

Accelerating Genetic Algorithm Using General Purpose GPU and CUDA

Rashmi Sharan Sinha¹, Satvir Singh², Sarabjeet Singh³ and Vijay Kumar Banga⁴

^{1,2}*Department of Electronics & Communication Engineering,
SBS State Technical Campus, Ferozepur–152004, (Punjab) India*

³*Department of Computer Science & Engineering,
SBS State Technical Campus, Ferozepur–152004, (Punjab) India*

⁴*Department of Electronics & Communication Engineering,
Amritsar College of Engg. & Tech., Amritsar–143001, (Punjab) India*

¹*sinharashmisinha@hotmail.com, ²drsativir.in@gmail.com*

³*sarabjeet_singh13@yahoo.co, ⁴v_banga@rediffmail.com*

Abstract

Genetic Algorithm (GA) is one of most popular swarm based evolutionary search algorithm that simulates natural phenomenon of genetic evolution for searching solution to arbitrary engineering problems. Although GAs are very effective in solving many practical problems, their execution time can become a limiting factor for evolving solution to most of real life problems as it involve large number of parameters that are to be determined. Fortunately, the most time-consuming operators like fitness evaluations, selection, crossover and mutation operations that involves multiple data independent computations. Such computations can be made parallel on GPU cores using Compute Unified Design Architecture (CUDA) platform. In this paper, various operations of GA such as fitness evaluation, selection, crossover and mutation, etc. are implemented in parallel on GPU cores and then performance is compared with its serial implementation. The algorithm performances in serial and in parallel implementations are examined on a testbed of well-known benchmark optimization functions. The performances are analyzed with varying parameters viz. (i) population sizes, (ii) dimensional sizes, and (iii) problems of differing complexities. Results shows that the overall computational time can substantially be decreased by parallel implementation on GPU cores. The proposed implementations resulted in 1.18 to 4.15 times faster than the corresponding serial implementation on CPU.

Keywords: Genetic Algorithm (GA), General Purpose Computing on Graphics Processing Unit (GPGPU), Compute Unified Device Architecture(CUDA)

1. Introduction

Genetic Algorithm (GA) is a swarm based global search algorithm inspired natural mechanism of genetical improvements in biological species [1], described by Darwinian Theory of *Survival of Fittest*. It was developed by John Holland in 1970 at University of Michigan [2]. It simulates the process for evolving solutions to arbitrary problems [3].

GA is algorithm involves multiple solutions represented by a string of variables, analogous to the chromosomes in Genetics. With a initially randomly generated population, every swarm member is a required to be evolved. Evolution is based on the fitness pairs of parent solutions that are selected randomly and reproduce next generation of solutions, stochastically. Each child chromosome has features of both the parent as an output of crossover. Another is limited alteration in feature values of the generation represents effect of mutation.

GA essentially strives to attain the global maximum (or minimum) of cost depending upon the nature of the problem. Over the period of advancements, GA is widely used and extensively researched as optimization and search tools in several fields such as, medical, engineering, and finance etc. The basic facts for their success are simple structure, broad relevance with problem [4]. Goldberg and Harik brought the term compact Genetic Algorithm (cGA) which represents the solution as a probability distribution over the wide space set of solutions, Huanlai and Rong well utilized the concept in minimization problem of resources of network codes [5]. Gas produces high-quality solutions through its high adaptation property with the environment changes [6]. Prakash and Deshmukh investigated the use of meta-heuristics for combinatorial decision-making problem in flexible manufacturing system with GA [7]. Prominent GA applications include pattern recognition [8], classification problems [9], protein folding [10] and neural network design [11] *etc.* GAs are also suitable for multi-objective optimal design problems [12], in solving multiple objectives.

Even though, GAs has powerful characteristic in determining many practical problems. However their execution time act as bottleneck in few real life problems. GA involve large number of trial vectors that needed to be evaluated. However, the major portion of time consuming function of fitness evaluations can be made parallel to perform independently due to data independency and, therefore, can be evaluated using parallel computational mechanisms.

With the advent of General Purpose GPU (GPGPUs), researchers have been evolving Evolutionary Computations [13-16] for parallel implementation. Similar advancements in the field of genetic programming are quickly adopted by GA researchers.

After this brief background, the remaining paper is organized as follows: Section II, describes GA Operators along with pseudo codes for implementation in parallel. Section III, introduces architecture detail of GPGPU and C-CUDA followed by Section IV of its implementation. Section V, summarizes performance evaluation of experimental result. Conclusion of this paper with future aspect is presented in Section VI.

2. GA Operators

GA provides number of solutions however best solution among them is one with least processing time [17]. The three primary operators involved in GA are: (1) Selection, (2) Crossover, (3) Mutation and (4) Elite Solutions operators described as follows.

