Abstract

Cuckoo Search is one of the recent swarm itelligence metaheuritics. It has been succesfuly applied to a number of optimization problems but is stil not very well researched. In this paper we present a parallelized version of the Cuckoo Search algorithm. The parallelization is implemented using CUDA architecture. The algorithm is significantly changed compared to the sequential version. Changes are partialy done to exploit the power of mass parallelization by the graphical processing unit and partialy as a consequence of the memory access restrictions that exist in CUDA. Tests on standard benchmark functions show that our proposed parallized algorithm greatly decreases the execution time and achieves similar or slightly better quality of the results compared to the sequential algorithm.

Parallelization of the Cuckoo Search using CUDA Architecture

Raka Jovanovic[1] and Milan Tuba[2]

March 5, 2013

1 Introduction

In recent years a wide range of nature inspired algorithms has been developed for solving hard optimization problems. Among such algorithms swarm intelligence is becoming prominent. Swarm intelligence is collective behavior of decentralized, self-organized systems. A wide range of animal and insect species like fish, birds and ants exhibit this type of behavior where many extremely primitive individuals exhibit remarkable collective intelligence and by doing so greatly increase their chance of survival in nature.

Swarm intelligence has inspired the development of many metaheuristics for solving hard combinatorial as well as continuous optimization problems. They include Ant Colony Optimization [1], Particle Swarm Optimization [2], Artificial Bee Colony Optimization [3] etc. with numerous improvements [4], [5], [6].

A new and very promising member of the swarm intelligence metaheuristics family is the Cuckoo Search (CS) algorithm which mimics the behavior of the brood parasites [7][8]. It has not yet been thoroughly researched, but there have already been successful applications to many different problems like component design[9], training neural models [10] [11], mesh optimization [12], test data generation[13], etc . There have also been several attempts to improving its performance by adding changes to the basic algorithm [14], [15], [16], [17] or by taking special consideration to the generation of the initial population [18]. One of the most successful is the Cuckoo Optimization Method but it has a significant increase in algorithm complexity [19].

Closely related to swarm intelligence algorithms are different types of older evolutionary algorithms that are also population based. Many of the concepts used to improve performance of evolutionary algorithms can also be applied to ones based on swarm intelligence. One of the common problems is transition from global to local search i.e. moving from wide search to a fine one localized near already found good solutions [20], [21]. Another approach to improving the performance of population based algorithms is hybridization that combine more than one of such algorithms [22], [23].

Population based algorithms are very suitable for parallelization. This is due to the fact that such algorithms always contain large number of population members each conducting very similar tasks. It has been shown that even superlinear improvement compared to the sequential version of the algorithm can be achieved [24] by using the island based approach where separate colonies are executed in parallel.

Until recently massive parallelization has been reserved for supercomputers only, however nowadays with the development of the powerful Graphic Processor Units (GPU) it become available even on average personal computers. The GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational power and very high memory bandwidth. Several tools have been created for developing software exploiting the power of the GPU like NVIDIAs CUDA (Compute Unified Device Architecture) [25], Khronos Groups OpenCL (Open Computing Language) [26] and Microsoft DirectCompute which is a part of Microsoft DirectX [27].

In this paper we focus on the parallelization of the Cuckoo Search algorithm on the GPU using CUDA. Previously the parallelization of the CS has been done using multi-core processors [28], but that approach yields much lower number of threads that are executed simultaneously, compared to the possibilities of the GPU.

In our algorithm parallelization is used on three levels. First, parallel reduction is used to speedup the calculation of the fitness function for colony members. Second, all members of one colony are calculated in parallel in one block which contains several threads. Finally, several colonies are run in parallel in separate thread blocks.

The developed algorithm has been designed in a way to comply with CUDA memory access restrictions. We show in our tests that proposed approach greatly increases the speed of calculation giving similar or slightly better quality results compared to the sequential algorithm. Slightly better quality results are achieved even though we only parallelized the basic version of the algorithm and did not introduce any communication among parallel entities, which would be another possibility. The improvement is due to more systematic exploration.

