Cellular Genetic Algorithm on Graphic Processing Units

Pablo Vidal and Enrique Alba

Abstract. The availability of low cost powerful parallel graphic cards has estimulated a trend to implement diverse algorithms on Graphic Processing Units (GPUs). In this paper we describe the design of a parallel Cellular Genetic Algorithm (cGA) on a GPU and then evaluate its performance. Beyond the existing works on master-slave for fitness evaluation, we here implement a cGA exploiting data and instructions parallelism at the population level. Using the CUDA language on a GTX-285 GPU hardware, we show how a cGA can profit from it to create an algorithm of improved physical efficiency and numerical efficacy with respect to a CPU implementation. Our approach stores individuals and their fitness values in the global memory of the GPU. Both, fitness evaluation and genetic operators are implemented entirely on GPU (i.e. no CPU is used). The presented approach allows us benefit from the numerical advantages of cGAs and the efficiency of a low-cost but powerful platform.

Keywords: Cellular Genetic Algorithm, Parallellism, GPGPU, CUDA.

1 Introduction

Cellular Genetic Algorithms (cGAs) are effective optimization techniques solving many practical problems in science and engineering [1]. The basic algorithm (cGA)

Pablo Vidal
LabTEm - Laboratorio de Tecnologías Emergentes, Unidad Académica Caleta Olivia,
Universidad Nacional de La Patagonía Austral,
Ruta 3 Acceso Norte s/n, (9011) Caleta Olivia Sta. Cruz - Argentina
Phone/Fax: +54 0297 4854888
e-mail: pablo.vidal.20@gmail.com

Enrique Alba
Dept. de Lenguajes y Ciencias de la Computación, University of Málaga, ETSI Informática,
Campus de Teatinos, Málaga - 29071, Spain
e-mail: eat@lcc.uma.es

J.R. González et al. (Eds.): NICSO 2010, SCI 284, pp. 223–232, 2010.
springerlink.com

is selected here because of its high performance and because of its swarm intelligence structure (i.e. emergent behavior and decentralized control flow). By evolving this kind of algorithm is able of keeping a high diversity in the population until reaching the region containing the global optimum. This kind of algorithms may benefit from parallelism as a way of speeding up its operations [2] when the instance of the problem is complex.

Graphic Processing Units (GPUs) are well-known hardware cards with a fixed function, being traditionally used for visualization purposes. However, the new generations of GPUs have also unleashed a promising potential for scientific computing, seen as a new hardware allowing the use of high arithmetic capacity and high performance.

Thus, researchers and developers have begun to harness GPUs for general purpose computation [4] [7]. In addition to their low cost and ubiquitous availability, GPUs have a superior processing architecture when compared to modern CPUs, and thus present a tremendous opportunity for developing lines of research in optimization algorithms especially targeted for GPUs, this its shown in present works such as [3] [5].

Therefore, we work here with a parallel cGA running entirely on GPU (i.e. no CPU is needed only to start and stop the algorithm), and demonstrate that the proposed optimization technique (called cGA GPU) is quite amenable for massive parallelism to obtain larger performances with reduction of times and improvements of the speedup. This approach offers the possibility to solve larges problem instances with the improved computing capacity of a GPU. All this will be shown on a benchmark of discrete and continuous problem to claim not only for time reductions but also for numerical advantages of this swarm intelligence algorithm.

The paper is structured as follows, The next section contains some background about the parallelism, we explain the Cellular Genetic Algorithm and its implementation in GPU. Section 3 describes the experimental setup, while Section 4 explains the test problems used, details of the cGA parameters, and the statistical tests performed.

Finally, Section 5 provides the obtained results and Section 6 offers our conclusions, as well as some comments on the future work.

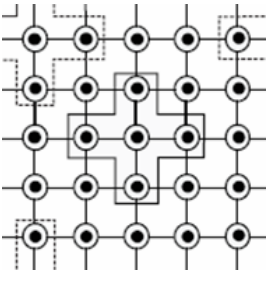

Fig. 1 Toroidal structure of a cGA population

2 Description of a Cellular Genetic Algorithm

Cellular GAs (cGAs) are a subclass of Genetic Algorithm (GAs) in which the population is structured in a specified topology defined as a connected graph, 2D toroidal grid, in which each vertex is an individual that communicates with its nearest neighbours (e.g, North, South, East, West) and use these individuals for crossover and mutation. Algorithm 1 (and Figure 1) presents the structure of a cGA.

