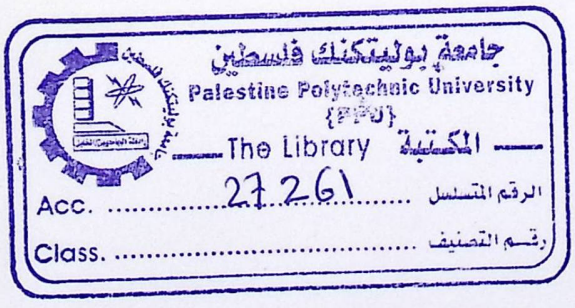
Palestine Polytechnic University
Deanship of Graduate Studies and Scientific Research
Master of Informatics

E-MOGA: A General Purpose Platform for Multi Objective Genetic Algorithm running on CUDA

Submitted By
Sami Mohammad Salamin

In Partial Fulfillment of the Requirements for the Degree
Master of Informatics

February, 2012



The undersigned hereby certify that they have read and recommended to the Deanship of Graduate Studies and Scientific Research at Palestine Polytechnic University the acceptance of a thesis entitled:
E-MOGA: A General Purpose Platform for Multi Objective Genetic Algorithm running on CUDA

Submitted by **Sami Mohammed Salamin**

In partial fulfillment of the requirements for the degree of Master in Informatics

Graduate Advisory Committee:

Dr. Ismail Romi, Palestine Polytechnic University, Committee Chair
Signature:.....Date:...5/3/2012

Dr. Hashem Tamimi, Palestine Polytechnic University, Committee Member (Supervisor)
Signature:..... Date:...6/3/2012

Dr. Mohammed Aldasht, Palestine Polytechnic University, Committee Member (Supervisor)
Signature:..... Date:...5/3/2012

Dr. Radwan Tahboub, Palestine Polytechnic University, Committee Member
Signature:..... Date:...5/3/2012..

Dr. Mohammed Awad, Arab American University, Committee Member
Signature:..... Date:...8/3/2012

Thesis Approved

Prof. Dr. Karim Tahboub
Dean of Graduate Studies and Scientific Research
Palestine Polytechnic University
Signature:..... Date:...19.3.2012

Abstract

The Multi Objective Genetic Algorithm is the process of simultaneously optimizing two or more conflicting objectives subject to certain constraints.

Compute Unified Device Architecture (CUDA) is a general purpose parallel computing architecture, with a new parallel programming model and instruction set architecture, which uses the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU

This thesis considers the development of an Enhanced Multi Objective Genetic Algorithm (E-MOGA) platform that runs on Compute Unified Device Architecture (CUDA) hardware as a general purpose tool that can solve conflict optimization problems. This tool demonstrates significant speed gains using affordable, scalable and commercially available hardware.

The objectives of the research are: to enhance the general purpose Multi Objective Genetic Algorithm (MOGA) to be ready for execution on CUDA by parallelizing the time consuming parts, to test the performance of the enhanced MOGA on many testing cases, to test the quality of the result of the enhanced MOGA and to study

the effect of the number of objectives on performance.

The implemented system works under MATLAB environment. It was tested on GPU GeForce GTX 9500 in CUDA platform. The experiment results showed an average Speedup with more than 28X. The quality measured, using the error with respect to optimal solution, outperform the sequential MOGA.

الملخص

تهدف فكرة البحث إلى إيجاد أداة تساعد في حل المشكلات متعددة الأهداف Multi Objective optimization باستخدام Genetic Algorithm والتي تعرف باسم MOGA. على خلاف الأدوات المستخدمة لحل مثل هذا النوع من المشاكل، ستعمل هذه الأداة بشكل أسرع بعد أن يتم تعديل التركيب البرمجي لها لتصبح قادرة على تنفيذ بعض الأجزاء منها بشكل متوازي بواسطة كرت الشاشة GPU وباستخدام نظام يدعى Compute Unified Device Architecture (CUDA) الذي تم تصميمه من قبل شركة nVidia.

ستتناول الرسالة مواضيع متعددة تندرج تحت الهدف الرئيسي وهي مقسمة كما يلي: تعديل الخوارزمية المستخدمة في حل مشكلة متعددة الأهداف ليصبح أجزاء منها متوازية التنفيذ وقادرة على أن تنفذ باستخدام كرت الشاشة، العمل على اختبار كفاءة النظام المستحدث من ناحية سرعة التنفيذ ومقارنته مع نظام قائم، العمل على تحسين كفاءة النتائج من حيث بعدها عن الأخطاء ومقارنة النتائج مع نظام قائم، أخيراً اختبار تأثير عدد الأهداف (objectives) والذي يعكس صعوبة المشكلة على أداء النظام.

تم إنهاء العمل على البحث باستخدام بيئة MATLAB وقد تم اختباره باستخدام كرت شاشه من نوع GeForce GTX 9500 والذي يدعم تقنية CUDA. النتائج التي تم التوصل إليها كانت مرضيه حيث حصلنا على كفاءة أفضل من النظام القائم حيث أن النظام الجديد كان أسرع بمعدل 28 مره كما تم إثبات أن النظام الجديد يعطي جودة أفضل في النتائج حيث تم قياس جودة النتائج من خلال قياس قرب الحلول الناتجة مع الحلول النموذجية والتي تم اختيارها إثناء مرحلة الاختبارات.

DECLARATION

I declare that the Master Thesis entitled "E-MOGA: A General Purpose Platform for Multi Objective Genetic Algorithm running on CUDA" is my own original work, and hereby certify that unless stated, all work contained within this thesis is my own independent research and has not been submitted for the award of any other degree at any institution, except where due acknowledgment is made in the text.

Sami Mohammed Salamin

Signature:..... Date: 20/12/2012

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for the master degree in Informatics at Palestine Polytechnic University, I agree that the library shall make it available to borrowers under rules of the library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made. Permission for extensive quotation from, reproduction, or publication of this thesis may be granted by my main supervisor, or in his absence, by the Dean of Graduate Studies and Scientific Research when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Sami Mohammed Salamin

Signature:..... Date: 20/12/2012

DEDICATION

I dedicate this work to my parents, brothers and my wife. Without their patience, understanding, there continuous support, and most of all love, the completion of this work would not have been possible during my vital educational years. Also their endless patience and encouragement when it was most required. Also I dedicate this work to my supervisors Dr. Hashem and Dr. Mohammed for their help and support during the work of my thesis.

