M.Tech. Project Report

ACCELERATING GENETIC ALGORITHM USING GPGPU AND CUDA

Submitted in partial fulfillment of the requirements for the degree of Master of Technology in Electronics & Communication Engineering

by

Rashmi Sharan Sinha (1269844)

Under the Supervision of

Dr. Satvir Singh

PUNJAB TECHNICAL UNIVERSITY

Jalandhar-Kapurthala Highway, Jalandhar

SHAHEED BHAGAT SINGH
STATE TECHNICAL CAMPUS
Moga Road (NH-95), Ferozepur-152004 (PB) INDIA

August 2013

CERTIFICATE

I, **Rashmi Sharan Sinha (1269844)**, hereby declare that the work being presented in this thesis on ACCELERATING GENETIC ALGORITHM USING GPGPU AND CUDA is an authentic record of my own work carried out by me during my course under the supervision of Dr. Satvir Singh. This is submitted to the Department of ECE at Shaheed Bhagat Singh State Technical Campus, Ferozepur (affiliated to Punjab Technical University, Jalandhar) as partial fulfillment of requirements for award of the degree of Master of Technology in Electronics & Communication Engineering.

Rashmi Sharan Sinha (1269844)

To the best of my knowledge, this thesis has not been submitted to Punjab Technical University, Jalandhar or to any other university or institute for award of any other degree or diploma. It is further understood that by this certificate, the undersigned do/does not endorse or approve any statement made, opinion expressed or conclusion drawn therein, however, approve the thesis only for the purpose for which it is submitted.

Dr. Satvir Singh [Supervisor]

The M.Tech Viva-Voce Examination of Rashmi Sharan Sinha (1269844) is held at Department of ECE, SBS State Technical Campus, Ferozepur on .....

Dr. Savina Bansal
Professor ECE, GZS PTU Campus Bathinda
(External Examinaer)

Dr. Sanjeev Dewra
(M.Tech. Coordinator, ECE)

I don't believe in taking right decisions. I take decision and then make them right.

- Ratan Tata

Dedicated to

My Family & Guide

ACKNOWLEDGEMENTS

Apart from my efforts, I could reach at this end of this project only because of the help, support, and guidance of many others. I take this opportunity to express my gratitude to those people. I would like to express the deepest gratitude to my supervisor, **Dr. Satvir Singh**, Professor, Department of Electronics & Communication Engineering , SBS State Technical Campus, Ferozepur (Punjab), India. He continually and convincingly conveyed a spirit of adventure in regard to research and scholarship, and an excitement in regard to teaching. Without his able guidance and persistent help this dissertation would not have been possible. I cant say thank you is enough for his tremendous support and help. I am becoming increasingly present to the fact that research can indeed be an enjoyable and rewarding experience, despite the tedium and hardwork involved. I am lucky to have M.Tech. thesis guide like him.

There are several other persons who made important contributions during this period. First and the foremost, I want to express my heartfelt gratitude to **Mr. Sarabjit Singh**, Assistant Professor, Computer Science and Engineering Department, SBS State Technical Campus, Ferozepur (Punjab) for helping me in programming. My solemn thanks to my teachers Dr. Vishal Sharma who used to guide us with their inspiring lectures of research methodology.

My sincere thanks to **Dr. T. S. Sidhu**, Director, SBS State Technical Campus, Ferozepur (Punjab) and to **Dr. Sanjeev Dewra**, Head, ECE Department, SBS State Technical Campus, Ferozepur (Punjab) for creating marvelous facilities and research environment in the institute.

I express my special thanks to **Mr. Gurtej Singh** for sparing hours of discussions on various research topics.

I wish to acknowledge the magnificent support I have received from my friend **Ms. Alisa Luna**, **Ms. Shivani Sharma**, **Ms. Garima Arora** and my fellow juniors **Ms. Jaspreet Kaur**, **Ms. Shivani Kakkar** in the form of useful discussions throughout this work.

I cannot forget to thank my sisters **Surbhi**, **Shalini**, **Priyanka** and **Aishwarya** for there marvelous support in the form of encouragement and love.

Most profound regards to my mother **Mrs. Poonam Sinha** and my father **Mr. Shanker Sharan Sinha**, who confined their needs all for my sake. It is their sacrifices and unconditional blessings that kept me motivated and committed, until I reached this end.

Finally, I must thank GOD for giving me the environment to study, people to help, opportunities to encash and potential to succeed.

Place: SBS STC Ferozepur
Date: March 16, 2015

Rashmi Sharan Sinha
(1269844)

LIST OF THESIS OUTCOMES

International Conferences Publications

1. R. S. Sinha and S. Singh, "Optimization Techniques on GPU", in ***IMTCS**, International Multi Track Conference on Science, Engineering & Technical Innovations,* Jalandhar, India, June-2014, pp 566-568. [Online] http://www.drsatvir.in.

2. S. Satvir and J. Kaur and R. S. Sinha, "A Comprehensive Survey on Various Evolutionary Algorithms on GPU", in ***ICCCS**, International Conference on Communication, Computing & System* Firozpur, India, 8-9 August, 2014, pp 83-88. [Online] http://www.hgpu.org.

3. R. S. Sinha, S. Singh, Sarabjeet Singh and V. K. Banga "Speedup Genetic Algorithm Using GPGPU", in *Proceedings of, **IEEE** International Conference On Communication Systems and Network Technologies,* Gwalior, India, 4-6 April, 2015. [Online] http://ieeexplore.ieee.org.

ABSTRACT

Genetic Algorithm (GA) is a family of computational models. It comes under the class of Evolutionary Algorithms (EA) and widely classified as a part of Artificial Intelligence (AI). It is swarm based stochastic search algorithm that simulates natural phenomenon of genetic evolution for searching solution to arbitrary engineering problems.

GA is algorithm involves a swarm of solutions represented by a string of parameters (genes), analogous to the chromosomes in Genetics. Initial swarm is generated randomly where each gene value is chosen randomly from the universe of discourse of respective parameter. Evolution is based on the fitness of parent solutions that are selected randomly and reproduce next generation of solutions, stochastically. Each child chromosome has features of both the parents as a resultant of crossover. Limited but random alteration in gene values of new generation represents effect of mutation. This process of solution is iterative and run until best new solution is not sufficiently good or maximum number of iterations has reached.

At the end, GA provides a number of solutions, however, best solution among them is one with maximum fitness. All three primary operators involved in GA viz. (1) Selection, (2) Crossover, and (3) Mutation, have data independency and hence, run can parallel.

Roulette Wheel Selection strategy is one of the most popular strategy used to search potential parent chromosomes based on fitness levels of individuals from the randomly generated initial population. This selection operator is expected to produce solutions with higher fitness in succeeding generations. On contrarily, Roulette Wheel Selection operator anticipated to produce relative probability of being selected according to their fitness in the swarm. This leads the GA to find global best solution rather than converging to its nearest local best solution. Selected parents are paired and undergo crossover operation for producing two

children solutions from them. It is significant operators which mimic biological crossover and reproduction mechanism. Mutation is GA operator which is responsible for maintaining genetic diversity in a generation. Mutation operator creates a new swarm of solutions where mutation factor controls the amplification of the difference between two generations so as to avoid search stagnation. The new generation is evaluate for the fitness level and then undergo Selection, Crossover and Mutation operators to produce another new generation with improved fitness levels and this process continues iteratively.

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 are data independent and can be performed in parallel. At the same time, with the recent advancement in programing techniques GPUs are possible to be used for general purpose computations, therefore, researchers have been working on parallel implementation of similar EAs and are presenting very encouraging results in their recent publications.

Compute Unified Device Architecture (CUDA) is a programming model that an interface for programmers for using general C language library, and it makes the General-Purpose computations possible on GPU cores in parallel. Moreover GPUs have low cost and higher computation power, that, attracted researchers and developers to harness GPUs for various nongraphic, applications in recent years.

Oiso and Matumura have evaluated Steady State GA with population size of 256. Performance of the algorithms was demonstrated on a set of benchmarking test functions, and they achieved a speedup rate of 6x compared to serial CPU implementation. Jiri and Jaros proposed an implementation of a Genetic Algorithm for Solving the Knapsack Problem with a multiple GPUs for population sizes of 128 to 2048. They achieved a speedup rate of 35-781 compared to serial CPU implementation. Arora, Tulshyan and Deb used the GPU to implement a binary and real encoded parallel GA and discussed the different data structures mapped to GPU textures, and archived a speedup of 40-400 for a population size of 128-16384 as compared sequential execution on CPU.

In this thesis, an implementation of GA on GPU using C-CUDA is reported. The massive parallel architecture of GPU is exploited to attain maximum speedups in evolving solution to arbitrary problems. In particular, it is observed that our implementation is more effective as it enables the execution of more threads than the work reported by others. The genetic

operator is applied to each solution in parallel and number of threads is kept equal to the population size. The new succeeding generations follow the same strategy as the GA evolution progresses. Hence, due to these benefits of GPU and improved thread planning takes less time as compared to its sequential execution time on CPU. This can suppress the frequency of data transfer between the host (CPU) and the device (GPU), which is probably the bottleneck to speed up by GPU.

Place: Ferozepur — Rashmi Sharan Sinha

Date: March 16, 2015 — (1269844)

ABBREVIATIONS

Abbreviations	Description
ACO	Ant Colony Optimization
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
arGA	adaptive resolution Genetic Algorithm
arLS	adaptive resolution Local Search
BBO	Biogeographical Based Optimization
BLX	Blended Crossover
Cg	C language Graphics
cGA	Cellular Genetic Algorithm
CPU	Central Processing Unit
CUDA	Compute Unified Design Architecture
DRAM	Dynamic Random Access Memory
EA	Evolutionary Algorithm
EC	Evolutionary Computation
ECC	Error Correcting Code
EP	Evolutionary Programming
ES	Evolutionary Strategies

Abbreviations	Description
FLOPS	FLoating Point Operations Per Second
FORTRAN	Formula Translation
GDE	Generalized Differential Evolution
GA	Genetic Algorithm
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
LLVM	Lower Level Virtual Machine
MINLP	Mixed Integer Nonlinear Programming
MMDP	Massively Multimodel Deceptive Problem
MOEAs	Multiobjective Evolutionary Algorithms
NLP	Non-linear Programming
OpenCL	Open Computing Language
Open ACC	Open Accelerator
PSO	Particle Swarm Optimization
SM	Streaming Multiprocessor
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
TFLOPS	Tera FLoating Point Operations Per Second

NOTATIONS

Symbols	Description
f_i	Fitness Value
P_i	Probability of selection
r	Uniform Random Number
p_c	Probability of Crossover
p_m	Probability of Mutation
f_{min}	Minima Benchmark Test Function
m_fact	Mutation Factor
new_Pop	New Population
pop_size	Population Size
CSum	Cumulative Sum
mixRatio	Mixing Ratio
chromoLength	Chromosome Length of String

Coding Notations	**Description**
L1	Level-1 Cache memory
host	Execution of programme at CPU
device	Execution of programme at GPU
blockDim	Block Dimension
gridDim	Grid Dimension
Malloc	Memory Allocation
Memcpy	Memory Copy
cudaMalloc	CUDA Memory Allocation
cudaMemcpy	CUDA Memory Copy

LIST OF FIGURES

LIST OF TABLES

CONTENTS

CHAPTER 1

INTRODUCTION

Genetic Algorithms (GAs) are population based stochastic search algorithms that are inspired by naturally established evolutionary systems. GAs are presenting very encouraging results in engineering and non-engineering applications as reported by various researchers. However, these algorithm involves intensive data independent calculation that have potential of making them parallel. This thesis presents a work on implementation of GAs in parallel on Graphic Processing Unit (GPU) using Compute Unified Design Architecture (CUDA) and especially, this chapter gives an overview of the this a whole.

