Communications in Advanced Mathematical Sciences
Vol. I, No. 1, 67-83, 2018

A metaheuristic optimization algorithm for multimodal benchmark function in a GPU architecture

Javier L. Mroginski[1]*, H. Guillermo Castro[1]

Abstract
It is well known that the numerical solution of evolutionary systems and problems based on topological design requires a high computational power. In the last years, many parallel algorithms have been developed in order to improve its performance. Among them, genetic algorithms (GAs) are one of the most popular metaheuristic algorithms inspired by Darwin's evolution theory. From the High Performance Computing (HPC) point of view, the CUDA environment is probably the parallel computing platform and programming model that more heyday has had in recent years, mainly due to the low acquisition cost of graphics processing units (GPUs) compared to a cluster with similar functional characteristics. Consequently, the number of GPU-CUDAs present in the top 500 fastest supercomputers in the world is constantly growing. In this paper, a numerical algorithm developed in the NVIDIA CUDA platform capable of solving classical optimization functions usually employed as benchmarks is presented. The obtained results demonstrate that GPUs are a valuable tool for acceleration of GAs and may enable its use in much complex problems. Also, a sensitivity analysis is carried out in order to show the relative weight of each GA operator in the whole computational cost of the algorithm.

Keywords: CUDA environment, Genetic algorithm, Mathematical function optimization, GPU architecture
2010 AMS: Primary 68T20, Secondary 80M50

[1] *Argentine Council for Science and Technology (CONICET) and Computational Mechanics Laboratory (LAMEC) - Northeast National University (UNNE) Av. Las Heras 727 (C.P. 3500), Resistencia - Chaco - Argentina*
***Corresponding author**: javiermro@gmail.com
Received: 12 July 2018, **Accepted:** 19 September 2018, **Available online:** 30 September 2018

1. Introduction

It is well known that Genetic Algorithms (GAs) are robust and efficient domain-independent searching techniques for global optimization problems inspired by the Darwinian evolution theory. Most classical GAs include three different operators: selection, mutation and crossover. In each iteration, individuals are evaluated using the objective (or fitness) function and a stopping criteria is also required in order to end the iterative process [5]. However, despite GA is very useful for many practical optimization problems, the execution time can become a limiting factor for some huge problems [4, 6, 9].

There are many possibilities to accelerate a GA code [1]. One of the most widespread strategies proposed in order to reduce the computational cost are the MPI-based parallelization techniques. In this case, a cluster computer is required [19, 20]. Probably, the main problem in the near future will be the platforms heterogeneity of data and applications. However, since the CPU maximum frequency seems to be reached [8], many-core parallel techniques appear to be an interesting option.

During the last decade, Graphic Processing Unit (GPU) has been used for computing acceleration due to the intrinsic

vector-oriented design of the chip set. This gave race to a new programming paradigm: the General Purpose Computing on Graphics Processing Units (GPGPU) [11, 16, 12]. In general, the GPGPU acronym is used for all techniques able to develop algorithms extending computer graphics applications but running on a GPU. This paradigm is widely used for a very large range of applications, in computational fluid dynamics (CFD) problems [24], environmental applications [2], advection transport [7], numerical analysis [22], optimization techniques [21, 29], among others.

Later on, the GPGPU programming paradigm was replaced by the Compute Unified Device Architecture (CUDA) in 2007. CUDA has several advantages compared to GPGPU and CPU which includes faster memory sharing and read backs, minimal threads creation overhead, etc. [17]. Since GAs are inherently parallel, the CUDA computing paradigm provides an interesting framework, allowing a strong optimization in terms of processing performance and scalability [15, 25, 23].

In this paper an open source code written in the C++ programming language that allows the optimization of n-dimensional space functions using GA classical metaheuristic techniques is developed. In order to reduce the computational cost of metaheuristic optimization, a parallelization of a classic GA code using a GPU in a CUDA environment is also proposed. Moreover, a sensitivity analysis is performed in order to identify the relative computational cost of each GA operator.

Three different objective functions are employed, which are often used as benchmark functions in optimization problems: De Jong's function, Rastrigin's function and Ackley's function. The global optimum is the same and is known for all test cases: this allows to make comparisons from the GA performance point of view. The paralellized algorithm propoed in this work is capable of optimize n-dimensional functions using CUDA programming paradigm. The obtained results using a GeForce GTX 750 Ti GPU show that the proposed code is a valuable tool for accelerating GAs, reducing its computational cost by about 92%.

The article is structured as follows. Section 2 presents some characteristics of classical GAs as well as the main features of the GA proposed in this work. Section 3 describes the characteristics of the employed hardware and some key aspects of the parallelization strategy using CUDA. The results are shown in Section 4 while the conclusions and final discussions are presented in Section 5.

2. Classical genetic algorithm

Based on the Darwin's evolution theory, genetic algorithms are analogous to the natural selection laws and survival of strongest individuals. Therefore, the individuals with higher fitness in a population have a greater chance of survival than the others. Each individual is a candidate solution of the optimization problem and its quality is assessed by evaluating an objective (or fitness) function which has to be minimized or maximized.

2.1 General characteristics

Some features of classical genetic algorithms are [5, 14]:

- Individuals can be defined with arrays of integer, real or binary numbers as well as a combination.
- The iterative process is usually known as *elitist algorithm*. Thus, the better adapted individuals pass to the next generation without going through the crossover and mutation procedures.
- Continuous functions for individual definition allow imposing conditions on lower and upper bounds of the variables in order to fulfill the objective function domain.
- Partial renewal of the population in order to prevent the saturation with the best individuals. In this step, the mutation procedure is no longer required.

Figure 1.1a shows the pseudocode of the GA developed in this article. Also, the main features of the classical GA are included, being *pop*, *sel* and *shoot* three entities accounting population, selection operator and the randomized shooting procedure, respectively.

2.2 Characterization of individuals

In Fig. 2.1, a generic coordinate x_i is presented. Therefore, each generic coordinate of an individual is composed by *nvdec* genes and *nvbin* chromosomes. The first gene is used for sign assignation. Thus, the negative sign of a generic coordinate is adopted when the first digit is less than five, otherwise the obtained coordinate is positive. Form example, Fig. 2.2 shows the determination of a real number from a binary random matrix.

Although individuals are defined from a binary matrix of $nvbin * nvdec$ elements, in the GA code developed in this work the population is organized as a vector of size $psize * nvbin * nvdec$ in order to optimize memory access. According to this definition of individuals, the domain of the generic coordinate is setting automatically in $x_i \in \ ]-1,1[$. Thus, no further penalty functions are required in order to impose boundary conditions.

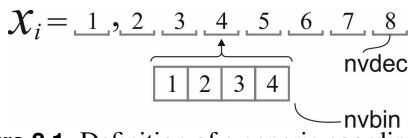


Figure 2.1. Definition of a generic coordinate x_i

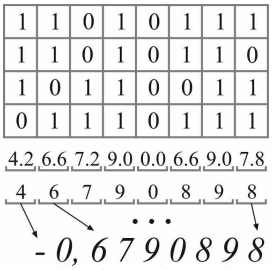


Figure 2.2. Example of a real number determination.

