# Parallelization of Binary and Real-Coded Genetic Algorithms on CUDA

Ramnik Arora, Rupesh Tulshyan, Kalyanmoy Deb

***Abstract*— It is a well-known fact that genetic algorithms (GAs) are ideal for parallel computers due to their ability to parallely evaluate population members. Most past parallel GA studies have exploited this aspect. Besides resorting to completely different algorithms, such as island models etc., a GA involves a number of other operations which, if parallelized properly, may also end up with a better parallelization to an existing serial GA implementation. In this paper, we parallelize binary and real-coded genetic algorithms for the CUDA platform using its C API extensions. Although, objective and constraint violations are *embarassingly parallel*, other algorithmic and code optimizations have been tested and suggested. The bottlenecks in the GA algorithms for a parallel implementation are identified and modified suitably. The results are compared with the serial algorithm on accuracy and clock time for varying problems by studying the effect of a number of parameters: (i) differing population sizes, (ii) differing number of threads, (iii) differing problem sizes, and (iv) problems having differing complexities, such as evaluation time and interactions among variables. The results indicate that proposed methods are more than 40 times faster than their serial counterparts.**

## I. Introduction

Evolutionary algorithms (EAs) are ideal for their use on a parallel computing platform due to a number of reasons:

- EAs use a population of solutions in each iteration. Before any EA operator is applied to the population, every member must be evaluated to find the objective and constraint values in an optimization problem solving task. Since the evaluation of solutions is an independent event, the evaluation task can be performed parallely.
- Most EA operators involve one or two solutions, thereby making them ideal for a parallel implementation.
- EA is a Markov chain, involving only the information of the previous population, thereby not requiring to store too many parameters iteration-wise.

Due to these reasons, EAs are often parallelized and used in practical applications. Majority of the parallel EA implementations involve a distributed evaluation of population members using a master-slave implementation [1], [2]. There are, however, other algorithmic changes which have also been suggested. Of them, the island EA [3] runs individual EA on each processor and allows occasional migration of better solutions among processors.

With the recent advancement of graphics processing units (GPUs), researchers have been modifying EAs for parallel applications on a single desktop computer [4], [5], [6], [7], [8], [9], [10]. The studies report speed-up factors as high as 200 to 300 compared to their serial applications.

This paper makes a contribution towards this direction and makes a detail study by investigating the effect of number of threads, problem size, population size, and problem complexity on the speed-up of two existing evolutionary algorithms: a binary-coded GA and a real-coded GA. A significant way in which our study differs from some past GPU-based EA studies is that we not only implement parallel evaluation of solutions, but also suggest modifications to the existing GA operators to suit the GPU architecture. In all our simulations, we have used Nvidia's C1060 Tesla computing processor and find 40 to 300 times speed-up in solving single-objective optimization problems.

In the remainder of this paper, we give a brief introduction to the GPU computing and CUDA architecture. Thereafter, we discuss the possibilities of improving an EA for using them appropriately with a GPU computing system. We suggest a number of modifications to various operators of a binary-coded and a real-coded GA. Then, we present simulation results on three test problems and show scale-up results using our proposed algorithms. Conclusions are made at the end.

## II. GPU Computing and CUDA

Recent advancements in parallel computing have seen the usage of Graphics Processing Units for general purpose computing (GPGPU). The Graphics Processing Units (GPUs) employ parallel multi-core architecture which favors the inherently data-parallel nature of graphics rendering. They are designed such that they can devote more transistors for arithmetic and logical operations rather than to data caching and flow control compared to a typical CPU. Driven by booming gaming industry, the GPUs have advanced rapidly and are today far ahead of the CPUs in the number of cores and hence, their computational power.

With the launch of Nvidia's Compute Unified Device Architecture - Software Development Kit (CUDA) in 2007, using the GPU's computational powers for general purpose computing has become easy. It provides an API built upon the 'C' language in which programs can be written. The GPU *device* operates as a coprocessor to the *host* running the C program.

The basic unit of a parallel code on CUDA is a thread, millions of which can be deployed simultaneously. Each thread has its own set of memory registers. With dynamic scheduling and fast creation and destruction of light-weight

R. Arora and R. Tulshyan are with the Department of Mathematics and Scientific Computing, Indian Institute of Technology Kanpur, Kanpur, India. (Email: {ramnik,tulshyan}@iitk.ac.in)

Prof. K. Deb is with the Department of Mechanical Engineering, Indian Institute of Technology Kanpur, Kanpur, India. He is also the Director of Kanpur Genetic Algorithms Laboratory. (Email: deb@iitk.ac.in)

threads, CUDA is best suited for problems whereby same instruction set is to be executed on different data via multiple threads. A group of threads can be bundled into blocks which can share information using a limited shared memory. A *function* or task to be done on a GPU is called a *Kernel*. A kernel begins by defining a grid of the above-mentioned blocks and contains the similar instruction set for all threads albeit depending on individual thread indexes and block indexes. The GPU has its own DRAM referred to as *device/global memory* using which data can be transferred to-and-forth between the CPU and the GPU (hereafter, referred to as *memcpy* operations). The global memory can hold the data throughout the program. For temporary variables (for the lifetime of a kernel) other memory levels may be used. Different memory levels though have different latencies and optimizations of all sorts are required to get good speed-ups.