The rest of the paper is organized as follows. In the next section the sequential Cuckoo Search algorithm is presented. The parallel version of this algorithm is presented in the following section. In the final section, experimental results and discussion are presented.

2 Cuckoo Search

Cuckoo search (CS) is an optimization algorithm that has been inspired by the brood parasitism of some cuckoo species. Cuckoos lay their eggs in the nests of other host birds (of other species). The shape and color of the cuckoo eggs have evolved to mimic the ones of the host. If a host bird discovers that the eggs are not their own, it will either throw these alien eggs away or simply abandon its nest and build a new one elsewhere. If the cuckoo eggs hatch, the cycle is repeated.

This type of behavior has been converted to a meta-heuristic called Cuckoo Search in the following way. Each egg in a nest represents a solution, and cuckoo egg represents a new solution. The idea is to create new, similar and potentially better solutions (cuckoos) to replace the not-so-good solutions in the nests. In the simplest form, each nest contains one egg.

CS is based on three idealized rules:

1. Each cuckoo in the colony lays one egg (solution) at a time, and dumps it in a randomly chosen nest.

2. The best solution will be carried to the next generation.

3. The number of available hosts nests is fixed, and the egg laid by a cuckoo is discovered by the host bird with a probability p_a. The discovering (discarding) operation is only done on some set of worst nests.

These rules can be converted to the standard CS algorithm given in the following pseudo code:

```
Objective function: $f(X)$, $X = (x_1, x_2, .., x_d)$

Generate an initial population of $n$ host nests;
**while** ($t < MaxGeneration$) or ($stopcriterion$) **do**
  Get a cuckoo randomly (say, $i$) and replace its solution by performing Levy
  flights;
```

```
        Evaluate its quality/fitness $F_i$

        Choose a nest among $n$ (say, $j$) randomly;
        **if** $F_i < F_j$ **then**
            Replace $j$ by the new solution;
        **end if**

        A fraction ($p_a$) of the worse nests are abandoned and new ones are built;
        Keep the best solutions/nests;
        Rank the solutions/nests and find the current best;
        Pass the current best solutions to the next generation;
    **end while**
```

Levy flight is of essential importance for the CS algorithm, it is performed using the following equation:

$$X_i(t+1) = X_i(t) + \alpha \bigotimes Levy(\lambda), \tag{1}$$

where α $(\alpha > 0)$ represents a step size. This step size should be closely related to the scale of the test function that the algorithm is applied on. In most cases, α can be set to the value of 1. Equation 1 basically represents a stochastic equation for a random walk which is a Markov chain. In it the next position (status) is dependent of two parameters: the current position $(X_i(t))$ and probability of transition $(\alpha \otimes Levy(\lambda))$. The product $\otimes$ is used for entry-wise multiplications. It has been shown that the use of Levy flight is much more efficient in exploring the search space as its step length is significantly longer when a large number of steps is performed compared to a simple random walk. The random step length is drawn from a Levy distribution which has an infinite variance with an infinite mean:

$$Levy \sim u = t^{-\lambda} \quad , \lambda \in (0, 3] \tag{2}$$

The consecutive positions generated through steps / iterations of a cuckoo, create a random walk process which obeys a power-law step length distribution with a heavy tail.

As presented in this pseudo code, the first step is to generate the initial population (set of random solutions). In the main loop of the algorithm one new solution F_i is generated using Levy flight (a heavy tailed random walk) from the appropriate nest i. This new solution is compared to a solution that corresponds to a randomly selected nest j and if the newly generated solution F_i is better, the nest becomes the same as i. This step is used to focus the search in more promising areas of the solution space. The final part of the main loop is used to diversify the search of the colony to new regions. This is done by changing the positions of p_a fraction of the worst nests using Levy flight. The Levy flight step in this stage, has a greater base step (multiplied by a scalar), compared to the one that is used when new cuckoos are generated to avoid trapping in local optimal solutions.

