Increasing the performance of AllToAll variant of self-organizing migration algorithm using CUDA

MICHAL PAVLECH, JAN SECKAR
Faculty of Applied Informatics
Tomas Bata University in Zlin
nam. T.G.Masaryka 5555, 760 01 Zlin
Czech Republic
pavlech@fai.utb.cz, seckar@fai.utb.cz

Abstract: - Modern graphics processing units offer general purpose parallel computing capabilities. Thus they have become a relatively low cost alternative for applications requiring extensive parallel computations. Evolutionary algorithms are especially well suited for parallel SIMD architecture. This paper deals with the modification of AllToAll variation of self-organizing migration algorithm, which has high computational demand for one round of algorithm, using the CUDA framework. The main goal is to speedup performance of the algorithm in comparison to CPU implementations.

Key-Words: - GPGPU, CUDA, SOMA, optimization

1 Introduction

Optimization by means of evolutionary algorithms (EAs) requires high amount of computational power. Therefore every possible speedup of the process is a welcome addition for the scientific community.

The introduction of general purpose computing frameworks for graphics processing units (GPUs) opened new possibilities as GPUs are devices with high number of processors capable of performing SIMD computations. Probably the best well-known and widely used framework is Compute Unified Device Architecture (CUDA) which only runs on GPUs from nVidia.

There were several previous works which used CUDA to increase performance of numerous EAs, some of them are: genetic algorithm [1], particle swarm optimization [2] and differential evolution [3].

The AllToAll variant of self-organizing migration algorithm (SOMA) is an EA which has very high computational demands per one algorithm round, due to high number of cost function evaluations needed. The aim of this paper is to create a suitable modification of this algorithm for GPUs so that its runtimes are lowered to more acceptable values.

All tests in this paper were repeated 5 times and results averaged in order to avoid random glitches influencing the results.

c-CUDA extends C programming language by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. The CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program. This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU. The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces, referred to as host memory and device memory, respectively [4].

2 Methods

2.1 SOMA and its variations

SOMA was first introduced by Zelinka in 2004 [5]. It is an evolutionary optimization technique which mimics the behavior of a cooperative pack of animals looking for food. It maintains population of solutions like most of the EAs but due to its different background it uses slightly different nomenclature, one round is called migration instead of generation. The basic version of SOMA, called AllToOne, finds better solutions by moving individuals in the solution space towards the one with the best fitness (the leader). The movement distance and coarseness of searching the space is controlled by two variables: *PathLength* controls how far from the leader will active individual stop its movement and *Step* controls the length of each discrete step. In order to introduce stochastic element into movement a third variable called *PRT* is present. *PRT* controls the creation of *PRTVector*,

which defines if a dimension of solution can or cannot be changed during this migration. *PRTVector* is generated for each individual separately according to equation:

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

The movement is then performed using equation:

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

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*,... The cost function for each step is evaluated

Apart from this version of SOMA there are two others which require higher computational power per migration round, but their convergence should be faster. First variant is called AllToAll. Individuals are not moving towards single leader, but to all remaining individuals in population. After moving to one individual the active individual is returned to its starting position and starts moving towards next one. After performing all the steps its position is set to the best position found during migrating. Second variant, AllToAllAdaptive is essentially the same as AllToAll, but after moving towards an individual the active individual is not restored to its starting position, but is set to best position found during the movement and from this position starts its new movement.

2.2 CUSOMA2A

The modification of SOMA for CUDA framework was named CUSOMA2A and is implemented as a C++ class with its parts written in c-CUDA.

In order to find the best adaptation of AllToAll algorithm for CUDA 3 different variants were programmed which differ in the way in which the CUDA threads are utilized for migration. All these variants share some common characteristics.

2.2.1 Population scheme

Population is in form of one dimensional array of float values and is a member of a C++ class. It has to be copied to the GPU prior to the migration and after a preset number of migrations is finished it is copied back to host memory where it can be further analysed. The internal scheme of population is depicted in **Fig. 1** and it contains genotype of each individual followed by its fitness value.