At the hardware level, the blocks of the grid are enumerated and distributed to cores with available execution capacity (a block cannot be shared by two cores). A core (also called as Symmetric Multiprocessor) consists of eight Scalar Processor cores, two special function unit for transcendentals, a multi-threaded instruction unit, and on-chip shared memory. The multiprocessor creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. The threads of a block execute with time-sharing and are instructed in batches of 32 called as *warps*. A warp executes one common instruction at a time and care must be taken to avoid logical divergence of threads in a warp. As thread blocks terminate, new blocks are launched on the vacated cores.

Overall performance optimization strategies include

- Structuring the algorithm so as to exploit maximum parallel execution and high arithmetic intensity.
- Minimizing data transfers with low-bandwidth i.e. between the host and the device.
- Achieving *global memory coalescing* - simultaneous memory accesses by threads in a half-warp can be coalesced into a single memory transaction. Assuming global memory to be partitioned and aligned into segments of 32, 64 or 128 bytes, *coalescing* occurs when 16 threads of a *half-warp* access variables of same bit-size (lying in same segment of global memory) in sequence.
- Maximizing use of shared memory albeit avoiding access conflicts.
- Minimizing the number of local variables per thread.
- Minimizing warp-level divergence.
- Minimizing the need of explicit synchronization among threads.

## III. Evolutionary Computation and CUDA

The need for parallel implementation of evolutionary algorithms has for long been acknowledged by the EA community to tackle computationally expensive and difficult problems. In particular, parallel GAs have attracted extensive research. A comprehensive survey of the research in techniques used to parallelize GAs before the change of millennium is available at [11], [12]. Recent efforts have been directed towards porting GAs onto the GPU and on MapReduce using Hadoop [13], a powerful abstraction for making scalable and fault tolerant applications.

Research on adapting EAs for consumer level graphics processing units were initiated by development of Brook stream programming language and Cg or C for Graphics language. [14] work was one of the foremost efforts to port fine-grained real coded GA on GPU with BLX-$\alpha$ crossover using Cg. [4] parallelized evolutionary programming porting the fitness evaluation, mutation and reproduction to the GPU, while tournament selection was performed on the host. [5] focused on faster evaluation of individual fitness function citing the memory transfer operations to be expensive.

In 2007, the release of initial Compute Unified Device Architecture SDK, the compute engine in Nvidia's GPUs spurred further research in the field. [15] used CUDA on the G80 GPU to efficiently interpret several GP programs in parallel obtaining impressive speed-ups for small training sets. [6] proposed a parallel multi-objective evolutionary algorithm on GPU inspired by previous work of PGA and NSGA-II. In the parallel MOEA, other than non-dominated selection procedure, all other procedures were ported to the GPU. Other studies [7], [9], [16], [17] have tried to use the consumer level graphics processing units to speed up EAs.

## IV. Algorithm and Code Optimizations

We concentrate our study on a particular GA implementation. The serial 'C' code for the GA (supporting both binary and real variables along with constraint handling) is available on the the KanGAL website **www.iitk.ac.in/kangal/codes.shtml**. The basic GA algorithm can be stated as follows:

**Algorithm** *B/RGA*($Problemset, inputparams$)
**Input:** Problem Definition, Set of initial GA parameters.
**Output:** Generation-wise best $x_t$, $objective(x_t)$

```
1.  InputParam(inputparams)
2.  for i = 1 to MaxRun
3.          InitPopulation()
4.          for t = 1 to MaxGen
5.                  Statistics()
6.                  Selection()
7.                  CrossOver()
8.                  Mutation()
    return
```

The CUDA algorithm exactly emulates the serial version except for *Selection*, *CrossOver(Real)* and *Mutation(Binary)* operators, wherein small changes are introduced to get better performance. In the following subsections, we state how the different GA operators work, how we have parallelized them, and discuss the intricacies involved in parallelizing and optimizing them.

### A. InputParam()

This method reads the basic GA parameters, including population and generation sizes, types of decision variables

and their lower and upper limits. This remains unchanged in parallel algorithm.

We now enumerate some of the variables and constants used in the algorithm, to have convenience in further discussions.

- $pop\_size \leftarrow$ Population Size
- $nvar \leftarrow$ Number of decision variables
- $chromsize \leftarrow$ Number of 32-bit integers needed to store choromosome string information for binary variables of one population individual
- $tourney\_size \leftarrow$ Tournament Size
- $x_{i_{(L)}}, x_{i_{(U)}} \leftarrow$ Lower and Upper limits of a decision variable $x_i \in (x_1, x_2, \ldots, x_{nvar})$
- $H\_seeds \leftarrow$ Host Array containing random seed information
- $D\_seeds \leftarrow$ Device Array containing random seed information

### B. Data Organization

We maintain separate 1-D integer and float arrays for storing population information and other data including variable boundaries, tournament schedule, objective values, constraint violations etc. If the decision space contains both real and binary variables, separate arrays are used to represent them, and separate kernels are called for corresponding operations. Since, in most of the kernels, different threads of a block cater to different individuals of the population, storing variables of one individual sequentially in an array does not permit efficient memory access. So, to achieve memory coalescing, variables of one type belonging to different population members are stored adjacently in arrays.