Some of the main advantages of CS compared to other population based methods is that it is relatively easy to implement and it has a very small number of parameters that control the method.

3 Parallelization of CS

In the parallelization of CS we consider its use on finding minimal values for standard tests like the Sphere, Rosenbrocks valley, Schwefels and other functions. These functions are mostly given in the form of the sums of functions on its elements. This makes it possible to greatly increase calculation speed using parallel reduction.

3.1 Analysis

When analyzing the sequential CS algorithm we notice two parts of it that are not natural to parallel algorithms. First is the generation of one new solution F_i from nest i and comparing it to a single solution F_j that corresponds to nest j. In the sequential algorithm this is done to minimize the number of calculated evaluations that are necessary to guide the colony to good areas of the solution space. In case of a parallel algorithm in which separate threads are dedicated to individual nests this is not an advantage for the calculation speed. The reason for this is that if the fitness function is calculated only for one nest (thread), a high level of divergence exits amongst threads. This in practical applications means that during this step all but one thread will be idle.

The second problematic part in the CS is the sorting of solutions corresponding to nests, and leaving the worst p_a fraction. In case of a parallel algorithm it is more natural to use a partial parallel reduction for the maximization problem. If parallel reduction is stopped at the third step we know that all the values that have passed to this stage are greater than at least 3 elements which means they have a high probability of belonging in the worst 25%. This makes them suitable for change using Levy flight as explained in previous section. We wish to point that it is not of essential importance in the CS that the worst p_a faction has been abandoned since this stage is just used for diversification of the search. What is important is that the best solution is passed to the next generation and that some ”relatively bad” nests are abandoned which is achieved with this reduction approach.

3.2 Implementation

The first step in implementing the parallel CS algorithm is to divide the calculations into separate threads. To do this using the GPU with CUDA architecture, we need to follow the organization of threads and memory that exist in it. In CUDA a group of threads are organized in thread blocks which have access to fast shared memory, an these blocks are organized into a grid. The separate blocks can communicate using global memory which is slow compared to the shared one. Because of this the threads are organized in the following way: separate memory blocks are dedicated to separate colonies since there is not much need for communication between them.

As mentioned in the previous section, for a single colony we wish to be able to calculate the fitness function using parallel reduction. On the other hand we wish to be able to calculate several nests at the same time. To achieve this we shall dedicate n threads to each of the m nest inside of the colony. The organization of threads in the proposed parallel algorithm can be seen in Figure 1.

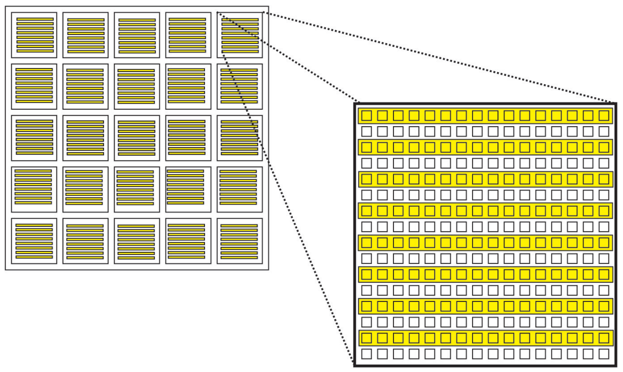

Figure 1: Organization of threads in the GPU. Left - Different thread blocks are dedicated to separate colonies. Right - In a single block (colony) each row is responsible for calculating the fitness function of a single nest

As previously mentioned, in the parallel version of the algorithm it is not an optimal approach to just select one random nest i with solution F_i and compare it to another random nest j and it's solution F_j, and if F_i is better replace j with i. It is much better to do this for a higher number of nest pairs. There are to main problems when this is attempted, bank conflicts and a very high speed of losing diversity.