2.1. Selection

There are three most popular different types of Selection Strategies viz., Tournament Selection, Ranked-Based Roulette Wheel, and Roulette Wheel Selection [18]. These strategies are used to search potential parent chromosomes based on the fitness level of individuals from the randomly generated population. The selection operator is expected to produce solutions with higher fitness in succeeding generations. On contrarily, selection operator anticipated to produce relative probability of being selected according to their fitness in the swarm. This leads the algorithm to find global best solution rather than converging to its nearest local best solution.

2.1.1. Tournament Selection: The mechanism of tournament selection is based upon random selection of solutions from current population. The selected solutions form a pool of solutions, which produces optimal solution among them for succeeding generation. The selection is done on the highest fitness among the pool of solution.

2.1.2. Rank-based Roulette Wheel Selection: The strategy of Rank-based Roulette Wheel Selection where fitness of each solution is given a rank relative to swarm, deals with the rank of solution rather than fitness value. The chance of selection of chromosome

is distributed rationally to the rank of individual solution. Rank-based Roulette Wheel Selection avoid premature conversions significantly.

2.1.3. Roulette Wheel: In this method, the selection of parent solutions for the next generation child solutions depend upon the probabilities of fitness values relative to portion of spinning a roulette wheel. The chromosomes are chosen for next generation on the basis of their values of fitness, i.e., a chromosome's selection is directly proportional to section of roulette wheel corresponding to the fitness level of the same. Let $f_1, f_2, f_3, \dots, f_n$ be fitness values of chromosomes 1, 2, 3, . . . n. Then the probability of selection P_i for chromosomes i is defined as (1),

$$P_i = \frac{f_i}{\sum_{j=1}^n f_j}$$

Advantage of proportional roulette wheel selection is that it selects all of the solutions of swarm with the probability relational to fitness values. Hence it maintains diversity of solution.

2.1.4. Parallel implementation of Selection: In Roulette wheel selection function there is a global call of kernel for execution of GA on GPU. The thread number per block *threadIdx* is equal to the respective dimension of population. The selection is done in parallel generating uniformly distributed random numbers between zero to max (cumulative sum) and thereby checking which of the fitness lies immediate greater than that of generated number. Then the corresponding fitness of the trial solution gets selected as parent chromosome for next generation as depicted in Algorithm 1.

Algorithm 1: Pseudocode for Roulette Wheel Selection

Global call of kernel for Roulette Wheel Selection function

No. of threads i is equal to *threadIdx*

Random number $r \leftarrow (0, \text{cumulative fitness})$

While *size of population* < *pop_size* **do**

 Generate random number r equal to *pop_size*

 Calculate fitness (p_i), cumulative sum of fitness (*CSum*)

 Spin the wheel *pop_size* times with random force

If *CSum* < r **then**

 Select the *first* chromosome, otherwise,

 Select the j^{th} chromosome

End if

End While

 Return solution with the fitness value proportion to
 the size of selected chromosome on roulette wheel

End

2.2. Crossover

The process of producing child chromosomes from parent chromosomes is termed as crossover. It is a significant operator which mimics biological crossover and reproduction of the nature. This operator in GA is broadly classified into three different techniques viz., single point, double point, and uniform distribution crossover.

2.2.1. Single Point Crossover: In single point crossover, the selected parent solution chromosome string get swapped from a randomly selected crossover point. The resulting chromosome after swapping form children population for next generation.

2.2.2. Double Point Crossover: Double point crossover is similar to that of single point crossover however; the crossover points are two rather than one.

2.2.3. Uniform Distribution Crossover: In Uniform Distribution Crossover technique, chromosomes of parent solution are mixed uniformly with a fixed ratio termed as mixing ratio. The process of mixing parent chromosomes produces child chromosomes mixed at gene level as compared with single and double point crossover where mixing is done at segment level. Therefore uniform crossover is more suitable for larger search space. Hence in this paper uniform distribution crossover is used.

2.2.4. Parallel Implementation Uniform Crossover: In uniform crossover, there is a global call of kernel for execution of the function on GPU. Uniformly distributed random number is generated at the interval 0 to 1 while probability of crossover is defined at 0.9. The mixing ratio of 0.8 is applied at gene level to produce child chromosome. The pseudocode for its parallel implementation is shown in Algorithm 2.

Algorithm 1: Pseudocode for Uniform Crossover

Global call of kernel for uniform crossover function

No. of threads i is equal to $threadIdx$

N is population size pop_size

L is chromosome length of string $chromoLength$

Crossover Probability is defined as $probCross$

Mixing Ratio is defined as $mixRatio$

$r \leftarrow$ random no. between 0 to 1

if $r \geq probCross$ **then**

if $r \geq mixRatio$ **then**

$crossPoint(i) \leftarrow$ random (0, L-1)

$crossPoint(i+1) \leftarrow crossPoint(i+1)$

Else

$crossPoint(i) \leftarrow$ no change

$crossPoint(i+1) \leftarrow$ no change

End if

End if

End

2.3. Mutation

Mutation operator is applied to preserve genetic variance (diversity) in succeeding generation of population in GA. Mutation operator creates a new solution for each possible trial solution. To avoid optimal search stagnation, the difference between two chromosome is increased by a factor termed as mutation factor. In this experiment, the factor is kept relative to the number of iteration between 0.01 to 1.