2.3 Initiation

At the beginning, the control variables should be defined, i.e. the population size (*psize*), the maximum number of generations (*ngen*), the number of genes (or variables) for each individual (*nvars*), which in this case is obtained by the product between the total amount of decimal digits plus one, for the sign assignation (*nvdec*) and the adopted quantity of binary digits (*nvbin*). Also, the mutation probability (*mut prob*) and the elite size (*nelit*) should be set at the beginning of the GA.

2.4 Initial population

The initial population is generated by a random algorithm specially designed to fulfil the boundary conditions regarding to the upper and lower bounds of the variables. Individuals are vectors of *nvars* dimension composed by eight integer digits (the first one used for sign assignation). Each integer digit is determined by a four digits binary number (see Fig. 2.1). Therefore, on the optimization problem of two three-dimensional functions (two independent variables), the individual dimension is $nvars = 2 * nvdec * nvbin = 128$ binary digits.

2.5 Cost functions

In a classical GA, the concept of fitness involves testing how "fit" a given individual is in comparison with other individuals using a cost function in order to obtain the best individual of the population. If a GA is used in unconstrained optimization problems, as in the case of this study, the fitness coincided with the actual cost function [26].

In the following, three different cost functions are considered. These functions are widely used to evaluate the performance of new optimization algorithms [15, 27, 28, 23]. Being n the problem dimension, the following cost functions are used:

- The simplest test function, the De Jong's function, also known as the sphere model. It is continuos, convex and unimodal (see Fig. 2.3a):

$$f_1 = \sum_{i=1}^{n} x_i^2 \ . \tag{2.1}$$

- The Ackley's function [3], widely used as a multimodal test function. It has many extreme values and a unique absolute minimum (see Fig. 2.3b):

$$f_2 = -a \exp\left[-b \left(\sum_{i=1}^{n} x_i^2 / n \right)^{-\frac{1}{2}} \right] - \exp\left(\sum_{i=1}^{n} \cos(c\, x_i)/n \right) + a + \exp(1) \ , \tag{2.2}$$

where, the adopted values of the constant parameters a, b and c are: 20, 0.2 and 2π, respectively.

- The Rastrigin's function, based on function f_1, Eq. (2.1), with the addition of a cosine modulation to produce many local minima. Therefore, is highly multimodal. However, the location of the minima are regularly distributed (see Fig. 2.3c):

$$f_3 = 10\,n + \sum_{i=1}^{n} x_i^2 - 10\,\cos(2\pi\,x_i)\ . \tag{2.3}$$

All the adopted cost functions share special characteristics. They all are continous functions with monotonic shape and their absolute minimum is perfectly determined at $x_i = 0$. This important feature allows an accurate evaluation of the GA performance analyzed in this work.

2.6 Selection

The selection process is also known as *Tournament* or *Ranking* selection method. In this method, a ranking-based competition is carried out in small groups of individuals (*ngroup*), randomly generated, in which the best of each group is selected according to its cost.

The main disadvantage of this procedure is that the worst individuals have nearly zero probability of being selected. In contrast, the population heterogeneity is better assessed in comparison with the classical *Simple Roulete* selection method [5, **?**].

2.7 Crossover

The crossover procedure is a crucial module for the GA performance. However, as the main purpose of this work is not focused on qualitatively improving convergence behavior of GA, an enhanced crossover technique is not required.

Furthermore, the GA parallelization in a GPGPU presented in this study allows computational cost of classical GA to be greatly reduced. Thus the simple well known *one-point crossover* method is used. All data beyond that point are swapped between the two parent individuals.

2.8 Mutation

Mutation is a genetic operator used to preserve the population diversity, behaving analogously to a biological mutation. Another purpose of this operator is to prevent the algorithm from being trapped into a local optimum, thus increasing GA search capability. The simplest mutation consists in modifying one or more gene values in a chromosome from its initial state according to a user-defined mutation probability during the evolutionary process. This probability should be set to a low value ($< 5\%$). Otherwise, the process could switch to a *primitive random search* [30].

In this research, the *Flip Bit* mutation operator was implemented. This technique is widely used and inverts the bits of a chosen genome according to its mutation probability (i.e. if the genome bit is 1, it is changed to 0 and viceversa).

3. Parallelization strategy in the CUDA environment

The main feature of graphics processing units is the ability to run a common process on a large number of cores working all together over different data. In order to improve the GPGPU paradigm, the CUDA environment has been developed by NVIDIA [18], allowing a more efficient programming. Therefore, there are two important issues that directly affect the performance of CUDA codes. The first one is related to a basic processing element: *threads*. Threads are labelled between 0 and *BlockDim*. The group of threads is called a *block*, and it contains a (recommended) 32 multiple number of threads.

The following aspect to be considered in the CUDA programming style is the device architecture. The minimum unit, where a single thread runs, is the *Streaming Processor* (SP). A *Streaming Multiprocessor* (SM) is a group of SPs.

The parallelization on GPU/CUDA architecture can be done on a *fine-grained* or *coarse-grained* level [18]. In order to reduce the data processing latency, a fine-grained parallelization strategy is adopted [10, 13], in which each data is assigned to each thread. This strategy in general improves GPU performance keeping busy at all times all multiprocessors cores, and consequently, the latency diminishes [25].

The coarsest granularity of synchronization occurs between commands in a queue or stream. Although this does provide synchronization across the entire kernel, it is very slow, as it often even involves a round trip to the CPU for API call completion. Finer-grained barriers can also be used to synchronize control (not data) within a thread block.

Figure 1.1a presents a standard GA flowchart. To qualitatively show the computational cost of each subroutine of this GA, the inner loops schemes required in each function of the GA are also depicted.

The flowchart of the CUDA GA algorithm implemented in this research is shown in Fig. 1.1b. The main difference observed between the classical sequential GA and the fully parallel CUDA-based GA is the absence of the inner loop in each GA subroutine. As will be seen later, this difference introduces a substantial improvement in the computational cost of the GA.

Table 1. Main features of GTX GeForce 750 Ti related to CUDA environment

CUDA Driver Version / Runtime Version:	7.0 / 6.5
Total amount of global memory:	2048 MBytes
(5) Multiprocessors, (128) CUDA Cores/MP:	640 CUDA Cores
L2 Cache Size:	2097152 bytes
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Warp size:	32
Maximum number of threads per multiprocessor:	2048
Maximum number of threads per block:	1024

Furthermore, in the Appendix A the main CUDA parallelized functions developed in this work are presented. This section was included in order to illustrate a simple way to introduce a CUDA parallelized function in a classical GA.

The graphics card used in this work is the GeForce GTX 750 Ti, and its main features can be obtained from the *deviceQuery* application found in the installation folder of NVIDIA CUDA enviroment [17]. In Table 1, the main characteristics of the GPU employed in this work are summarized.

4. Numerical results

In this section, the numerical results obtained from the computational tests using the CUDA GA algorithm developed in this work are presented.

4.1 Sensitivity analysis on the computational cost of the different operations entailed by GA

Firstly, a sensitivity analysis similar to the proposed in [**?**] is carried out. In this case, the main objective is the determination of the relative computational cost of each GA subroutine. For that purpose, each subroutine is parallelized while keeping the traditional sequential scheme for the remaining of the code. In order to have a homogeneous basis of comparison, $ngen = 100$ and $psize = 320$ are adopted for this computational test as control variables.

Figure 4.1 shows the computational cost of each GA subroutine compared with the fully parallelized CUDA version considering only the De Jong's cost function. It can clearly be seen that the most expensive process is the cost function evaluation (*Func*), followed by the crossing operator (*Crossover*).