### C. Random Number Generation

The performance of an evolutionary algorithm depends a lot on the quality of its random number generations. The serial versions of GAs have typically used LCGs, which have proven to be sufficiently random for GA purposes. However, LCGs are inherently not parallelizable and parallel random number generation otherwise, is not trivial. Though some efficient PRNGs have been suggested in literature (and CUDA SDK), which can generate billions of random numbers in seconds; their usage seems exorbitant for GAs (which require significantly less random numbers), and might bring needless computation overheads.

We work around this problem by using an approach similar to Langdon [18], who had recently suggested a simple and efficient implementation of Park-Miller random number generator on GPUs. This generator works as follows:

$$r_{k+1} = (r_k * 16807) mod (2^{31} - 1)$$

Note that for a given problem, the maximum number of threads in a kernel computing random numbers (say it is $m$), can be calculated. Using this information, we can spawn $m$ Park-Miller generators with different initial seed values, in the device and then invoke them sequentially for subsequent random number generations.

### D. Control Flow

The parallel code starts with taking the input parameters for the GA, using which it does the memory allocation for data arrays. Note that, for each population attribute, we need two arrays - one for the host CPU, and the other for the device (GPU).

The algorithm then generates the initial seeds for Park-Miller random number generation on device, using *rand.c* C library. This is done only once, and the seed data is then copied from the host ($H\_seeds$) to the device ($D\_seeds$) once. $D\_seeds$ is maintained throughout the program in the device global memory. When a thread needs a random number, it loads the value $D\_seeds[h(thread\_id)]$ from global memory to its memory registers. Here, $h(r)$ is a function, which ensures that each thread in the kernel accesses different seed value, since $thread\_id$ are identical for different blocks. Now, the next random number is computed, using the loaded seed value and is stored in $D\_seeds[h(thread\_id)]$ for subsequent random number generations.

The host then calls the subsequent operations of initialization, evaluation and new population generation, for different generations and GA runs, which are all executed on the device sequentially. The data resides on the device memory inbetween different generations and is iteratively updated. The statistics computation, particularly finding the best fitness individual of each generation, are also done on the device and only the attributes of the best population member per generations are copied from device to host. Thus, the host-device data transfer is minimized which brings huge efficiency.

### E. InitPopulation()

This method initializes the initial population members and is easily parallelizable. Most parallel EA applications modify this part of the algorithm such that every processor gets a fraction of the population members for evaluation, thereby parallelizing the whole computational effort. We use a similar strategy here but we suggest that the generation of variables be done within each thread independently, thereby parallelizing the effort even more.

For initialization of binary variables, each thread works on a 32-bit integer of the chromosome array, and uniformly randomly assigns 0 or 1 to each bit. For real variables, a 2-D grid of $pop\_size/no\_threads \times nvar$ blocks is created, where each block contains $no\_threads$ threads. Thus, each block assigns a random value to the variable $x_i$, for a subset of whole population, using the limits $x_{i_{(L)}}$ and $x_{i_{(U)}}$ which are stored in shared memory.

### F. Selection()

We use a modified form of the tournament selection operator in our parallel implementation. The aim of the basic tournament selection operator is to conduct a tournament of $pop\_size$ matches, each match involving $tourney\_size$ individuals. In a match between $tourney\_size$ individuals, the one with the best fitness value wins, and is kept in the mating pool.

In the serial code, each individual of the population participates in exactly $tourney\_size$ matches. This is implemented by uniformly shuffling the individual indexes in the population array, and then choosing consecutive individuals $(x_i, x_{i+1}, \ldots, x_{tourney\_size-1)}$ for $i^{th}$ match.

Note that, shuffling of an array is not an efficiently parallelizable task, as it involves a consideration of the complete population. Hence, in the parallel code, for $i^{th}$ match, $tourney\_size$ individuals are uniformly randomly chosen out of the whole population. This procedure may not allow exactly $tourney\_size$ copies of each population member for tournaments, our simulation results do not show any large bias and the advantage gained for parallelization out-weighs the slight randomness introduced by this implementation.

### G. CrossOver()

We use the Single-Point Crossover for binary variables, and the Simulated Binary Crossover (SBX) technique for real variables [19]. The crossover is done with the predefined $p\_xover$ probability.

For binary crossover, the crossover sites are uniformly randomly generated on the device. A grid of $pop\_size/2$ blocks is evoked, with each block containing $chromsize$ threads. The $i^{th}$ block basically conducts the crossover between $p_{2i}$ and $p_{2i+1}$, population members of the mating pool. The crossover site information is stored in shared memory. Each thread corresponds to one 32-bit integer of the chromosome string, for both the parents and depending on the crossover site handles the swapping of bits, if required. This grid structure is optimum in the following sense:

- Global coalescing for chromosome string accesses can be achieved.
- Each block can use shared memory to store crossover site information. Also, there is only one load per crossover site information from the global memory.
- Warp divergence is minimum, especially for large variable sizes. For consecutive threads of a *half-warp* handling the chromosome string bits on either side of the crossover site, there is no branching (in the operation of swapping of bits).