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my supervisors Dr. Hashem Tamimi and Dr. Mohammad Aldasht. they offered me great freedom on choosing my favorite research topic and developing my research interests. they continuously provided me with help, encouragement, and with extensive knowledge. I thank all of my colleagues and relatives for their support.

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my supervisors Dr. Hashem Tamimi and Dr. Mohammad Aldasht. they offered me great freedom on choosing my favorite research topic and developing my research interests. they continuously provided me with help, encouragement, and with extensive knowledge. I thank all of my colleagues and relatives for their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Before 2005 Graphical Processing Unites (GPUs) were used only in the video cards for 3D rendering. Their main applications are in personal computers, mobile phones, workstations and embedded systems. The architecture of the GPU differs from CPU; the GPU is designed and specialized for intensive and highly parallel computation involved in graphics rendering [17]. Tens or hundreds of cores (processors), found in the single card devoted to data processing, are related to graphical operations such as rendering, shading... etc, rather than data caching and flow control [17].

With the trends of exploitation, all the available resources found inside computer cases, and since the GPU offers both high computing power and memory bandwidth for a reasonable price, many researchers start thinking to use the Graphical Processing Unit (GPU) for not only displaying videos or gaming but also for using them as parallel processors [27].

The GPUs manufacturers open the way for developers to use their cargo for

General-Purpose Computation on Graphical Processing Units (GPGPU) by different programming language levels. Compute Unified Device Architecture (CUDA) is one of the most powerful programming tools introduced by Nvidia corporation in 2006 and was first publicly released in the end of 2007 [17]. Before that, programming platform such as DirectX, OpenGL, HLSL, GLSL had already been introduced [30].(see Chapter 2).

Multi-objective optimization is the process of simultaneously optimizing two or more conflicting objectives subject to certain constraints [18] such as finding the tradeoff between the weight and the speed of a car. The Genetic Algorithm is one of the most popular heuristic techniques and Evolutionary Algorithms to solve Multi-Objective Design and Optimization problems [21].

The Genetic Algorithm (GA) is a subfield of artificial intelligence that involves combinatorial optimization problems by exploring all the possible solutions to get the optimal one. In many cases, it is a time consuming to get the optimal solution, so, the computational time can be reduced by modifying the algorithm to explore the solution search space in parallel [6].

1.1 Background

Multi-objective formulations are realistic models for many complex engineering optimization problems. In many real-life problems, objectives under consideration conflict with each other, and optimizing a particular solution with respect to a single objective can result in unacceptable results regarding to the other objectives [25].

A reasonable solution to a multi-objective problem is to investigate a set of solutions, each of which satisfies the objectives at an acceptable level without being dominated by any other solution [23]. In other words, the Multi-objective optimization can be applied in many fields and to many cases, wherever optimal decisions need to be taken in the presence of trade-offs between two or more conflicting objectives, such as maximizing profit and minimizing the cost of a product [13].

Genetic Algorithms (GAs) are powerful, domain-independent search techniques that can be modified to work in parallel computational environments [43]. GAs employ selection, mutation, and crossover to generate new search points in the state space [44]. A genetic algorithm starts with a set of individuals that forms the population of the algorithm, for each iteration, each individual is evaluated by using the fitness function while the termination function is invoked to determine whether the termination criteria have been satisfied [45].

Although GAs are very effective in solving many practical problems, the execution time can become a limiting factor for some huge problems since a lot of candidate solutions must be evaluated. Fortunately, the most time consuming part is for fitness evaluation, and this can be performed independently for each individual in the population using various types of parallelization [44].

GPUs are massively multi threaded multi core chips [14] with a separate memory. GPUs have a different architecture than CPU, they contain hundreds of cores, and can handle thousands of threads at the same time. That is called massively-parallel programs and this type of programming is expected to improve all the complex

algorithm performance in many fields.

The performance enhancement by parallelization shows great results, but not any more; due to multi cores technologies, "Massively parallel computing lost momentum to the inexorable advance of commodity technology" [14], the enhancement achieved by parallelization is not cheap therefor, so it is not the best solution for low budget research [53]. Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by Nvidia and it is the computing engine in Nvidia graphics processing units (GPUs) that is accessible to software developers through many of standard programming languages like C/C++.

1.2 Scope

With the increase of developing powerful computer systems, we are still facing a higher increase in the complexity of the software. Many applications may need to solve complex operations and algorithm in reasonable time. The Multi-objective Genetic Algorithm (MOGA) is one of the most commonly used techniques in many important fields, such as engineering [38], chemistry [31], scheduling [41] ...etc, that needs to be solved as fast as possible.

As mentioned before, GPUs are capable of manipulating and calculating parallel structure operations such as matrix and vector operations faster than CPU [17]. Hence, our scope is to help the researchers and engineers in their work fields by designing a general purpose tool to find the best trade off between conflict objectives through implementing Enhanced MOGA (E-MOGA). E-MOGA will execute

on CPU and GPU to speedup the system.

1.3 Literature Review

Many researchers attempt to improve the performance of many complex and time consuming algorithm by using the power of GPU in the CUDA device. The Genetic Algorithm is one of the most important algorithm in the artificial inelegance field that considered as time consuming algorithm. The researchers try to modify the genetic algorithm to be involved in the parallel form and they got a better performance than the old style. When they execute the algorithm in parallel or distributed processors environment, a serious step to increase the performance and speedup with less cost is taken to modify the algorithm and test it on CUDA device.

Returning to many implementations using CUDA with genetic algorithm, one can see different successful applications for this idea in different areas such as: solving theoretical computer science problems including the SATisability application [37]. Biological application such as Autodock (a Drug Discovery Tool) [22], or algorithmic problems such as parallel 0/1 knapsack [45].