Each individual interacts only with their neighbours. The resulting overlapped small neighbourhoods help in exploring the search space because the induced slow diffusion of solutions through the population provides a kind of exploration, while exploitation takes place inside each neighbourhood by genetic operations. The reader can find a deeper estudy on cGAs in [1].

Algorithm 4. Pseudocode of Canonical Cellular GA

```
1: pop ← initializePopulation(pop)
2: pop ← evaluatePopulation(pop)
3: while not stop criterion do do
4:    for each individual do do
5:       neighbours ← calculateNeigbourhood(individual)
6:       parents ← selection (neighbours)
7:       offspring ← Recombination(parents,prob_Recombination);
8:       offspring ← Mutation(offspring,prob_Mutation);
9:       pop' ← evaluate(offspring);
10:      replacement(pop',individual,offspring);
11:   end for
12: end while
```

2.1 The Proposal

The basic idea behind most parallel programs is to divide a task into subtasks and solve the subtasks simultaneously using multiple processors. This divide and conquer approach can be applied to GAs in many different ways, and the literature contains many examples of successful parallel implementations [2]. Some parallelization methods use a single population, while others divide the population into several relatively isolated subpopulations. Some methods exploit massively parallel computer architectures, while others are better suited to multicomputers with fewer and more powerful processing elements.

In the case of NVIDIA GPUs have (currently) up to 30 Streaming Multiprocessors (SM); each SM has eight parallel thread processors called Streaming Processors (SP). The SPs run synchronously, meaning all eight SPs run a copy of the same program, and actually execute the same instruction at the same time by each thread created (see Figure 2). Different SMs run asynchronously, much like commodity multicore processors. For achieving this, the notion of kernel is defined. A kernel is a function callable from the host and executed on the specified GPU simultaneously by several SPs in parallel.

Algorithm 5. Pseudocode of Cellular GA on a GPU

```
1: initialize_cGA(Input_param)
2: generate_random_numbers(seeds)
3: allocate problems, seeds for random numbers and data inputs on GPU device memory
4: for each individual in parallel do do
5:     individual ← initializeOnGPU(individual)
6:     individual ← evaluateOnGPU(individual)
7: end for
8: while not Stop Criterion do do
9:     neighbours ← calculateNeigbourhoodOnGPU(individual)
10:    parents ← selectionOnGPU (neighbours)
11:    offspring ← RecombinationOnGPU(parents,prob_Recombination);
12:    offspring ← MutationOnGPU(offspring,prob_Mutation);
13:    evaluateOnGPU(offspring);
14:    replacementOnGPU(individual,offspring);
15: end while
```