For real variables, a 2-D grid of $pop\_size/(2*no\_threads)$ $\times$ $nvar$ blocks is created, where each block contains $no\_threads$ threads. Blocks along the horizontal axis cater to different subsets of complete population, whereas blocks along the vertical axis cater to different variables of same population subset. Each thread is responsible for crossover of a variable $x_i$ of two mates. Again, this structure is optimal as it supports coalescing and uses shared memory for storing $x_{i_{(L)}}$ and $x_{i_{(U)}}$ values. Warp divergence, here cannot be avoided as it depends on random numbers.

An alternative grid structure, similar to the binary crossover could have also been implemented for real crossover, wherein each block handled the crossover of two mates. But this increases the number of global loads for upper and lower limits of different variables, as each block now needs all limits. Expecting a tradeoff, we choose the former structure.

### H. Mutation()

The serial code works on each bit of the chromosome string and flips it with predefined $p\_mutation$ probability, requiring immense random number generations. Deb et al. suggest $p\_mutation = 1/nvar$ to achieve proper balance between crossover and mutation in their study. This probabilistically means we mutate only one variable out of $nvar$ variables for a population member. For binary case, this means flipping only one bit of the whole chromosome. Hence, to reduce random number generation which is quite computation intensive task, we modify the procedure such that every thread chooses one random bit for the complete chromosome string of a individual and flips it.

As is seen in Table II this tweak lends speedups of order of hundreds. To keep things in perspective, we also undertake a speedup study with the modified binary mutation operator employed in serial code also.

For real variables, each thread mutates the corresponding $x_i$ with $p\_mutation$ probability. The grid structure is same as the one used in real variable initialization.

### I. Statistics()

This method may involve varied generation-wise statistics computation, involving averages, maximum or minimum of different population attributes. All these operations of sum, maximum or minimum finding are not data parallel, and it also becomes very tedious task, to manage intermediate arrays if we do parallelize them. However, shifting these computations to host brings sufficient overheads in terms of memory-transfers and serial computations.

Currently, we implement the computation of best population member of a generation on the device, as it is the most essential information for the user. Assuming a minimum population size of 32 for any optimization problem, we first evoke a kernel-grid of $pop\_size/32$ blocks with each block containing 32 threads. These 32 threads load the objective and penalty data of a population subset into the shared memory and use parallel reduction to find the best fit individual. Every block copies its best individual data onto an intermediate array. After that, we call a grid containing only 1 thread to serially find the best fit individual, working on the intermediate array of length $pop\_size/32$. Besides, we also give user the flexibility of copying all population data from the device to host and do other relevant statistic computations on the host, if required.

## V. Test Problems

Three well-known problems in literature have been taken up for this study. We are interested in showing the scale-up advantage of our proposed parallelization and the objective functions are important from the point of view of an algorithm's ability to find the minimum solution.

### A. One-Max Function

First, we consider a separable linear function, which can be scaled to any number of variables (real or binary):

$$\begin{aligned} \text{Minimize} \quad & f(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} x_i, \\ \text{Subject To:} \quad & -10 < x_i < 10, \quad \forall i. \end{aligned} \tag{1}$$

In the case of binary-coded GA application, we restrict $x_i \in \{0, 1\}$ and $n$ becomes the string length. The optimal solution has a function value equal to zero.

### B. Ellipsoidal Function

Next, we consider the ellipsoidal function:

$$\begin{aligned} \text{Min} \quad & f(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} i x_i^2, \\ \text{Subject to:} \quad & -10 < x_i < 10, \quad \forall i. \end{aligned} \tag{2}$$

The optimal solution has a function value equal to zero.

### C. Rosenbrock Function

$$\begin{aligned} \text{Minimize} \quad & f(x_1, x_2, \ldots, x_n) \\ & = \sum_{i=1}^{n/2} [100(x_{2i-1}^2 - x_{2i})^2 + (x_{2i-1} - 1)^2], \\ \text{subject to} \quad & -10 < x_i < 10, \quad \forall i. \end{aligned} \tag{3}$$

The optimal solution is $x_i = 1$ for each $i$ and has a function value equal to zero.

## VI. Results

In this section, we present the results obtained by our parallel RGA and BGA implementations. We use Tesla C1060 Nvidia processor with 30 SMs (240 cores). The serial C code for GAs is run on the AMD Athlon 64 X2 Dual Core (3800+) Processor. For the BGA and RGA, we have used $p\_xover = 0.8$. For RGA, we use SBX recombination operator with $\eta_c = 2$ and polynomial mutation with $\eta_m = 100$ for all three problems. For BGA, we use 16 bits to represent a variable.

### A. Convergence of Proposed Parallel Algorithms

First and foremost, any parallelization effort must ensure that finally achieved solution is similar in quality to that obtained from the serial algorithm. The parallelization is not expected to find a better solution in terms of its function value and constraint violation (if any) than that of serial algorithm. Thus, here we first demonstrate the ability of both our real-coded and binary-coded parallel GA implementations for their ability to find a solution close to the true optimal solution.