2.3.1. Parallel Implementation Mutation:

Pseudocode represents the process carried out for mutation in GA on GPU Algorithm 3. There is a global call of kernel for execution of the function on GPU. Each solution of

the population get mutated by a single thread operations. Scheduling a block with a sufficient number thread is needed to mutate the whole population.

Algorithm 1: Pseudocode for Mutation

Global call of kernel for mutation function
 No. of threads i is equal to $threadIdx$
 Mutation factor is defined as m_fact
 Obtain population after crossover new_Pop
 Random no. r is generated between 0 to 1
for $i=0$ to n
 if $r < m_fact$
 $new_Pop = 1 - new_Pop$
Else
 $new_Pop = new_Pop$
End if
End

2.4. Elite Solutions

After the crossover and mutation operation, elite solution is applied. In this solution elite string is compared with parent chromosomes and current child chromosomes of entire solution. Elite solution is updated, if any solution in the child population is superior to the solution in elite string. When elite string stop showing any further improvement, it reflects the convergence of the swarm.

3. Basic GPGPU and C-CUDA

3.1. General Purpose GPU

The architectures of General Purpose Graphics Processing Unit (GPGPU) is highly parallel, data processing unit endorsed with multiple number of streaming processors. GPGPU interfaced with Compute Unified Device Architecture (CUDA). It was developed by nVIDIA Corporation in 2006 along with Geforce80 and programming model on CUDA platform [17]. It supports execution of arithmetic operations at higher rate, substantial hardware is computationally powerful then CPUs.

Libraries such as *curand kernel.h*, *cuda.h* and *curand.h* etc., along with C language libraries provide high freedom of accessibility to interface user with General Purpose GPU. Researches and experiments in last few years prove its significance in several fields. On contrary it also have low cost and higher power-to-watt ratio as compared to CPUs [19]. In recent years, such features attracted lots of researcher and developers to harness GPUs for various general purpose computations (GPGPU).

Oiso Matumura achieved a speedup rate of 6x in evaluation of Steady State GA with population size of 256, taken benchmarking test functions to compared to the CPU implementation [19]. Jiri and Jaros proposed the application of GA by achieving speedup of 35-781 for determining the Knapsack Problem with a multi-GPU while population size was kept 128 to 2048 individuals per island [20]. Arora, Tulshyan and Deb employed GPGPU to apply a real and binary coded GA and talked about several data structures application on GPU, and archived a speedup of 40-400 for a population size of 128-16384 as compared to its sequential execution [21].

GA proves suitable in determining several realistic problems [22]. Therefore in this experiment, simultaneous kernel process taken out on GA using GPU. The performance

evaluation of single objective GA on set of benchmark test function using nVIDIA GeForce GT 740M GPU is used specifically. It gives speedup of 1.18-4.15 times as compared to CPU execution time.

3.2. Compute Unified Device Architecture (CUDA)

The program based on GPGPU can be easily developed using CUDA architecture. The execution of CUDA program composes of two parts: *host* section and *device* section. The *host* section is executed on CPU while the *device* section is executed on GPU respectively. However the execution of *device* section on GPU managed by kernel. The kernel handles synchronization of executing threads. It is invoked by *device* call for GPU.

The threads in GPU architecture can be grouped into *blocks* and *grids* as depicted in Figure 1. In GPU *grid* is with group of thread *blocks*, and a thread *block* comprises definite number of threads per *block*. Differentiating between unique threads, thread *block* and *grid* may be done by using a set of identifiers *threadIdx*, *blockIdx* and *gridIdx* variables respectively. Thread per *block* can exchange information to synchronize with each other. Per-*block* shared memory can be used for communication between each thread within a thread *block*, however there no direct interaction or synchronization possible between the threads of different *blocks* [19].