1.1 Introduction

GA introduced in **[Holland, 1975]** is powerful, domain-independent search techniques inspired by Darwinian Theory of *Survival of Fittest.* In GAs, the initial population of solutions is generated randomly using a uniform distribution. Each individual solution in the swarm is evaluated using a fitness function. An algorithm termination criteria is defined to determine whether the best solution in the swarm is present or not. The algorithm ends if acceptable solution has been found or the computational resources have been spent. Otherwise, the individuals in the population are manipulated by three natural operator of evolution, viz. (1) Selection, (2) Crossover and (3) Mutation. Individuals from the previous population are called parents while those created by applying the evolutionary operators are called children or offsprings. These process steps are iterated to generate improved as selection process involves probabilistically better solutions in present pool.

Although GAs are very effective in solving many practical problems, their execution time is a limiting factor for a host of problems, a candidate solutions must be evaluated. Fortunately, the most time-consuming fitness evaluations can be performed independently for each individual in the population using various types of parallelization.

Nowadays modern Graphic Processing Units (GPU), although originally designed for real-time 3D rendering can be seen as very fast highly parallel general purpose systems [**Pospíchal et al., 2010**], [**Kirk and Wen-mei, 2012**] and hence, employed with advantage to accelerate GAs.

1.1.1 General Purpose Computations on GPU

GPU is evolving into very powerful and flexible processors, while their price remained in the range of consumer market. GPUs are in fact very powerful massively parallel computers that have (among others) one main drawback: all the elementary processors on the card are organized into larger multi-processors. They have to execute the same instruction at the same time but on different data (SIMD model, for Single Instruction Multiple Data). They now offer floating-point calculation much faster than todays CPU and, beyond graphics applications; they are very well suited to address general problems that can be expressed as data-parallel computations (i.e. the same code is executed on many different data elements). Consequently, several general purpose programming modals for GPUs have become available, e.g. Compute Unified Design Architecture (CUDA) [**Pospíchal et al., 2010**] and Open Computing Language (OpenCL) [**Nickolls and Dally, 2010**] and thus developers do not need any more to master the extra complexity of graphics programming APIs (Application Program Interfaces) when they design non graphics applications [**Van Dam and Feiner, 2014**].

GAs need to run an identical evaluation function on different individuals (that can be considered as different data), meaning that this is exactly what GPUs have been designed to deal with. The most basic idea that comes to mind when one wants to parallelize an evolutionary algorithm is to run the evolution engine in a sequential way on some kind of master CPU (potentially the host computer CPU), and when a new generation of offsprings have been created, get them all to evaluate rapidly on a massively parallel computer. This approach has been examined in [**Pospíchal et al., 2010**]. The proposed evolutionary algorithm reaches the speedup about 100. But, the bottleneck can be seen in slow data transfers from host memory to GPU and back, especially for small transactions [**Farber, 2011**].

Another way, how to parallelize GA is to move the whole algorithm on GPU. However, very few researchers so far have gone this way. They usually used Cg language [**Brodtkorb et al., 2013**] which does not allow access to some GPU features (i.e. manual thread and

block control). A parallel genetic algorithm targeted to numerical optimization has been published in [**Belegundu and Chandrupatla, 2011**]. Unfortunately, this implementation reached only small speedups between 1.16 and 5.30 depending on population size. Several interesting publication can be also found in [**Pospíchal et al., 2010**].

1.2 Genetic algorithm on GPU

nVidia's CUDA (Compute Unified Device Architecture) [**Farber, 2011**] programming modal is most commonly used for GPGPU and use here as well to implement GA in parallel on GPU. This toolkit promises best achieved speedups on GPU so far and vast community of developers. CUDA can be performed most of nVidia graphics card on both Linux and Windows platforms. Natural parallelism of computation on GPU is expressed by a few compiler directives added on to the well known C programming language.

As mentioned earlier, nVidia GPUs consist of multiprocessors capable of performing same tasks in parallel on multiple core with different data vactors. Threads running in parallel on these cores are very lightweight and can be synchronized using barriers so that data consistency is maintained. This can be done with very low impact on the performance in a multiprocessor, however, not between multiprocessors. This limitation forces us to evolve islands either completely independent or perform migrations between them asynchronously. The memory attached to GPU cards is divided into two levels; (i) Main Memory and (ii) On-chip memory. Main memory has a big capacity (hundreds of MB) and holds a complete set of data as well as user programs. It also acts as an entry/output point during communication with CPU. Unfortunately, big capacity is outweighed with high latency. On the other hand, on-chip memory is very fast, however, has very limited size. Apart from per-thread local registers, onchip memory contains particularly useful per-multiprocessor shared segments.

This 16KB array acts as a user managed L1 cache. The size of on-chip memory is a strongly limiting factor for designing efficient GA, however, existing CUDA applications greatly benefit here. This is why, our primary concern during designing GA accelerated by GPU is to create its efficient mapping to CUDA software model with a special focus on massive parallelism, usage of shared memory within multiprocessors and avoiding the system bus bottleneck. This approach will help in accelerating GA performance otherwise may lead to deteriorate.

The proposed algorithm begins with the input population initialization on the CPU side. Then, chromosomes and GA strategy parameters are transferred to the GPU (device) main memory using system bus. Next, the CUDA kernel perform genetic operations on GPU is launched. Depending on kernel parameters, the input population is distributed to several

blocks (islands) of threads (individuals). All threads on each island read their chromosomes from the main memory to the fast shared (on-chip) memory within a multiprocessor.

memories maintain local island populations. The process of evolution then proceeds for a certain number of generations in isolation, whereas, the islands as well as individuals are evolved on the graphics card in parallel. Each generation consists of fitness function evaluation and application of the selection, crossover and mutation. The operators are separated by CUDA block barriers with zero overhead so that data consistency is ensured [**Pospíchal et al., 2010**].

The algorithm iterates until a terminating condition is met (currently the maximum number of generations is set). Finally, every thread writes its evolved chromosome back to the main memory from where it will be read by CPU through the system bus.

1.3 Problem in Brief

GA like other Evolutionary Algorithms BBO and PSO has already shown its ability to solve optimization problems upon sequential CPU Processors. This kind of algorithm is capable of maintaining a high diversity in the population until reaching the region containing the global optimum. Furthermore, it may benefit from parallelism as a way of speeding up its operations when the instance of the problem is complex. Hence, in order to improve this advantage relative to other heuristic algorithms,it is necessary to improve this algorithm by implementing upon GPU CUDA.

1.4 Objectives

The primary objectives of this research work are summarized as follows:

1. To implement Genetic Algorithm (GAs) on CPU and to do various experiments with genetic operators.
2. To Parallelize Genetic Algorithm using CUDA to get higher speedup results.
3. To compare the results of this parallelized GAs with that of sequential GA using various Benchmark test functions.

1.5 Methodology

The methodology followed in this research project is discussed in brief as follows:

1. One of the foremost requirement is to conduct deep study about the algorithm and understand various key features and strategy parameters.
2. Second difficult requirement is to design programs for genetic algorithms and to implement it on nVIDIA CUDA and get the output.
3. Compute Unified Device Architecture (CUDA), a powerful parallel programming model for issuing and managing computations on the GPU without mapping them to a graphics API.

1.6 Contributions

The main contributions of this report are:

1. To study GA for design issues.
2. To create code on C-CUDA for GA.
3. To explore various benchmark test functions and achieving required fitness of swarm.

1.7 Thesis Outline

After the brief introduction to M.Tech project report given in Chapter 1, Chapter 2 starts with the literature survey giving an overview of GPGPU advancement in the field of evolutionary algorithms viz BBO,PSO and GA . It also presents a gentle introduction to nVIDIA CUDA and developments in the domain of GPGPU.

Chapter 3 is devoted to study of Genetics, literature of GA including flow and its variants reported till date. Implementation Platform of Optimization. Chapter 4 is dedicated to study of implementation platform and its parameter.

Chapter 5, Firstly, GA operators in parallel along with architectural details of CUDA is discussed. Secondly, implementation flow of GA on C-CUDA is discussed, and a brief introduction of CUDA environment is also presented in this chapter.

Chapter 6 represents simulation results of convergence performance of GA on various testbed benchmark functions for optimization. Best results in tabulated and graphical form are also represented in this chapter. Lastly, conclusion and future scopes of this research are discussed in Chapter 7.

CHAPTER 2

LITERATURE SURVEY

This chapter presents investigational study of Genetic Algorithms, General purpose Computing on GPU and C-CUDA. This Chapter contains the overview of GA and its variants on testbed of various benchmark functions.

2.1 Introduction

Genetic algorithms (GAs) are search methods based on principles of natural selection and genetics [**Holland, 1975**]. We start with a brief introduction to simple genetic algorithms and associated terminology.

GAs encode the decision variables of a search problem into finite-length strings of alphabets of certain cardinality. The strings which are candidate solutions to the search problem are referred to as chromosomes, the alphabets are referred to as genes and the values of genes are called alleles. For example, in a problem such as the traveling salesman problem, a chromosome represents a route, and a gene may represent a city. In contrast to traditional optimization techniques, GAs work with coding of parameters, rather than the parameters themselves.

To evolve good solutions and to implement natural selection, we need a measure for distinguishing good solutions from bad solutions. The measure could be an objective function that is a mathematical model or a computer simulation, or it can be a subjective function where humans choose better solutions over worse ones. In essence, the fitness measure must

determine a candidate solutions relative fitness, which will subsequently be used by the GA to guide the evolution of good solutions.

Another important concept of GAs is the notion of population. Unlike traditional search methods, genetic algorithms rely on a population of candidate solutions. The population size, which is usually a user-specified parameter, is one of the important factors affecting the scalability and performance of genetic algorithms. For example, small population sizes might lead to premature convergence and yield substandard solutions. On the other hand, large population sizes lead to unnecessary expenditure of valuable computational time.

Once the problem is encoded in a chromosomal manner and a fitness measure for discriminating good solutions from bad ones has been chosen, we can start to evolve solutions to the search problem using the following steps:

1. *Initialization* The initial population of candidate solutions is usually generated randomly across the search space. However, domain-specific knowledge or other information can be easily incorporated.