The convergence of the algorithm can be best studied by investigating the average objective function values of the whole population generation-wise. Taking average objective values of a generation as a parameter, we plot its average, minimum and maximum values for 10 different runs of the GAs. This is done for the One-Max problem (Figures 1 for RGA and 2 for BGA) and the Rosenbrock functions (Figures 3 for RGA and 4 for BGA). The results are compared with CPU-RGA and CPU-BGA, their serial counterparts. It is clear from the plots that a similar convergence performance is obtained for both problems.

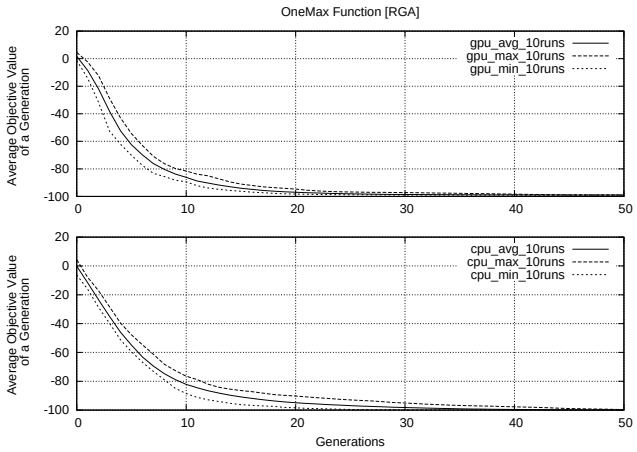


Fig. 1. Objective function value versus generation counter for OneMax problem using RGA. The top figure is obtained with a GPU based computer and the bottom figure is obtained with a serial computer.

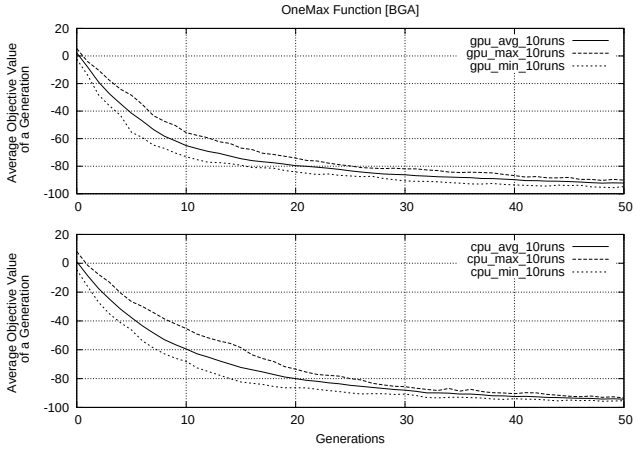


Fig. 2. Objective function value versus generation counter for OneMax problem using BGA. The top figure is obtained with a GPU based computer and the bottom figure is obtained with a serial computer.

### B. Effect of Number of CUDA Threads

Having demonstrated similar convergence characteristics, we are now ready to discuss the scale-up performance of the parallelized algorithms.

First, we perform a study of running times of GPU parallel code with varying number of threads in a block. For the study, the number of threads per block in all kernels has been kept same. We show the scale-up results using 4 to 512 threads on the one-max problem in Table I. It is observed that 64-256 threads per block take similar amount of computational time, for a problem of relatively smaller dimensions. Since, on increasing the number of dimensions the shared memory usage in one of the kernels increases, hence 64 threads per block is taken as optimal size for our subsequent studies.

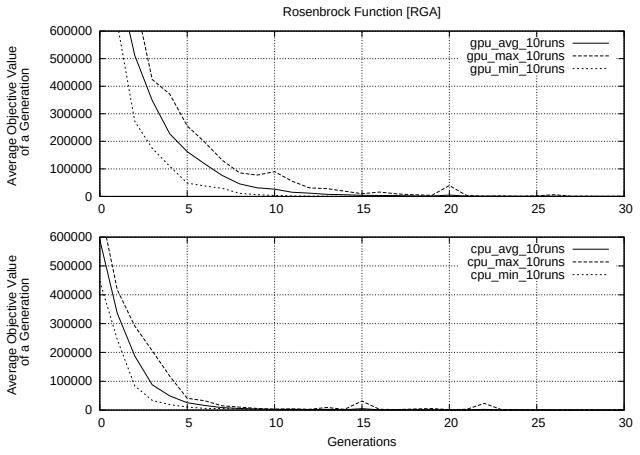


Fig. 3. Objective function value versus generation counter for Rosenbrock problem using RGA. The top figure is obtained with a GPU based computer and the bottom figure is obtained with a serial computer.