The results show a clear improvement in decreasing the running time in many applications by speeding up the system. When we refer to the Autodock implementations, we can get up to 50x on the fitness function evaluation and 10x-47x speedup on the core genetic algorithm [51]. Also, an average Speedup around 462x can be achieved in the parallel genetic 0/1 Knapsack problem for 4-bit to 40-bit instance [45]. Another example of improvement is the SATisfiability which is con-

sidered as NP-hard problem with a Speedup around 25x.

Many researchers working in the multi-objective optimization field try to improve the performance of the system by using the genetic algorithm in order to speedup the systems. As we can see in [5]; the authors proposed an interactive approach based method for solving multi-objective optimization problems modeled in fuzzy environment. The experiments were done by using the proposed algorithm for solving five test examples and they showed that the performance of the proposed method is quite satisfactory.

Further more, a good application can be found in [51], in the field of information retrieval systems, where the keyphrases have been used extensively to make information exchange operation easier and to organize information as well as assisting information retrieval. In their implementation, they proposed an automated keyphrase extraction algorithm using a non-dominated sorting multi-objective genetic algorithm. The objective of such approach is to find the smallest phrase set that has the best precision, and the result shows that 90% of the generated phrases are considered appropriate for use in a thesaurus engineering design.

Moreover, a good implementation that were used to map the parallel island genetic algorithm to the CUDA software model. The proposed mapping is tested using Rosenbrock, Griewank and Michalewicz benchmark functions. The obtained results indicate that the new approach leads to a good speedups with reasonable results quality [44].

Another research found in [40], the researchers implemented and mapped GAs

to the CUDA environment. The major characteristic point of this study is that the parallel processing is adopted not only for individuals but also for the genes in an individual. The proposed implementation is evaluated through eight test functions. The reported results using the proposed implementation method yields 7, 6-18 and 4 times faster results than those of a CPU implementation.

A very related work can be found in [20]. They introduce a Multi Objective Parallel Genetic Algorithm (MOPGA) using the Compute Unified Device Architecture (CUDA). This work tries to highlight the power of GPU to implement the parallel MOGA in document search problem. The algorithm demonstrates significant speed gains vary from 2X to 55X.

1.4 Reseach Methodologies

In order to make a good and successful research we have to follow the scientific research process and steps which describe the methodologies that were used during our research. The used methodologies should be valid before and during the research process. Many steps or guide lines should be considered as follow:

1. Formulating the research problem; to make an efficient tool considered as a general purpose tool that will be used for solving the multi objective optimization problem. In order to avoid starting from the scratch, we will work on a previous implementation of sequential execution MOGA and enhance it. The enhancement will affect the performance and the error in results of MOGA.

The enhancement in performance will be implemented by parallelizing some parts of code and execute it locally (on the same computer) on the parallel architecture found in GPU.

2. Extensive literature review; to search and read what other researcher do, and figure out the used technology and the resulted output. We found many related work, but, up to our knowledge, there is no similer work with ours.

3. Developing the objectives; the objectives of the research should be cleared and abstracted as mentioned previously(see next section).

4. Collecting the data and select the suitable benchmarks; depending on the literature review, we will select the data and benchmarks from related problems used by other researchers in their works for experimental purpose as will show later.

5. Experiments; making many experiments on our implementation in order to test the objectives and hypothesis using the selected benchmarks.

6. Analysis of results; analyzing the results and drawing the needed graphs and tables.

7. Preparation of the report and presentation,formal write of conclusions reached.

1.5 Research Objectives

In our research, we will make a set of modifications on MOGA to make it ready to run on CUDA, we call the new platform Enhanced Multi Objective optimization using Genetic Algorithm (E-MOGA). The objectives of our research can be summarized as follow:

- Enhance the performance of the sequential execution multi objective optimization using Genetic Algorithm (MOGA) to produce the new platform that we call E-MOGA by re-designing MOGA to be ready for execution on CUDA.
- Propose a criteria for measuring the quality of the solutions and demonstrate the quality enhancement in E-MOGA over MOGA.
- Test the performance and the quality of E-MOGA, using different problems with different properties.
- Study the effect of the number of objectives on performance.

Chapter 2

Theoritecal Background

This chapter is concerned with the main theoretical background needed for the reader to understand the next chapters, the theoretical background will divide into three main categories that should cover all topics as follow: Evolutionary Algorithm, General Purpose computation on GPU and speedup.

2.1 Evolutionary Algorithm

Evolutionary Algorithms (EAs) are population-based meta-heuristic optimization algorithms that use biology-inspired mechanisms like mutation, crossover, natural selection, and survival of the fittest in order to refine a set of solution candidates iteratively [49].

In computer science and engineering, one is often faced with a problem where the possible set of solutions is exceptionally big. It is impossible to go through the

solutions set by a simple brute force algorithm, and a deterministic algorithm that would be fast enough to get the solution, or it may be a complex problem that is hard to define. So, different techniques were used to find a good enough solution in these cases, one of the best methods is EA, and one of most well known algorithm used as searching technique is the Genetic Algorithm (GA). The good solution characteristic means that we do not look for optimality and there is no need to go through all the possible solutions.

The neighborhood of a solution is the set of all available solutions that can be reached with one technique moving from the current solution. The concept of neighborhood is especially used in local search algorithms, such as hill-climbing (is a mathematical optimization technique, It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found) [28], tabu search (take a potential solution to a problem and check its immediate neighbors in the hope of finding an improved solution) [24] and simulated annealing(is a generic probabilistic met heuristic for the global optimization problem of locating a good approximation to the global optimum of a given function in a large search space. It is often used when the search space is discrete) [3]. The fitness of a solution indicates how good the solution is. In rare cases, when the optimum is known, one tries to get the fitness value as close to the optimum as possible. So, it is usually attempted to maximize or minimize a fitness function [35].

2.1.1 Genetic Algorithms

Genetic algorithms were invented by John Holland in the 1960s. Holland's original goal was not to design application specific algorithms, but to study the ways of evolution and adaptation in nature and develop ways to import them into computer science [35]. Holland's 1975 book "Adaptation in Natural and Artificial Systems" [12] presents the genetic algorithm as an abstraction of biological evolution and gives the theoretical framework of adaptation under the genetic algorithm [34].