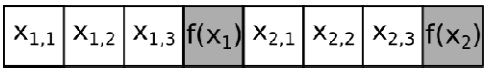

Fig. 1 Scheme of population in global memory

2.2.2 Population initialization

The initialization of population to random values and the first evaluation of all individuals is realized by CUDA kernels.

The random numbers are generated using the cuRAND from CUDA SDK, which allows the generation of random numbers with various characteristics from within the kernels. If each kernel should generate unique sequence of random numbers a separate state must be initialized and stored for each. These states are stored inside the C++ class in one dimensional array and copied to device prior to migration. In order to speedup the initialization of states a kernel responsible for this is launched before the first call of cuRAND generators which does this in parallel. The seed of these generators is obtained using the standard C++ random numbers generator.

Generation of initial values and evaluation of individuals requires launch of *populationSize* kernels with each kernel generating the whole genotype of an individual, evaluates it and stores in global population.

Fig. 2 shows schematic view of initialization phase.

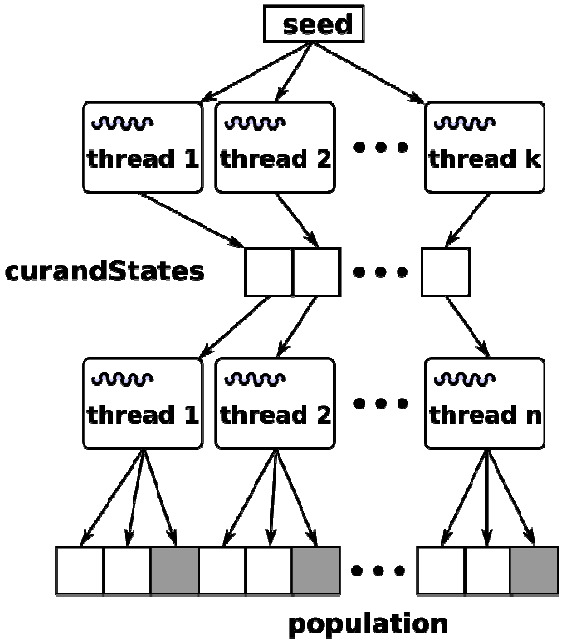

Fig. 2 Scheme of population initialization, *k* is equal to number of migration kernels, *n* to *populationSize*

2.2.3 Variant A

The CPU is only responsible for memory copies and launching of a number of kernels equal to a population size. Each CUDA thread is responsible

for migration of one individual towards all the other individuals in population.
This variant puts a lot of load on each thread resulting in long running kernels. If the program is run on a device which also serves as an output device for GUI, long running kernels may be terminated by system watchdog timer. Therefore the most complex problems should be solved without GUI or another GPU should be present for displaying the GUI.

2.2.4 Variant B

This variant shifts a part of load from GPU to the CPU. The CPU loops over all individuals in the population and launches *populationSize* kernels. Each CUDA thread computes migration of one individual towards one target individual and stores the best found solution in a global array. After the kernels finish, another kernel is launched which finds the best individual among the kernels and writes it into temporary population. To speedup the operation, the search for the best individual is implamented as a parallel reduction. After the migration of all individuals ends, the individuals from the temporary population are copied into population and new round can begin.
The logic behind this variant is that CUDA streams are running at lower clock speeds than CPUs and single streams should be used for tasks which are computationally less demanding that those running on CPU. Thus kernels in this variant run only migration of one individual to another one.