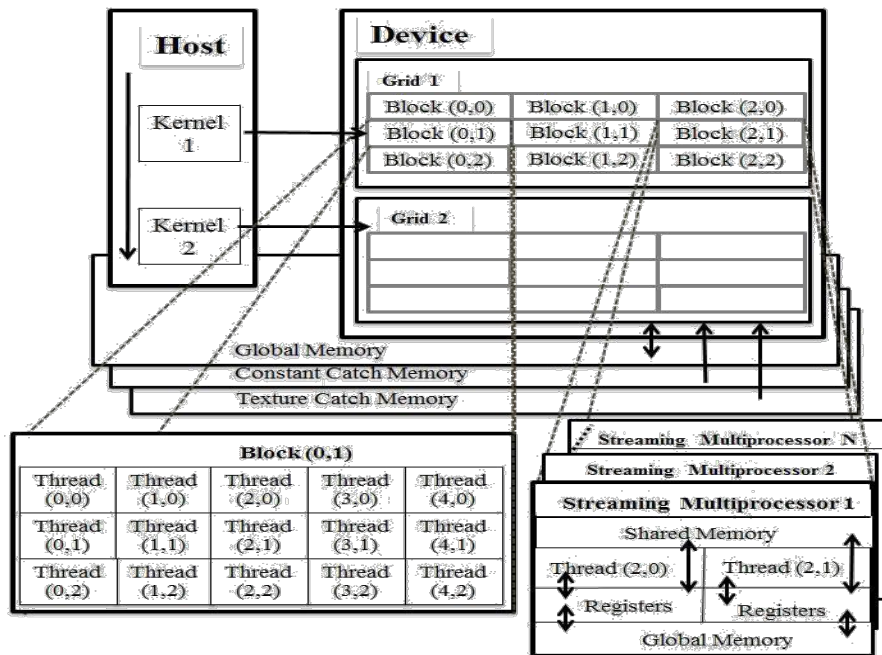


Figure 1. CUDA Architecture

The entire shared memory in CUDA architecture is divided into four types viz., texture memory, constant memory, perthread private local memory and global memory for data shared by all threads. Between these memories, texture memory and constant memory can be accumulated as fast read only caches; while registers and shared memories are the overall fastest memories.

For constant memory, the optimal access strategy adopts reading of same memory location by all threads. The threads can read neighboring thread addresses using texture cache with a high reading efficiency. CUDA functions for allocation and deallocation of memory *cudaMalloc* and *cudaFree*, respectively are used. CUDA function *cudaMemcpy* is used to copy data from host to device.

There are multiple Streaming Processors to handle GPU computations. It is considered as fundamental processor of *device* architecture. While Streaming Multiprocessors had to run on group of streaming processors. The number of thread *block* in streaming processors is scheduled by GPU *device* when kernel function is called.

When threads executing in a group of 32 streaming multiprocessors it is called *wrap* under the Single Instruction, Multiple Thread (SIMT) scheme, i.e., in nVIDIA GeForce GT 740M have 16KB of shared memory per streaming multiprocessor with 16384 64-bit registers. Shared memory and registers limits the thread *block* per streaming multiprocessors while executing threads. Hence streaming multiprocessors is limited up to 8 *blocks*.

4. Implementing GA Using C-CUDA

Implementation of GA includes parallel flow of algorithm to find global optimal solution using C-CUDA. The major implementation of algorithm consists generation of initial population using GPU, randomly generated numbers to find global best solution, selection of parent solution, implementation of genetic operators and elite solution and finally coping child population back to parent population. The overview of GA execution is depicted in Figure 2. The implementation of C-CUDA kernels on GA is based on under mentioned principle:

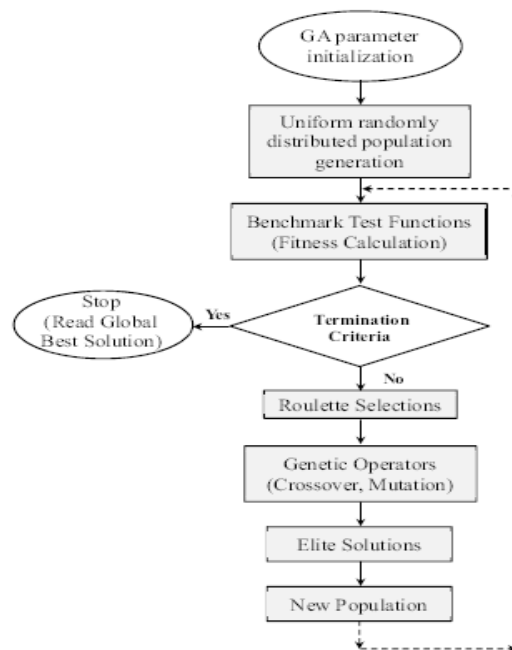


Figure 2. Implementation GA Flowchart (Shaded Modules Represents GPU Computation, Non-shaded Modules Represents Computation on CPU)

4.1. All GA solution is calculated using thread block at each generation. The maximum size of GA population at each generation is limited to the total number of threads which is currently $(2^{16}-1)^2$.

4.2. Every trial solution uses threads to compute possible outcome. GPU's computation capability is 512 threads per block for 1 x 1024. Hence it is directly proportional to the hardware capability.

4.3. GPU access all the trial solution in one step i.e., with each kernel call C-CUDA launches number of threads per block equivalent to the population size of the generation.

These characteristics turn out to be best feature for massive implementation of such algorithms. It easily provides speedup in overall computation time of GA. C-CUDA kernels generates population in one step then computes their respective fitness values. The genetic operator is applied to each solution where number of thread kept equal to its population size. The new generation of succeeding population follow same strategy to find solution. Hence due to these benefits of GPU takes less time as compared to its sequential execution on CPU.