GA Architecture

We have to clarify some biological terminology in order to make the genetic algorithm easier. All living organisms consist of cells, and every cell contains a set of chromosomes, which are strings of DNA and give the basic information of the particular organism. A chromosome can be further divided into genes, which in turn are functional blocks of DNA, each gene represents some particular property of the organism. The different possibilities for each property, is located at a particular locus of the chromosome. When reproducing, crossover occurs: genes are exchanged between the pair of parent chromosomes. The offspring is subject to mutation, where single bits of DNA are changed. The fitness of an organism is the probability that the organism will live to reproduce and carry on to the next generation [50, 49]. The set of chromosomes at hand at a given time is called a population.

In computer sciences, the genetic algorithms are applied in many fields as a searching tool to find a good solution from a very big solution set, and obviously,

the goal is to find a solution as good as possible. Since the computer problem is different from the fields where the genetic algorithm exists, we have to re-form the problem into a form of genetic, so one needs encoding of the solution, to represent the solution in a form that can be interpreted as a chromosome,such as, initial population, mutation, crossover operators, fitness function and a selection operator for choosing the survivors for the next generation [8].

The big advantage of genetic algorithms is that you do not have to specify all the details of a problem in advance. Potential solutions are evaluated by a fitness function representing the problem we want to solve [8]. Then we define an evolution procedure to produce new candidate solutions. The idea is that combining good solutions (solutions with highest fitness scale) should lead to better solutions. By adding some natural noise (mutating), we hope to find better solutions [8].

GA Pseudo-code

The GA is a popular algorithm and used in many application, the GA pseudo code can be as in Algorithm 1: The GA stops when certain criteria are met: optimal

Algorithm 1 Genetic Algorithm pseudo code

Begin
Draw a random population of n candidates. This is the first population
Generate an intermediate population of n-number of candidates by using genetic operations.
Compare and rank the $2n$ candidates by using the fitness function.
Choose the n highest scoring candidates as the new population.
Go to second step.
End

solution found, number of cycles reached, time passed or good score reached...etc.

Encoding

The most common and traditional way of encoding is to use a bit vector (string of ones and zeros) [49]. Thus every bit in the chromosome represents a gene in that position. This has the advantage of being very easy to interpret. Usually such encoding is used for combinatorial problems [8].

Another common way of forming a chromosome is to have a string of natural numbers [49]. Such solutions are good for permutation problems for example, the traveling salesman problem (TSP) [49]. The nodes in the graph are numbered and the travel route will be the order of the nodes in the chromosome.

Initial Population

As we know, GA is a type of meta-heuristic algorithms and some processes are executed on a probabilistic basis. Since the GA is a diverse algorithm (always become closer to the solution), the first generation that will start the GA operation always created randomly and no matter what are the starting values [40].

Fitness Function

In order to evaluate how good the different individuals in the population are, a fitness function needs to be defined. A fitness function assigns each chromosome a

value that indicates how well that chromosome solves the given problem. [34, 49], so the fitness function is a subject of the problem.

A common application of genetic algorithms is optimizing a function. Unfortunately, optimizing problems are rarely straightforward. In fact, genetic algorithms are usually used in an attempt to optimize complex multi variable functions or non-numerical data [49]. So, the more complex the problem is, the more complex the fitness function usually becomes. So, the developer must find the metrics or ways to find a numerical evaluation of the problem.

Selection Operator

Since the number of individuals in a population is always increasing with the result of crossovers and mutation, a selection operator is needed to manage the size of the population. The selection operator will determine the individuals who will survive to the next generation, and should be defined so the ones with the best fitness are more likely to survive in order to increase the average fitness of the population [47, 49].

The simplest way of defining a selection operator is to use a purely elitist (the best of the best) selection. This selects only the "elites", i.e., the individuals with the highest fitness. Elitist selection is easy to understand and simple to implement; one can simply discard the weakest individuals in the population. However, elitist selection is not the best choice, as it may result in getting stuck to a local optimum.

Another common way of defining the selection operator is to use a "roulette wheel" sampling [47]. Each individual is given a slice of the "wheel" in proportion

of the area that its fitness has in the overall fitness of the population. In this way, the individuals with higher fitnesses have a larger area in the wheel, and so have a higher probability of getting selected. The wheel, spun as many times as there are individuals needed for the population.

A common selection operator is a crossing of the two methods presented above; the survival of the fittest is guaranteed by choosing the best individual with elitist methods, while the rest of the population is selected with the roulette wheel in order to ensure variety within the population [47]

Crossover

The crossover operator is applied to two chromosomes; the parents to create two new chromosomes and their offspring, which combine the properties of their parents. Like mutation, the crossover operator is applied to a certain randomly selected position in the chromosome. The crossover operator will then exchange the subsequences before and after the selected genes to create the offspring [34, 49].

As an example, we can see in Figure 4.14, suppose we have two chromosomes on the left side of the arrows; the first 10101010100101001101 and 00110100000010101010, and the selected position is in position k, $k < n$, where n is the chromosome length. The offspring resulted after the crossover operation are on the right side of the arrows. It is also possible to execute a multi-point crossover [34], where the crossover operator is applied to several positions in the parent chromosomes.

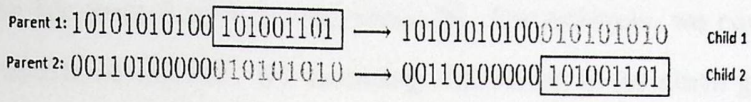


Figure 2.1: Crossover in binary vector

Like mutation, the crossover operator also has a crossover probability, which determines how likely the crossover operator applied to a chromosome, the crossover probability is related to the fitness of the chromosome. The fitter the individual is; the more likely it will survive to the next population, the bigger the chance that the offspring will also have a high fitness-value [8]. Whether the offspring will actually have a higher fitness value depends on how well the crossover-operation is defined. Unfortunately, this can not always be guaranteed. Thus, the probability of a crossover increases in some correlation with the fitness-value of the chromosome [35].