Furthermore, it can be observed that the computational cost of the fully parallelized GA (3.54 sec) is only the 7.66% of the computational cost of the sequential GA scheme (46.21 sec). Hence, the parallel implementation allows to achieve an 92% improvement.

The following sensitivity analysis test consists in evaluate the De Jong's cost function with different dimensions of the independent variable (x_i). The results obtained were presented in Fig. 4.2a in order to qualitatively show the non-linear dependence of the total speedup of the fully parallelized GA with the independent variable dimension. Furthermore, the table included in Fig. 4.2b shows mean values as well as standard deviation of both generations and time.

4.2 Effect of population size

The effect on population size on the computational cost of the whole GA optimization process is then analyzed. For that purpose, both sequential and parallel codes are used with different population sizes. Also, in order to optimize the memory access, the chosen population size (*psize*) is proportional to the *warp size* (which is equal to 32 for the graphic card used in this work) in each numerical test.

Figure 4.3 shows a comparison between the numerical results obtained from the classical sequential GA and the fully parallelized CUDA version, for different population sizes. In addition, a third order polynomial fitting equation is plotted. It can be noted that the non-linear coefficients (corresponding to x^3 and x^2) are very small in both cases, therefore a linear function adjustment is enough to fit the numerical test. Hence, the average speedup of the GA can be deduced through the ratio between the values given by the two curves plotted in Fig. 4.3. Thus, the speedup obtained is equal to 14.4, neglecting higher order terms. These results were obtained with a fixed number of iterations ($ngen = 100$) in order to show the average computational cost of the sequential code compared with the fully parallelized implementation.

Actually, in classical GA optimization problems the number of generations required for obtaining the optimal solution is not a deterministic variable. Therefore, optimized design and computational cost should be evaluated in terms of computation time statistics (mean and standard deviation). In this regard, Fig. 4.4 illustrates the behaviour of the mean computation time for each cost function defined in section 2.5 considering different population sizes. It can be observed that the computation time dispersion decreases as the population size increases.

4.3 Computational cost analysis of the optimization process

This section analyzes the computational cost required for the optimization benchmark problem. In order to obtain a probabilistic estimation, one hundred samples were obtained by running the fully parallelized GA code considering the following population sizes: 320, 640, 960, 1280, 1600, 1920, 2240.

In this analysis the number of generations is not set previously. Therefore, a stopping criteria for the iterative procedure must be adopted. Since the global minimum is known in all benchmark functions described in Section 2.5 ($x_i = 0$), the stopping criteria is $|f(x_i)| < tol$, being the adopted tolerance $tol = 10^{-6}$.

Figure 4.5 shows the numerical results of this computational test for the De Jong's function with different population sizes. Similar to the results discussed in the previous section, the points representing each global optimum, obtained in successive runs with the same population size are fairly close to a linear function. Furthermore, it can noted that the population size directly affects the average slope of the linear function. However, the mean values for each population size is better fitted by an exponential function (discontinuous line in Fig. 4.5). Furthermore, the statistics analysis of the fully parallel GA considering different population sizes for each cost functions defined in section 2.5 are presented in Table 2, 3 and 4 for De Jong, Ackley and Rastrigin's cost functions, respectively. Also, in the above mentioned tables $\overline{g}$, Sd_g are the mean and standard deviation of the generations number, respectively, as well as $\bar{t}$, Sd_t are the mean and standard deviation of computation time, respectively.

Table 2. Comparative study of the parallel algorithm considering different population sizes and De Jong's cost functions.

psize	$\overline{g}$	Sd_g	$\bar{t}$	Sd_t
320	1240.13	670.51	40.33	21.75
640	299.23	209.60	20.00	11.49
960	153.28	122.32	12.51	9.75
1280	74.08	54.36	8.38	5.87
1600	47.92	34.81	6.76	4.55
1920	38.54	28.82	6.54	4.44
2240	27.91	18.63	5.79	3.37
2560	40.59	14.67	9.41	3.09
2880	161.00	125.12	13.00	2.80

Table 3. Comparative study of the parallel algorithm considering different population sizes and Ackley's cost functions.

psize	$\overline{g}$	Sd_g	$\bar{t}$	Sd_t
320	2602.90	930.13	84.81	30.28
640	724.31	299.81	39.85	16.39
960	346.78	141.93	27.91	11.29
1280	202.62	83.81	22.21	11.00
1600	146.13	84.14	19.55	10.96
1920	131.43	61.64	20.78	9.46
2240	100.49	47.17	18.91	8.52
2560	98.46	39.04	21.56	8.20
2880	351.00	143.20	28.00	10.10

Table 4. Comparative study of the parallel algorithm considering different population sizes and Rastrigin's cost functions.

psize	$\bar{g}$	Sd_g	$\bar{t}$	Sd_t
320	1850.04	841.17	60.23	27.35
640	512.10	217.98	28.23	11.96
960	227.37	103.85	18.33	8.25
1280	128.94	60.58	14.10	6.68
1600	96.71	36.58	13.07	5.00
1920	73.03	27.07	11.81	4.16
2240	61.40	22.33	11.84	4.02
2560	52.43	15.26	11.86	3.20
2880	126.00	58.50	14.00	3.01

4.4 Effect of cost function complexity on computational cost

In this section, the effect of cost function complexity on the computational cost of the optimization process is studied for each test problem running 100 times the fully parallelized GA code considering the following population sizes: 320, 640, 960, 1280, 1600, 1920 and 2240. The numerical results are summarized in Table 5.

Table 5. Statistical data on the CUDA GA computational cost for the different test problems.

psize	De Jong's function			Ackley's function			Rastrigin's function		
	$\bar{t}$	Sd_t	FA	$\bar{t}$	Sd_t	FA	$\bar{t}$	Sd_t	FA
320	1262.56	710.36	0	2428.01	971.40	4	1840.53	804.28	42
640	299.23	209.60	0	750.02	314.20	0	519.47	205.38	42
960	153.28	122.32	0	338.75	154.05	0	232.11	86.19	28
1280	78.82	69.34	0	208.82	118.89	0	134.49	66.45	23
1600	47.92	34.81	0	162.56	104.04	0	100.24	41.52	21
1920	38.54	28.82	0	126.61	50.95	0	74.65	32.20	17
2240	27.91	18.63	0	92.05	50.93	0	59.47	19.50	13

$\bar{t}$, Sd_t: mean and standard deviation of computation time, respectively.
FA: Failed attempts.

Figure 4.6 shows the mean lapsed computational time required for the convergence of the fully parallelized GA code against the generations numbers considering each cost function adopted in Section 2.5.

It can be seen that the De Jong's function optimization is faster than the others. Although the optimization of the Rastrigin's function requires less CPU time than the Ackley's problem, it should be noted that this test case has a much higher number of failed attempts, where the target global optimum could not be reached (see Table 5).