In our implementation we solve the problem of bank conflicts in the following way. We consider a bank conflict if we have two pairs (i_1, j) and (i_2, j) which try to write to the same location j. In the sequential CS algorithm we use one pair of nests (i_0, j_0), and check if the fitness function corresponding to nest i_0 is greater than the one of j_0. In the case of the parallel version of the algorithm we wish to do the same with several pairs of (i_k, j_k) and avoid bank conflicts. A second requirement for the selected pairs is for them to be relatively random. The pairs will be generated in the following way, first we shall use one random variable α from which two variables will be generated using Equations 3, 4.

$$Offset = \alpha^2 \bmod BlockSize \tag{3}$$

$$Dist = 1 + (\alpha^3 \bmod \frac{BlockSize}{2}) \tag{4}$$

Variable $BlockSize$ is used to specify the number of nests existing in a colony. Variable $Dist$ is used to define the distance (difference in nest indexes), more precisely a pair will be defined $(k, (k + dist)\%BlockSize)$. Finally the nests that will be used for the initial indexes k of the pair will be the ones that satisfy the following equation

$$(k + Offset)/Dist) \bmod 2 == 0 \tag{5}$$

In Equation 5 variable $Offset$ is used to randomize the selection of the first element of the pair.

In the CS algorithm for the initial nest k, in the pair, a new test position X^*_k is generated and the corresponding fitness function F^*_k is calculated. We compare this value to the value of the fitness function of the second element of the pair F_{k+Dist}, if it is better the nest position X_{k+Dist} is overwritten by X^*_k. It is obvious that if at each iteration of the algorithm there is a large number of pairs, or in other words a large number of positions (solutions) that are overwritten, the diversity of solutions in the colony will disappear very quickly. This has a consequence that the colony is easily trapped in local optimal solutions. Because of this effect, instead of overwriting a solution we will generate new test ones using the following formula.

$$X_{k+Dist} = pX_k^* + (1-p)X_{k+Dist} \tag{6}$$

In Equation 6 p is a fixed constant from the interval $(0, 1)$. When using this approach it is important not to loose good newly generated solutions. In the case that $F_k > F_k^*$, the nest position X_k needs to be overwritten by X_k^*.

To better understand this approach we give the following pseudo code that is executed by an individual thread.

```
$i$ - thread index gives index in vector $X_j(x_1, x_2...)$
$j$ - thread index defines which nest or parameter $X_j$

Load all $X_j$ to shared memory
syncthreads();

Calculate initial function values $F_j$
using parallel reduction for nest(row of threads)
syncthreads();

**while** (Not All Evaluations Done) **do**

  calculate levi flight step $S_j$
  $X_j^* = X_j + S_j$

  Check Bounds
  syncthreads();

  Calculate $F_j^* = F(X_j^*)$ using parallel reduction for nest(row of threads)
  syncthreads();

  **if** ($F_j^* < F_j$) **then**
    $X_j = X_j + S_j$;
  **end if**
  syncthreads();

  Use partial parallel reduction for maximum of all $F_j$ (column of threads)
  **for all** nest has passed reduction **do**
    $X_j = X_j + d * S_j$
```

```
        end for
        syncthreads();

        Get block level random variables $Offset$ and $Dist$

        if $((j + Offset)/Dist) \bmod 2 == 0)$ then
            if $(F_{j+Offset} < F_j^*)$ then
                $X_{j+Offset} = p * X_{j+Offset} + (1 - p) * X_j^*$
            end if
        end if

        syncthreads();
    end while

    copy all $X_j$ from shared to Output memory
```

In the presented pseudo code operations similar to $X_j = X_j + S_j$ represent the appropriate operation that is done by the thread, more precisely the thread (i, j) will execute the following $X_j[i] = X_j[i] + S_j[i]$. This notation has been chosen to have a more clear pseudo code. The parameter d appearing in the pseudo code represents a scalar value greater than 1, that is used to generate new solutions that are far from existing ones to avoid trapping in local optimal solutions.