Where and how the crossover operator is used it varies based on the application and the developer. Many researchers consider that the selection operator always selects parents, and thus all chromosomes selected to the next generation are subject to the crossover operator.

Mutation operation is very important specially for local minima problem where far away solution is needed to jump from the portion where the algorithm stuck.

Mutation

Mutation are a way of creating new individuals from the population by administering a minor change to some of the existing individuals by changing a gene in a random position. When the chromosome is represented by a bit vector, a basic mutation is

to change one bit from 0 to 1 or vice versa [8]. For example, we could have a bit string 10101010100101001101. By mutating this string in its ninth gene the result would be 10101010100001001101 as seen in Figure 2.2.

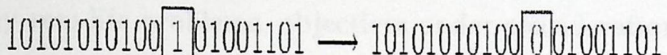


Figure 2.2: Mutation in binary vector

When the string contains natural numbers, a mutation will switch the places of two numbers, and we have to remember that the more complex the encoding of the chromosome is, the more there are possible mutation that can be applied and the mutation may become more complex.

The result (new chromosome) should always still be a legitimate individual, i.e., it should solve the defined problem. So it is possible to have a separate "mutation correction" that will check the chromosome after a mutation to see that it still solve the problem that it is supposed to. If the mutation causes the chromosome to be unnatural, i.e., it does not belong to the solution space anymore, corrective actions will take place [35].

For every mutation there is always a defined probability that the mutation in question would be applied to an individual, this is called the mutation probability or mutation rate [49]. As in nature, mutation are unwanted in most cases, thus the mutation probabilities are usually quite low.

2.1.2 MOGA

The multi-objective optimization also known as multi-criteria or multi-attribute optimization, are realistic models for many complex engineering optimization problems. In many real-life problems, objectives under consideration conflict with each other, and optimizing a particular solution relating to a single objective can result in unacceptable results with respect to the other objectives. A reasonable solution to a multi-objective problem is to investigate a set of solutions, each of which satisfies the objectives at an acceptable level without being dominated by any other solution. [13]

Another definition of multi-objective optimization is the process of simultaneously optimizing two or more conflicting objectives subject to certain constraints [18]. The GA are the most popular heuristic search technique to solve multi-objective and optimization problems [21].

Multi-objective optimization problems can be found in various fields: product and process design [41], finance, aircraft design [46], the oil and gas industry [38], automobile design [26], or wherever optimal decisions need to be taken in the presence of trade-offs between two or more conflicting objectives. Maximizing profit and minimizing the cost of a product; maximizing performance and minimizing fuel consumption of a vehicle; and minimizing weight while maximizing the strength of a particular component are examples of multi-objective optimization problems [13].

For nontrivial multi-objective problems, one cannot identify a single solution that simultaneously optimizes each objective. While searching for solutions, one

reaches points such that, when attempting to improve an objective further, other objectives suffer as a result. A tentative solution is called *non-dominated*, *Pareto optimal*, or *Pareto efficient* [25] if it cannot be eliminated from consideration by replacing it with another solution which improves an objective without worsening another one. Finding such non-dominated solutions, and quantifying the trade-offs in satisfying the different objectives, is the goal when setting up and solving a multi-objective optimization problem.

Multiobjective Optimization

When the optimization process is applied on a single objective, the result will show one optimal or sub-optimal solution by comparing the solution space and selecting the best solution. In the case of multi objective problems, the evaluation function f will take a vector of values that represent the objectives and returns a vector of results. Assuming we have $k > 1$ where k indicate number of objectives, the evaluation function f will return a vector $Y \in \Re^k$. This make the comparison between two solutions x_1 and x_2 very difficult [55].

In this case the Pareto dominance can be used to find the optimal solution that is called Pareto front.

Definition. *Pareto dominance*: an objective vector y_1 is said to dominate another objective vectors $y_2 (y_1 \succ y_2)$ if no component of y_1 is smaller than the corresponding component of y_2 and at least one component is greater [55].

Accordingly, we can say that a solution x_1 is better to another solution x_2, i.e.,

x_1 dominates x_2, $(x_1 \succ x_2)$, if $f(x_1)$ dominates $f(x_2)$. Here, optimal solutions, i.e., solutions that are not dominated by any other solution, may be mapped to different objective vectors. In other words: several optimal objective vectors may exist to representing different trade-offs between the objectives [55].

Assume that we have k objectives, all to be maximized without priority. The solution of this problem can be described in terms of a decision vector $(x_1, x_2, ..., x_n)$ in the decision space $\mathbf{X}$. A function $f : (X \rightarrow Y)$, evaluates the quality of a specific solution by assigning it to an objective vector $(y_1, y_2, ..., y_k)$ in the objective space $\mathbf{Y}$ [31, 55](see Figure 2.3 [31]).

The set of optimal solutions in the decision space $\mathbf{X}$ is in general denoted as the Pareto set $\mathbf{X}^* \subseteq \mathbf{X}$, and we denote its image in objective space as Pareto front $\mathbf{Y}^* = f(\mathbf{X}^*) \subseteq \mathbf{Y}$.

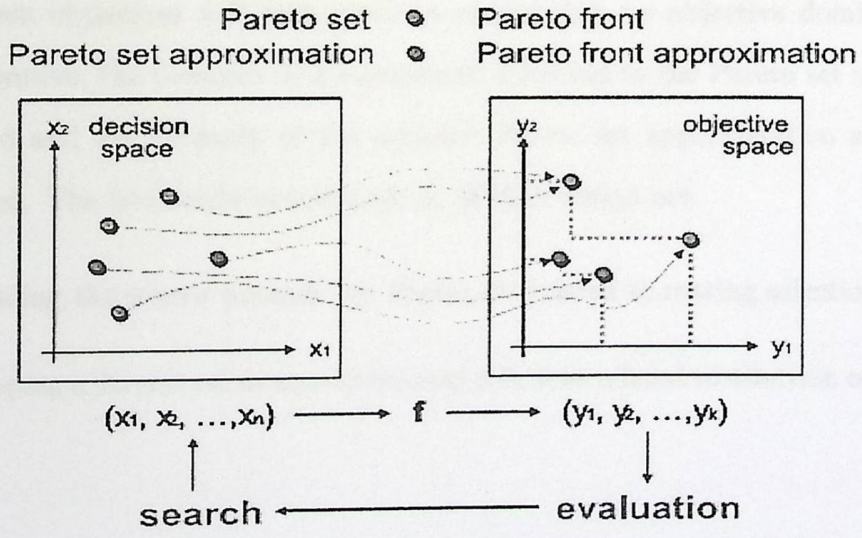


Figure 2.3: Multi-objective optimization problem [31]