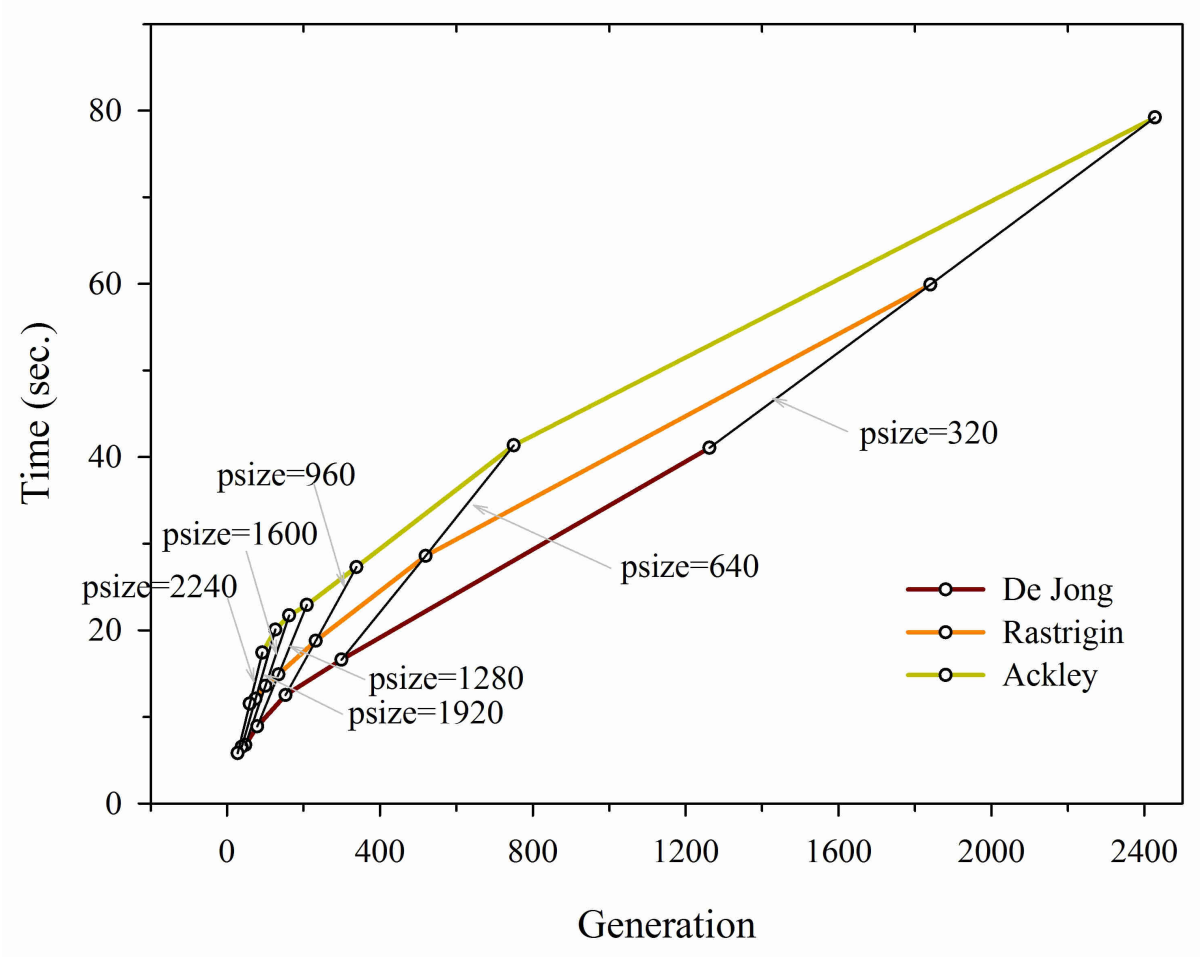


Figure 4.6. Comparison of computational cost of parallel CUDA GA for the different test problems.

5. Conclusions

A general GA code for global optimization implemented in the GPU CUDA environment was developed in this study. The code is capable of optimize *n*-dimensional functions which are frequently used as benchmark problems. Using CUDA programming paradigm, an interesting increment in the convergence rate of about 92% compared with the sequential code is obtained, and also, the speedup reported is about 13.05.

Furthermore, a sensitivity analysis performed on each GA operator allowed to determine its relative influence on the total computational cost of the algorithm and enables the programmer to focus on certain routines of the GA when the goal is not to fully parallelize the GA. The obtained numerical results show the efficiency and scalability of the proposed algorithm for the chosen cost functions. The presented GA is very general and may be extended to more complex problems or real-life applications in an straightforward manner, by rewriting the objective function.

6. Acknowledgment

This work has received financial support from Secretaría de Ciencia y Tecnología de la Universidad Tecnológica Nacional Facultad Regional Resistencia (UTN-FRRe, Argentina, Grants PID 25/L057 (2012), Secretaría General de Ciencia y Técnica de la Universidad Nacional del Nordeste (PID D001/11), Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT, Argentina, PICT-2015-2904, PICT-2013-0790).

1. Parallelized functions

Main device functions parallelized in CUDA framework are presented in this appendix.

A.1 InitPop

This is the first device function, included in any classical GA, with the aim of generate the random initial population

```
__global__ void InitPop(int *d_Pop, int nvars, curandState* globalState){
int it = blockDim.x * blockIdx.x + threadIdx.x;
for(int i=0; i<nvars ; i++) {
float k = generate(globalState, i+it)*1;
d_Pop[it*nvars + i] = lroundf(k);
}
}
```

A.2 Func_n

This device subroutine contain precisely the cost functions adopted in this article for n-dimensional generic variables x_i.

```
__global__ void Func_n(float *d_Sol, int *d_Pop, int nvbin, int nvdec, int ndim){
int it = blockDim.x * blockIdx.x + threadIdx.x;
int function = 3 ; // 1=DeJong, 2=Ackley , 3= Rastrigin
int ipop=0, two=2, ten=10, bin[4];
float x=0, y=0, X=0, Y=0, sum = 0, PI=3.141592653, xi[2], lim = 9/(powf(2,nvbin)-1);
for(int ix=0; ix<ndim; ix++){
for (int inum=1; inum<nvdec; inum++){
for(int i = 0; i<nvbin; i++) bin[i] = d_Pop[it*nvbin*nvdec*ndim + nvbin*nvdec*ix +
    nvbin*inum + i];
X = lroundf(BinDec(bin, nvbin)*lim); x = x + powf(ten,(0-inum))*X;
}
xi[ix] = x, x=0, X=0;
}

for(int i=0; i<nvbin; i++) bin[i]=d_Pop[it*nvbin*nvdec*ndim+i];
if(BinDec(bin,nvbin) < (powf(two,nvbin)-1)*0.5) x = (-1)*x;

d_Sol[it]=0;
float a=20, b=0.2, c=2*PI, d=10, x1=0, x2=0;
switch(function) {
case 1: // De Jong's function
for(int ix=0; ix<ndim; ix++) d_Sol[it] = d_Sol[it] + xi[ix]*xi[ix];
break;
case 2: // Ackley's function
for(int ix=0; ix<ndim; ix++){
x1 = x1 + xi[ix]*xi[ix]; x2 = x2 + cos(c*xi[ix]);
}
d_Sol[it] = -a*expf(-b*sqrt(x1/ndim)) - expf(x2/ndim) + a + expf(1);
break;
case 3: // Rastrigin's function
for(int ix=0; ix<ndim; ix++) x1 = x1+xi[ix]*xi[ix]-d*cos(two*PI*xi[ix]);
d_Sol[it] = d*ndim + x1;
break;
}
}
```