4 Tests and Results

In this section we compare the performance of the sequential and proposed CUDA implementation of the CS algorithm. Both algorithms are implemented using Microsoft Visual Studio 2010 combined with CUDA version 4.0 for the parallel implementation. The calculations have been done on a machine with Intel(R) Core(TM) i7-2630 QM CPU @ 2.00 Ghz, 4GB of DDR3-1333 RAM, with Nvidia GTX 540M 1GB graphics card running on Microsoft Windows 7 Home Premium 64-bit). The graphics card had 96 CUDA Cores.

Due to the restraints given by the amount of shared memory that a thread block can efficiently use, the size used in our implementation is 25x16. The optimal size of the blocks has been calculated using the CUDA occupancy calculator [29]. The block size means that each colony had 25 nests and that 16 threads are dedicated to each nest. In our tests we first compare the quality of results, to verify that the proposed approach does not degrade the performance. In the tests we have used only three of the simpler benchmark functions for which there is no necessity for fine tuning of the algorithm which is common in the improved versions of the CS. We do this since the goal of our proposed approach is to significantly increase the calculation speed and not the improvement of the quality of results compared to the sequential CS.

The sequential CS algorithm that is used for comparison is presented in the article [7]. The tests are done on the standard test functions Sphere, Rosenbrocks valley, Schwefel having 5, 10 and 16 parameters. In Table 1, we present the average and best found solutions for 25 independent runs of the algorithm with 10 000 evaluations for each test function. The values for the sequential algorithm are calculated using the Matlab code provided by Yang which has also been used in article [7].

Table 1: Comparison of quality of results achieved by sequential and parallel CS algorithm.

Function	Number of	Parallel CS			Sequential CS		
	Parameters	Average	Stdev.	Best	Average	Stdev.	Best
Sphere	5	3.54E-05	2.88E-05	4.11E-06	**9.67e-13**	1.36e-12	**2.71e-14**
Min=0	10	0.0055	0.0110	**0.0004**	**5.34e-4**	0.0021	0.0029
	16	**0.0383**	0.0312	**0.0050**	0.1356	0.0476	0.0522
Rastrigin	5	**0.7926**	0.7712	**0.0011**	2.8088	0.8766	0.6566
Min=0	10	**2.787**	2.069	**0.111**	21.962	3.896	14.583
	16	**6.526**	3.633	**1.127**	58.491	5.064	50.303
Schwefell	5	**-2051.39**	41.52	**-2092.0**	-1873.1	65.1	-2037.7
Min=	10	**-3615.0**	144.5	**-4063.5**	-3.094.3	162.9	-3434.1
-418.9*NP	16	**-5224.5**	289.6	**-5833.6**	-4272.6	239.4	-4730.1

It is noticeable that the parallel version of the CS in most of the test cases out performs the sequential version. The parallel algorithm performs worst in the case of the Sphere function which we believe is due to the fact that in our implementation the step in the Levy's flight does not decrease fast enough. The expected reason for improvement is that a more diverse solutions are generated and tested. This is due to the fact that lower quality solutions (nests) are not simply overwritten but are used in the creation of new ones. The newly created nests are a linear combination of the good and the lower quality solution, and in a sense represent a transition between them. This has a consequence that in a relatively small number of iterations, the lower quality solution will become very close to the good one, but the space between them will be also checked. The number of intermediate solutions checked is low enough not to decrease the effectiveness of the basic algorithm, but provides a more systematic exploration of the solution space.

In the second part of our experiments we analyze the calculation speed of our parallel algorithm and the sequential one. In Table 2, we compare the time needed for the execution of 25 CS when 100 000 function evaluations are conducted for each of the test functions. For these tests we have implemented the sequential algorithm given by Yang using C++.

From the results in Table 2, we can see a tremendous decrease in calculation time of even 10-25 times. The big difference in the level of speedup 10 times in the case of 5 parameters and 25 times in case of 16 parameters is due to the fact of using a fixed block size in all of our tests. This has been done for simplicity of implementation. In the case of smaller problems with only 5 parameters a large number of threads would be idle during the program execution. This dramatic speedup is similar to the case of parallelization of the PSO algorithm given in article [30]. This parallelization of PSO using CUDA also involves the use of multiple threads for function evaluation for individual colony members using parallel reduction.

