTVEctor** means, that corresponding dimension of individual is locked – individual cannot change its value during this migration. If all elements of **PRTVector** are set to 1, individual moves straight towards the leader.

Creation of new individuals for next population is implemented using vector operations. Active individual moves towards the leader according to equation:

$$\mathbf{x}_{i,j,t}^{ML+1} = \mathbf{x}_{i,j,start}^{ML} + \left(\mathbf{x}_{L,j}^{ML} - \mathbf{x}_{i,j,start}^{ML}\right).t.\mathbf{PRTVector}_j \tag{2}$$

where *ML* is the number of current migration round, $\mathbf{x}^{ML}_{i,j,start}$ is position of active individual at beginning of current migration, $\mathbf{x}^{ML}_{L,j}$ is the position of the leader, $t \in [0; pathLength]$, $t = 0, Step, 2*Step,...$

This equation is applied to all individuals except the leader, which does not move. For each step in solution space (denoted by *t*) newly created individual is evaluated. At the end of migration (individual made all steps lower than *PathLength*) individual is set to a position which had the best fitness value during the current migration. Therefore the fitness value of an individual cannot deteriorate.

2.2 cuSOMA

The aim of this work was to create the fastest possible implementation of SOMA using the CUDA toolkit without compromising its functionality and convergence ability. This algorithm was named cuSOMA. cuSOMA is implemented as a C++ class with its core components in c-CUDA.

CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads. The CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program, the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively [13].

Early implementations of EAs for GPUs reported problems with generation of random numbers which was time consuming on GPU hardware [6]. More recent nVidia CUDA SDK features a library called CURAND which deals with efficient pseudorandom generators on GPUs [14]. CURAND random generators need to preserve their states using a data structure *curandState* in order to avoid generating the same number sequences for each thread and for each kernel call. Separate states for each thread are stored in an array and have to be copied from host to device prior to kernel calls. Initialization of states is done only once during the class initialization and uses standard C++ random generator. After a seed is generated a kernel is launched, which initializes random generators for each thread in parallel.

Population of candidate solutions for cuSOMA is stored in one dimensional array on the device. To minimize the impact of memory transfers on algorithm performance the population and states are copied from device only after a user definable amount of migrations finished. The migration of individuals requires 2 additional arrays to store intermediary positions and the best position discovered during the migration. Both arrays for temporary values have the same structure as the original population. The fitness values of each individual are stored inside a population after the phenotype. The scheme of population is depicted in Fig 1.

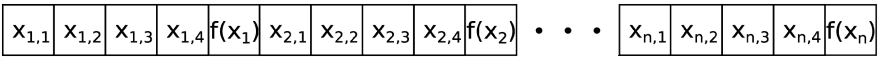

Fig. 1. Scheme of population stored on device

One possible bottle neck of this solution is the transfer of data between device and host. To quantify this problem a test was conducted which identified the amount of time needed for data transfers and time for actual computation on device. The data transfer time should be proportional to a size of copied data and there are two parameters which influence this amount: population size and dimension of a solution. Data transfer times include: allocation of arrays on device, copying of population and states to and from device and freeing of allocated memory. Two tests were conducted to explore the influence of these two parameters with all the time measurements conducted with CUDA events. First test was run with dimension locked at 50 and population size growing steadily from 100 to 2,500 with a step of 100 and with population size from 5,000 to 25,000 with step of 5,000. All attempts were run for 100 migration rounds. Second test was run with population size and migration rounds locked to 1,000 and 100 respectively and with dimension of the test functions growing from 25 to 250 with step of 25 and additionally from 500 to 2,500 with step of 500. All tests were repeated for 10 times and averaged in order to avoid random glitches. The share of data transfer times on total run time can be seen in Fig. 2 and Fig. 3 where it is displayed as a percentage of total runtime with x axis displayed in logarithmic scale for better readability.