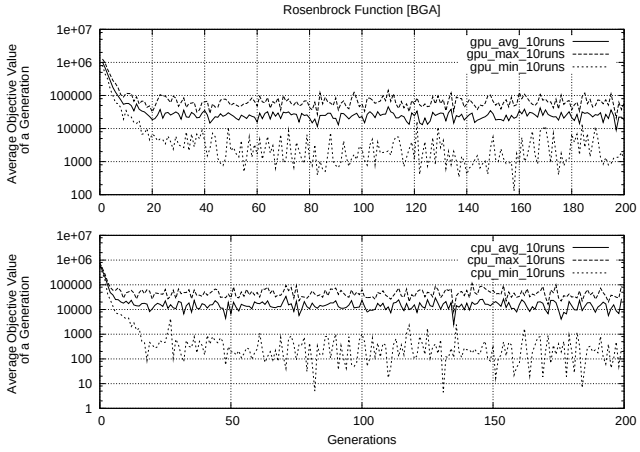


Fig. 4. Objective function value versus generation counter for Rosenbrock problem using BGA. The top figure is obtained with a GPU based computer and the bottom figure is obtained with a serial computer.

### *C. Effect of Problem and EA Parameters*

Next, we apply both our parallel implementations to the three test problems for studying the scale-up property of the algorithms:

1) Effect of number of decision variables
2) Effect of population size
3) Effect of problem complexity

Each run is terminated after 50 generations. Table II presents the results. The scale-up value is computed by dividing the overall computational time of serial implementation with respect to the time taken by the parallel implementations. Following observations can be made.

As the number of decision variables increase, the parallel RGA shows an increase in speed-up, while the parallel BGA shows its best performance for an intermediate size of the problem.

TABLE I
ONEMAX FUNCTION (10 VARIABLES, 25600 POPULATION, 50 GENERATIONS)

| Threads Per Block | GPU RGA Time | GPU BGA Time |
|---|---|---|
| 4 | 0.396730 | 0.220304 |
| 8 | 0.228432 | 0.141849 |
| 16 | 0.141425 | 0.091726 |
| 32 | 0.122048 | 0.086557 |
| 64 | 0.116688 | 0.085488 |
| 128 | 0.119170 | 0.085660 |
| 256 | 0.121322 | 0.086627 |
| 512 | 0.125622 | 0.087817 |

As the population size is increased, the speed-up increases and eventually for large population sizes the speed-up stabilizes to value around 40 for RGA and around 250-300 for BGA.

Interestingly, with the problem complexity (OneMax to Ellipsoidal to Rosenbrock), the speed-up is smaller for more complex problem, but as the population size is increased the speed-up gets more or less identical.

These results clearly indicate that for large population size applications, the use of parallel implementations of existing GA codes in a GPU platform is beneficial to a large extent.

A number of previous studies reported similar scale-up results on different problems. Yu et al [20] use a large population size (262,144) and reported a speed-up of about 15 with a real-coded GA with the BLX operator. While parallelizing genetic programming, another study [15] reported a speed-up of 8 to 80 using a 128 cores GeForce 8800GTX card. In the parallelization of NSGA-II, a recent study [7] reported a speed-up of 5.62 to 10.75 using the GeForce 9600GT (64 cores) card. Tsutsui and Noriyuki [9] reported a speed-up of 12 to 15 times in solving quadratic assignment problems using a parallelized GA using the GeForce GTX285 (240 cores). Maitre et al. [17] reported a speed-up of 30 to 100 with parallelization of solution evaluations alone. Compared to these results, our results (reporting a speed-up of 40 to 400) are comparable and in tune with the reported results.

We now describe the reasons for the discrepancy in the scale-up values of RGA and BGA implementations. As discussed in the *Mutation()* section , we have used a fast mutation scheme for BGA implementation, in which a single variable (or bit) from a solution is chosen at random and is mutated. In the serial BGA implementation, the mutation was implemented by checking whether to mutate every variable or bit, thereby consuming a lot of random number of generations. Since the BGA scale-up results in Table II are computed by comparing with a highly computationally expensive algorithm, the speed-ups are much higher compared to that in the RGA study. Table II (last three columns) presents the new scale-up values with BGA implementation where the serial BGA is run with the same modified mutation procedure. Now, the scale-up values are somewhat smaller than that with the original mutation scheme, still a speed-up of as high as 272 is achieved. The speed-up values reported

TABLE II
COMPARATIVE STUDY OF SPEED-UPS WITH VARYING PROBLEM COMPLEXITIES.