5 Conclusion

In this paper we have presented a massive parallel version of the CS. The implementation is done using CUDA for execution on the GPU. The new algorithm consists of parallelization on several levels, first numerus threads are used for evaluating fitness function for individual nests in the colony, all the nests in one colony are executed in parallel and finally several colonies are simulated at the same time. The sequential algorithm has been significantly changed for its parallel implementation. The new version of the CS algorithm has been able to improve the quality of results for the same number of function evaluations. The proposed parallel algorithm has had a significant decrease in calculation time of even 25

Table 2: Comparison of execution speed of the sequential and parallel CS algorithm for 100 000 function evaluations.

Function	Number of	Execution time	
	Parameters	Sequential	Parallel
Sphere	5	15.32	1.42
	10	20.52	1.42
	16	27.34	1.43
Rastrigin	5	17.44	1.50
	10	22.21	1.50
	16	33.50	1.51
Schwefell	5	19.43	1.51
	10	24.45	1.51
	16	35.67	1.52

times compared to the sequential CS.

Acknowledgement

The authors acknowledge the financial support by the Serbian Ministry of Science and Education Project 171006. The author is grateful to the anonymous referee for a careful checking of the details and for helpful comments that improved this paper.

References

[1] M. Dorigo et L.M. Gambardella, Ant Colony System : A Cooperative Learning Approach to the Traveling Salesman Problem, IEEE Transactions on Evolutionary Computation, Vol. 1, No. 1, 1997, pp. 53-66

[2] J. Kennedy, R.Eberhart, Particle Swarm Optimization. Proceedings of IEEE International Conference on Neural Networks. IV, 1995, pp. 19421948

[3] D. Karaboga,B. Akay, A Comparative Study of Artificial Bee Colony Algorithm, Applied Mathematics and Computation, vol 214, 2009, pp. 108-132

[4] R.Cheng, M. Yao, Particle Swarm Optimizer with Time-Varying Parameters based on a Novel Operator Applied Mathematics & Information Sciences, Vol. 5, No. 2 Special Issue: SI, MAR 2011, pp. 33-38

[5] R. Jovanovic, M.Tuba, An ant colony optimization algorithm with improved pheromone correction strategy for the minimum weight vertex cover problem, Applied Soft Computing, Vol. 11, No 8, 2011, pp. 5360-5366

[6] I. Brajevic, M. Tuba, An upgraded artificial bee colony algorithm (ABC) for constrained optimization problems, Journal of Intelligent Manufacturing, published Online First, 2012, DOI: 10.1007/s10845-011-0621-6

[7] Yang, X. S. and Deb, S., Cuckoo search via Levy flights, in: Proc. of World Congress on Nature & Biologically Inspired Computing (NaBIC 2009), 2009, pp. 210-214

[8] Yang, X. S., and Deb, S. Engineering Optimisation by Cuckoo Search, Int. J. of Mathematical Modelling and Numerical Optimisation, Vol. 1, No. 4, 2010, pp. 330343.

[9] I. Durgun, A. R. Yildiz, Structural Design Optimization of Vehicle Components Using Cuckoo Search Algorithm, MP Materials Testing, No.3, 2012, pp. 185-188

[10] R.A. Vazquez, Training spiking neural models using cuckoo search algorithm, IEEE Congress on Evolutionary Computation (CEC), 2011 , pp. 679-86

[11] Ehsan Valian, Shahram Mohanna and Saeed Tavakoli, Improved Cuckoo Search Algorithm for Feedforward Neural Network Training, International Journal of Artificial Intelligence & Applications (IJAIA), Vol.2, No.3, 2011, pp. 36-43

[12] S. Walton, O. Hassan, K. Morgan,Reduced order mesh optimisation using proper orthogonal decomposition and a modified cuckoo search, International Journal for Numerical Methods in Engineering, DOI: 10.1002/nme.4400, 2012