5. Performance Evaluation

5.1.Experimental Setup

The experiments were conducted on two different PCs (PC1 and PC2) and with same nVidia cards (Refer Table I for System Specifications) for separate performance evaluations. PC1 is tested with active background applications. PC2 is kept ideal until complete performance evaluation carried out. The total number of streaming processors and streaming multiprocessors are 16 hence 256 streaming processors in each PC. Entire GA code of C-CUDA is written in Visual Studio C++ (2012 release mode) and complied on nvcc compiler. The result of above experiment is evaluated using two different iteration size of 10,000 and 100,000. The dimension size of experiment is kept fixed. The acceleration of GA on GPU is seen efficient with large number dimension size and maximum iteration. The result Table III and Table IV in next section shows a significant speedup.

Table I. Computational Systems Specification

Platform		PC1	PC2
CPU	Processor	Intel Core i5 4200(U)+ 2.6 GHz	Intel Core(TM) i5 333TU+ 1.8 GHz
	Catch	3072KB	3072KB
	Memory	16GB DDR3L	32 GB DDR3/L
	Interface	PCI-E 2.0	PCI-E 3.0
GPU	Graphic Card	nVIDIA GeForce GT 740M	nVIDIA GeForce GT 740M
	Version	9.18.13.2057	9.18.13.2702
	CUDA Version	5.5	5.5

5.2.Benchmark Test Functions

Benchmark Test Functions for our experiment with distinct minima (f_{\min}) is given in Table II, which are numbered from $f1(x)$ up to $f7(x)$ and correspond to the functions in [23]. Here, range is the lower and upper limits of the universal discourse for every function.

Table II. Benchmark Functions with Different Local Minima [23]

Test Functions	Range of x_i	f_{\min}
$f_1(x) = \sum_{i=1}^n x_i^2$	± 5.12	0
$f_2(x) = \sum_{i=1}^n x_i^4$	± 100	0
$f_3(x) = \frac{1}{4000} \sum_{i=1}^n x_i^2 - \prod_{i=1}^n \cos(x/\sqrt{i}) + 1$	± 2048	0
$f_4(x) = \sum_{i=1}^n x_i^5 - 3x_i^4 + 4x_i^3 + 2x_i^2 - 10x_i - 4 $	± 10	0
$f_5(x) = \sum_{i=1}^n x_i \sin(x_i) + 0.1x_i $	± 10	0
$f_6(x) = -\exp\left(-0.5 \sum_{i=1}^n x_i^2\right)$	± 1	1
$f_7(x) = \sum_{i=0}^{n-1} [100(x_i - x_{i+1}^2)^2 + (1 - x_i)^2]$	± 2048	0

5.3.Result and Discussion

The ratio of total time consumed on CPU for serial implementation of code to the time consumed by GPU C-CUDA in parallel implementation gives speedup for the same experiment on different platforms.

5.3.1. Case Study 1: In this experiment, the dimension size of the population generation is kept first 32 and then 64. Each dimension size iterated for maximum number of iteration, which was set to 10,000. The performance shown in result table is average value of 20 trials. The speedup of GPU over CPU for all seven benchmark test functions are shown in Table III and indicated in Figure 3 & Figure 4.

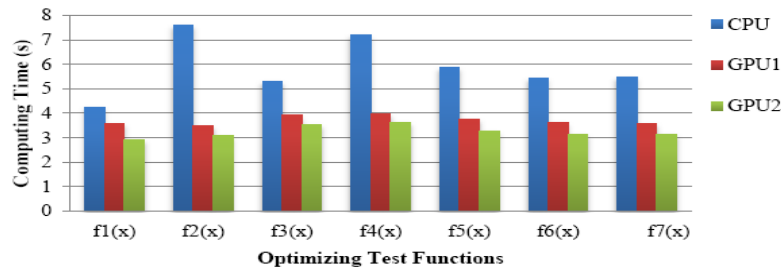


Figure 3. Computing Time for 10,000 Iterations with 32 Dimension Size

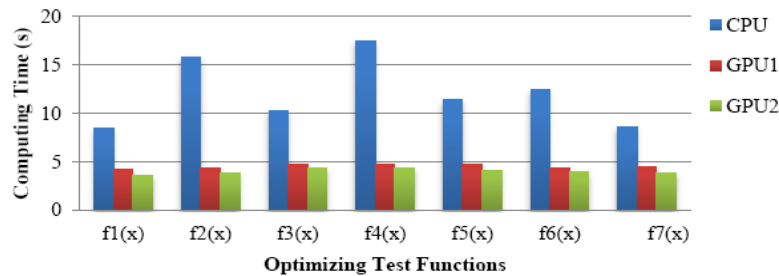


Figure 4. Computing Time 10,000 Iteration with 64 Dimension Size

The best computational performance achieved among GPU1 and GPU2 for dimension size of 32, is 2.17 times for f2(x) on GPU1, while on GPU2 with the dimension size of 64,

$f_2(x)$ shows a speedup of 4.15 times higher than its CPU execution time. The Table III depicts the best result for 10,000 iterations along with quality.