Multi Objectives and Evolutionary Algorithm

Generating the Pareto set by exploring all solutions can be computationally expensive and is often infeasible due to time and resources, because the complexity of the underlying application prevents exact methods from being applicable. For this reason, a number of stochastic search strategies such as evolutionary algorithms, tabu search [24], simulated annealing [3] and ant colony optimization [29] have been developed, these method do not usually guaranteed to identify optimal trade-offs but to find a good approximation, i.e., a set of solutions whose objective vectors are (hopefully) not too far away from the optimal objective vectors [31].

MOGA Design Issues

Since the MOGA generates many candidate solutions in each iteration and a trade off between objectives will take place to ensure that no objective dominates an other objective, the distance of the generated solutions to the Pareto set should be minimized and the diversity of the achieved Pareto set approximation should be maximized. The two fundamental goals in MOGA design are:

1. Guiding the search towards the Pareto set related to mating selection.

2. Keeping a diverse set of non-dominated solutions related to selection operation.

Diversity Preservation

Most MOEAs try to maintain diversity within the current Pareto set approximation by incorporating density information into the selection process [31]: An individuals chance of being selected is decreased the greater the density of individuals in its neighborhood. The methods used in MOEAs can be classified into kernel, nearest neighbor and histogram, Each of the three approaches is visualized in Figure 2.4 [55].

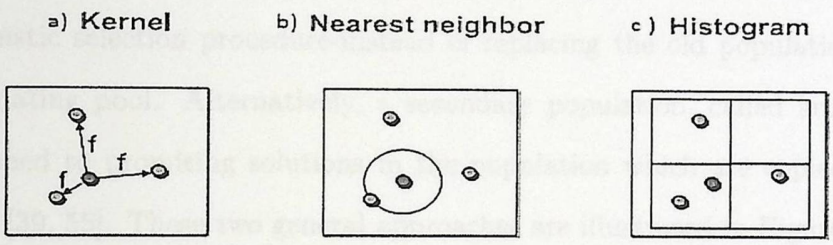


Figure 2.4: Diversity preservation techniques [55]

Kernel methods define the neighborhood of a point in terms of a Kernel function $\boldsymbol{K}$ which takes the distance to another point as an argument; for each individual the distances *di* to all other individuals i are calculated and after applying $\boldsymbol{K}$ the resulting values $\boldsymbol{K}$(di) are summed up. The sum of the $\boldsymbol{K}$ function values represents the density estimate for the individual. This method is popular in MOGA, and in the implementation it is called crowding distance and it is shown in Figure 2.4.a.

Nearest neighbor techniques take the distance of a given point to its k^{th} nearest neighbor in order to estimate the density in its neighborhood and it is shown in Figure 2.4.b.

Histograms use a hyper grid to define neighborhoods within the space. The density around an individual is simply estimated by the number of individuals in the same box of the grid and it is shown in Figure 2.4.c.

Elitism

Elitism is the way to solve the problem of losing good solutions during the optimization process due to random effects. The first method is to combine the old population with the offspring, i.e., the mating pool after variation, and to apply a deterministic selection procedure-instead of replacing the old population by the modified mating pool. Alternatively, a secondary population, called archive, can be maintained to promising solutions in the population which are copied at each generation [39, 55]. These two general approaches are illustrated in Figure 2.5 [55].

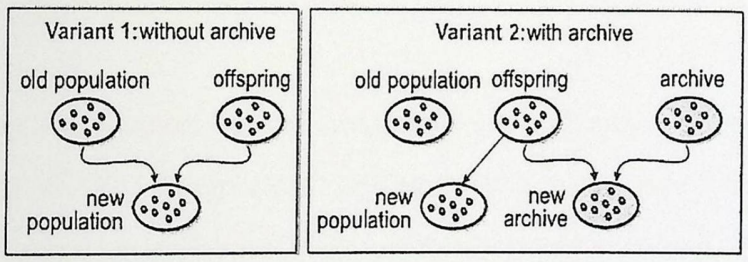


Figure 2.5: Two possible ways to implement elitism [55]

Fitness assignment

Pareto ranking is the technique that used for fitness assignment in MOGA. Pareto-ranking approaches explicitly utilize the concept of Pareto dominance in evaluating

fitness or assigning selection probability to solutions. The population is ranked according to a dominance rule, and then each solution is assigned a fitness value based on its rank in the population, not its actual objective function value [25]. The whole operation shown in the following pseudocode [25]:

Algorithm 2 Pareto Ranking

Begin
Set $i = 1$ and $TP = P$.
Identify non-dominated solutions in TP and assigned them set to F_i.
Set $TP = TP * F_i$. If $TP = \phi$ go to Step 4, else set $i = i + 1$ and go to Step 2.
For every solution $x \in P$ at generation t, assign rank $r_1(x, t) = i$ if $x \in F_i$.
End

MOGA Pseudo Code

MOGA is one of the hardest algorithm to implement and it's contains many steps that could be summarized referring to our implementation as in Algorithm 3.