2.2.5 Variant C

The last variant fully utilizes the ability of GPUs to run high numbers of thread simultaneously. This variant launches the highest number of kernels equal to *populationSize* × *populationSize*. Each kernel computes migration of one individual towards another one, as in variant B, but migration of the whole population is not implemented via CPU loop, but by utilizing high number of kernel launches. ker
Each kernel performs migration of an individual towards another individual and writes best found position and fitness value into global memory. The index of active individual is computed according to equation:

$$kernelIndex / populationSize \quad (3)$$

And the index of target individual according to:

$$kernelIndex \bmod populationSize \quad (4)$$

If active individual is the same as target individual kernel only copies values from active individual into temporary population.
After the kernels finish, another kernel is launched which searches for the best value of the active individual using parallel reduction. These best values are copied into population and become the basis for next migration round. The schematic representation of migration is in **Fig. 3**.
The nature of this variant means that it requires high amount of memory – array of size of an individual for temporary storage and an array of the same size plus 1 for the best position and fitness for each kernel.
The amount of memory needed for the run of variant C is defined by equation:

$$[popSize \times popSize \times \dim + popSize \times popSize \times (\dim + 1)] \times sizeOf(float) \quad (5)$$

These requirements may prove restricting for optimization with big populations on GPUs with low amount of RAM.

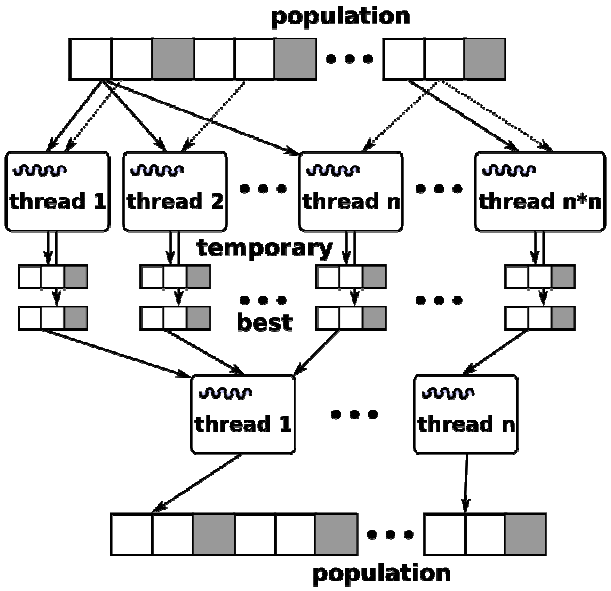

Fig. 3 The scheme of variant C migration, *n* is equal to *populationSize*

2.2.6 Performance of variants

In order to determine which variant offers the best performance a simple test was conducted with 3 artificial test functions.
De Jong's function:

$$f(x) = \sum_{i=1}^{n} x_i^2$$

$$-5.12 < x_i < 5.12 \quad (6)$$

Schwefel's function

$$f(x) = \sum_{i=1}^{n} [-x_i \sin(\sqrt{|x_i|})]$$

$$-500 < x_i < 500 \quad (7)$$

Griewangk's function

$$f(x) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1, \quad (8)$$

$$-600 < x_i < 600$$

Dimension of test functions was set to 100 and each algorithm had to perform 10 migrations. The population size varied from 50 to 450 with a step of 100. The settings of SOMA variables were S*tep* =0.11, *PRT* =0.8, *pathLength* = 3, the threads per block was set to 256. Time needed for 10 migrations was measured using the cudaEvents from CUDA SDK.

