Speedup Genetic Algorithm using Parallel Processing Method

Hak Kun Ri*, Chol Hun Pak and Nam Song An
*Faculty of Information Science, **Kim Il Sung** University, Taesong District, Pyongyang, DPR Korea*

Abstract
Genetic Algorithm (GA) is one of most popular swarm based evolutionary search algorithms that involve multiple data independent computations. Such computations can be made in parallel processing method 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.
Result shows that the overall computational time can substantially be decreased by parallel implementation on GPU cores. The proposed implementations resulted in 1.18 to 3.68 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 for genetical improvements in biological field. 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 resea rched as optimization and search tools in several fields such as, medical, engineering, and finance etc. The basic fact 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].

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] and protein folding [10] etc. GAs are also suitable for multi-objective optimal design problems. Even though GAs has powerful characteristic are capable of determining many practical bottleneck problems, their execution time acts as bottleneck in some practical problems. GA is accompanied by a large number of trial vectors to compute. However, most of the execution time is spent for evaluating fitness and data being available for parallel processing due to data independency, whose performance can be evaluated using parallel computing mechanism.

* Corresponding author
Email addresses: *LHK1972612@star-co.net.kp* (Hak Kun Ri)

2. Implementation of GA operators

Amidst many solutions provided by GA, it is the best one that has the shortest processing time.GA consists of 4 major parallel processes; selection, crossover, mutation and operators.

1) Selective parallel processing

Roulette function is used for selective parallel processing. 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 by generating uniformly distributed random numbers from 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 get selected as parent chromosome for next generation as depicted in Algorithm 1.

Algorithm 1 Pseudocode for Roulette Selection

Global call of kernel for Roulette Wheel Selection function
No. of threads i is equal to threadIdx
Random number r ← (0, cumulative fitness)
While size of population < pop_size do
Generate random number r equal to pop_size
Calculate fitness (pi), 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 jth chromosome
End if
End While
Return solution with the fitness value proportion to the size of selected chromosome on roulette wheel
End

2) Parallel Implementation of Uniform Crossover

In Uniform Distribution Crossover technique, chromosome of parent solution is mixed uniformly with a fixed ratio in terms of 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. 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 2: 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 ← random no. between 0 to 1
if r ≥ probCross **then**
if r ≥ mixRatio **then**
crossPoint(i)← random (0, L-1)
crossPoint(i+1)← crossPoint(i+1)
Else

crossPoint(i) ← no change
crossPoint(i+1) ← no change
End if
End if
End

3) Parallel Implementation of Mutation

seudocode 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. In this experiment, the mutation factor is kept relative to the number of iteration from 0.01 to 1. Scheduling a block with a sufficient number thread is used to mutate the whole population. 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. If there is not any further improvement, it means the convergence of the swarm.)

Algorithm 3: Pseudocode Mutation

Global call of kernel for mutation function
Number of thread 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 and 1
For i =0 to n
If r < m_fact **then**
new_Pop = 1- new_Pop
Else
new_Pop = new_Pop
End if
End

3. GA Implementation using CUDA C

1) General Purpose GPU

The structure of General Purpose Computing on Graphics Processing Unit (GPGPU) is characterized by high level of parallel, data processing unit with multiple number of streaming processors. The program based on GPGPU can be easily developed using CUDA architecture. The execution of CUDA program is composed 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 is managed by kernel. The threads in GPU architecture can be grouped into blocks and grids . In GPU grid is (made up of)with group of thread blocks, and a thread block comprises(is made up of) 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.

When threads are executed in a group of 32 streaming multiprocessors,wrap is called by the Single Instruction Multiple Thread (SIMT) scheme, i.e., in nVIDIA GeForce GT 740M has 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 are limited up to 8 blocks.

2) Implementing GA using CUDA C

Implementation of GA includes parallel flow of algorithm to find global optimal solution using CUDA C. The major implementation of algorithm consists of 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 copying child population back to parent population. The overview of GA execution is depicted in Fig 1. The implementation of CUDA C kernels on GA is based on the following principle:

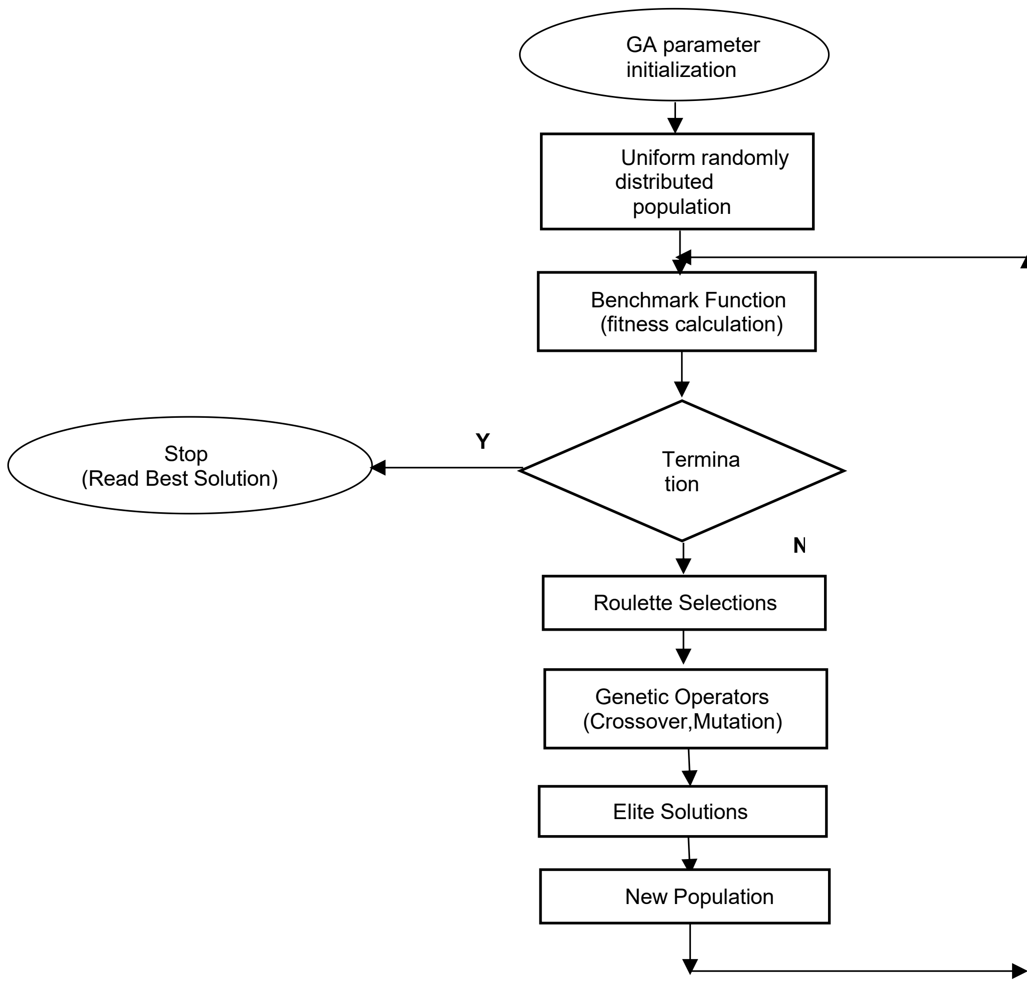

Fig 1. Implementation GA

All GA solution is calculated using thread block at each generation.

The maximum size of GA population during each generation is limited to the total number of threads to be currently $(2^{16}-1)^2$. For every trial solution, threads are used to compute possible results. GPU's computation capability is 512 threads per block for 1×1024. Hence, it is directly proportional to the hardware capability.

GPU accesses all the trial solution in one step i.e., with each kernel call CUDA C launches number of threads per block equivalent to the population size of the generation. CUDA C kernels generates population in one step and then computes their individual fitness values. The genetic operator is applied to each solution, where number of thread kept (to be) equal to its population size. The same stages are repeated to generate the following population and find a

solution.)Hence, due to these benefits of GP, it takes less time as compared to its sequential execution on CPU.

4. Performance evalution

1) Setting up experimental condition

The experiments are made on PC and NVIDIA card for performance evaluations. (Refer Table 1) The total number of streaming processors and streaming multiprocessorsare 16, hence, (there are) 256 streaming processors in each PC. Entire GA code of CUDA C is written in Visual Studio C++ 2013 release mode and complied on nvcc compiler. The result of (the)above experiment is evaluated using two different iteration size of 10,000 and 100,000. The dimension size of experiment is fixed. The larger the dimension size of its area and iteration, the more effective the acceleration of GA on GPU is.

TABLE 1 Computational Systems Specification

CPU	Processor	Intel Core(TM)i3 3320(U)+3.3GHz
	catch	3072KB
	Memory	2GB
GPU	Graphic Card	NVIDIA Geforce GT 730
	Version	9.18.13.2057
	CUDA Version	8.0

2) Benchmark Functions

Benchmark Test Functions for our experiment with distinct minima (f_{min}) is given in Table 2, which are numbered from $f_1(x)$ up to $f_7(x)$.

TABLE 2 Benchmark Functions

Benchmark 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
$f_3(x) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos(\frac{x}{\sqrt{i}})-1$	± 2048	0
$f_4(x) = \sum_{i=1}^{n} \lvert x_i^5 - 3x_i^4 + 4x_i^3 + 2x_i^2 - 10x_i - 4 \rvert$	± 10	0
$f_5(x) = \sum_{i=1}^{n} \lvert x_i \sin x_i + 0.1x_i \rvert$	± 10	0
$f_6(x) = -\exp(-0.5\sum_{i=1}^{n} x_i^2)$	± 1	0
$f_7(x) = \sum_{i=0}^{n-1} [100(x_i - x_{i+1}^2) + (1 - x_i)^2]$	± 2048	0

3) Experiment Result

As serial implementing time using CPU is compared with parallel implementing time using GPU and CUDA C, there is a remarkable difference under the same condition.