2. *Evaluation* Once the population is initialized or an offspring population is created, the fitness values of the candidate solutions are evaluated using different Benchmark Test Functions.

3. *Selection* Selection allocates more copies of those solutions with higher fitness values and thus imposes the survival-of-the-fittest mechanism on the candidate solutions. The main idea of selection is to prefer better solutions to worse ones, and many selection procedures have been proposed to accomplish this idea, including roulette-wheel selection, stochastic universal selection, ranking selection and tournament selection, some of which are described in the next section.

4. *Crossover* Recombination combines parts of two or more parental solutions to create new, possibly better solutions (i.e. offspring). There are many ways of accomplishing this (some of which are discussed in the next section), and competent performance depends on a properly designed recombination mechanism. The offspring under recombination will not be identical to any particular parent and will instead combine parental traits in a novel manner **[Goldberg, 2002]**.

5. *Mutation*While recombination operates on two or more parental chromosomes, mutation locally but randomly modifies a solution. Again, there are many variations of mutation, but it usually involves one or more changes being made to an individuals trait or traits. In other words, mutation performs a random walk in the vicinity of a candidate solution.

6. *Replacement* The offspring population created by selection, recombination, and mutation replaces the original parental population. Many replacement techniques such as elitist replacement, generation-wise replacement and steady-state replacement methods are used in GAs.

7. Repeat steps 2 to 6 until a terminating condition is met.

2.2 Variants of GA

GA is an efficient Algorithm in solving problems based upon swarm optimization. Over the last few decades, many different variants of GA has been introduced for solving optimization problem. The comprehensive study of their performance is shown below.

2.2.1 Real coded Genetic Algorithm (RCGA)

This section presents a theory of convergence for Real Coded Genetic Algorithms (RCGA) that use floating point or other high-cardinality codings in their chromosomes. The theory is consistent with the theory of schemata and postulates that selection dominates early GA performance and restricts subsequent search to intervals with above-average function value dimension by dimension. These intervals may be further subdivided on the basis of their attraction under genetic hillclimbing. Each of these subintervals is called a virtual character, and the collection of characters along a given dimension is called a virtual alphabet. It is the virtual alphabet that is searched during the recombinative phase of the GA, and in many problems this is sufficient to ensure that good solutions are found. Although the theory helps suggest why many problems have been solved using RCGA, it also suggests that real-coded GAs can be blocked from further progress in those situations when local optima separate the virtual characters from the global optimum [**Goldberg, 1989**].

2.2.2 Binary Coded Genetic Algorithm (BCGA)

The Binary Coded Genetic Algorithm (BCGA) is a probabilistic search algorithm that iteratively transforms a set (called a population) of mathematical objects (typically fixed-length binary character strings), each with an associated fitness value, into a new population of offspring objects using the Darwinian principle of natural selection and using operations that are patterned after naturally occurring genetic operations, such as crossover and mutation. Following the model of evolution, they establish a population of individuals, where each individual corresponds to a point in the search space. An objective function is applied to each individual to rate their fitness. Using well conceived operators, a next generation is formed

based upon the survival of the fittest. Therefore, the evolution of individuals from generation to generation tends to result in fitter in individuals, solutions, in the search space.

2.2.3 Cellular Genetic Algorithm (cGA)

Cellular Genetic Algorithms (cGAs) are a kind of evolutionary algorithm (EAs). cGAs are robust search algorithms inspired by the analogy of natural evolution from the point of view of the individual solution. They have demonstrated to be particularly effective optimization techniques solving many practical problems in science and engineering. The basic algorithm (cGA) shows high performance and because of its swarm intelligence structure (i.e. emergent behavior and decentralized control flow). cGA is able of keeping a high diversity in the population until reaching the region containing the global optimum. Furthermore, it may benefit from parallelism as a way of speeding up its operations when the instance of the problem is complex.

2.2.4 Mixed Integer Non-Linear Programming (MINLP)

Mixed Integer Non-Linear Programming (MINLP) problems are the most generalized form of single-objective global optimization problems. They contain both continuous and integer decision variables, and involve non-linear objective function and constraints setting no limit to the complexity of the problems. MINLPs deals with three major parameters. 1) They involve both discrete (integer) and continuous (floating point) variables. 2) Objective function & constraints are non-linear, generating potential non-convexities. 3) They can involve active equality and inequality constraints. Many real world constrained optimization problems are modeled as MINLPs e.g. heat and mass exchange networks, batch plant design and scheduling, design of interplanetary spacecraft trajectories etc.

2.3 GPGPU and GA

General Purpose Computing on Graphics Processing Units (GPGPUs) are widely used among developers and researchers as accelerators for applications outside the domain of traditional computer graphics. In particular, GPUs have become a viable parallel accelerator for scientific computing with low investment in the necessary hardware. In this section, various Variants of GA is described and their gained efficient speedup on GPU using CUDA.

2.3.0.1 Island Based Genetic Algorithm

Island based genetic algorithm (GA) is implemented on Multi-GPU in [**Jaros, 2012**], for solving the Knapsack problem. The main motives to speed up the GA by using a cluster of nVIDIA GPU and comparing the execution time of single GPU with a multicore CPU.

The population of proposed GA is organized in two one-dimensional arrays. First array representing the genotype and the other array represents the fitness value. The GA parameter such as population and chromosome size, the crossover and mutation rates, the statistic collection and migration interval, the total number of evaluated generations etc. are filled with command line parameters, this maintained structure is stored at GPU constant memory [**Singh et al., 2014**]. The basic concept to maximize GPU utilization is to control thread divergence and amalgamate all memory accesses using this algorithm Singh et al. [2014]. Firstly, a hash function generator based stateless random number is generated [**Salmon et al., 2011**]. Then, the genetic material is exchanged of two parents using crossover and mutation process by performing the binary tournament selection to create a new individual. As the new offspring is created fitness evaluation is carried out. Next, the parent is replaced by the offspring with the help of entire warp if the fitness value of latter is higher than the former. The good individuals are migrated from the adjacent lower index island to the higher index island which is arranged in the unidirectional ring topology. Lastly, all the statistical data from the local island and from the global gathering process are collected [**Sanders and Kandrot, 2010**].

The analysis illustrates that as the individual per GPU and number of islands increases the fitness value increases. Secondly, the execution time is invariant for island size up to 512 and then elevate linearly beyond 512. All in all, the implemented Island based GA leads to the GPU performance of 5.67 TFLOPS.

2.3.0.2 Advanced Genetic Algorithm

With the increasing advent of GPGPU using CUDA, the stochastic algorithm of advanced Genetic Algorithm is used to solve non-convex MINLP and non-convex Non-linear Programming (NLP) problems. MINLP refers to mathematical programming algorithms that can optimize both continuous and integer variables, in a context of nonlinearities in the objective function and/or constraints. MINLP problems involve the simultaneous optimization of discrete and continuous variables. These problems often arise where one is trying to simultaneously optimize the system structure and parameters. This is difficult because optimal topology is dependent upon parameter levels and vice versa [**Munawar et al., 2011**].

In many design optimization problems, the structural topology influences the optimal parameter settings so a simple de-coupling approach does not work: it is often not possible to isolate these and optimize each separately. Finally, the complexity of these problems depends upon the form of the objective function. In the past, solution techniques typically depended upon objective functions that were single-attribute and linear (i.e., minimize cost). However, real problems often require multi-attribute objectives such as minimizing costs while maximizing safety and/or reliability, ensuring feasibility, and meeting scheduled deadlines. In these cases, the goal is to optimize over a set of performance indices which may be combined in a nonlinear objective function. For efficient utilization of GPU parallel resources adaptive resolution genetic algorithm (arGA) is developed. Through this algorithm the intensity of each individual is beamed using entropy measures. The algorithm is tested for different benchmark problems having different levels of difficulty. Parallelization of arGA and the arLS (local search) operators is done to gain significant speedup. The results of the tests shows a speedup of 42x with single precision and 20x with double precision over nVIDIA Fermi C2050 GPU [**Munawar et al., 2011**].

2.3.0.3 Steady State Genetic Algorithm

Steady-State GA is implemented on GPU using CUDA in [**Oiso et al., 2011**], where population individual data is accessed parallel to effectively speedup the process. The optimization problem is effectively solved by the means of Evolutionary Computing [**Stentiford, 2001**]. The steady state Genetic Algorithm is used to access optimization algorithms. These algorithms basically have selection for the reproduction and selection of survival implementation with concurrent kernel execution [**Arora et al., 2010**].

The implementation of Steady-State GA is done as follows: Firstly, from the population two individual (parents) are received by Streaming Multiprocessors (SM). Then, the kernel generates random number as GAs are stochastic search processes. BLX- is adopted as blend crossover in the crossover process. The crossover operation is executed with two parents in SM, and the two offspring yielded are stored in the shared memory. Next, uniform mutation, fitness based sorting process and selection process is executed. This whole process is repeated until the loop terminates. Four test functions of the optimization problem Hyper sphere, Rosen rock, Ackley, and Griewank were used for comparing GPU and CPU computation on implementing Steady-State GA. The study is first performed upon CPU then with nVIDIA GeForce GTX480GPU gives a speedup of 3x to 6x then the previous implementation on CPU [**Vidal and Alba, 2010**]. Moreover, the speed up ratio for Generational GA is much better than Steady-State GA on GPU since computational granularity is very small in the latter. So a large amount of execution time is occupied by the latency caused by the kernel calls. However, in terms of function values Steady-State GA are more efficient.

2.3.0.4 Cellular Genetic Algorithm

Genetic Algorithm have a subclass known as Cellular Genetic Algorithm (cGA) which provides the data of population structured in several specified topologies [**Vidal and Alba, 2010**]. Cellular Genetic Algorithm (cGA) is implemented for multi-GPU to accelerate the execution process so that the system could be more efficient. cGA is used because of its high performance and swarm intelligence structure. Until the global optimum region is reached, cGA is able of keeping a high diversity in population.

To manage the multi-GPU utilization each CPU thread is held responsible for one GPU device which is known as multi-threaded mode. Firstly, each CPU thread is associated to one GPU. This can be done if common structure (toroidal grid) is designed for all CPU threads. Then, the population is divided into subpopulation which is stored in the global memory of each GPU. Each GPU works individually irrespective of other GPU and the process is same as performed with single GPU implementation. To ensure that every GPU had finished its work a synchronization barrier is used and lastly, data is collected and transferred to other GPUs. The process executes until the while loop in pseudo code of cGA terminates.

Three discrete optimization problems: Colville Minimization, Error Correcting Codes Design Problem (ECC) and Massively Multimodal Deceptive Problem (MMDP), and three continuous ones, Shifted Griewank function, Shifted Restringing function and Shifted Rosenbrock function [**Suganthan et al., 2005**] [**Jamil and Yang, 2013**] were selected for comparing the algorithm in terms of efficiency and efficacy. Statistical tests [**Vidal and Alba, 2010**] are performed for each problem to ensure that the results are statistically significant. A common parameter, population size is used to make a meaningful comparison among all the algorithms.