| Parameters | | GPU Speed-Up (RGA) | | | GPU Speed-Up (BGA) Original Mutation | | | GPU Speed-Up (BGA) Modified Mutation | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Pop Size | $nvar$ | One Max | Ellip-soid | Rosen-brock | One Max | Ellip-soid | Rosen-brock | One Max | Ellip-soid | Rosen-brock |
| 128 | 10 | 4.530 | 4.108 | 4.352 | 24.504 | 14.259 | 11.527 | 14.005 | 15.787 | 14.918 |
| | 50 | 6.816 | 7.054 | 8.706 | 74.795 | 39.111 | 22.827 | 45.765 | 24.285 | 24.285 |
| | 100 | 19.406 | 9.579 | 5.154 | 75.694 | 58.885 | 40.135 | 28.691 | 28.247 | 16.072 |
| 256 | 10 | 8.595 | 8.138 | 8.413 | 48.158 | 27.375 | 22.812 | 28.593 | 36.040 | 29.978 |
| | 50 | 24.020 | 12.776 | 16.419 | 89.691 | 81.441 | 44.524 | 72.452 | 51.354 | 31.224 |
| | 100 | 33.283 | 16.638 | 9.604 | 74.910 | 83.474 | 47.321 | 49.090 | 52.278 | 30.874 |
| 512 | 10 | 17.285 | 16.263 | 16.686 | 90.032 | 52.212 | 43.246 | 54.208 | 49.852 | 47.671 |
| | 50 | 38.284 | 20.057 | 25.103 | 125.202 | 133.726 | 83.546 | 77.135 | 86.572 | 55.619 |
| | 100 | 31.202 | 23.949 | 16.027 | 125.877 | 131.413 | 83.111 | 77.912 | 81.117 | 38.442 |
| 1024 | 10 | 30.571 | 30.347 | 30.930 | 83.328 | 93.035 | 81.341 | 49.747 | 63.547 | 53.501 |
| | 50 | 27.733 | 27.817 | 23.137 | 206.838 | 218.282 | 147.199 | 125.300 | 141.291 | 96.589 |
| | 100 | 31.328 | 31.754 | 24.914 | 171.379 | 179.083 | 129.943 | 105.838 | 112.854 | 84.537 |
| 2048 | 10 | 41.781 | 30.778 | 41.880 | 134.379 | 153.357 | 136.563 | 76.187 | 99.525 | 90.765 |
| | 50 | 30.474 | 31.563 | 30.125 | 301.810 | 322.046 | 236.445 | 184.379 | 203.610 | 155.352 |
| | 100 | 33.423 | 33.926 | 31.683 | 213.982 | 224.893 | 183.618 | 132.571 | 143.277 | 118.676 |
| 4096 | 10 | 31.647 | 31.861 | 34.159 | 192.200 | 221.112 | 204.687 | 112.701 | 145.078 | 135.182 |
| | 50 | 38.150 | 39.405 | 40.326 | 359.632 | 387.417 | 323.082 | 217.122 | 236.503 | 214.313 |
| | 100 | 40.135 | 41.280 | 41.446 | 229.276 | 241.694 | 221.386 | 142.748 | 152.668 | 143.615 |
| 8192 | 10 | 36.933 | 37.235 | 40.683 | 232.932 | 270.718 | 254.431 | 134.116 | 175.833 | 171.211 |
| | 50 | 40.952 | 42.277 | 45.370 | 386.951 | 414.549 | 379.928 | 234.846 | 262.600 | 249.188 |
| | 100 | 42.084 | 43.129 | 45.625 | 245.041 | 255.465 | 248.968 | 153.114 | 162.053 | 160.923 |
| 16384 | 10 | 40.176 | 40.851 | 43.536 | 254.440 | 289.063 | 273.735 | 149.438 | 194.114 | 183.355 |
| | 50 | 42.163 | 43.623 | 45.913 | 401.129 | 429.285 | 383.653 | 245.436 | 272.473 | 252.158 |
| | 100 | 42.889 | 43.922 | 45.666 | 254.614 | 267.727 | 255.964 | 158.879 | 170.433 | 165.722 |

in the table are phenomenal and clearly indicate an excellent match of a GPU facility (hardware) with an evolutionary algorithm (software) for solving optimization problems.

### *D. Effect of Evaluation Time*

Finally, we consider the effect of evaluating a single solution to the scale-up performance of both algorithms. To simulate such a case, we add an artificial delay in computing the objective function. We study the effect by introducing different delays in evaluating a solution (zero second to 1 sec). Table III shows computational time using RGA and the achieved scale-up values for the OneMax problem. It is evident from the table that as the delay in evaluation (simulating the evaluation time) increases, the speed-up achievable by our proposed parallelized RGA increases. This is because with the increase in evaluation time, the relative computational time for performing GA operations get more and more diminished and an algorithm will start to exhibit a linear speed-up. Since most problems in practice are computationally demanding, the high speed-up demonstrated in this study is encouraging and indicates the usefulness of practical EA applications on GPU based computing platforms.

TABLE III
ONEMAX FUNCTION (10 REAL VARIABLES, 960 POPULATION, 10 GENERATIONS). 64 THREADS ARE USED.

| Time Delay (sec) | CPU RGA Time | GPU RGA Time | SpeedUp |
|---|---|---|---|
| 0.0 | 0.071639 | 0.003365 | 21.289 |
| 0.00001 | 0.24069 | 0.002779 | 86.610 |
| 0.0001 | 1.09861 | 0.004818 | 228.022 |
| 0.001 | 4.91567 | 0.025108 | 195.781 |
| 0.01 | 48.812629 | 0.227881 | 214.202 |
| 0.05 | 238.70936 | 1.129175 | 207.854 |
| 0.1 | 478.14445 | 2.255732 | 211.969 |
| 0.2 | 952.91689 | 4.508898 | 211.341 |
| 1.0 | 4851.69365 | 22.534169 | 215.304 |

## VII. CONCLUSIONS

In this paper, we have modified two genetic algorithm implementations – binary-coded GA and real-coded GA – for their application with a General Purpose Graphics Processing Unit (GPGPU) facilitated by Nvidia's Tesla C1060 card.