It can be seen that data transfer times are smaller than 1% of the overall computation time for all test cases. Increase in dimension resulted in longer computation times and thus the influence of data transfers decreases. Increase in population size lead to opposite results up to 2,500 individuals where data transfer share rose steadily up to 0.36% for De Jong function. For larger population sizes, on the contrary, this share declined. Results show that computational time rose faster than data transfer time for a number of individuals which was significantly larger than number of thread processors on GPU and thus the share of data transfer time decreased.

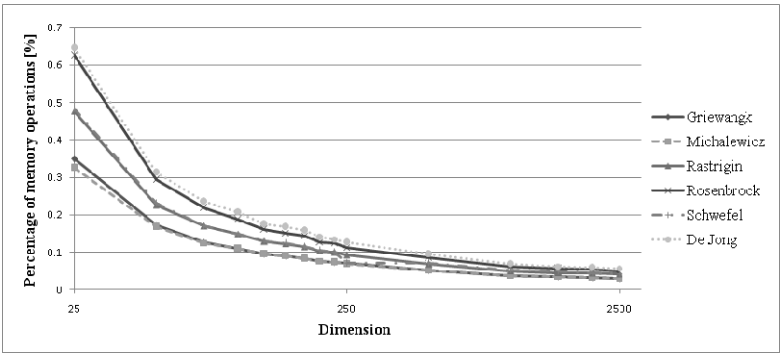

Fig. 2. Share of data transfer time on algorithms runtime with relation to dimension of cost function

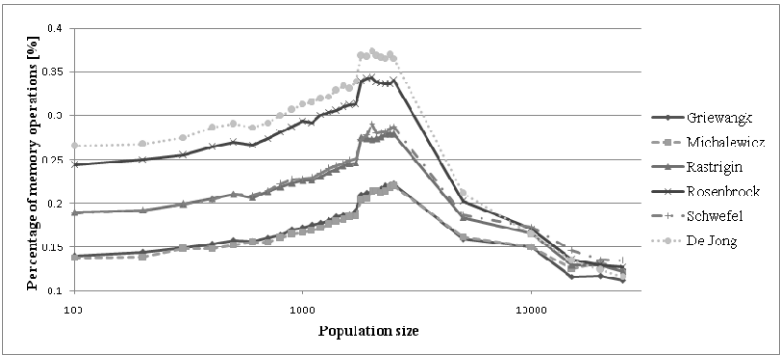

Fig. 3. Share of data transfer time on algorithms runtime with relation to population size

For larger populations it is not possible to execute all the operations using only one CUDA block because the thread per block limit is set to 1024 (512 for devices with compute capacity lower than 2). Large populations require use of several separate blocks. Number of blocks is decided using equation:

$$blocks = \frac{(population\_size + threads\_per\_block - 1)}{threads\_per\_block} \tag{3}$$

During the computation each thread calculates the index of an individual which it should migrate according to equation:

$$index = (block\_id \times threads\_per\_block) + thread\_id \tag{4}$$

where *block_id* is the index of current block in a grid and *thread_id* is thread index in current block.

The influence of threads per block on performance of cuSOMA was investigated in three tests which had population size set to 1,000, 3,000 and 5,000, in order to simulate spawning of more blocks, dimension set to 50 and number of migrations to 100.

Table 1 shows the best performing number of threads per block for 3 different population sizes. The number of threads has considerable impact on cuSOMA performance. The test showed that 16 threads per block was the setting with the most consistent performance across population sizes and cost functions and therefore was used in further tests.

Table 1. Number of threads with fastest execution with relation to population size and test function

Population	Griewangk	Michalewicz	Rastrigin	Rosenbrock	Schwefel	De Jong
1,000	16	16	16	16	16	16
3,000	32	32	32	16	32	16
5,000	16	256	16	16	16	16

The population is stored in global device memory during computation but this memory is the slowest type of memory available on device. Therefore it is desirable to move as much data as possible from global to either shared memory or registers. Both, registers and shared memory, are too small to contain whole population so it was decided to move only the leader for each migration, which is common for all individuals, into shared memory. The contents of shared memory are accessible by all threads from one block so it is sufficient if only first thread from the block copies the leader while other threads in the block are idle, waiting for this operation to finish.