A.3 GroupSelection

This device function generate the group selection for the Ranking selection method adopted in this classical genetic algorithm.

```
__global__ void GroupSelection(int *sel, int *group, int gsize, int ngroup){
int it = blockDim.x * blockIdx.x + threadIdx.x;
if(it<ngroup){
sel[it] = group[it*gsize];
for(int i=0; i<gsize; i++){
if (sel[it] < group[it*gsize + i] ) sel[it] = group[it*gsize + i];
}
```

```
}
}
```

A.4 CrossoverSingle

The crossover operation is easily performed in this device function.

```
__global__ void CrossoverSingle(int *NPop, int *Male, int *Female, int *Pop, int psize, int
    nvars, int ndim){
int it = blockDim.x * blockIdx.x + threadIdx.x ;
if(threadIdx.x < nvars/2){
NPop[it*2] = Pop[Male[blockIdx.x]*nvars + threadIdx.x*2];
NPop[it*2+1] = Pop[Female[blockIdx.x]*nvars + threadIdx.x*2+1];
}else{
NPop[it*2] = Pop[Female[blockIdx.x]*nvars + (threadIdx.x - nvars/2)*2];
NPop[it*2+1] = Pop[Male[blockIdx.x]*nvars + (threadIdx.x - nvars/2)*2+1];
}
}
```

A.5 Mutation

The mutation operation is performed in this device function.

```
__global__ void Mutation(int *Pop, float mutp, int nvec, int nvars, curandState*
    globalState){
int it = blockDim.x * blockIdx.x + threadIdx.x;
if (it < nvec) {
float ran = generate(globalState, it)*(1.0);
if(ran<mutp){
int ivar = lroundf(generate(globalState, it)*(nvars-0.5));
if(Pop[it*nvars+ivar]==0){
Pop[it*nvars+ivar] = 1;
}else{
Pop[it*nvars+ivar] = 0;
}
}
}
}
```

References

[1] A. Aalaei, V. Kayvanfar, and H. Davoudpour. A multi-objective optimization for preemptive identical parallel machines scheduling problem. Computational and Applied Mathematics, pages doi:10.1007/s40314-015-0298-0, 2015.

[2] M. Abouali, J. Timmermans, J. E. Castillo, and B. Z. Su. A high performance GPU implementation of surface energy balance system (SEBS) based on CUDA-C. Environmental Modelling and Software, 41:134-138, 2013.

[3] D. H. Ackley. An empirical study of bit vector function optimization. In Lawrence Davis, editor, Genetic Algorithms and Simulated Annealing, Pitman, London, 1987, pages 170- 204. 1987.

[4] E. Alba, G. Luque, C.A.C. Coello, and E.H. Luna. Comparative study of serial and parallel heuristics used to design combinational logic circuits. Optimization Methods and Software, 22:485-509, 2007.

[5] A. Belegundu and T. Chandrupatla. Optimization Concepts and Applications in Engineering. Prentice Hall, 1999.

[6] A. Cano, A. Zafra, and S. Ventura. Speeding up the evaluation phase of GP classiffication algorithms on GPUs. Soft Computing, 16(2):187-202, 2012.

[7] S. D. Costarelli, M. A. Storti, R. R. Paz, L. D. Dalcin, and S. R. Idelsohn. GPGPU implementation of the BFECC algorithm for pure advection equations. Cluster Computing, 17(2):243-254, 2014.

[8] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. CPU DB: recording microprocessor history. Acm Queue, 10(4):10-27, 2012.

[9] D.P.S. dos Santos and J.K. Silva Formiga. Application of a genetic algorithm in orbital maneuvers. Computational and Applied Mathematics, 34:437-450, 2015.

[10] Anthony Gregerson. Implementing fast MRI gridding on GPUs via CUDA. Nvidia Tech. Report on Medical Imaging using CUDA, 2008.

[11] M. J. Harris. Real-time cloud simulation and rendering. University of North Carolina. Technical Report TR03-040, 2003.

[12] C. Lederman, R. Martin, and J. Cambier. Time-parallel solutions to differential equations via functional optimization. Computational and Applied Mathematics, pages doi:10.1007/s40314-016-0319-7, 2016.

[13] Daniel Lustig and Margaret Martonosi. Reducing GPU ooad latency via

ne-grained CPU-GPU synchronization. In Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), pages 354-365, Washington, DC, USA, 2013. IEEE Computer Society.

[14] J. L. Mroginski, P. A. Beneyto, G. Gutierrez, and H. A. Di Rado. A selective genetic algorithm for multiobjective optimization of cross sections in 3D trussed structures based on a spatial sensitivity analysis. Multidiscipline Modeling in Materials and Structures, 12:423-435, 2016.

[15] H. Muhlenbein, M. Schomisch, and J. Born. The parallel genetic algorithm as function optimizer. Parallel Computing, 17(6-7):619-632, 1991.

[16] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. Queue, 6(2):40-53, 2008.

[17] NVIDIA Corporation. Cuda c programming guide, September 2015.

[18] NVIDIA Corporation. Cuda toolkit documentation v7.5, September 2015.

[19] R. R. Paz, M. A. Storti, H. G. Castro, and L. D. Dalcin. Using hybrid parallel programming techniques for the computation, assembly and solution stages in finite element codes. Latin American Applied Research, 41:365-377, 2011.

[20] R. R. Paz, M. A. Storti, L. D. Dalcin, H. G. Castro, and P. A. Kler. Fastmat: A C++ library for multi-index array computations. Advances in Engineering Software, 54:38-48, 2012.

[21] D. Robilliard, V. Marion-Poty, and C. Fonlupt. Genetic programming on graphics processing units. Genetic Programming and Evolvable Machines, 10(4):447-471, 2009.

[22] G. Sharma, A. Agarwala, and B. Bhattacharya. A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA. Computers and Structures, 128:31-37, 2013.

[23] S. Siva Sathya and M.V. Radhika. Convergence of nomadic genetic algorithm on benchmark mathematical functions. Applied Soft Computing, 13(5):2759-2766, 2013.

[24] T. Tomczak, K. Zadarnowska, Z. Koza, M. Matyka, and L. Miroslaw. Acceleration of iterative Navier-Stokes solvers on graphics processing units. International Journal of Computational Fluid Dynamics, 27:201-209, 2013.

[25] M. Ujaldon and J. Saltz. Exploiting parallelism on irregular applications using the GPU. In G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, and E. Zapata, editors, Parallel Computing: Current & Future Issues of High-End Computing, John von Neumann Institute for Computing, Julich, (Germany), volume 33, pages 639-646, 2006.

[26] Y. Yang. A globally and quadratically convergent algorithm with efficient implementation for unconstrained optimization. Computational and Applied Mathematics, 34:1219-1236, 2015.

[27] Liu Yong, Kang Lishan, and D.J. Evans. The annealing evolution algorithm as function optimizer. Parallel Computing, 21(3):389-400, 1995.

[28] H. Zhang and M. Ishikawa. The performance verification of an evolutionary canonical particle swarm optimizer. Neural Networks, 23(4):510-516, 2010.

[29] K. Zhang, S. Yang, L. Li, and M. Qiu. Parallel genetic algorithm with opencl for traveling salesman problem. Communications in Computer and Information Science, 472:585-590, 2014.

[30] Y. Zhang, M. Sakamoto, and H. Furutani. Effects of population size and mutation rate on results of genetic algorithm. In 4th International Conference on Natural Computation (ICNC), volume 1, pages 70-75, 2008.