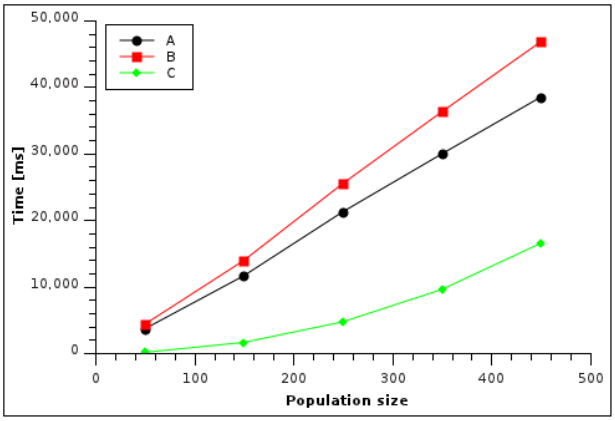

Fig. 4 Comparison of 3 variants, De Jong's function

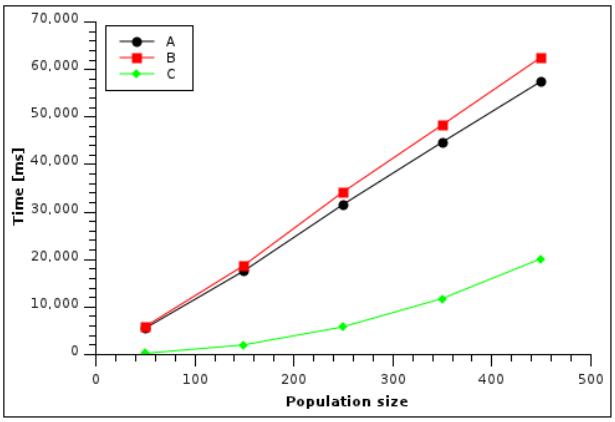

Fig. 5 Comparison of 3 variants, Schwefel's function

As can be seen in **Fig. 4**, **Fig. 5** and **Fig. 6** variant C consistently outperformed the other variants and thus offers the best performance in tested range. The rest of the tests in this article will be run with variant C exclusively.

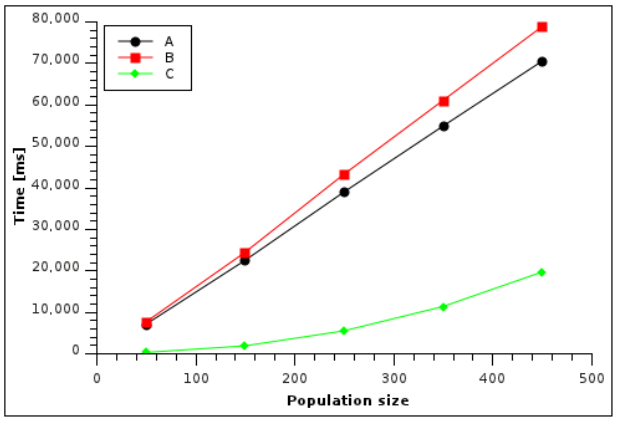

Fig. 6 Comparison of 3 variants, Griewangk's function

2.2.7 Memory consumption

Although being the fastest of 3 variants, variant C requires the highest amounts of GPU memory which can be limiting for older and lower end GPUs.

The exact amount of required memory with relation to increase in population size was tested with De Jong's function, dimension set to 200 and population size changing from 100 to 500 with a step of 50. Other functions were not tested, because selection of one of the 3 test functions had none effect on memory consumption.

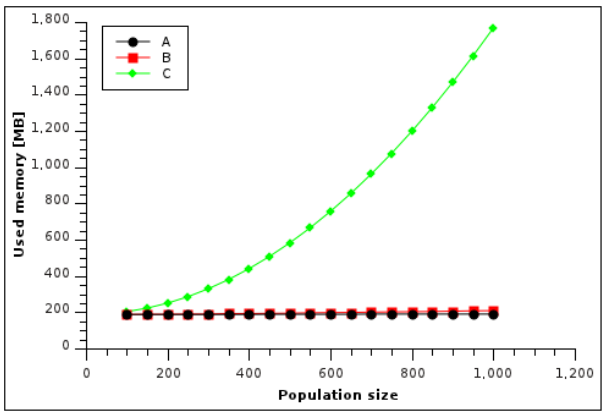

Fig. 7 Memory consumption of all 3 variants

Figure **Fig. 7** show the difference between the amounts of memory needed to sustain all 3 variants of CUSOMA2A. And **Table 1** shows amounts of memory for selected population sizes. It can be seen that variant C has much larger memory needs than previous variants and this fact should be taken into consideration when running optimization problems with high number of individuals in population.