Algorithm 3 MOGA psuedo code

Begin
STEP 1: Random initial population.
STEP 2: Perform GA operations.
STEP 3: Fitness assignments (Pareto Rank).
STEP 4: Diversity (Kernel).
STEP 5: Elitism (Archive).
STEP 6: Selection
STEP 7: If stopping criteria met then go to step 8, else go to step 2.
STEP 8: Return Pareto Front.
End

2.2 GPGPU

GPGPU stands for General-Purpose computation on Graphics Processing Units, and also known as GPU Computing. And as a result of the insatiable market demand for real-time, high-definition 3D graphics, the programmable Graphic Processor Unit (GPU) has evolved into highly parallel, multi threaded, high-performance many-core processors capable of very high computation and data throughput. It is specially designed for computer graphics and it is difficult to program, as illustrated by Figure 2.6 [17], this figure shows that,the GPU performance is much better than CPU measured by the number of floating point operation per second, the G92 GPU which is present on famous 9800 GTX graphics card can give a massive performance

when compared to a Core 2 Duo processor. The memory bandwidth for GPUs is also higher than CPUs as shown in Figure 2.7 [17] which is measured by the Giga bytes transfer per second. todays GPUs are general-purpose parallel processors with support for accessible programming interfaces and industry-standard languages such as C [17].

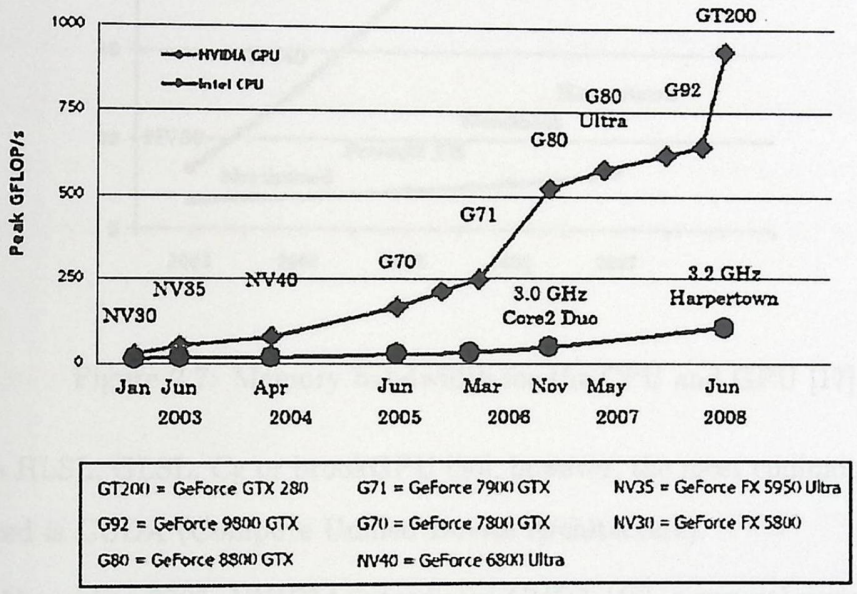


Figure 2.6: Floating-point operations per second for the CPU and GPU [17]

2.2.1 CUDA

In order to get the benefit from the power of GPU for general purpose computation, many companies developed an interfacing language to access the complex structure of the GPU card, and resulted in many programming interface and IDEs for GPU

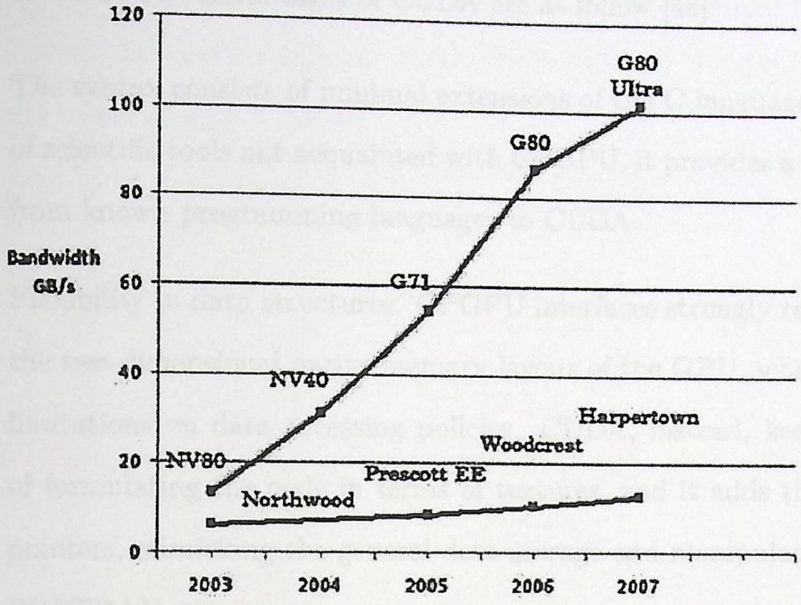


Figure 2.7: Memory bandwidth for the CPU and GPU [17]

such as HLSL, GLSL, Cg or brookGPU [30], however, the most commonly used and advanced is CUDA (Compute Unified Device Architecture).

In November 2006, NVIDIA introduced CUDA [48], a general purpose parallel computing architecture, with a new parallel programming model and instruction set architecture, which uses the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C as a high-level programming language [17].

However CUDA works only within environments that contain NVIDIA graphic cards [16]. By programming in CUDA the operations can be addressed to either

GPU or CPU. The advantages of CUDA are as follow [48]:

1. The syntax consists of minimal extensions of the C-language. For a developer of scientific tools not acquainted with GPGPU, it provides a smooth transition from known programming languages to CUDA.

2. Flexibility in data structures. GPGPU interfaces strongly rely on the texture, the two-dimensional native memory layout of the GPU, which imposes severe limitations on data accessing policies. CUDA, instead, keeps the possibility of formulating the code in terms of textures, and it adds the option of using pointers, mimicking the general data storage and manipulation schemes of C, FORTRAN, etc.

3. Latency times for data transfer between the CPU and the GPU have been dramatically improved.

4. Explicit access on the different physical memory levels of the GPU is allowed. This enables the fine-tuning of the codes data patterns to optimize the performance.

5. A complete, well documented development platform, including a compiler, scientific libraries, debugger and profiler utilities is provided.

The new concept that programmers are familiar with C needs to know is: the thread. This concept takes a place in the GPU as a SIMD (Single Instruction, Multiple Data) technique of vector computers, SIMD Architecture are shown in

Figure 2.8 [17]. A thread is simply a sequence of instructions (functions in C) that can be executed on different data units in parallel; these threads are explicitly grouped and managed in blocks.