The analysis shows the average speed up with respect to CPU version ranges from 8 to 771 and for single GPU it is alike multi-GPU, with a little overhead in the latter case. The multi-GPU is more prominent in paralleling the algorithm and producing accurate results as there is a need of special maintenance to perform same experiment upon single GPU.

Genetic Algorithm is tested and evaluated on parallel implementation on C-CUDA API on the parameters like population size, number of threads, problem size and problem of differing complexities with variation in the population individuals [**Vidal and Alba, 2010**]. For an efficient implementation on GPGPU the solution is thoroughly implemented along with the operators like random number generation, initialization, selection operation, and mutation operations [**De Veronese and Krohling, 2010**]. The nVIDIA GeForce 8800GTX shows overall speedup of 40-400 on three different test problems [**Arora et al., 2010**]. Thus parallel implementation is more effective then sequential process as compared with clock time and accuracy.

2.4 GPGPU and Evolutionary Algorithms

GPGPU-based architecture, aiming at improving the performance of computationally demanding optimizations for identifiable specific mapping parameters, one can reduce total execution time drastically and also, improve greatly the optimization process convergence. In this section, different Evolutionary Algorithms (EAs) and their variants is shown with their specific speedups.

2.4.1 Particle Swarm Optimization (PSO)

PSO is a meta heuristic algorithm works by having a swarm of particles. These particles are moved around in the search-space according to a few simple formulae. The movements of the particles are guided by their own best known position in the search-space as well as the entire swarm best known position. When improved positions are being discovered these will then come to guide the movements of the swarm. PSO is one of the types of Evolutionary Algorithm used to optimize the multiple objective problems. When an optimization problem involves more than one objective function, the task of finding one or more optimal solutions is known as multiobjective optimization For implementing PSO code in C-CUDA the allocation of vector/matrix is done on the device.

Random numbers are generated using Mersenn Twister code and then based on objective functions are evaluated. After evaluation the global best particle of whole swarm is updated. Next, the sum and multiplication operations are performed on the vectors which describe the particle.

2.4.2 Particle Gradient Multi-objective Evolutionary Algorithm (PGMOEA)

The general Purpose GPU is efficiently used in optimizing the multiple objective problems. The particle gradient Multi-objective Evolutionary Algorithm (PGMOEA) is used to solve optimization problems. PGMOEA is first experimented on CPU and then after parallelizing the algorithm executed upon GPU which formed a great speedup results. The first step to implement PGMOEA is to read parameters such as population size, dimension size, maximum iterative generations, crossover rate, mutation rate and initialize particle texture array. Blank texture array i.e objective, rank value, entropy and free energy array are then generated to store different results. Next, the particle texture array is is loaded to GPU to calculate the rank of all the particles and the results are then stored in rank value texture array. The particles are sorted in the decreasing order of their ranks to make a mating pool. The higher order rank value particles are selected to perform crossover and mutation operation using

Guos algorithm. The new particles generated through this process are then replaced with the last particles which have lower rank in mating pool to get a new population. The program is terminated if the halt condition is satisfied else particle texture array is again loaded to GPU and the process is repeated again.

2.4.3 Central Force Optimization (CFO)

The metaheurestic algorithm CFO is implemented upon GPGPU using local neighborhood and implemented CFO concepts. The calculation of CFO independent upon the movement of probes which are scattered all over the space. The probes then slowly move towards the probe having highest mass or fitness. CFO is the most evaluated algorithm with the measures of initial position and acceleration vectors, fitness evaluation and probe movements. The test problems are having the dimension of 30 to 100 of four different examples of Pseudo random CFO (PR-CFO). The PR-CFO is tested with four test types i.e. Ring, Standard, CUDA, CUDARing. PR-CFO shows a speedup of 4 to 400 using CUDA.

2.4.4 Benchmark Test Functions

TABLE 2.1: Benchmark Functions with Different Local Minima

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

The test function benchmarks and their diverse properties such as modality and separability. A function with more than one local optimum is called multimodal. These functions are

used to test the ability of an algorithm to escape from any local minimum. If the exploration process of an algorithm is poorly designed, then it cannot search the function landscape effectively. This, in turn, leads to an algorithm getting stuck at a local minimum. Benchmark Test Functions for our experiment with distinct minima (f_{min}) is given in Table 2.1, which are numbered from $f_1(x)$ up to $f_7(x)$ and correspond to the functions in [Jamil and Yang, 2013].

Test functions are important to validate and compare optimization algorithms, especially newly developed algorithms. Here, we have attempted to provide the most comprehensive list of known benchmarks or test functions. It can be expected that all these functions should be used for testing new optimization algorithms so as to provide a more complete view about the performance of any algorithms of interest. Here, range is the lower and upper limits of the universal discourse for every function.

2.5 Conclusion

TABLE 2.2: Comparison Table Of Different Evolutionary Algorithms On GPU and CPU

Algorithm	**Experimental Set up**		**Time**		**Speed up**	
	GPU (nVIDIA)	***CPU***	***GPU***	***CPU***	***GPU***	***CPU***
Island based GA	GTX580	Intel Xeon Six-Core	5.67 TFLOPS	--	653.68	11.32
Advanced GA	C2050 (Double precision)	Intel Core 2 Duo	--	--	20x	--
	C2050 (Single precision)	Intel Core i7			40x	
Steady-state GA	Geforce GTX480	Intel Core i7	4.874 - 4.780s	14.46-28.56s	3.0x-6.0x	--
Cellular GA	GTX-285	Intel Quad processor	0.021-1.821s	0.266 - 1450.415s	8 - 771	--
Binary and Real coded GA	Tesla C1060	AMD Athlon 64 X2 Dual Core	RGA 0.003 – 22.534s	RGA 0.071-4851.69s	--	RGA 21.28 – 215.30 s

In this chapter we present different optimization algorithm with tremendous speedups in the computation time. The overall GPU performance of multi-GPU Island-based GA for solving Knapsack problem reaches 5.67 TFLOPS. MINLP archived an overall speedup of 20x to 42x using nVidia Tesla C2050 GPU as compared to Intel Core i7 920 CPU processor. On implementing Steady state GA on a GPU approximately 6 times faster results are obtained than the corresponding CPU implementation. The implementation of Cellular Genetic Algorithm for a multi-GPU platform leads to speedup range from 8 to 771 with respect to the CPU version. The new binary-coded and real-coded Genetic Algorithm using CUDA leads to a performance improvement with the speedup of 40x to 400x.

CHAPTER 3

GENETIC ALGORITHM

Genetic Algorithms are a family of computational models inspired by evolution. These algorithms encode a potential solution to a specific problem on a simple chromosome-like data structure and apply recombination operators to these structures as as to preserve critical information. Genetic algorithms are often viewed as function optimizer, although the range of problems to which genetic algorithms have been applied are quite broad.
An implementation of genetic algorithm begins with a population of (typically random) chromosomes. One then evaluates these structures and allocated reproductive opportunities in such a way that these chromosomes which represent a better solution to the target problem are given more chances to 'reproduce' than those chromosomes which are poorer solutions. The 'goodness' of a solution is typically defined with respect to the current population.

3.1 Introduction

Genetic Algorithms are a family of computational models inspired by evolution. These algorithms encode a potential solution to a specific problem on a simple chromosome-like data structure and apply recombination operators to these structures as as to preserve critical information. Genetic algorithms are often viewed as function optimizer, although the range of problems to which genetic algorithms have been applied are quite broad. The GA operators i.e., Population Generation, selection, crossover and mutation is widely discussed in coming sections.

3.1.1 Population Initialization

The major questions to consider are firstly the size of the population, and secondly the method by which the individuals are chosen. The size of the population has been approached from several theoretical points of view, although the underlying idea is always of a trade-off between efficiency and effectiveness. Intuitively, it would seem that there should be some optimal value for a given string length, on the grounds that too small a population would not allow sufficient room for exploring the search space effectively, while too large a population would so impair the efficiency of the method that no solution could be expected in a reasonable amount of time.

A slightly different question that we could ask is regarding a minimum population size for a meaningful search to take place. The initial principle was adopted that, at the very least, every point in the search space should be reachable from the initial population by crossover only. This requirement can only be satisfied if there is at least one instance of every allele at each locus in the whole population of strings. On the assumption that the initial population is generated by a random sample with replacement.

3.1.2 Basic GA Operators

In this section some of the selection, recombination, and mutation operators commonly used in genetic algorithms is described.

3.1.2.1 Selection Methods

There are three major types of selection schemes [**Noraini and Geraghty, 2011**] which are as follows:

1. ***Tournament Selection***
   In tournament selection, n individuals are selected randomly from the larger population, and the selected individuals compete against each other. The individual with the highest fitness wins and will be included as one of the next generation population. The number of individuals competing in each tournament is referred to as tournament size, commonly set to 2 (also called binary tournament). Tournament selection also gives a chance to all individuals to be selected and thus it preserves diversity, although keeping diversity may degrade the convergence speed.

   In tournament selection, larger values of tournament size lead to higher expected loss of diversity [**Blickle and Thiele, 1995**], [**Whitley et al., 1989**]. The larger tournament

size means that a smaller portion of the population actually contributes to genetic diversity, making the search increasingly greedy in nature. There might be two factors that lead to the loss of diversity in regular tournament selection; some individuals might not get sampled to participate in a tournament at all while other individuals might not be selected for the intermediate population because they lost a tournament.

2. ***Rank-based Roulette Wheel Selection***
   Rank-based roulette wheel selection is the selection strategy where the probability of a chromosome being selected is based on its fitness rank relative to the entire population. Rank-based selection schemes first sort individuals in the population according to their fitness and then computes selection probabilities according to their ranks rather than fitness values. Hence rank-based selection can maintain a constant pressure in the evolutionary search where it introduces a uniform scaling across the population and is not influenced by super-individuals or the spreading of fitness values at all as in proportional selection. Rank-based selection uses a function to map the indices of individuals in the sorted list to their selection probabilities.

3. ***Roulette Wheel Selection***
   In this selection scheme, individuals are selected with a probability that is directly proportional to their fitness values i.e. an individual.s selection corresponds to a portion of a roulette wheel. The probabilities of selecting a parent can be seen as spinning a roulette wheel with the size of the segment for each parent being proportional to its fitness. Obviously, those with the largest fitness (i.e. largest segment sizes) have more probability of being chosen. The fittest individual occupies the largest segment, whereas the least fit have correspondingly smaller segment within the roulette wheel. The circumference of the roulette wheel is the sum of all fitness values of the individuals. The roulette wheel selection mechanism is depicted in Fig. 3.1.