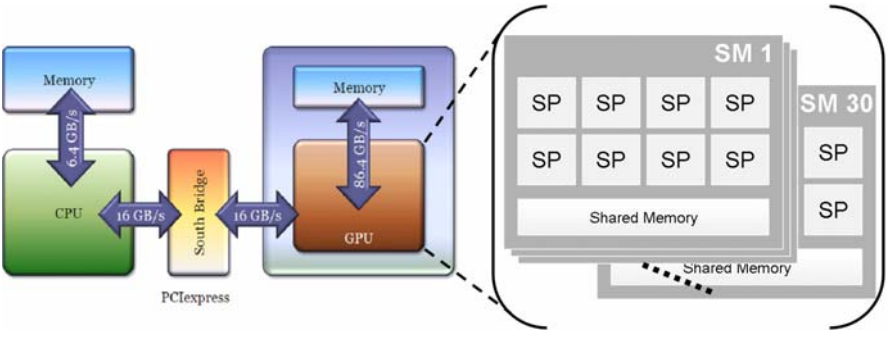

Fig. 2 Description of the architecture between CPU and GPU

In our present work the proposed algorithm exploits the inherent parallelism of a GPU using a direct mapping between the population structure and the threads of the GPU. First of all, at initialization stage, the memory allocations on GPU have to made. Input parameters for the algorithm (the population generated in the CPU and the configuration parameters for the algorithm), are stored in the global memory of the GPU. The population generated is transfered from the CPU to the device memory, this is a synchronous operation. Since we are not having a Pseudo Random Number Generator (PRNG) for GPUs, we used a PRNG that is provided by the SDK of CUDA named Merseinne Twister; the only condition for its use is to initially copy from CPU to GPU a group of seeds neccesary for execute the PRNG. Once the copies are done, we execute a series of subtasks implemented only in the GPU called through a kernel function (that allows to invoke functions implemented in the GPU) and these are executed for every thread. As a second step, for each individual, we need to identify its neighrbourhood. Third, we proceed to apply the GA operators on the solution neighbourhood in each thread. Now, we synchronize

all threads for taking the fourth step: replacement of the individual with the offspring (if a condition is satisfied). Finally, this process is repeated until a stop condition is satisfied. This algorithm is synchronous, as the individuals of the population of the next generation are formally created all at the same time. We can see a general model of the proposal algorithm for GPU in the Figure 3.

The implementation for this algorithm was done with CUDA [6] for GPU .

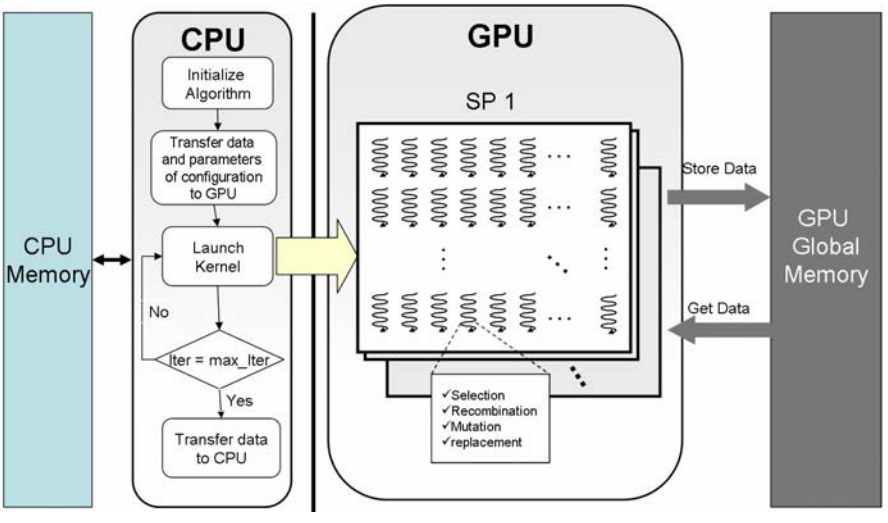

Fig. 3 Description of the architecture between CPU and GPU

3 Experimental Setup

This section is devoted to describing the methodology that we have used in the experiments carried out in this work. First, we present the benchmark problems used to compare the cGA GPU. In order to show the performance on a wide spectrum of problems we encompass tests both in discrete and continuous domains. Also, we try to use standard benchmarks as the ones reported in CEC 2005 [8] and 2008 [9] standards.

4 Methodology and Configurations Used

We have selected for our tests the following problems: Colville Minimization, ECC, MMDP (discrete optimization) and Shifted GriewankĄfs function, Shifted RastriginĄfs function and Shifted RosenbrockĄfs function (continuous optimization). These problems were selected because they are generally popular in GAs and/or used in previous works on GPUs [1] [12].

Our GPU-based implementation is compared against previous software implementations on a CPU implemented in JCell [1].

Now, we explain the statistical test that we have applied to ensure the confidence of the obtained results. Since we are dealing with stochastic algorithms and we want to provide the results with confidence, we have made 30 independent runs of each experiment, and the following statistical analysis has been performed throughout this work. Firstly, a Kolmogorov Sminorv test was performed in order to check whether the values of the result follow a normal (Gaussian) distribution or not. If the distribution is normal, we will apply Levene test for the homogeneity of the variances. If samples have equals variance (positive Levene test), an ANOVA test is performed, otherwise a Welch test is performed. For non Gaussian distributions, the non-parametric Kruskal-Wallis test is used to compare the medians of the algorithms.