Table 1 Memory consumption in mega bytes of all 3 variants

Pop. size	A	B	C
100	187.33	187.33	202.83
500	188.33	194.33	581.45
1,000	189.33	208.33	1,766.57

ISBN: 978-1-61804-117-3

2.2.8 Choosing the number of threads per block

Number of threads per CUDA block can have significant influence on program performance. This issue was tested for all 3 test functions, dimension of 200 and population size 400. Time needed for 10 migration rounds was measured.

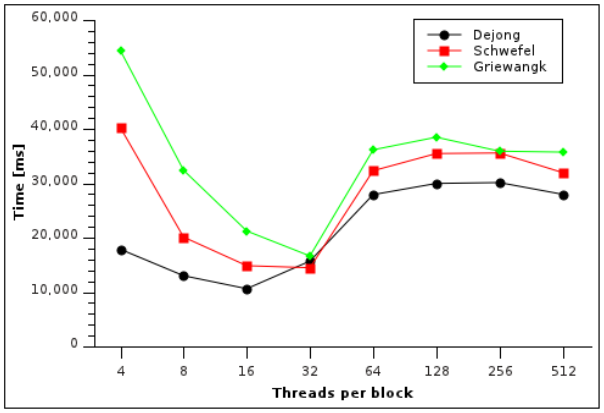

Fig. 8 Time needed for 10 migration rounds with relation to threads per block.

As can be seen in **Fig. 8**, the optimal number of threads per block is dependent on the cost function. However the best setting for these 3 functions can be chosen between 16 and 32 threads. The performance tests were run with 32 threads for all functions in order to maintain consistency between tests.

2.3 Testing the performance of CUSOMA2A

The main drive behind the modification of SOMA AllToAll for GPUs was the promise of speedup over the CPU version of algorithm.

The CUSOMA2A was tested with 3 artificial test functions and with varying dimensions and population sizes. Dimension was changing from 25 to 225 with a step of 50, the population size was changing from 100 to 500 individuals with a step of 100. The test was run for all combinations of population size / dimension with CUSOMA2A and CPU version of SOMA. The speedup over the CPU version was measured and used to model a 3D graphs.