[13] P. R. Srivastava, R. Khandelwal,S. Khandelwal, K. Shobhit, S. S. Sanjay, Automated Test Data Generation Using Cuckoo Search and Tabu Search

(CSTS) Algorithm, Journal of Intelligent Systems, Vol. 21, No. 2, 2012, pp 195224

[14] N. Bacanin, An object-oriented software implementation of a novel cuckoo search algorithm, Proc. of the 5th European Conference on European Computing Conference (ECC'11), 2011, pp. 245-250

[15] M. Tuba, M. Subotic, and N. Stanarevic, Modified cuckoo search algorithm for unconstrained optimization problems, Proc. of the 5th European Conference on European Computing Conference (ECC'11), 2011, pp. 263-268

[16] M. Tuba, M. Subotic, N. Stanarevic, Performance of a modified cuckoo search algorithm for unconstrained optimization problems, WSEAS Transactions on Systems, Vol. 11, No. 2, 2012, pp. 62-74

[17] S. Walton, O. Hassan, K. Morgan and M. R. Brown, Modified cuckoo search: A new gradient free optimisation algorithm, Chaos, Solitons & Fractals. Vol 44, Is. 9, 2011, pp. 710718

[18] M. Shatnawi, Starting configuration of Cuckoo Search algorithm using Centroidal Voronoi Tessellations, 11th International Conference on Hybrid Intelligent Systems (HIS), 2011, pp. 40-45

[19] R. Rajabioun, Cuckoo Optimization Algorithm, Applied Soft Computing, Vol. 11, Is. 8,2011, pp. 5508–5518

[20] A.C. Martinez-Estudillo, C. Hervas-Martinez, F.J. Martinez-Estudillo, N. Garcia-Pedrajas, Hybridization of Evolutionary Algorithms and Local Search by Means of a Clustering Method, IEEE Transactions on Systems, Man and Cybernetics, Part B: Cybernetics, Vol. 36, No. 3, 2006, pp. 534-545

[21] Hecheng Li,Yuping Wang, , An Evolutionary Algorithm with Local Search for Convex Quadratic Bilevel Programming Problems Applied Mathematics & Information Sciences, Vol. 5, No. 2, 2011, pp. 139-146

[22] Jinghui, Gao, Rui Shan, A New Method for Modification Consistency of the Judgment Matrix Based on Genetic Ant Algorithm Applied Mathematics & Information Sciences, Vol. 6, Special Issue: 1, 2012, pp. 35-39

[23] N. Bacanin, M. Tuba, Artificial Bee Colony (ABC) Algorithm for Constrained Optimization Improved with Genetic Operators, Studies in Informatics and Control, Vol 21, Issue 2, 2012, pp. 137-146

[24] Thomas Sttzle, Parallelization strategies for Ant Colony Optimization, Parallel Problem Solving from Nature PPSN V, Springer Berlin/Heidelberg, 1998, pp. 722-731

[25] NVIDIA CUDA C Programming Guide Version 4.0, 2011.

[26] P.O. Jaaskelainen, C.S. de La Lama, P. Huerta, and J.H. Takala, OpenCL-based design methodology for application-specific processors, *Embedded Computer Systems (SAMOS), 2010 International Conference on* , 2010, pp. 223–230.

[27] DirectCompute Lecture Series, http://channel9.msdn.com/tags/DirectCompute-Lecture-Series/, 2011

[28] M. Subotic, M. Tuba, N. Bacanin, and D. Simian, Parallelized cuckoo search algorithm for unconstrained optimization, Proceedings of the World Congress: Applied Computing Conference (ACC'12), University of Algarve, Faro, Portugal, 2012, pp. 151–156

[29] CUDA Occupancy Calculator, http:// news.developer.nvidia.com /2007 /03 /cuda _occupancy _.html/, 2007

[30] Luca Mussi, Fabio Daolio, Stefano Cagnoni, Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture Information Sciences, 2011, pp. 4642-4657