   In Fig.3.1, when the wheel is spun, the wheel will finally stop and the pointer attached to it will point on one of the segment, most probably on one of the widest ones. However, all segments have a chance, with a probability that is proportional to its width. By repeating this each time an individual needs to be chosen, the better individuals will be chosen more often than the poorer ones, thus fulfilling the requirements of survival of the fittest. Let $f_1, f_2, f_3, ...f_n$ be fitness values of chromosomes $1, 2, 3, \ldots n$. Then the probability of selection P_i for chromosomes i is defined as (5.1),

$$P_i = \frac{f_i}{\sum_{j=1}^{n} f_i} \tag{3.1}$$

   The basic advantage of proportional roulette wheel selection is that it discards none of the individuals in the population and gives a chance to all of them to be selected. Therefore, diversity in the population is preserved.

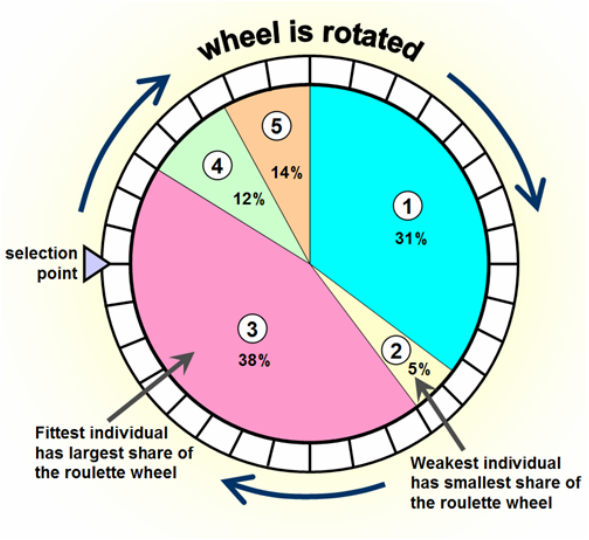

FIGURE 3.1: Roulette Wheel

3.1.2.2 Crossover Operator

After selection, individuals from the mating pool are recombined (or crossed over) to create new, hopefully better, offspring. In the GA literature, many crossover methods have been designed [**Agrawal et al., 1994**] and some of them are described in this section. Many of the recombination operators used in the literature are problem-specific and in this section we will introduce a few generic (problem independent) crossover operators. It should be noted that while for hard search problems, many of the following operators are not scalable, they are very useful as a first option. Recently, however, researchers have achieved significant success in designing scalable recombination operators that adapt linkage which will be briefly discussed in Section 6.1.

In most recombination operators, two individuals are randomly selected and are recombined with a probability p_c, called the crossover probability. That is, a uniform random number, r, is generated and if $r \leq p_c$, the two randomly selected individuals undergo recombination. Otherwise, that is, if $r > p_c$, the two offspring are simply copies of their parents. The value of pc can either be set experimentally, or can be set based on schema-theorem principles [**Agrawal et al., 1994**].

1. ***k-point Crossover***
   One-point, and two-point crossovers are the simplest and most widely applied crossover methods. In one-point crossover, illustrated in Figure 4.1, a crossover site is selected at random over the string length, and the alleles on one side of the site are exchanged between the individuals. In two-point crossover, two crossover sites are randomly selected. The strings between the two sites are exchanged between the two randomly paired individuals.

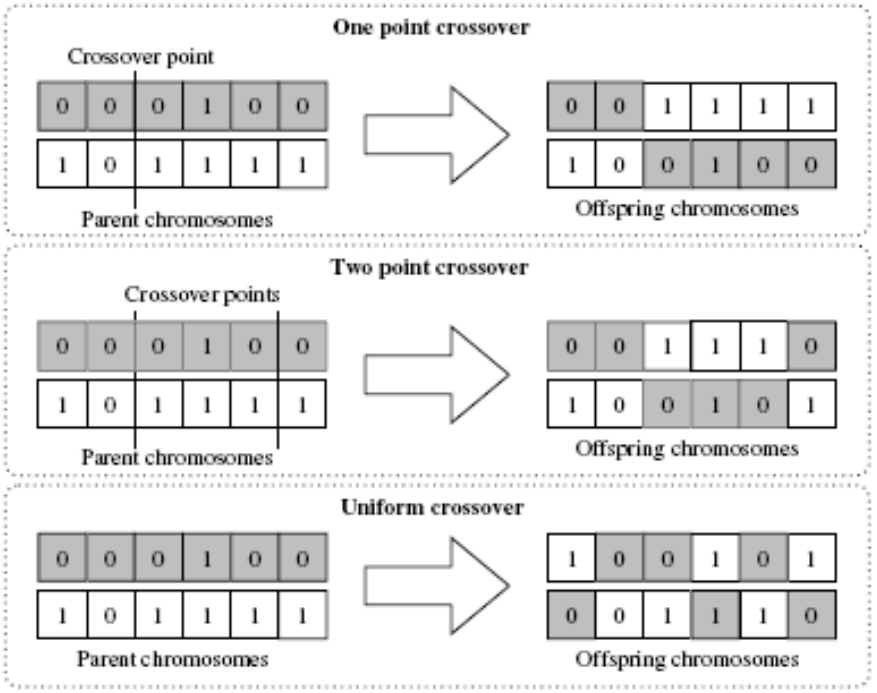

FIGURE 3.2: One-point, two-point, and uniform crossover methods.

Two-point crossover is also illustrated in Fig.3.2. The concept of one-point crossover can be extended to k-point crossover, where k crossover points are used, rather than just one or two.

2. ***Uniform Crossover***
   Another common recombination operator is uniform crossover [**Chambers, 2010**]. In uniform crossover, illustrated in Figure 4.1, every string is exchanged between the a pair of randomly selected chromosomes with a certain probability, p_c, known as the crossover probability. Usually the crossover probability value is taken to be 0.5.

3.1.2.3 Mutation Operators

If we use a crossover operator, such as one-point crossover, we may get better and better chromosomes but the problem is, if the two parents (or worse, the entire population) has the same strings at a given gene then one-point crossover will not change that. In other words, that gene will have the same string forever. Mutation is designed to overcome this problem in order to add diversity to the population and ensure that it is possible to explore the entire search space.

In evolutionary strategies, mutation is the primary variation/search operator. For an introduction to evolutionary strategies see, for example, [**Beyer et al., 2002**]. Unlike evolutionary strategies, mutation is often the secondary operator in GAs, performed with a low

probability. One of the most common mutations is the bit-flip mutation. In bitwise mutation, each bit in a binary string is changed (a 0 is converted to 1, and vice versa) with a certain probability, p_m, known as the mutation probability. As mentioned earlier, mutation performs a random walk in the vicinity of the individual. Other mutation operators, such as problem-specific ones, can also be developed and are often used in the literature.

3.1.2.4 Replacement

Once the new offspring solutions are created using crossover and mutation, we need to introduce them into the parental population. There are many ways we can approach this. One of them is Elite Solution.

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 than the solution in elite string. When elite string stop showing any further improvement, it reflects the convergence of the swarm.

3.2 Conclusion

Genetic Algorithms are easy to apply to a wide range of problems, from optimization problems like the traveling salesperson problem, to inductive concept learning, scheduling, and layout problems. The results can be very good on some problems, and rather poor on others. If only mutation is used, the algorithm is very slow. Crossover makes the algorithm significantly faster. GA is a kind of hill-climbing search; more specifically it is very similar to a randomized beam search. As with all hill-climbing algorithms, there is a problem of local maxima. Local maxima in a genetic problem are those individuals that get stuck with a pretty good, but not optimal, fitness measure. Any small mutation gives worse fitness. Fortunately, crossover can help them get out of a local maximum. Also, mutation is a random process, so it is possible that we may have a sudden large mutation to get these individuals out of this situation. (In fact, these individuals never get out. It's their offspring that get out of local maxima.) One significant difference between GAs and hill-climbing is that, it is generally a good idea in GAs to fill the local maxima up with individuals. Overall, GAs have less problems with local maxima than back-propagation neural networks.

If the conception of a computer algorithms being based on the evolutionary of organism is surprising, the extensiveness with which this algorithms is applied in so many areas is no

less than astonishing. These applications, be they commercial, educational and scientific, are increasingly dependent on this algorithms, the Genetic Algorithms. Its usefulness and gracefulness of solving problems has made it the a more favorite choice among the traditional methods, namely gradient search, random search and others. GAs are very helpful when the developer does not have precise domain expertise, because GAs possess the ability to explore and learn from their domain.

CHAPTER 4

CUDA PROGRAMMING MODAL

In this chapter, the programming modal of CUDA as a parallel computing platform is defined. C-CUDA enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). With millions of C-CUDA-enabled GPUs sold to date, software developers, scientists and researchers are finding broad-ranging uses for GPU computing with C-CUDA.

4.1 General Purpose GPU (GPGPU)

At the start of multicore CPUs and GPUs the processor chips have become parallel systems. But speed of the program will be increased if software exploits parallelism provided by the underlying multiprocessor architecture. Hence there is a big need to design and develop the software so that it uses multithreading, each thread running concurrently on a processor, potentially increasing the speed of the program dramatically. To develop such a scalable parallel applications, a parallel programming model is required that supports parallel multicore programming environment.

nVIDIAs graphics processing units (GPUs) are very powerful and highly parallel. GPUs have hundreds of processor cores and thousands of threads running concurrently on these cores, thus because of intensive computing power they are much faster than the CPU as shown in Fig.4.3. At start, they were used for graphics purposes only. But now GPUs are becoming more and more popular for a variety of general-purpose, non-graphical applications too. For example they are used in the fields of computational chemistry, sparse matrix solvers, physics models, sorting, and searching etc. The programs designed for GPGPU (General

Purpose GPU) run on the multi processors using many threads concurrently. As a result, these programs are extremely fast.

4.2 CUDA

CUDA stands for Compute Unified Device Architecture. It is a parallel programming paradigm released in 2007 by nVIDIA **"http://www.nvidia.com/"**. It is used to develop software for graphics processors and is used to develop a variety of general purpose applications for GPUs that are highly parallel in nature and run on hundreds of GPUs processor cores.

CUDA uses a language that is very similar to C language and has a high learning curve. It has some extensions to that language to use the GPU-specific features that include new API calls, and some new type qualifiers that apply to functions and variables. CUDA has some specific functions, called kernels. A kernel can be a function or a full program invoked by the CPU. It is executed N number of times in parallel on GPU by using N number of threads. CUDA also provides shared memory and synchronization among threads.

CUDA is supported only on nVIDIAs GPUs based on Tesla architecture. The graphics cards that support CUDA are GeForce 8-series, Quadro, and Tesla. These graphics cards can be used easily in PCs, laptops, and servers. More details about CUDA programming model are described in the next section.

4.2.1 System Architecture

Graphics processors were mainly used only for graphics applications in the past. But now modern GPUs are fully programmable, highly parallel architectures that delivers high throughput and hence can be used very efficiently for a variety of general purpose applications.

nVIDIA's graphics card is a new technology that is extremely multithreaded computing architecture. It consists of a set of parallel multiprocessors, that are further divided into many cores and each core executes instructions from one thread at a time as described in Fig.4.1.