Despite some preliminary studies in the past, our study has considered three different unconstrained optimization problems commonly used in the EA literature and has followed a detailed approach. First, along with parallelizing the task of solution evaluation (which is commonly achieved in parallel EA studies), we have parallelized the respective GA operators (such as the random number generation, initialization, selection operation, and mutation operation) such that the overall procedure is effective for the GPGPU application. Second, by performing a study on the effect of number of threads, we have found that around 64 threads provided the best speed-up performance. Third, a study on the effect of population size and problem size has revealed a speed-up of 40 to 400 on all three problems. Fourth, a study on the effect of evaluation time has indicated that the parallel implementation gets more effective in problems requiring more computational time per solution evaluation.

This study gives us inspiration and hope of integrating and running an evolutionary algorithm on a GPGPU aided desktop computer for achieving high-performing parallel applications. The achieved speed-ups are extremely encouraging. We are now exploring and extending the study for other EA applications, such as evolutionary multi-objective optimization, evolutionary multi-modal optimization, evolutionary combinatorial optimization and evolutionary robust and reliability based optimization.

## ACKNOWLEDGMENT

This work is supported by the Department of Science and Technology, India under its SERC funding scheme.

## REFERENCES

[1] D. E. Goldberg, *Genetic Algorithms for Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[2] E. Cantu-Paz, *Efficient and accurate parallel genetic algorithms*. Kluwer Academic Publishers, 2000.

[3] M. Bessaou, A. Petrowski, and P. Siarry, "Island model cooperating with speciation for multimodal optimization," in *PPSN VI: Proceedings of the Sixth International Conference on Parallel Problem Solving from Nature (PPSN-6)*. Springer-Verlag, 2000, pp. 437–446.

[4] M.-L. Wong, T.-T. Wong, and K.-L. Fok, "Parallel evolutionary algorithms on graphics processing unit," in *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, vol. 3, Sept. 2005, pp. 2286–2293 Vol. 3.

[5] S. Harding and W. Banzhaf, "Fast genetic programming on GPUs," in *Genetic Programming*, 2007, pp. 90–101.

[6] M. L. Wong, "Parallel multi-objective evolutionary algorithms on graphics processing units," in *GECCO '09: Proceedings of the 11th annual conference companion on Genetic and evolutionary computation conference*. New York, NY, USA: ACM, 2009, pp. 2515–2522.

[7] M. Wong and T. Wong, "Implementation of parallel genetic algorithms on graphics processing units," in *Intelligent and Evolutionary Systems*, 2009, pp. 197–216.

[8] G. Wilson and W. Banzhaf, "Deployment of cpu and gpu-based genetic programming on heterogeneous devices," in *GECCO '09: Proceedings of the 11th annual conference companion on Genetic and evolutionary computation conference*. New York, NY, USA: ACM, 2009, pp. 2531–2538.

[9] S. Tsutsui and N. Fujimoto, "Solving quadratic assignment problems by genetic algorithms with gpu computation: a case study," in *GECCO '09: Proceedings of the 11th annual conference companion on Genetic and evolutionary computation conference*. New York, NY, USA: ACM, 2009, pp. 2523–2530.

[10] W. Banzhaf, P. Nordin, R. Keller, and F. D. Francone, *Genetic Programming: An Introduction*. San Fransisco, CA: Morgan Kaufmann Publishers Inc., 1998.

[11] E. Cant-Paz, "A survey of parallel genetic algorithms," *CALCULATEURS PARALLELES*, vol. 10, 1998.

[12] E. Alba and J. M. Troya, "A survey of parallel distributed genetic algorithms," *Complex.*, vol. 4, no. 4, pp. 31–52, 1999.

[13] A. Verma, X. Llorà, D. E. Goldberg, and R. H. Campbell, "Scaling genetic algorithms using mapreduce," in *ISDA '09: Proceedings of the 2009 Ninth International Conference on Intelligent Systems Design and Applications*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 13–18.

[14] Q. Yu, C. Chen, and Z. Pan, "Parallel genetic algorithms on programmable graphics hardware," in *Advances in Natural Computation*, 2005, pp. 1051–1059.

[15] D. Robilliard, V. Marion-Poty, and C. Fonlupt, "Population parallel GP on the g80 GPU," in *Genetic Programming*, 2008, pp. 98–109.

[16] W. Banzhaf and S. Harding, "Accelerating evolutionary computation with graphics processing units," in *GECCO '09: Proceedings of the 11th annual conference companion on Genetic and evolutionary computation conference*. New York, NY, USA: ACM, 2009, pp. 3237–3286.

[17] O. Maitre, L. A. Baumes, N. Lachiche, A. Corma, and P. Collet, "Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA," in *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. New York, NY, USA: ACM, 2009, pp. 1403–1410.

[18] W. B. Langdon, "A fast high quality pseudo random number generator for nvidia cuda," in *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*. New York, NY, USA: ACM, 2009, pp. 2511–2514.

[19] K. Deb and R. B. Agrawal, "Simulated binary crossover for continuous search space," *Complex Systems*, vol. 9, no. 2, pp. 115–148, 1995.

[20] P. L. Yu, "A class of solutions for group decision problems," *Management Science*, vol. 19, no. 8, pp. 936–946, 1973.