3 Results

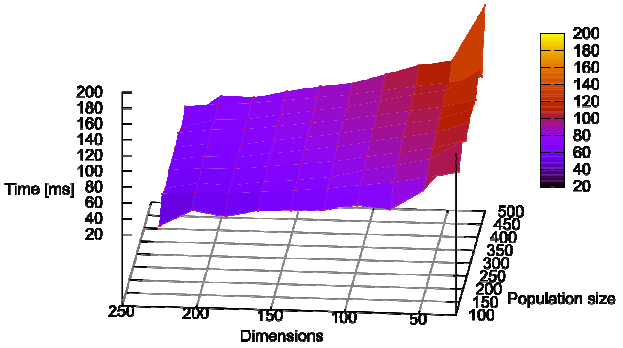

Fig. 9 Speedup over CPU version, De Jong's function

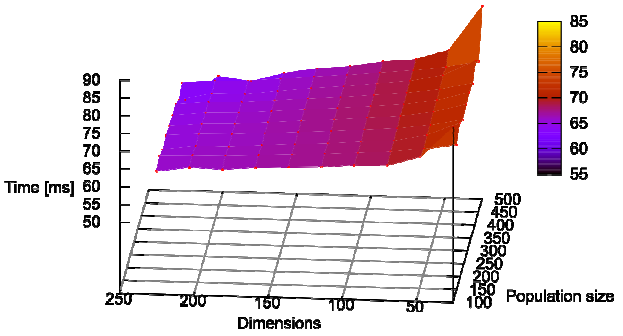

Fig. 10 Speedup over CPU version, Schwefel's function

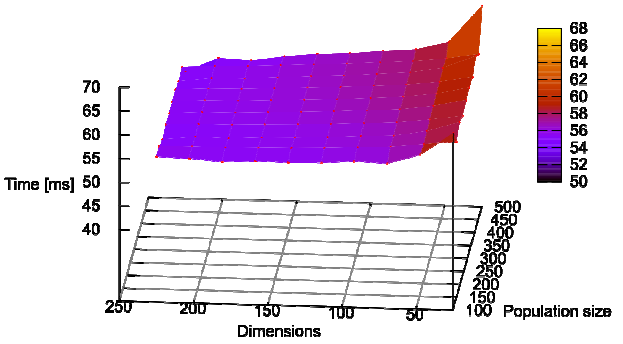

Fig. 11 Speedup over CPU version, Griewangk's function

Fig. 9, Fig. 10 and **Fig. 11** show the speedup of CUSOMA2A over the CPU version of SOMA. The speedup is significant and is in range from 26 to 190. **Table 2**and **Table 3** show the lowest and highest speedups for each test function.

Table 2 The highest values of speedup

	De Jong	Schwefel	Griewangk
Speedup	190.64	84.55	67.23
Dimensions	25	25	25
Pop. size	500	400	500

Table 3 The lowest values of speedup

	De Jong	Schwefel	Griewangk
Speedup	26.68	55.6	51.63
Dimensions	225	225	225

ISBN: 978-1-61804-117-3

Pop. size	400	500	500

Two conclusions can be made from the results. Firstly, CUSOMA2A scales very well to increase in population size and tests show consistent results across different population sizes with the same dimensions. Secondly, CUSOMA2A struggles with functions with high dimensionality, **Table 3** shows that lowest values of speedup were achieved for the highest dimensions. Both characteristics of CUSOMA2A are linked to the nature of GPU. It excels at problems which require high number of parallel yet computationally undemanding operations, but lower clock speed of CUDA cores may have problems with complex operations. Good example is comparison of De Jong's and Griewangk's functions. The lower complexity of De Jong's function allowed for much higher speedups in comparison to later. Even with these performance obstacles, the performance CUSOMA2A is superior to CPU implementation.

4 Conclusion

The AllToAll variant of self-organizing migration algorithm was successfully modified for CUDA framework. 3 different modifications were proposed and the one with the lowest run times was selected for further testing against the CPU version. Tests of the selected variant showed that number of CUDA threads per block can significantly influence performance and therefore should be tested prior to running time-critical applications. The biggest drawback of the proposed variant is its high memory usage which can be limiting for older GPUs.

The tests with 3 artificial test functions showed speedups ranging from 26.68 to 190.64 depending on test function and dimensions / population size combination. These results show that CUSOMA2A is an improvement over the CPU implementation in terms of run speed and can significantly lower times needed for optimization purposes.

Acknowledgements

The research described in this paper was supported by funding from European Regional Development Fund under project CEBIA-Tech No. CZ.1.05/2.1.00/03.0089 and by the internal grant agency project IGA/FAI/2012/033.

References:

[1] Pospichal P, Jaros J, Schwarz J, Parallel Genetic Algorithm on the CUDA Architecture, Applications of Evolutionary Computation, Springer Berlin / Heidelberg, 6024, 2010, 442-451

[2] Zhou Y, Tan Y (2009) GPU-based parallel particle swarm optimization. Proc. IEEE Congress Evolutionary Computation CEC '09, 1493-1500

[3] de Veronese LP, Krohling RA, Differential evolution algorithm on the GPU with C-CUDA, Proc. IEEE Congress Evolutionary Computation (CEC), 1-7, 2010

[4] NVIDIA CUDA C Programming Guide, [online] Available at: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf (Accessed 20 March 2012)

[5] Zelinka I, SOMA—self organizing migrating algorithm, In: New optimization techniques in engineering, Berlin: Springer 2004