Hence all those computations in which many threads have to execute the same instruction concurrently, also called data-parallel computations, are well-suited to run on GPU. nVIDIA has designed a special C-based language CUDA to utilize this massively parallel nature of GPU. CUDA contains a special C function called kernel, which is simply a C code that is

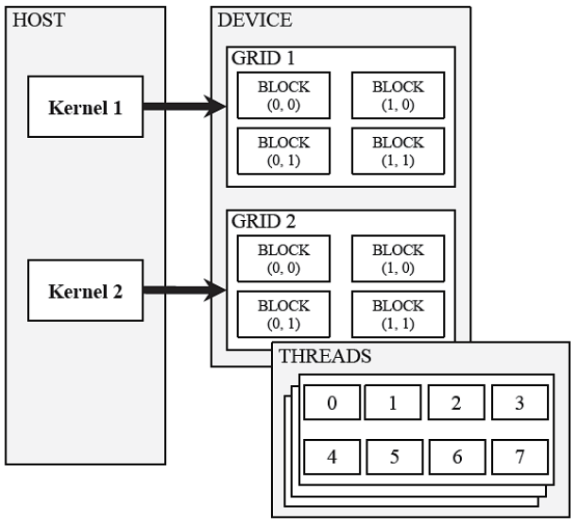

FIGURE 4.1: Basic CUDA Architecture

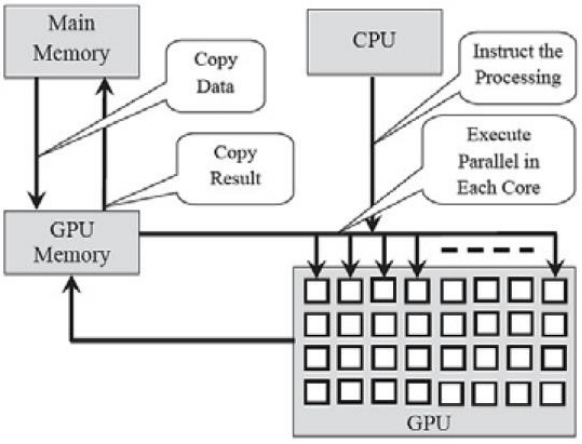

FIGURE 4.2: Memory Architecture

executed on graphics card on fixed number of threads concurrently. For defining threads, CUDA uses a grid structure.

4.2.2 Heterogeneous Architecture

CUDA programming paradigm is a combination of serial and parallel executions. Fig.4.2 shows an example of this heterogeneous type of programming. The simple C code runs serially on CPU also called the host.

Parallel execution is expressed by the kernel function that is executed on a set of threads in parallel on GPU; GPU is also called device. This kernel code is a C code for only one thread. The numbers of thread blocks, and the number of threads within those blocks that execute this kernel in parallel are given explicitly when this function is called.

The Grid consists of one-dimensional, two-dimensional or three-dimensional thread blocks. Each thread block is further divided into one-dimensional or two-dimensional threads. A thread block is a set of threads running on one processor. Fig.4.2 describes a two-dimensional grid structure and a two-dimensional block structure. Within a thread block, threads are organized together in warps. Normally 32 threads are grouped in one warp. All threads of a warp are scheduled together for execution.

All threads of a single thread block can communicate with each other through shared memory; therefore they are executed on the same multiprocessor. In this way it becomes possible to synchronize these threads.

The CUDA paradigm provides some built-in variables to use this structure efficiently. To access the *id* of a thread block the *blockIdx* variable (values from 0 to *gridDim*-1) is used and to access its dimension the *blockDim* variable is used while *gridDim* gives the dimensions of the grid. Each individual thread is identified by *threadIdx* variable, can have values from 0 to *blockDim*-1. Warp Size specifies warp size in the threads. All these variables are built-in in kernel. The maximum allowed sizes of each dimension of grid is 65535, and x, y, and z dimensions of a thread block are 512, 512, and 64, respectively .

The allocation of the number of thread blocks to each multiprocessor is dependent on the necessity of the shared memory and registers by each thread block. More memory and registers requirement by each thread block means allocation of less thread blocks to each multiprocessor. In this case the remaining thread blocks have to wait for their turn for execution.

All this threads creation, their execution, and termination are automatic and handled by the GPU, and is invisible to the programmer. The user only needs to specify the number of threads in a thread block and the number of thread blocks in a grid.

4.2.3 CUDA Programming Model

CUDA is a parallel computing platform and programming model introduced by nVIDIA that increases computing performance substantially by harnessing the power of parallelism of the GPU. CUDA gives program developers the direct access to the virtual instruction set and memory of the parallel computational elements in CUDA enabled GPUs. The CUDA platform is accessible to software developers through CUDA accelerated libraries, compiler directives (such as Open ACC), and extensions to industry-standard programming languages, including C, C++ and FORTRAN. C/C++ programmers use CUDA C/C++, compiled with nvcc which is nVIDIAS LLVM-based C/C++ compiler. A C/C++ program using CUDA

can interface with one GPU or multiple GPUs and can be identified and utilized in parallel, allowing for unprecedented processing power on desktop computers.

CUDA allows multiple kernels to be run simultaneously on GPU cores. CUDA refers to each kernel as a grid. A grid is a collection of blocks. Each block runs the same kernel, however, is independent of each other (this has significance in terms of access to memory types). A block contains threads, which are the smallest divisible unit on a GPU.

A thread block is a number of SIMD threads that work on core at a given time. Threads can exchange information through the shared memory and can be synchronized. The operations are systematized as a grid of thread blocks. For parallel operation the programming model allows a developer to partition a program into several subprograms, each of which is executed independently on a block. Each subprogram can be further divided into finer pieces that perform the same function but execute on different threads independently within the same block. For data set parallelism, data sets can be divided in to smaller chunks that are stored in the shared memory, and each chunk is visible to all threads of the same block. This local data arrangement approach reduces the need to access off-chip global memory, which reduces data access time.

The next critical component of a CUDA application is the memory model. There are multiple types of memory and each has different access times. The GPU is broken up into read-write per thread registers, read-write per thread local memory, read-write per-block shared memory, read-write per-grid global memory, read-only per-grid constant memory, and read-only per-grid texture memory. Texture and constant memory have relatively small access latency times, while global memory has the largest access latency time. Applications should minimize the number of global memory reads and writes. This is typically achieved by having each thread read its data from global memory and store its content into shared memory.

The basic structure of a CUDA code comprises of allocation of memory space (using *cudaMalloc* function) on device (GPU) and (using regular *malloc* function) on host (CPU). Data which is copied from the host to the device for the call of kernel routine to be executed on the GPU (using function *cudaMemcpy*) also defines the number of threads and their physical structure. Kernel is prefixed with the global keyword. Results are transferred from GPU to CPU in the same fashion as data is copied from host to device.

4.3 CPU Vs GPU

In recent years, the Graphics Processing Unit (GPU) has emerged as a powerful computation device and the main question that arises is why the GPU is much faster for computation than the Central Unit Processing (CPU). The differences lie in the architecture. While the CPU

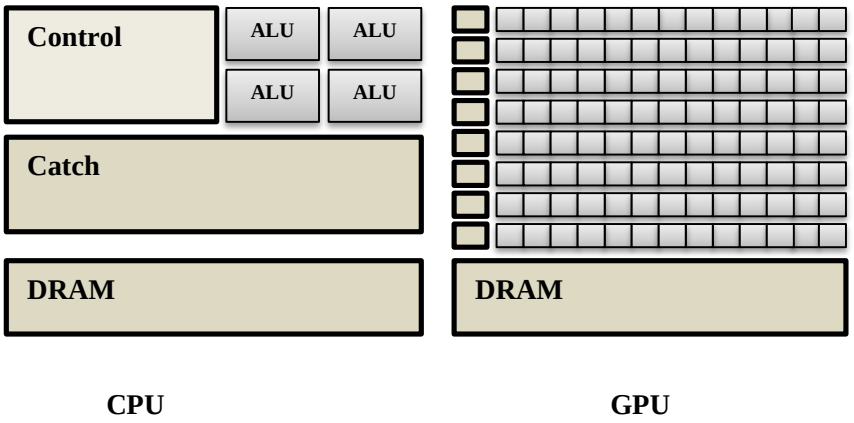

FIGURE 4.3: CPU Vs GPU

is conceptualized for general purposes carrying out arbitrary operations like input/output access, processing, etc; the GPU is conceptualized for performance optimization for defined tasks. The central issue is that not all algorithms can be effectively implemented on the GPU. Only numerical problems that are inherently parallel may have profit of this technology as described in Fig.4.1.

4.4 Conclusion

The difference in speed between CPU and GPU is due to the architecture of GPU. While the CPU is conceptualized for general purposes carrying out arbitrary operations like input/output access, processing, etc; the GPU is conceptualized for performance optimization for defined tasks. However, not all algorithms can be effectively implemented on the GPU. Only numerical problems that are inherently parallel may have profit of this technology. CUDA, which allows almost the direct translation of C code onto the GPU. The CUDA C extends the syntax of C language. Thus it has become easy to convert the existing code from C language to CUDA.

CHAPTER 5

BUILDING BLOCK OF GA ON GPU

In this chapter, 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 performance 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.

5.1 Introduction

Genetic Algorithm (GA) is a swarm based global search algorithm inspired natural mechanism of genetical improvements in biological species [**De Giovanni and Pezzella, 2010**], described by Darwinian Theory of *Survival of Fittest*. It was developed by John Holland in 1970 at University of Michigan [**Holland, 1973**]. It simulates the process for evolving solutions to arbitrary problems [**Ghoseiri and Ghannadpour, 2010**].

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. The 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 fact for their success are simple structure, broad relevance with problem [**Murphy, 2012**]. **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 [**Xing and Qu, 2012**]. GAs produces high-quality solutions through its high adaptation property with the environment changes [**Yang et al., 2010**]. **Prakash** and **Deshmukh** investigated the use of meta-heuristics for combinatorial decision-making problem in flexible manufacturing system with GA [**Prakash et al., 2011**]. Prominent GA applications include pattern recognition **Adams et al.** [**2010**], classification problems [**Quteishat et al., 2010**], protein folding [**Zhang and Wu, 2012**] and neural network design [**Magnier and Haghighat, 2010**] etc. GAs are also suitable for multi-objective optimal design problems [**Omkar et al., 2011**], 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 [**Fabris and Krohling, 2012**], [**Maitre et al., 2012**], [**Maitre et al., 2010**], [**Franco et al., 2010**] 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 5.2, describes GA Operators along with pseudo codes for implementation in parallel. Section 5.3, introduces architecture detail of GPGPU and C-CUDA followed by Section 5.4 of its implementation.

5.2 GA Operators

GA provides number of solutions however best solution among them is one with least processing time [**Ryoo et al., 2008**]. The three primary operators involved in GA are: (1) Selection, (2) Crossover, (3) Mutation and (4) Elite Solutions operators described as follows.