We always consider in this work a confidence level of 95% (i.e., significance level of 5% or p-value under 0.05) in the statistical tests, which means that the differences are unlikely to have occurred by chance with a probability of 95%. Successful tests are marked with "+" symbols in the last column in the first table; conversely, "•" means that no statistical confidence was found (p-value 0:05).

In order to make a meaningful comparison among the algorithms, we have used a common parameterization. The details are described in Table 1, where we include the maximum number of generations as the stop condition for all the algorithms in each execution (500). The toroidal grid has different sizes for evaluate the behavior of the algorithms and compare that exist some advantage or not to use different population sizes for each problem. So, we define four population sizes: 32×32, 64×64, 256×256 and 512×512 individuals. The neighbourhood used is composed of five individuals: the considered individual plus those located at its North, East,West and South (see Fig. 1). One selection method have been used in this work: on parent is always the cosidered individual itself, while the other one is obtained by using Roulette Wheel (RW) selection in its 4-neighbourhood. For the recombination operator, we obtain just one offspring from two parents: the one having the largest portion of the best parent. The DPX recombination is applied always (probability $p_c = 1.0$), this operator is a crossover of two points, keeping the largest part of the best parent. The bit mutation probability is set to $p_m = 0.05$. We will replace the considered individual on each generation only if its offspring has a better fitness value, called Replace if Better [11]. All these parameters are selected after previous works [1] and an own initial setting study.

Table 1 SpeedUp in seconds obtained with different population size

Parameters	**Value**
Max. Number of Generations	500
Population Size	$\{32^2, 64^2, 256^2, 512^2\}$
Neighborhood	N - E - W - S
Selection of Parents	itself + Roulette Wheel (RW)
Recombination	DPX = 1.0
Mutation	0.05
Replacement	Replace if the new individual is better

Experiments were run on a machine equiped with a Intel R . CoreTM Quad processor running at 2.67GHz , under Windows XP operating system, and having 4 GB of memory. The GPU used is a nVIDIA GeForce GTX 285 equipped with 1GB of RAM. The environment used is Microsoft Visual C++ 2008 Express Edition together with the Toolkit SDK for CUDA v2.1 with the nVIDIA driver v180.49.

5 Results

In this section we present the results obtained when solving the problems selected with the proposed cGA GPU algorithm. We here describe the numeric and time performance of the cGA on GPU. In order to compare the time performance we use a sequential version of a cGA implemented in JCell [1] that is executed in the CPU.

The results of speedup are summarized in Table 2: for each problem, the average speedup of the 30 executions is shown. This value is the result of the average of the time for the algorithm in CPU divided by the average time of the algorithm on GPU. Thus, a value over 1.0 means a more efficient performance of the GPU versus the CPU.

The results of our tests show that the speedup ranges from 5 to 24. In general, as the population size increase we see that the GPU can achieve a better performance.

For a population space as 32×32, the CPU implementation still remains faster than those in a GPU; the reason is probably because the population are very small and the existency the some overhead between the CPU and GPU to call the kernel functions affects the time performance. We would like to point out that the efficiency showed for the GPU is equivalent to 24 proccessors, an a insteresting benefit drawn from a commodity computer.

Another interesting observation is that there is not significant difference between the speedup of the discrete and continuous domains. This indicates that the GPU is effective to evaluate problem instances of both domains.

The result of the statistical tests are in column Test of the Table 2, where the symbol "+" means that statiscally significant differences exist. In most of the instances of the problems, the existing statistically significant differences favor the cGA implemented in the GPU versus the CPU. As well, Table 3 gives for each problem the time (in seconds) of the algorithm executed in CPU and in GPU respectively (each column shows the time of CPU and the GPU time separated by a "$-$"). As expected, the time of the GPU is very small (between 0.14 and 0.35 seconds) while for the CPU the execution time range between 0.11 and 7.89 seconds. In most of the cases, the time of the GPU is shorter than the one on CPU (an exception occured just for the population of 32×32). Table 5 gives results about of the average of fitness solutions obtained for each problem. This table shows the average value and the standard deviation of the averaged best final fitness value for each problem and algorithm configuration. The values obtained show that the algorithm gets very frequently a near optimal value for every problem. Also, those values are competitive against other algorithms in the literature [10]. Table 4 show the results of the average of fitness solutions obtained in CPU. This table show that the values obtained