Table IV. C-CUDA Vs. C Performances for 100,000 Iterations

n	Function	CPU		GPU ₁			GPU ₂		
		Time (sec)	Std. Div.	Time (sec)	Std. Div.	Speed Up	Time (sec)	Std. Div.	Speed Up
32	$f_1(x)$	4.24	11.59	3.58	0.1300	1.18	2.92	0.0006	1.45
	$f_2(x)$	7.60	2.43	3.50	0.0003	2.17	3.07	0.0005	1.47
	$f_3(x)$	5.31	0.82	3.91	0.0057	1.35	3.54	0.0006	1.50
	$f_4(x)$	7.20	9.74	3.97	0.0004	1.81	3.61	0.0007	1.99
	$f_5(x)$	5.86	0.91	3.75	0.0212	1.56	3.27	0.0004	1.79
	$f_6(x)$	5.43	17.43	3.62	0.0006	1.50	3.15	0.0080	1.72
	$f_7(x)$	5.48	16.26	3.60	7.9E-05	1.52	3.13	0.0015	1.75
64	$f_1(x)$	8.42	11.73	4.28	0.0007	1.96	3.65	0.0004	2.30
	$f_2(x)$	15.84	14.77	4.30	0.0018	3.68	3.81	0.0090	4.15
	$f_3(x)$	10.30	6.61	4.78	0.0002	1.55	4.41	0.0003	2.33
	$f_4(x)$	17.47	10.20	4.75	0.0180	3.68	4.39	0.0007	3.97
	$f_5(x)$	11.43	10.81	4.68	0.0271	2.44	4.08	0.0004	2.80
	$f_6(x)$	12.49	4.04	4.33	0.0280	2.88	3.96	0.0013	3.15
	$f_7(x)$	8.58	4.41	4.50	0.0006	1.91	3.85	0.0010	2.23

5.3.2. Case Study 2: In this experiment the dimension size is kept same as Case Study 1, however the iterations size increased to 100,000. The respective speedup for all seven benchmark test functions is shown in Table IV and indicated in Figure 5 and Figure 6.

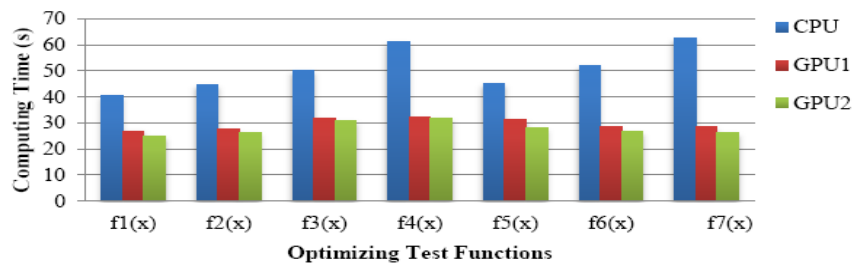


Figure 5. Computing Time for 100,000 Iterations with 32 Dimension Size

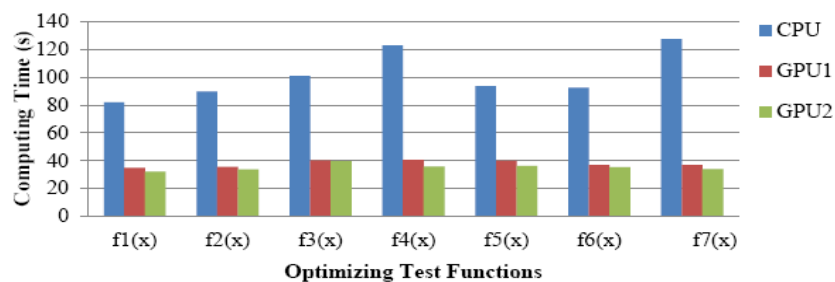


Figure 6. Computing Time for 100,000 Iterations with 64 Dimension Size

The highest speedup achieved in this case for dimension size 32 among GPU1 and GPU2 is 2.39 for test function $f_7(x)$. On the other hand, dimension size 64 has best speedup of 2.78 times for $f_7(x)$. The GPU average execution time is 33.70 seconds while 127.52 seconds in CPU.