5.2.1 Selection

There are three most popular different types of Selection Strategies viz., Tournament Selection, Ranked-Based Roulette Wheel, and Roulette Wheel Selection [**Noraini and Geraghty, 2011**]. 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 there fitness in the swarm. This leads the algorithm to find global best solution rather then converging to its nearest local best solution.

5.2.1.1 Tournament Selection

The mechanism of tournament selection is based upon random selection of solutions from current population. The selected solutions forms 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.

5.2.1.2 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 then 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.

5.2.1.3 Roulette Wheel Selection

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 the fitness level of the same. Let $f_1, f_2, f_3, ...f_n$ be fitness values of chromosomes $1, 2, 3, \ldots n$. Then the probability of selection P_i for chromosomes i is defined as (5.1),

$$P_i = \frac{f_i}{\sum_{j=1}^{n} f_i} \tag{5.1}$$

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 maintain diversity of solution.

5.2.1.4 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

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 ← (0, 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 (C*Sum*)
  Spin the wheel *pop_size* times with random force
   If C*Sum* < *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

solution get selected as parent chromosome for next generation as depicted in Algorithm 1.

5.2.2 Crossover

The process of producing child chromosomes from parent chromosomes is termed as crossover. It is a significant operator which mimic 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.

5.2.2.1 Single Point Crossover

In single point crossover, the selected parent solution chromosome string get swaped from a randomly selected crossover point. The resulting chromosome after swapping form children population for next generation.

5.2.2.2 Double Point Crossover

Double point crossover is similar to that of single point crossover however, the crossover points are two rather than one.

5.2.2.3 Uniform Distribution Crossover

In Uniform Distribution Crossover technique, chromosomes of parent solution is 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.

5.2.2.4 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

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
```

chromosome. The pseudocode for its parallel implementation is shown in Algorithm 2.

5.2.3 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.

5.2.3.1 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

Algorithm 3 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
```

number thread is needed to mutate the whole population.

5.2.4 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 than the solution in elite string. When elite string stop showing any further improvement, it reflects the convergence of the swarm.

5.3 Basic GPGPU and C-CUDA

5.3.1 General Purpose Computation on 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 [**Ryoo et al., 2008**]. It support 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 [**Oiso et al., 2011**]. In recent years, such features attracted lots of researcher and developers to harness GPUs for various general purpose computation (GPGPU).

Oiso and **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 [**Oiso et al., 2011**]. **Jiri** and **Jaros** achieved a speedup of 35-781 as compared to the CPU implementation, proposed the application of GA for determining the Knapsack Problem with a multi-GPU for population size of 128 to 2048 individuals per island [**Pospíchal et al., 2010**]. **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 [**Arora et al., 2010**].

GA proves suitable in determining several realistic problems [**Oiso et al., 2010**]. 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.

5.3.2 Application Programme Interface (API) of GPU

The program based on GPGPU can be easily developed using CUDA API of GPU 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.

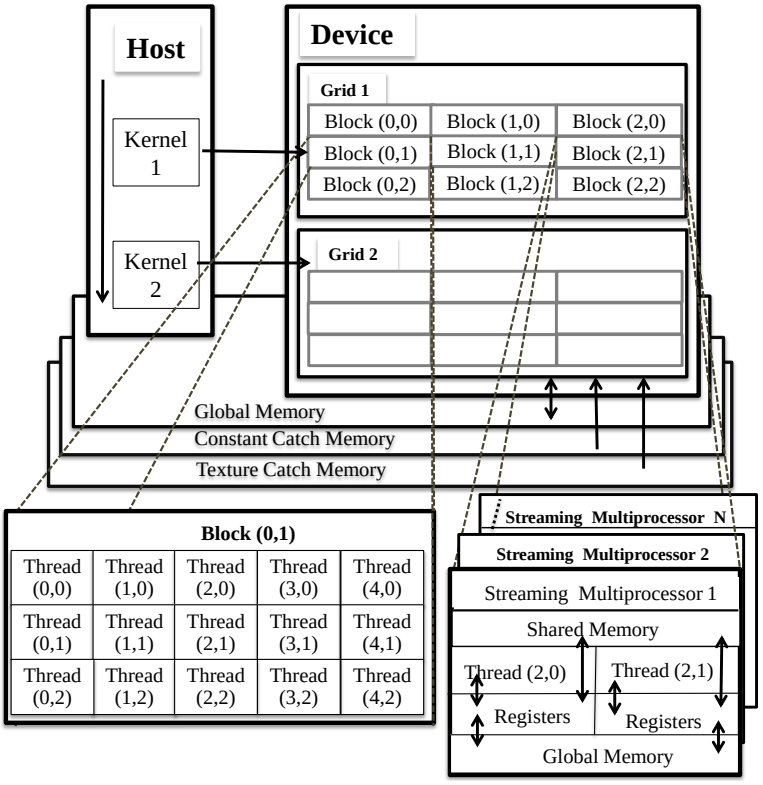

FIGURE 5.1: Typical CUDA Memory Processing and Architecture

The threads in GPU architecture can be grouped into *blocks* and *grids* as depicted in Fig.5.1. 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* [**Oiso et al., 2011**].

The entire shared memory in CUDA architecture is divided into four types viz., texture memory, constant memory, per-thread 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 catch 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.

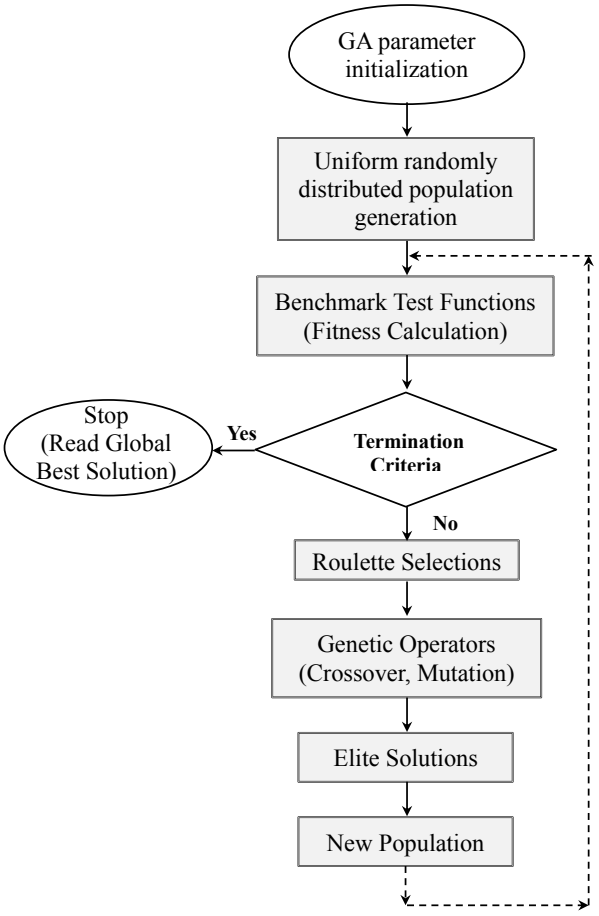

FIGURE 5.2: Implementation GA Flowchart (Shaded Modules represents GPU computation, Non-shaded Modules represents computation on CPU)

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*.

5.4 Implementing GA using C-CUDA

Implementation of GA include 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 Fig. 5.2. The implementation of C-CUDA kernels on GA is based on under mentioned principle:

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 $(2^{16} - 1)^2$.

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.

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 turns 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.

CHAPTER 6

SIMULATION RESULTS FOR LIMITATIONS

In this chapter, the ratio of total time consumed on CPU for serial implementation of GA to the time consumed by GPU in parallel implementation is shown. It also shows speedup for the same experiment on different platforms.

6.1 Introduction

The simulation result for 10,000 iteration with the dimension size of 32 and 64 analyzed in this section. The simulation speed of GPU with 10,000 iteration is greater as compared to 100,000 iteration of same algorithm. The best speedup result is shown with dimension size of 64 and 100,000 iteration.

6.2 Performance Evaluation

6.2.1 Experimental setup

The experiments were conducted on two different PCs (PC1 and PC2) and with same nVidia cards (Refer Table 6.1 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 multipro-

TABLE 6.1: 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

cessors 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 6.2 and Table 6.3 in next section shows a significant speedup.

6.3 Case study 1

In this experiments, 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 6.2 and indicated in Fig. 6.1 & Fig. 6.2.

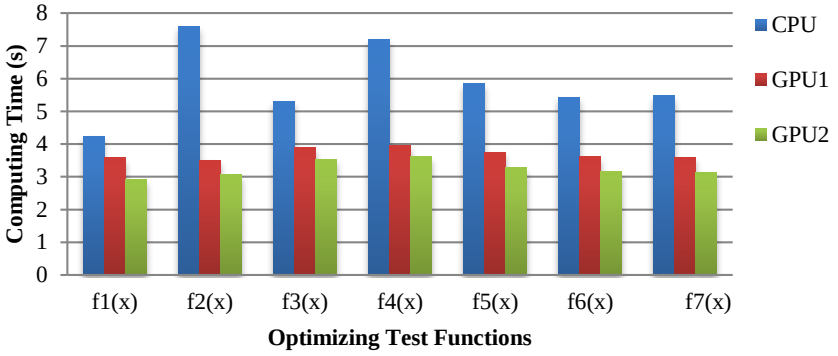

FIGURE 6.1: Computing time for 10,000 iteration with 32 dimension size.

TABLE 6.2: C-CUDA Vs. C Performances for 10,000 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

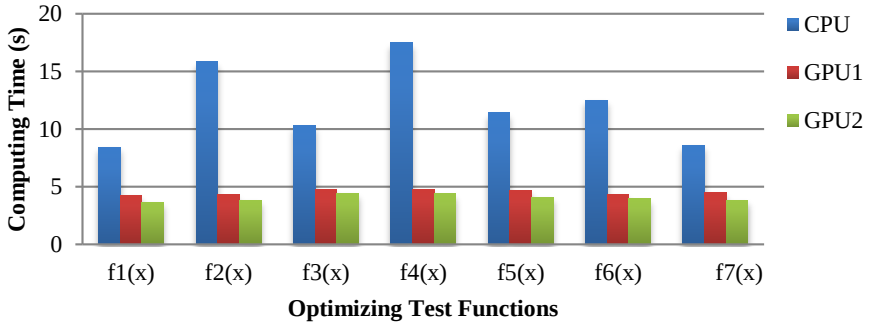

FIGURE 6.2: Computing time 10,000 iteration with 64 dimension size.

The best computational performance achieved among GPU_1 and GPU_2 for dimension size of 32, is 2.17 times for $f_2(x)$ on GPU_1, while on GPU_2 with the dimension size of 64, $f_2(x)$ shows a speedup of 4.15 times higher then its CPU execution time. The Table 6.2 depicts the best result for 10,000 iteration along with quality.

6.4 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 are shown in Table 6.3 and indicated in Fig. 6.3 & Fig. 6.4.