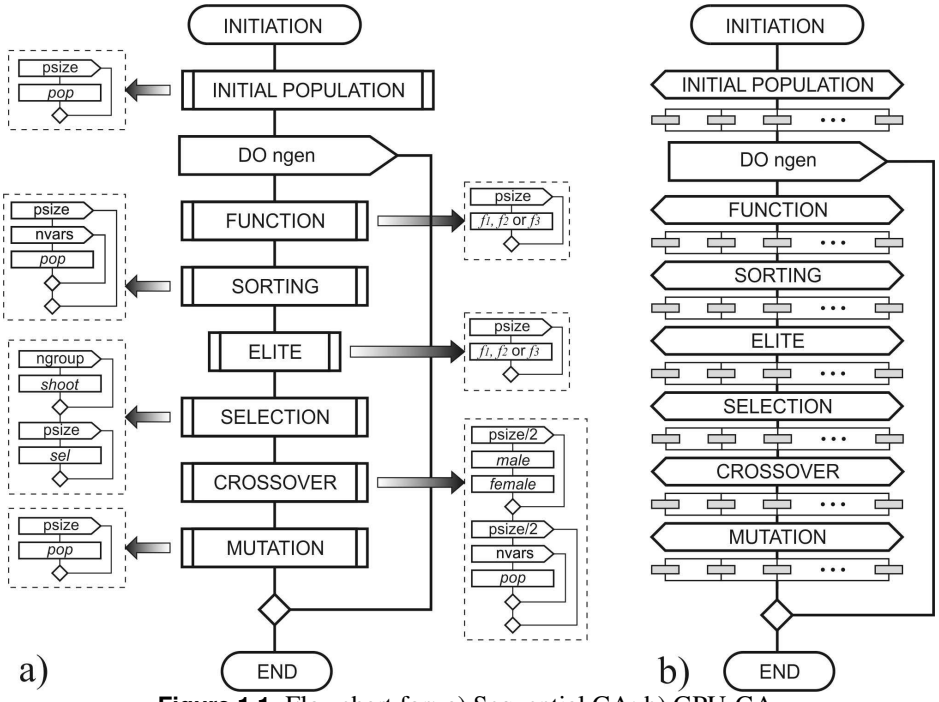


Figure 1.1. Flowchart for: a) Sequential GA; b) GPU-GA.

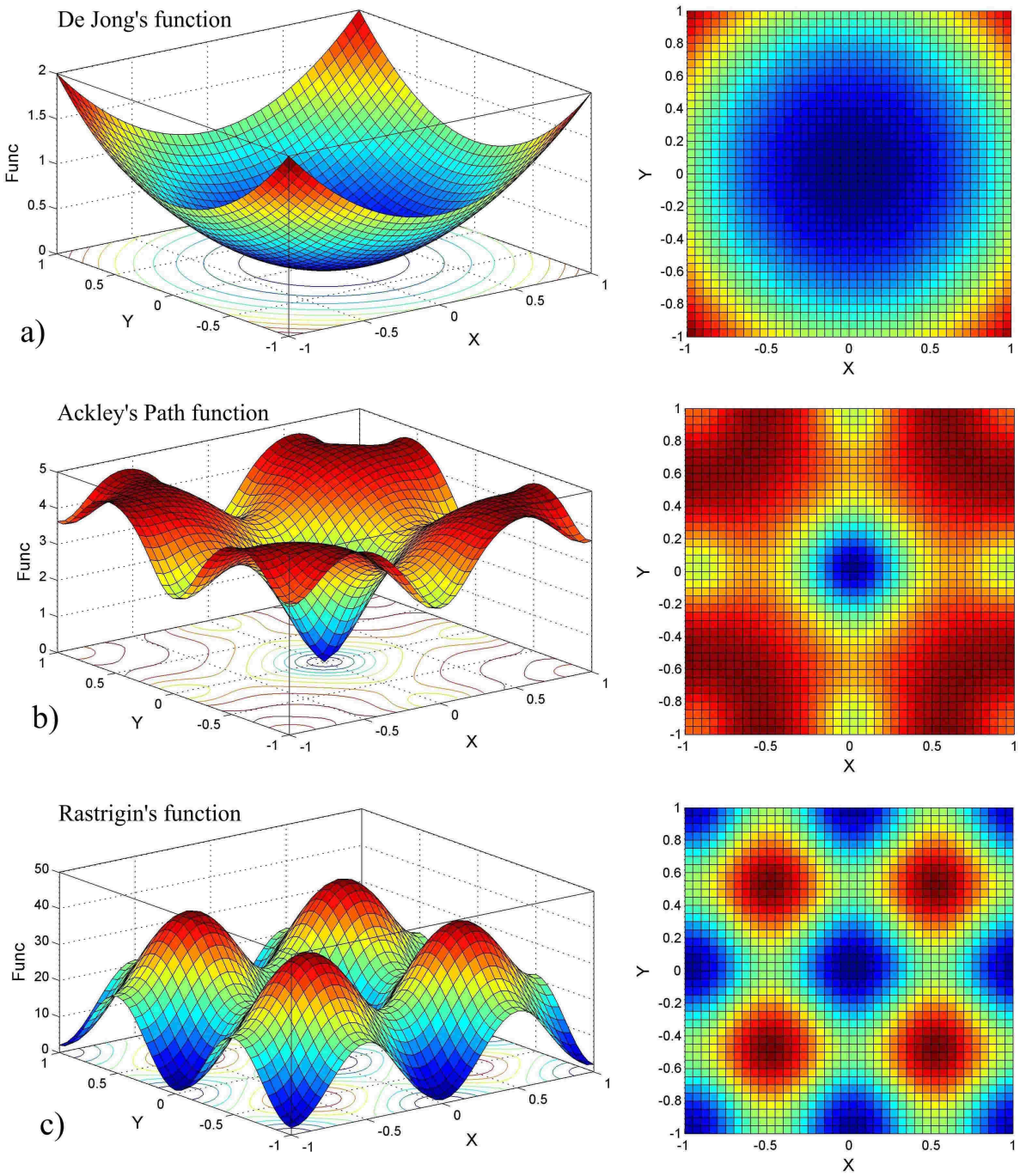


Figure 2.3. Representation of the cost function for two-dimensional optimization ($n = 2$): a) De Jong's function; b) Ackley's function; c) Rastrigin's function.