Table IV. C-CUDA Vs. C Performances for 100,000 Iterations

n	Function	CPU		GPU ₁			GPU ₂		
		Time (sec)	Std. Div.	Time (sec)	Std. Div.	Speed Up	Time (sec)	Std. Div.	Speed Up
32	$f_1(x)$	40.61	0.11	26.92	0.0150	1.51	24.73	0.0016	1.62
	$f_2(x)$	44.81	0.48	27.55	0.0286	1.63	26.05	0.0019	1.72
	$f_3(x)$	50.06	0.30	31.90	0.0133	1.57	30.96	0.1046	1.62
	$f_4(x)$	61.31	0.23	32.22	0.0006	1.90	31.57	0.0009	1.94
	$f_5(x)$	45.24	2.07	31.14	0.0087	1.45	28.10	0.0019	1.61
	$f_6(x)$	52.16	0.44	28.61	0.0580	1.82	26.76	0.0026	1.95
	$f_7(x)$	62.69	0.09	28.76	0.0105	2.18	26.27	0.0031	2.39
64	$f_1(x)$	81.86	0.62	34.84	0.1179	2.35	32.06	0.0012	2.55
	$f_2(x)$	89.77	0.31	35.42	0.0779	2.53	33.43	0.0017	2.68
	$f_3(x)$	101.13	0.75	40.01	0.0510	2.53	39.51	0.0007	2.56
	$f_4(x)$	122.97	0.13	40.53	0.0047	3.03	35.61	0.0030	3.45
	$f_5(x)$	93.61	2.07	39.63	0.0142	2.36	36.25	0.0198	2.58
	$f_6(x)$	92.36	0.44	36.83	0.0090	2.51	35.30	0.6523	2.61
	$f_7(x)$	127.52	0.08	36.71	0.1229	3.47	33.70	0.0026	3.78

It is considered that the GPU implementation can conceal the latency of memory access by executing many threads in parallel, while the CPU implementation executes the GA calculation sequentially. In particular, it is notable that our implementation to parallelize the process of both individuals and their data is more effective, because the implementation enables the execution of more threads than others. In addition, most GA processes are executed on GPU. This can suppress the frequency of data transfer between the host and the device, which is probably the bottleneck to speed up by GPU.

6. Conclusion

In this paper, the implementation of GA on GPGPU using C-CUDA is carried out. The massive parallel architecture of GPU exploited to attain required speedups in GA. It shows acceleration of 1.18-4.15 times as compared to its sequential execution on CPU on variety benchmark test functions. From this result it is concluded that, the algorithm can be made more optimized for several search problems to enhance its wide variety of features. In future work, the performance of GA model will be more improved by modifying single objective GA to multi-objective GA. Further improvement of this model will be done by implementing multi-objective GA model with Fuzzy logic system which is expected to a fast parallel approach.

References

- [1] L. De Giovanni and F. Pezzella, "An improved genetic algorithm for the distributed and flexible job-shop scheduling problem", European journal of operational research, vol. 200, no. 2, (2010), pp. 395-408.
- [2] J. H. Holland, "Genetic algorithms and the optimal allocation of trials", SIAM Journal on Computing, vol. 2, no. 2, (1973), pp. 88-105.
- [3] K. Ghoseiri and S. F. Ghannadpour, "Multi-objective vehicle routing problem with time windows using goal programming and genetic algorithm", Applied Soft Computing, vol. 10, no. 4, (2010), pp. 1096-1107.
- [4] K. P. Murphy, "Machine learning: a probabilistic perspective", MIT press, (2012).
- [5] H. Xing and R. Qu, "A compact genetic algorithm for the network coding based resource minimization problem", Applied Intelligence, vol. 36, no. 4, (2012), pp. 809-823.
- [6] S. Yang, H. Cheng and F. Wang, "Genetic algorithms with immigrants and memory schemes for dynamic shortest path routing problems in mobile ad hoc networks", Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on, vol. 40, no. 1, (2010), pp. 52-63.
- [7] A. Prakash, F. T. Chan and S. Deshmukh, "Fms scheduling with knowledge based genetic algorithm approach", Expert Systems with Applications, vol. 38, no. 4, (2011), pp. 3161-3171.