The highest speedup achieved in this case for dimension size 32 among GPU_1 and GPU_2 is 2.39 for test function $f_7(x)$. On the other hand, dimension size 64 have best speedup of 2.78

TABLE 6.3: 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

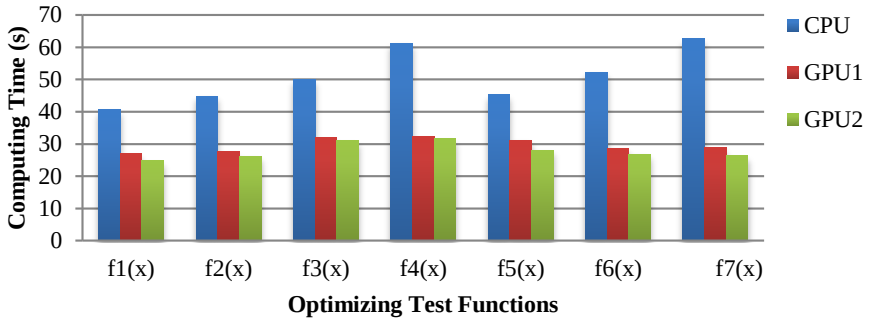

FIGURE 6.3: Computing time for 100,000 iteration with 32 dimension size.

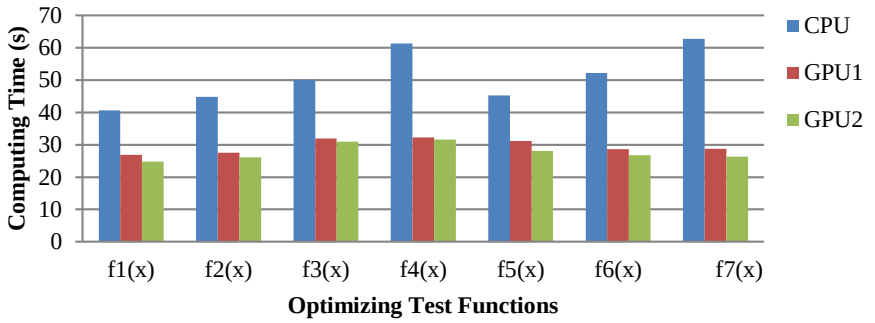

FIGURE 6.4: Computing time for 100,000 iteration with 64 dimension size.

times for $f_7(x)$. The GPU average execution time is 33.70 seconds while 127.52 seconds in CPU.

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.

CHAPTER 7

CONCLUSIONS AND FUTURE SCOPE

7.1 Introduction

In this chapter, the demonstration of implementation and comparative runtime performances of GA on GPU and CPU with the use of a graphics API is shown. However, API which is flexible, scalable, and can be used by any researcher with knowledge of C. It has been demonstrated that the CPU works equally fast as GPU when the system is small. As the number of genes or the number of inputs increase the GPU outperforms the CPU runtime. The notable contributions and conclusions of this work are:

1. Along with parallelizing the task of solution evaluation, we have parallelized the respective GA operators (the random number generation, initialization, selection operation, and mutation operation) such that the overall implementation is effective for the GPGPU.

2. Performing a study on the effect of number of threads, we have found that around 64 threads per block provide the best performance.

3. The highest speedup achieved in this case for dimension size 32 among GPU_1 and GPU_2 is 2.39 for test function $f_7(x)$. On the other hand, dimension size 64 have 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.

4. Study on the effect of evaluation time has indicated that the parallel implementation is effective in problems requiring more computational time per solution evaluation

This study inspires us to port GAs on consumer level graphics card for computationally expensive problems. The speed-ups achieved in this study are extremely encouraging. The GPU has an initial setup overhead of kernel loading and memory transfer, however, subsequent parallel computations leads to a small increase in processing time despite a substantial increase in computational load. On the other hand, CPU has no initial cost, but computation time grows linearly with computational load much beyond GPGPU runtime.

7.2 Future Research Agenda

Most of the times, a solution to a problem gives many issues to be investigated. The following remains on our future agenda:

1. In this thesis, the optimization algorithms GA will be more improved by modifying single objective GA to multi-objective GA.
2. This research can be extended to multi-objective NSGA.
3. 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

Adams, J., Woodard, D. L., Dozier, G., Miller, P., Bryant, K., and Glenn, G. (2010). Genetic-based type ii feature extraction for periocular biometric recognition: Less is more. In *Pattern Recognition (ICPR), 2010 20th International Conference on*, pages 205–208. IEEE.

Agrawal, R. B., Deb, K., and Agrawal, R. B. (1994). *Simulated binary crossover for continuous search space*. Citeseer.

Arora, R., Tulshyan, R., and Deb, K. (2010). Parallelization of binary and real-coded genetic algorithms on gpu using cuda. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE.

Belegundu, A. D. and Chandrupatla, T. R. (2011). *Optimization concepts and applications in engineering*. Cambridge University Press.

Beyer, H.-G., Schwefel, H.-P., and Wegener, I. (2002). How to analyse evolutionary algorithms. *Theoretical Computer Science*, 287(1):101–130.

Blickle, T. and Thiele, L. (1995). A comparison of selection schemes used in genetic algorithms.

Brodtkorb, A. R., Hagen, T. R., Schulz, C., and Hasle, G. (2013). Gpu computing in discrete optimization. part i: Introduction to the gpu. *EURO Journal on Transportation and Logistics*, 2(1-2):129–157.

Chambers, L. D. (2010). *Practical handbook of genetic algorithms: complex coding systems*, volume 3. CRC press.

De Giovanni, L. and Pezzella, F. (2010). An improved genetic algorithm for the distributed and flexible job-shop scheduling problem. *European journal of operational research*, 200(2):395–408.

De Veronese, L. and Krohling, R. A. (2010). Differential evolution algorithm on the gpu with c-cuda. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–7. IEEE.

Fabris, F. and Krohling, R. A. (2012). A co-evolutionary differential evolution algorithm for solving min–max optimization problems implemented on gpu using c-cuda. *Expert Systems with Applications*, 39(12):10324–10333.

Farber, R. (2011). *CUDA application design and development*. Elsevier.

Franco, M. A., Krasnogor, N., and Bacardit, J. (2010). Speeding up the evaluation of evolutionary learning systems using gpgpus. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1039–1046. ACM.

Ghoseiri, K. and Ghannadpour, S. F. (2010). Multi-objective vehicle routing problem with time windows using goal programming and genetic algorithm. *Applied Soft Computing*, 10(4):1096–1107.

Goldberg, D. E. (1989). Real-coded genetic algorithms virtual alphabets and blocking. *University of Illinois at Urbana Champaign*, 1:21.

Goldberg, D. E. (2002). *The design of innovation: Lessons from and for competent genetic algorithms*. Kluwer Academic Publishers.

Holland, J. H. (1973). Genetic algorithms and the optimal allocation of trials. *SIAM Journal on Computing*, 2(2):88–105.

Holland, J. H. (1975). *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. U Michigan Press.

Jamil, M. and Yang, X.-S. (2013). A literature survey of benchmark functions for global optimisation problems. *International Journal of Mathematical Modelling and Numerical Optimisation*, 4(2):150–194.

Jaros, J. (2012). Multi-gpu island-based genetic algorithm for solving the knapsack problem. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pages 1–8. IEEE.

Kirk, D. B. and Wen-mei, W. H. (2012). *Programming massively parallel processors: a hands-on approach*. Newnes.

Magnier, L. and Haghighat, F. (2010). Multiobjective optimization of building design using trnsys simulations, genetic algorithm, and artificial neural network. *Building and Environment*, 45(3):739–746.

Maitre, O., Krüger, F., Querry, S., Lachiche, N., and Collet, P. (2012). Easea: specification and execution of evolutionary algorithms on gpgpu. *Soft Computing*, 16(2):261–279.

Maitre, O., Lachiche, N., and Collet, P. (2010). Fast evaluation of gp trees on gpgpu by optimizing hardware scheduling. In *Genetic Programming*, pages 301–312. Springer.

Munawar, A., Wahib, M., Munetomo, M., and Akama, K. (2011). Advanced genetic algorithm to solve minlp problems over gpu. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 318–325. IEEE.

Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.

Nickolls, J. and Dally, W. J. (2010). The gpu computing era. *IEEE micro*, 30(2):56–69.

Noraini, M. R. and Geraghty, J. (2011). Genetic algorithm performance with different selection strategies in solving tsp. In *Proceedings of the World Congress on Engineering 2011*, volume II.

Oiso, M., Matsumura, Y., Yasuda, T., and Ohkura, K. (2010). Evaluation of generation alternation models in evolutionary robotics. In *Natural Computing*, pages 268–275. Springer.

Oiso, M., Yasuda, T., Ohkura, K., and Matumura, Y. (2011). Accelerating steady-state genetic algorithms based on cuda architecture. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 687–692. IEEE.

Omkar, S., Senthilnath, J., Khandelwal, R., Narayana Naik, G., and Gopalakrishnan, S. (2011). Artificial bee colony (abc) for multi-objective design optimization of composite structures. *Applied Soft Computing*, 11(1):489–499.

Pospíchal, P., Jaros, J., and Schwarz, J. (2010). Parallel genetic algorithm on the cuda architecture. In *Applications of Evolutionary Computation*, pages 442–451. Springer.

Prakash, A., Chan, F. T., and Deshmukh, S. (2011). Fms scheduling with knowledge based genetic algorithm approach. *Expert Systems with Applications*, 38(4):3161–3171.

Quteishat, A., Lim, C. P., and Tan, K. S. (2010). 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*, 40(3):641–650.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W. (2008). Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM.

Salmon, J. K., Moraes, M. A., Dror, R. O., and Shaw, D. E. (2011). Parallel random numbers: as easy as 1, 2, 3. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE.

Sanders, J. and Kandrot, E. (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

Singh, S., Kaur, J., and Sinha, R. S. (2014). A comprehensive survey on various evolutionary algorithms on gpu. In *International Conference on Communication, Computing & Systems (ICCCS2014)*. hgpu. org.

Stentiford, F. (2001). An evolutionary programming approach to the simulation of visual attention. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 2, pages 851–858.

Suganthan, P. N., Hansen, N., Liang, J. J., Deb, K., Chen, Y.-P., Auger, A., and Tiwari, S. (2005). *Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization*. PhD thesis, KanGAL Report.

Van Dam, A. and Feiner, S. K. (2014). *Computer graphics: principles and practice*. Pearson Education.

Vidal, P. and Alba, E. (2010). A multi-gpu implementation of a cellular genetic algorithm. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–7. IEEE.

Whitley, L. D. et al. (1989). The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *ICGA*, volume 89, pages 116–123.

Xing, H. and Qu, R. (2012). A compact genetic algorithm for the network coding based resource minimization problem. *Applied Intelligence*, 36(4):809–823.

Yang, S., Cheng, H., and Wang, F. (2010). 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*, 40(1):52–63.

Zhang, Y. and Wu, L. (2012). Artificial bee colony for two dimensional protein folding. *Advances in Electrical Engineering Systems*, 1(1):19–23.

INDEX