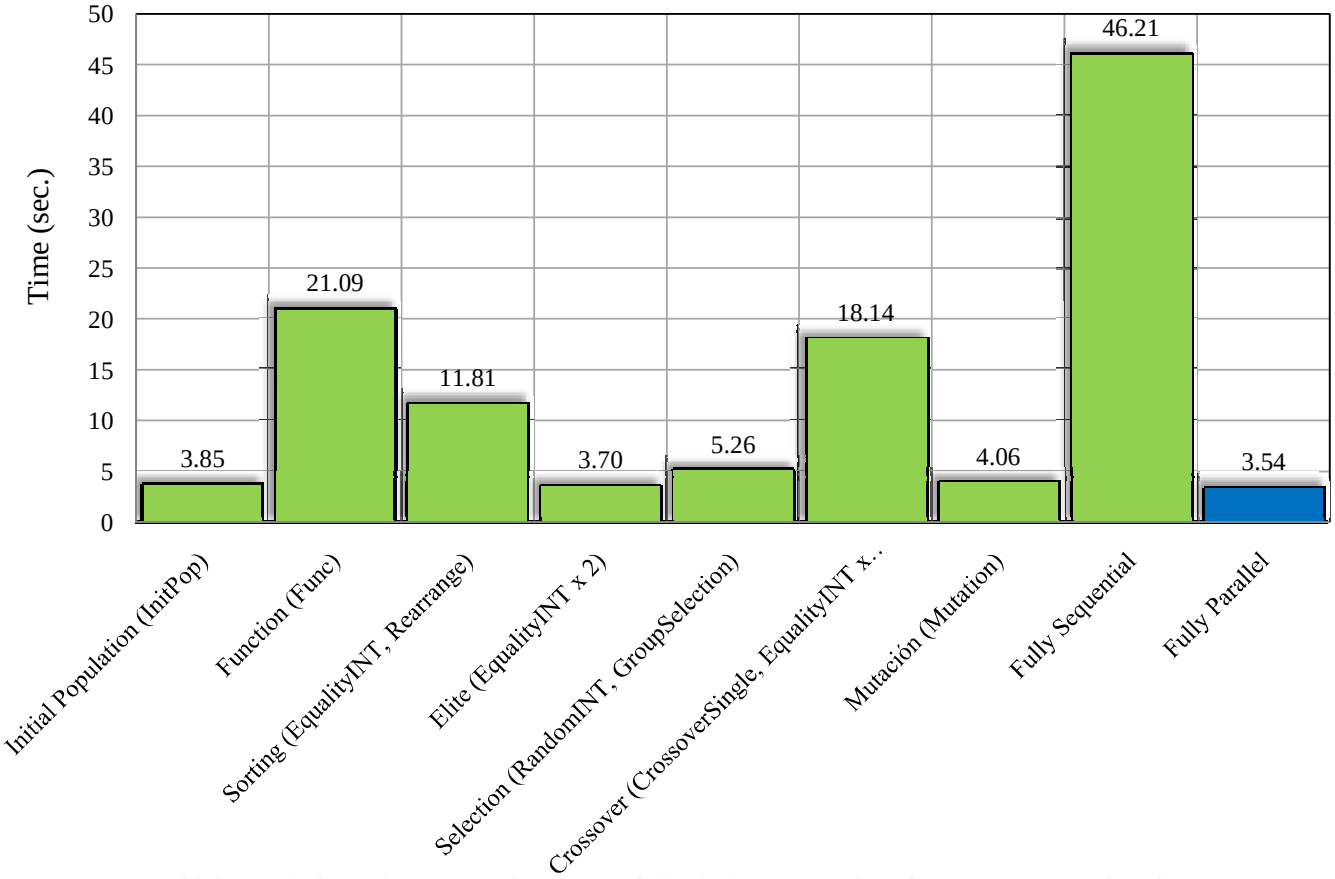
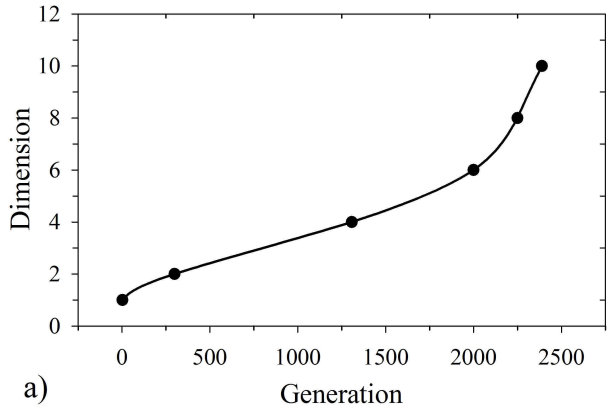


Figure 4.1. Sensitivity analysis on the computational cost of classical GA operations for De Jong's cost function ($ngen = 100$, $psize = 320$)



ndim	ngen	desvio	time	desvio
1	2,48	0,76	0,23	0,08
2	299,23	209,60	16,59	11,49
4	1307,49	517,97	72,99	28,76
6	2000,46	698,60	112,71	39,11
8	2250,00	1278,36	116,34	73,38
10	2389,49	1427,61	138,93	82,31

b)

Figure 4.2. Influence of the variable dimension on speedup of the fully parallelized GA: a) Qualitative representation; b) Data table

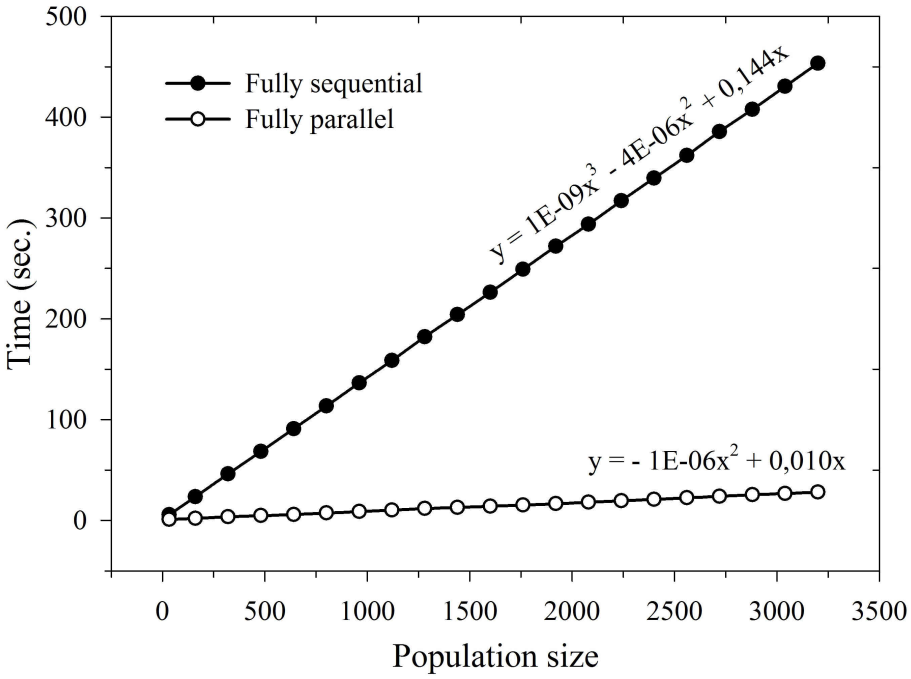
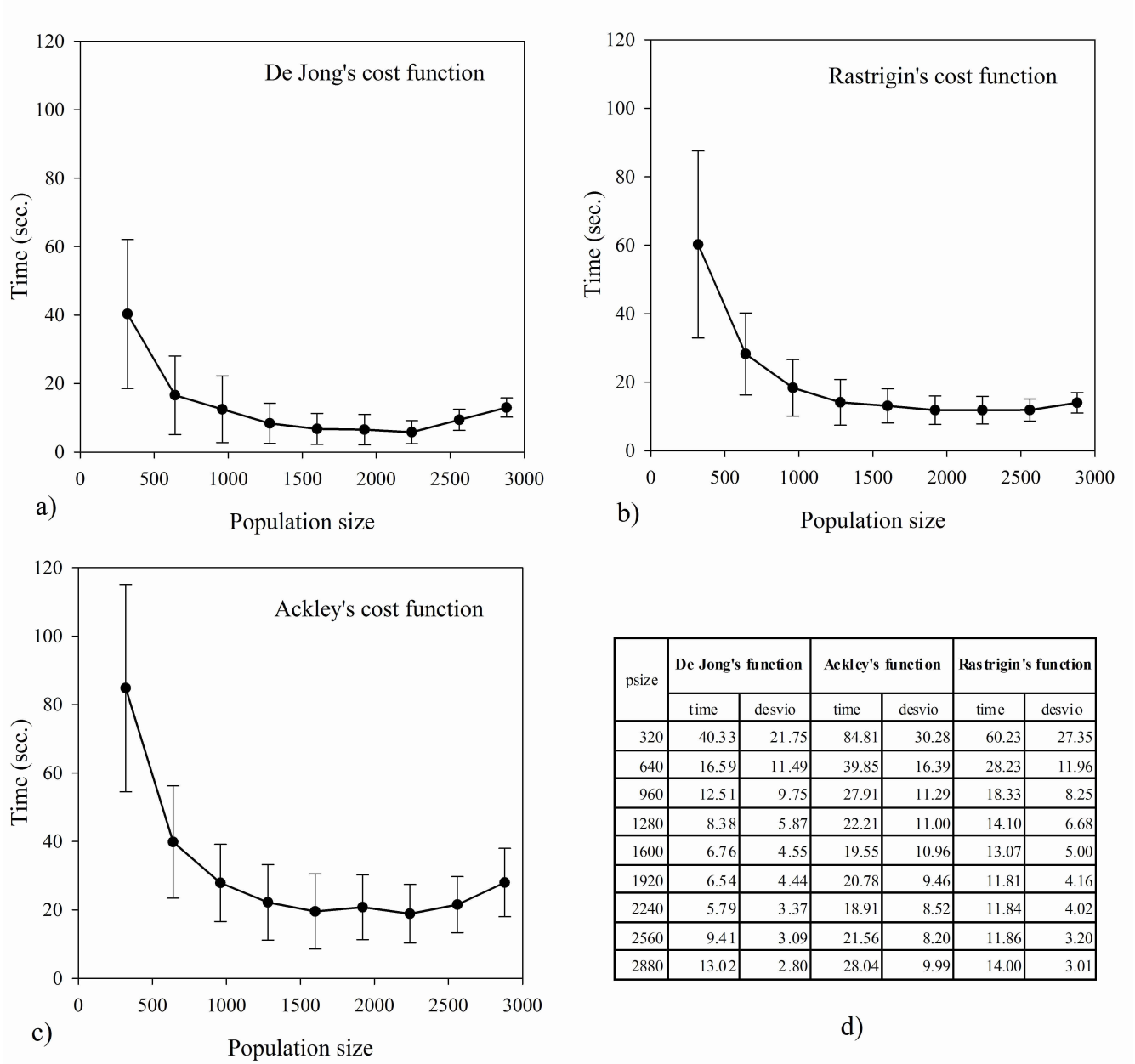