- [8] J. Adams, D. L. Woodard, G. Dozier, P. Miller, K. Bryant and G. Glenn, "Genetic-based type ii feature extraction for periocular biometric recognition: Less is more", Pattern Recognition (ICPR), 2010 20th International Conference on. IEEE, (2010), pp. 205-208.
- [9] A. Quteishat, C. P. Lim and K. S. Tan, "A modified fuzzy min-max neural network with a genetic-algorithm-based rule extractor for pattern classification", Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions, vol. 40, no. 3, (2010), pp. 641-650.
- [10] Y. Zhang and L. Wu, "Artificial bee colony for two dimensional protein folding," Advances in Electrical Engineering Systems, vol. 1, no. 1, (2012), pp. 19-23.
- [11] L. Magnier and F. Haghighat, "Multiobjective optimization of building design using trnsys simulations, genetic algorithm, and artificial neural network", Building and Environment, vol. 45, no. 3, (2010), pp. 739-746.
- [12] S. Omkar, J. Senthilnath, R. Khandelwal, G. Narayana Naik and S. Gopalakrishnan, "Artificial bee colony (abc) for multi-objective design optimization of composite structures", Applied Soft Computing, vol. 11, no. 1, (2011), pp. 489-499.
- [13] F. Fabris and R. A. Krohling, "A co-evolutionary differential evolution algorithm for solving min-max optimization problems implemented on gpu using c-cuda", Expert Systems with Applications, vol. 39, no. 12, (2012), pp. 10 324-10 333.
- [14] O. Maitre, F. Krüger, S. Querry, N. Lachiche and P. Collet, "Easea: specification and execution of evolutionary algorithms on gpgpu", Soft Computing, vol. 16, no. 2, (2012), pp. 261-279.
- [15] O. Maitre, N. Lachiche and P. Collet, "Fast evaluation of gp trees on gpgpu by optimizing hardware scheduling", Genetic Programming, Springer, 2010, (2011), pp. 301-312.
- [16] M. A. Franco, N. Krasnogor and J. Bacardit, "Speeding up the evaluation of evolutionary learning systems using gpgpus", Proceedings of the 12th annual conference on Genetic and evolutionary computation. ACM, (2010), pp. 1039-1046.
- [17] S. Ryoo, C. I. Rodrigues, S. S. Bagsorkhi, S. S. Stone, D. B. Kirk and W.-M. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda", Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, ACM, (2008), pp. 73-82.
- [18] M. R. Noraini and J. Geraghty, "Genetic algorithm performance with different selection strategies in solving tsp", Proceedings of the World Congress on Engineering 2011, vol. II, (2011).
- [19] M. Oiso, T. Yasuda, K. Ohkura and Y. Matsumura, "Accelerating steadystate genetic algorithms based on cuda architecture", Evolutionary Computation (CEC), 2011 IEEE Congress on. IEEE, (2011), pp. 687-692.
- [20] P. Pospíchal, J. Schwarz and J. Jaros, "Parallel genetic algorithm solving 0/1 knapsack problem running on the gpu", 16th International Conference on Soft Computing MENDEL, vol. 2, (2010), pp. 64-70.
- [21] R. Arora, R. Tulshyan and K. Deb, "Parallelization of binary and real-coded genetic algorithms on gpu using cuda", Evolutionary Computation (CEC), 2010 IEEE Congress, IEEE, (2010), pp. 1-8.
- [22] M. Oiso, Y. Matsumura, T. Yasuda and K. Ohkura, "Evaluation of generation alternation models in evolutionary robotics", Natural Computing. Springer, (2010), pp. 268-275.
- [23] M. Jamil and X.-S. Yang, "A literature survey of benchmark functions for global optimisation problems", International Journal of Mathematical Modelling and Numerical Optimisation, vol. 4, no. 2, (2013), pp. 150-194.

Authors



Rashmi Sharan Sinha was born on Apr 6, 1987. She received her Bachelor's degree (B.Tech.) from Lala Lajpat Rai Institute of Engineering and Technology, Moga, in year 2011 and Master's degree (M.Tech.) from Shaheed Bhagat Singh State Technical Campus (formerly, SBS College of Engineering & Technology), Ferozepur, Punjab (India) with specialization in Electronics & Communication Engineering in year 2015. Her research interests include Evolutionary Algorithms and Fuzzy Logic System.



Satvir Singh was born on Dec 7, 1975. He received his Bachelor's degree (B.Tech.) from Dr. B. R. Ambedkar National Institute of Technology, Jalandhar, Punjab (India) with specialization in Electronics & Communication Engineering in year 1998, Masters degree (M.E.) from Delhi Technological University (Formerly, Delhi College of Engineering), Delhi (India) with distinction in Electronics

& Communication Engineering in year 2000 and Doctoral degree (Ph.D.) from Maharshi Dayanand University, Rohtak, Haryana (India) in year 2011.

His fields of special interest include Evolutionary Algorithms, High Performance Computing, Type-1 & Type-2 Fuzzy Logic Systems, Wireless Sensor Networks and Artificial Neural Networks for solving engineering problems. He is active member of an editorial board of International Journal of Electronics Engineering and published nearly 15 research papers in International Journals and Conferences. He has delivered nearly 15 Invited Talks during National and International Conferences, Seminar, Short Term Courses and Workshops. He completed two AICTE funded projects under MODROB Scheme worth 15 Lacs and conducted a 2-Week Staff Development Programme on “Intelligent Computational Techniques”.



Sarabjeet Singh received the B.Tech degree in Computer Science & Engineering from National Institute of Technology Jalandhar, M.Tech degree in Computer Science & Engineering from Panjab University Chandigarh. He is currently working in Shaheed Bhagat Singh State Technical Campus, Ferozepur. His current research interests include Computer Graphics and High Performance Computing.



Vijay Kumar Banga received the B. Tech degree in Electronics and Communication engineering from Punjabi University Patiala, M. Tech degree in Electronics and Communication Engineering from Panjab University Chandigarh and Phd. from Thapar University Patiala. He is currently working as Principal in Amritsar College of Engineering and Technology, Amritsar. His current research interests include Soft Computing, Image Processing, Autonomous Robots.