Possible performance gains of using shared memory may be hindered by time needed to copy the leader from population (global memory) hence a test was conducted to find if shared leader technique brings performance improvements. The test was run with two algorithm variants, one which was using leader in shared memory and one which was accessing leader from global memory. The population size was set to 3,000, dimension to 50 and number of migrations to 100, number of threads per block was changed in powers of 2. Fig. 4 shows the difference in time between the two algorithm's versions. All values above zero mean that version with shared leader was faster for a given number of threads per block.

The leader in shared memory turned out to be a questionable improvement. There are certainly performance benefits for all test functions, but they are in range of milliseconds. For 16 threads the highest improvement was 3.25ms (Griewangk function) and the highest performance loss was 7.19ms (Michalewicz function). Overall results of shared leader test can be summed as follows: algorithm with shared leader was faster in 33 cases but slower in 15. Although it adds only minor speed improvements the shared leader was left intact as a part of cuSOMA.

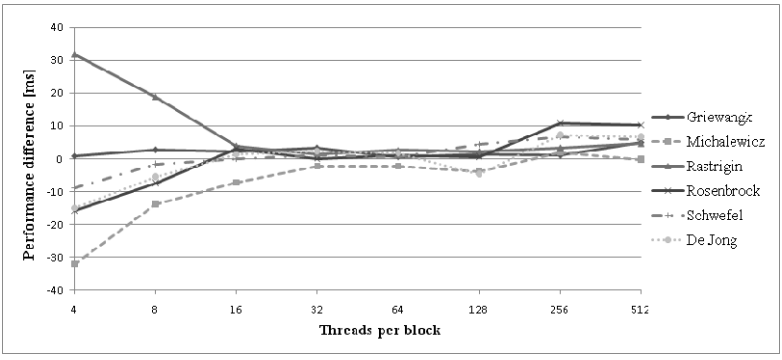

Fig. 4. Performance gains from leader in shared memory with relation to threads per block

In order to determine which individual will be the leader for next migration round it is necessary to search the whole population for individual with the best fitness. Because the population during migrations stays in the global memory of GPU it is possible to perform parallel search for the best individual. cuSOMA uses the parallel reduction to find the index of the leader and store it in the global memory where the threads performing migration can access it.

2.3 Performance Tests

The most important reason for implementing SOMA on GPU is the promise of possible speedup when compared to running the algorithm on CPU. To test these possible gains two types of tests were conducted with 6 different artificial test functions. Both tests were conducted on the same hardware: nVidia Tesla C2075, 448 thread processors at 1150 MHz and Intel Xeon E5607, at 2.26 GHz.

All the test cases were run for 100 migration rounds, repeated 10 times and their results averaged. The measured parameter was the speedup of cuSOMA when compared to CPU implementation.

First test was aimed at how well the cuSOMA scales to increase in population size.

Second test used constant population size and the dimension of the test functions was increasing. The purpose of this test was to decide if cuSOMA is suitable for functions which require higher computational power for evaluation.

In addition to speedup tests a simple test to determine if the convergence ability of SOMA was not compromised was conducted with all 6 test functions set to 25 dimensions, population size to 1,000 and number of migrations to 100.

Detailed setup of all tests is in Table 2.

Table 2. details of performance tests

Test	Population size	dimension
1	100-25,000	50
2	1,000	25-2500
3	1,000	25

3 Results

Fig. 5 shows that cuSOMA scales extremely well to the size of population in values from 100 to 2500. Each increase of population size enlarged the performance gap between CPU and GPU implementation of SOMA. Further increase in population size showed that the initial steady growth of speedup is much less significant for higher volumes of individuals. However no test function showed substantial decrease in speedup and cuSOMA stayed superior to CPU implementation.