(1) Case study 1

In this experiment, the dimension size of the population generation is kept first 32 and then 64. Each dimension size is iterated for maximum number of iteration, which was set as 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 3.The best computational performance achieved for dimension size of 32, is 2.17 times for $f_4(x)$ on GPU, while on GPU with the dimension size of 64, $f_4(x)$ shows a speedup of 3.68 times higher than its CPU execution time.

TABLE 3 CUDA C Vs. C Performances for 10,000 iterations

N	Function	CPU	GPU	
		Time(sec)	Time(sec)	Speedup
32	$f_1(x)$	4.24	3.58	1.18
	$f_2(x)$	7.60	3.50	2.17
	$f_3(x)$	5.31	3.91	1.35
	$f_4(x)$	7.20	3.97	1.81
	$f_5(x)$	5.86	3.75	1.56
	$f_6(x)$	5.43	3.62	1.50
	$f_7(x)$	5.48	3.60	1.52
64	$f_1(x)$	8.42	4.28	1.96
	$f_2(x)$	15.84	4.30	3.65
	$f_3(x)$	10.30	4.78	1.55
	$f_4(x)$	17.47	4.75	3.68
	$f_5(x)$	11.43	4.68	2.44
	$f_6(x)$	12.49	4.33	2.88
	$f_7(x)$	8.58	4.50	1.91

(2) Case study 2

In this experiment the dimension size is the same as Case Study 1, but the iterations size increases to 100,000. The highest speedup achieved in this case for dimension size 32 is 2.18 for test function $f_7(x)$. On the other hand, dimension size 64 has best speedup of 3.47 times for $f_7(x)$.

The GPU average execution time is 3370 seconds while 12752 seconds in CPU.

TABLE 4 CUDA C Vs. C Performances for 100,000 iterations

N	Function	CPU	GPU	
		Time(sec)	Time(sec)	Speedup
32	$f_1(x)$	40.61	26.92	1.51
	$f_2(x)$	44.81	27.55	1.63
	$f_3(x)$	50.06	31.90	1.57
	$f_4(x)$	61.31	32.22	1.90
	$f_5(x)$	45.24	31.14	1.45
	$f_6(x)$	52.16	28.61	1.82
	$f_7(x)$	62.69	28.76	2.18
64	$f_1(x)$	81.86	34.84	2.35
	$f_2(x)$	89.77	35.42	2.53
	$f_3(x)$	101.13	40.01	3.03
	$f_4(x)$	122.97	40.53	2.36
	$f_5(x)$	93.61	39.63	2.51

	$f_6(x)$	92.36	36.83	2.88
	$f_7(x)$	127.52	36.71	3.47

5. Conclusion

In this paper, the implementation of GA on GPGPU using CUDA C is carried out.

It shows acceleration of 1.18-3.68 times as compared to its sequential execution on CPU on various benchmark test functions. From this result it is concluded that the algorithm can be made 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.

Acknowledgment

We thank the following organizations for offering the wonderful basic data and software:

- Applied Mathematics and Computation Press Publication for providing basic project and software
- Computers and Electrical Engineering Press Publication for providing a large number of useful data and experiments.

References

[1] Esraa Shehab b, Alsayed Algergawy a,b, Amany Sarhan b "Accelerating relational database operations using both CPU and GPU co-processor" Computers and Electrical Engineering 57 (2019) 69–80

[2] Vincent Roberge, Mohammed Tarbouchi ,Gilles Labonté, " Fast Genetic Algorithm Path Planner for Fixed-Wing Military UAV Using GPU" 10.1109/TAES.2019.2807558, IEEE Transactions on Aerospace and Electronic Systems

[3] Abhijit Ghosh, Chittaranjan Mishra, " Highly efficient parallel algorithms for solving the Bates PIDE for pricing options on a GPU" Applied Mathematics and Computation 409 (2021) 126411

[4] Simon Zhang, Mengbai Xiao, and Hao Wang. GPU-Accelerated Computation of Vietoris-Rips Persistence Barcodes. In 36th International Symposium on Computational Geometry (SoCG 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[5] Sun, X.; Wu, C.-C.; Liu, Y.-F, "The Design and Implementation of an Improved Lightweight BLASTP on CUDA GPU" , Symmetry 2021, 13, 2385. sym13122385

[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, pp.52–63, 2015.

[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, pp. 3161–3171, 2016.

[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," in Pattern Recognition (ICPR),2010 20th International Conference on. IEEE, 2017, 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 on, vol. 40, no. 3, pp. 641–650, 2015.

[10] Y. Zhang and L. Wu, "Artificial bee colony for two dimensional protein folding," Advances in Electrical Engineering Systems, vol. 1, no. 1, pp.19–23, 2016.