Table 2 SpeedUp in seconds obtained with different population sizes

Population	SpeedUp Discrete Problems			SpeedUp Continuos Problems			Test
	Colville Minimization	ECC	MMDP	Rastrigin	Rosenbrock	Griengwak	
32×32	0.561	0.660	0.784	0.494	0.539	0.826	•
64×64	5.441	5.450	5.645	5.417	5.688	5.783	+
256×256	16.433	16.830	14.485	15.463	17.830	16.964	+
512×512	23.593	22.419	22.789	20.810	20.982	20.421	+

Table 3 Average of time performance in seconds with different population sizes

Population	SpeedUp Discrete Problems			SpeedUp Continuos Problems		
	Colville Minimization	ECC	MMDP	Rastrigin	Rosenbrock	Griengwak
32×32	0.10-0.18	0.11-0.17	0.11-0.14	0.12-0.22	0.11-0.19	0.12-0.15
64×64	1.19-0.21	1.25-0.23	1.21-0.21	1.17-0.21	1.20-0.21	0.21-0.20
256×256	4.16-0.25	4.63-.0.27	4.09-0.28	4.31-0.27	4.66-0.26	4.53-0.26
512×512	7.46-0.32	7.89-0.35	7.66-0.33	7.21-0.34	7.12-0.33	7.35-0.35

Table 4 Average of solutions fitness obtained with different population sizes for CPU

Population	Average Solutions			Average Solutions Continuos Problem		
	Colville Minimization	ECC	MMDP	Rastrigin	Rosenbrock	Griengwak
32×32	$0.133_{\pm 5.176e\text{-}5}$	$0.066_{\pm 1.065e\text{-}3}$	$39.896_{\pm 8.709e\text{-}6}$	$4.637e\text{-}5_{\pm 5.512e\text{-}5}$	$2.600e\text{-}5_{\pm 3.075e\text{-}5}$	$3.733e\text{-}5_{\pm 3.750e\text{-}3}$
64×64	$0.111_{\pm 3.684e\text{-}6}$	$0.066_{\pm 0.633e\text{-}3}$	$39.900_{\pm 9.145e\text{-}6}$	$2.978e\text{-}5_{\pm 1.136e\text{-}6}$	$1.645e\text{-}6_{\pm 5,170e\text{-}6}$	$2.687e\text{-}5_{\pm 3.410e\text{-}3}$
256×256	$0.100_{\pm 1.033e\text{-}6}$	$0.067_{\pm 0.361e\text{-}6}$	$39.911_{\pm 1.365e\text{-}6}$	$1.218e\text{-}5_{\pm 7.872e\text{-}6}$	$1.639e\text{-}5_{\pm 2.816e\text{-}6}$	$2.350e\text{-}5_{\pm 3.590e\text{-}6}$
512×512	$0.010_{\pm 0.310e\text{-}6}$	$0.067_{\pm 0.003e\text{-}6}$	$39.999_{\pm 2.713e\text{-}5}$	$1.749e\text{-}6_{\pm 4.350e\text{-}6}$	$3.311e\text{-}5_{\pm 4.997e\text{-}6}$	$1.356e\text{-}6_{\pm 1.450e\text{-}6}$

Table 5 Average of solutions fitness obtained with different population sizes for GPU