The highest recorded speedup was 126.9 for Michalewicz function and 25,000 individuals. cuSOMA seems to be more suitable for more computationally demanding functions as can be seen from comparison of results for De Jong and Michalewicz functions. This effect is probably caused by higher clock speed of CPU in comparison to thread processor clock speed of GPU, with this difference becoming more obvious for less demanding functions.

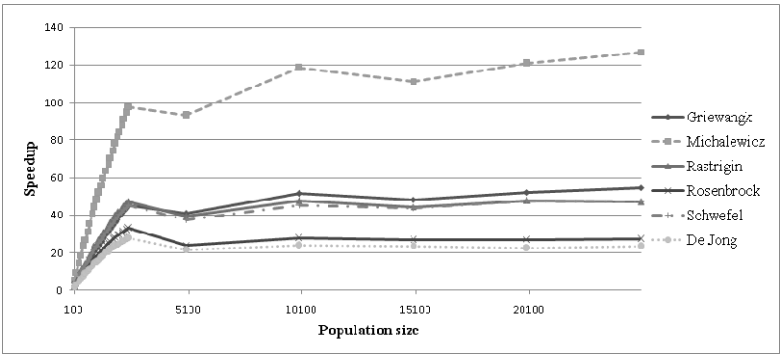

Fig. 5. Speedup measured with relation to population size

Fig. 6 shows that cuSOMA does not scale very well to increase in cost function dimension, which can be also viewed as an increase in computational complexity of cost function evaluation. The GPU implementation is considerably faster than its CPU counterpart with speedups ranging from 13 (De Jong function) to 56 (Michalewicz function). Results from higher dimensions show that the value of speedup remained almost constant witch exception of Michalewicz function which constantly showed improvements with each dimension increase.

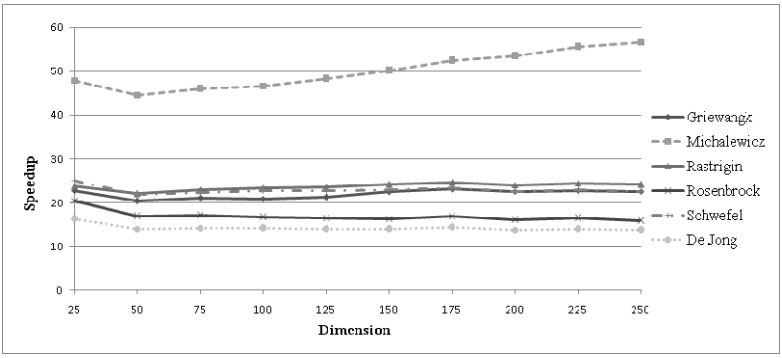

Fig. 6. Speedup measured with relation to test function's dimension

As shown in Table 3, cuSOMA was able to find known global minima [15] for 5 test functions with Rosenbrock's function being an exception. Further test runs with more migration rounds were able to find global optimum even for this function. Value of global extreme of Michalewicz's function for 25 dimensions is not known.

Table 3. The best found values in 10 test runs

	Griewangk	Michalewicz	Rastrigin	Rosenbrock	Schwefel	De Jong
Best value	0	-24.633	0	0.0114101	-10474.6	1.66679E-13
Best known value	0	Not known	0	0	-10474.6	0

4 Conclusion

An implementation of self-organizing migration algorithm for GPUs using nVidia CUDA was presented. This new version was named cuSOMA and provides significant improvements in computation time. The test with artificial test functions showed high speedups in comparison to CPU implementation of SOMA with highest recorded speedup of 126.9. cuSOMA scales very well to large population sizes and therefore it should be especially suitable for functions where finding global minimum requires a high number of individuals in order to satisfactory search the solution space. On the other hand, increase in computational complexity of test functions did not produce further speedup gains in comparison to CPU version but nevertheless it was still considerably faster.

The influence of block size on algorithm performance was investigated and it can be seen, that it can have significant impact on runtime but it is also dependent on the nature of optimized function. Also the share of data transfer times on overall runtime was investigated and tests showed that for large populations this time grows slower that actual computational time.

Further research will focus on fine tuning cuSOMA and finding other optimization algorithms which could be even better suited for general computing on graphical processing units.