Figure 4.3. Comparison of the computational cost of both the fully parallelized and sequential GA implementations, assuming the De Jong's cost function with different population sizes.



psize	De Jong's function		Ackley's function		Rastrigin's function	
	time	desvio	time	desvio	time	desvio
320	40.33	21.75	84.81	30.28	60.23	27.35
640	16.59	11.49	39.85	16.39	28.23	11.96
960	12.51	9.75	27.91	11.29	18.33	8.25
1280	8.38	5.87	22.21	11.00	14.10	6.68
1600	6.76	4.55	19.55	10.96	13.07	5.00
1920	6.54	4.44	20.78	9.46	11.81	4.16
2240	5.79	3.37	18.91	8.52	11.84	4.02
2560	9.41	3.09	21.56	8.20	11.86	3.20
2880	13.02	2.80	28.04	9.99	14.00	3.01

Figure 4.4. Mean speedup of the fully parallelized GA code vs population size. a) De Jong's function, b) Rastrigin's function, c) Ackley's function, d) Mean values and standard deviations.

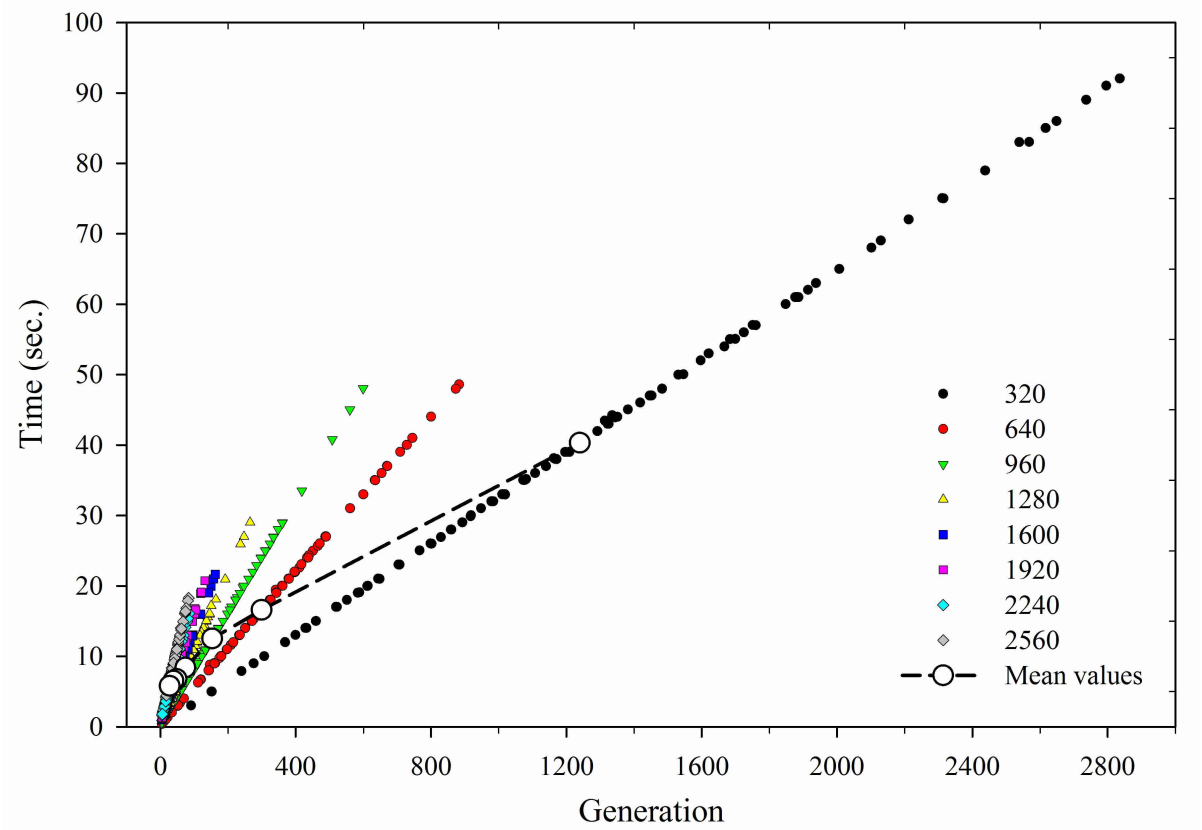


Figure 4.5. Actual computational cost of the fully parallelized CUDA GA implementation for the De Jong's function with different population sizes.