Population	Average Solutions			Average Solutions Continuos Problem		
	Colville Minimization	ECC	MMDP	Rastrigin	Rosenbrock	Griengwak
32×32	$0.330_{\pm 9.660e\text{-}2}$	$0.065_{\pm 0.865e\text{-}3}$	$39.590_{\pm 1.070}$	$4.850e\text{-}5_{\pm 2.970e\text{-}5}$	$2.600e\text{-}5_{\pm 9.330e\text{-}5}$	$3.733e\text{-}5_{\pm 3.750e\text{-}3}$
64×64	$0.330_{\pm 3.122e\text{-}2}$	$0.066_{\pm 0.633e\text{-}3}$	$39.720_{\pm 0.080}$	$4.560e\text{-}5_{\pm 6.810e\text{-}5}$	$1.645e\text{-}5_{\pm 7.360e\text{-}5}$	$2.687e\text{-}5_{\pm 3.410e\text{-}3}$
256×256	$0.130_{\pm 1.030e\text{-}3}$	$0.066_{\pm 0.361e\text{-}5}$	$39.860_{\pm 0.080}$	$4.540e\text{-}5_{\pm 6.330e\text{-}5}$	$1.639e\text{-}5_{\pm 3.870e\text{-}5}$	$2.391e\text{-}5_{\pm 1.830e\text{-}3}$
512×512	$0.100_{\pm 1.000e\text{-}3}$	$0.067_{\pm 0.003e\text{-}5}$	$39.940_{\pm 8.000e\text{-}3}$	$4.210e\text{-}5_{\pm 1.090e\text{-}5}$	$1.500e\text{-}5_{\pm 3.600e\text{-}5}$	$2.375e\text{-}5_{\pm 1.050e\text{-}3}$

for the CPU and GPU are very similar with a approximation very similar. As a conclusion, the algorithm implemented in GPU presents a robust numerical behavior because the values are very near or they reached the optimal. So, we can conclude that in general the cGA GPU is better than the sequential cGA, bothnumerically and in time.

6 Conclusions

In this work we have presented a novel implementation of a cGA running on a GPU. All operators have been implemented directly in the GPU. We test the performance of the algorithm with 6 different problems in continuous and discrete domain, and we compare against a standard cGA. We showed that the inherent parallelism of the GPU can be exploited to accelerate a cGA.

In the future, we will apply the presented approach to other complex real-world problems. Especially those that remains open because at their large dimensions, as well as to applications in industry. Another future work will be to implement other families of evolutionary algorithms and evaluate its performance in multiGPU architectures.

Acknowledgements. Authors acknowledge funds from the Spanish Ministry of Sciences and Innovation European FEDER under contract TIN2008-06491-C04-01 (M* project `http://mstar.lcc.uma.es`) and CICE, Junta de Andalucía under contract P07-TIC-03044 (DIRICOM project `http://diricom.lcc.uma.es`).

References

[1] Alba, E., Dorronsoro, B.: Cellular Genetic Algorithms. Operations Research / Computer Science, vol. 42. Springer, Heidelberg (2008)
[2] Alba, E.: Parallel metaheuristics: A new class of algorithms (August 2005)
[3] Lewis, T.E., Magoulas, G.D.: Strategies to minimise the total run time of cyclic graph based genetic programming with gpus (2009)
[4] Luebke, D., Harris, M., Krüger, J., Purcell, T., Govindaraju, N., Buck, I., Woolley, C., Lefohn, A.: Gpgpu: general purpose computation on graphics hardware. In: SIGGRAPH 2004: ACM SIGGRAPH 2004 Course Notes, vol. 33. ACM, New York (2004)
[5] Maitre, O., Baumes, L.A., Lachiche, N., Corma, A., Collet, P.: Coarse grain parallelization of evolutionary algorithms on gpgpu cards with easea (2009)
[6] Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with cuda. In: SIGGRAPH 2008: ACM SIGGRAPH 2008 classes, pp. 1–14. ACM, New York (2008)
[7] Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26(1), 80–113 (2007)
[8] Suganthan, P.N., Hansen, N., Liang, J.J., Deb, K., Chen, Y.-P., Auger, A., Tiwari, S.: Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization (2005)
[9] Tang, K., Yao, X., Suganthan, P.N., MacNish, C., Chen, Y.P., Chen, C.M., Yang, Z.: Benchmark functions for the CEC 2008 special session and competition on large scale global optimization (November 2007)

[10] Tseng, L.-Y., Chen, C.: Multiple trajectory search for large scale global optimization. In: Evolutionary computation, CEC 2008 (IEEE World Congress on Computational Intelligence). IEEE Congress (2008)
[11] Whitley, D.L.: The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In: Proceedings of the 3rd international conference on genetic algorithms (1989)
[12] Yu, Q., Chen, C., Pan, Z.: Parallel genetic algorithms on programmable graphics hardware. In: Wang, L., Chen, K., S. Ong, Y. (eds.) ICNC 2005, part III. LNCS, vol. 3612, pp. 1051–1059. Springer, Heidelberg (2005)