Acknowledgments. This paper was created as a part of internal grant agency project number IGA/FAI/2012/033 at Tomas Bata University in Zlin, Faculty of Applied Informatics and of the European Regional Development Fund under project CEBIA-Tech No. CZ.1.05/2.1.00/03.0089.

References

1. Wong, M.L., Wong, T.T., Fok, K.L.: Parallel evolutionary algorithms on graphics processing unit. In: Proc. IEEE Congress Evolutionary Computation, vol. 3, pp. 2286–2293 (2005)
2. Fok, K.L., Wong, T.T., Wong, M.L.: Evolutionary Computing on Consumer Graphics Hardware. IEEE_M_IS 22, 69–78 (2007)
3. Pospichal, P., Jaros, J., Schwarz, J.: Parallel Genetic Algorithm on the CUDA Architecture. In: Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcazar, A.I., Goh, C.-K., Merelo, J.J., Neri, F., Preuß, M., Togelius, J., Yannakakis, G.N. (eds.) EvoApplicatons 2010, Part I. LNCS, vol. 6024, pp. 442–451. Springer, Heidelberg (2010)
4. Zhang, S., He, Z.: Implementation of Parallel Genetic Algorithm Based on CUDA. In: Cai, Z., Li, Z., Kang, Z., Liu, Y. (eds.) ISICA 2009. LNCS, vol. 5821, pp. 24–30. Springer, Heidelberg (2009)
5. Langdon, W.B.: Large Scale Bioinformatics Data Mining with Parallel Genetic Programming on Graphics Processing Units. In: de Vega, F.F., Cantú-Paz, E. (eds.) Parallel and Distributed Computational Intelligence. SCI, vol. 269, pp. 113–141. Springer, Heidelberg (2010)
6. de Veronese, L.P., Krohling, R.A.: Differential evolution algorithm on the GPU with C-CUDA. In: Proc. IEEE Congress Evolutionary Computation (CEC), pp. 1–7 (2010)

7. Fu, J., Lei, L., Zhou, G.: A parallel Ant Colony Optimization algorithm with GPU-acceleration based on All-In-Roulette selection. In: 2010 Third International Workshop on Advanced Computational Intelligence (IWACI), pp. 260–264 (2010)
8. Zhou, Y., Tan, Y.: GPU-based parallel particle swarm optimization. In: Proc. IEEE Congress Evolutionary Computation CEC 2009, pp. 1493–1500 (2009)
9. de Veronese, L.P., Krohling, R.A.: Swarm's flight: Accelerating the particles using C-CUDA. In: Proc. IEEE Congress Evolutionary Computation CEC 2009, pp. 3264–3270 (2009)
10. Zelinka, I.: SOMA—self organizing migrating algorithm. In: Onwubolu, G.C., Babu, B.V. (eds.) New Optimization Techniques in Engineering. Springer, Berlin (2004)
11. Senkerik, R., Zelinka, I., Oplatkova, Z.: Comparison of Differential Evolution and SOMA in the Task of Chaos Control Optimization - Extended study. In: Complex Target cf 2009 IEEE Congress on Evolutionary Computation, vols. 1-5, pp. 2825–2832. IEEE (2009)
12. Tupy, J., Zelinka, I., Tjoa, A., Wagner, R.: Evolutionary algorithms in aircraft trim optimization. In: Dexa 2008: 19th International Conference on Database and Expert Systems Applications, Proceedings, pp. 524–530. IEEE Computer Soc. (2008)
13. NVIDIA CUDA C Programming Guide. NVIDIA Developer Zone, `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf` (accessed March 13, 2012)
14. CUDA Toolkit 4.1 CURAND Guide. NVIDIA Developer Zone, `http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CURAND_Library.pdf` (accessed March 13, 2012)
15. Molga, M., Smutnicki, C.: Test functions for optimization needs (2005), `http://www.zsd.ict.pwr.wroc.pl/files/docs/functions.pdf` (accessed March 13, 2012)