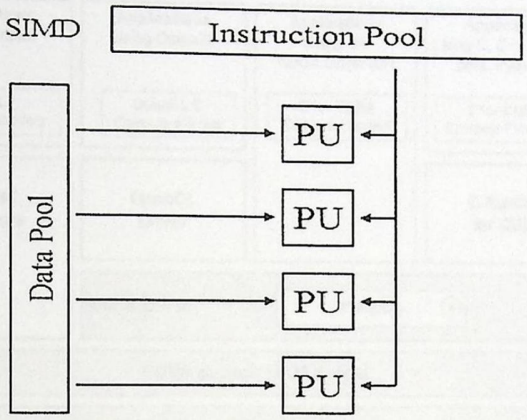


Figure 2.8: SIMD Architecture [17]

Threads in a block (up to a hardware delimited maximum of 512 threads) are processed by the same processing unit in the GPU and share a small amount of 16 kb fast on- chip memory [17].

2.2.2 The CUDA Architecture

The CUDA-enabled GPU (so-called device) is exposed to the CPU (so-called host) as a co-processor. This means that each GPU is considered to have its own memory and processing elements that are separate from the host computer [17]. To perform useful work, data must be transferred between the memory space of the host computer and CUDA device(s). For this reason, performance results must include input and output

(I/O) time to be informative.The CUDA architecture could be cleared in Figure 2.9 [15].

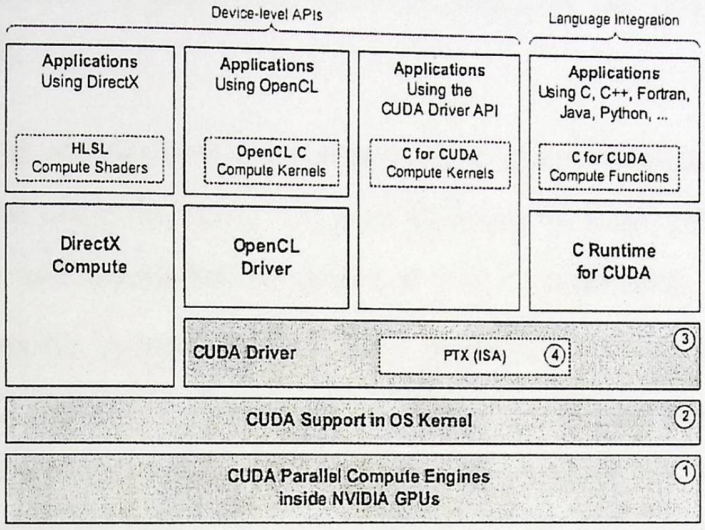


Figure 2.9: CUDA Architecture [17]

2.2.3 CUDA Development Environment

The CUDA software development environment supports two different programming interfaces:

1. A device-level programming interface, where the application uses DirectX Compute, OpenCL or the CUDA Driver API directly to configure the GPU, launch compute kernels (functions that will be executed on GPU), and read back results [15].

2. A language integration programming interface, where an application uses the C Runtime for CUDA and developers use a small set of extensions to indicate which compute functions should be performed on the GPU instead of the CPU [15].

When using the device-level programming interface, developers write kernels in separate files using the kernel language supported by their API of choice. The CUDA Driver API accepts kernels written in C or PTX assembly.

When using the language integration programming interface, developers write functions in C and the C Runtime for CUDA automatically handles setting up the GPU and executing the functions [16]. This programming interface enables developers to take advantage of native support for high-level languages such as C, C++, Fortran, Java, Python, and more, reducing code complexity and development costs [16].

The performance of applications developed on CUDA is varied and depends on programmers to get better performance. The programmers have to keep thousands of threads busy. The current generation of NVIDIA GPUs can efficiently support a very large number of threads, and as a result they can deliver one to two orders of magnitude performance increase in application performance [17].

2.2.4 CUDA Programming Structure

C for CUDA extends C that allows the programmer to define C functions, in parallel format that is called Kernel, and when called, is executed N times in parallel by N

different CUDA threads. A kernel is defined using the __global__ declaration specifier and the number of CUDA threads for each call is specified using a new <<< ... >>> syntax.

Each of the threads that execute a kernel is given a unique thread ID that is accessible within the kernel through the built-in thread Idx variable. For example, the following simple code shown in Figure 2.10 adds two vectors A and B of size N and stores the result into vector C.

```
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Kernel invocation
    vecAdd<<<1, N>>>(A, B, C);
}
```

Figure 2.10: CUDA example

2.2.5 Kernel

A kernel is a function callable from the host and executed on the CUDA device simultaneously by many threads in parallel. In fact CUDA executes a function in the Single Program Multiple Data (SPMD) model, which means that a user-configured number of threads run the same program on different data. Each thread will execute the same kernel function and will operate on a single data element. Each thread is distinguished from all the others by block ID (BID) and thread ID (TID) indicates that it can be used to determine the data element the thread will

access. CUDA organizes a parallel computation using the abstractions of threads, blocks and grids [17, 16], and the simple definitions are as follows:

- Thread is an execution of a kernel with a given index. Each thread uses its index to access data elements [52].
- Block is a group of threads. Threads within a block can execute concurrently or serially and in no particular order. They can be coordinated using the synchronization function that makes a thread stop at a certain point in the kernel until all the other threads in its block reach the same point [52].
- Grid is a group of blocks. There is no synchronization at all between the blocks [52].

These multiple blocks are organized into a one-dimensional or two-dimensional grid of thread blocks [17] as illustrated in Figure 2.11 [17]. On the other hand, the computation of threads, blocks and grids are distributed as follows:

The execution on GPU can be summarized as follow:

1. Delivering the code to the processor [52]:
   - Grid $\longrightarrow$ GPU: An entire grid is handled by a single GPU chip.
   - Block $\longrightarrow$ MP: The GPU chip is organized as a collection of multiprocessors (MPs), with each multiprocessor responsible for handling one or more blocks in a grid. A block is never divided across multiple MPs.

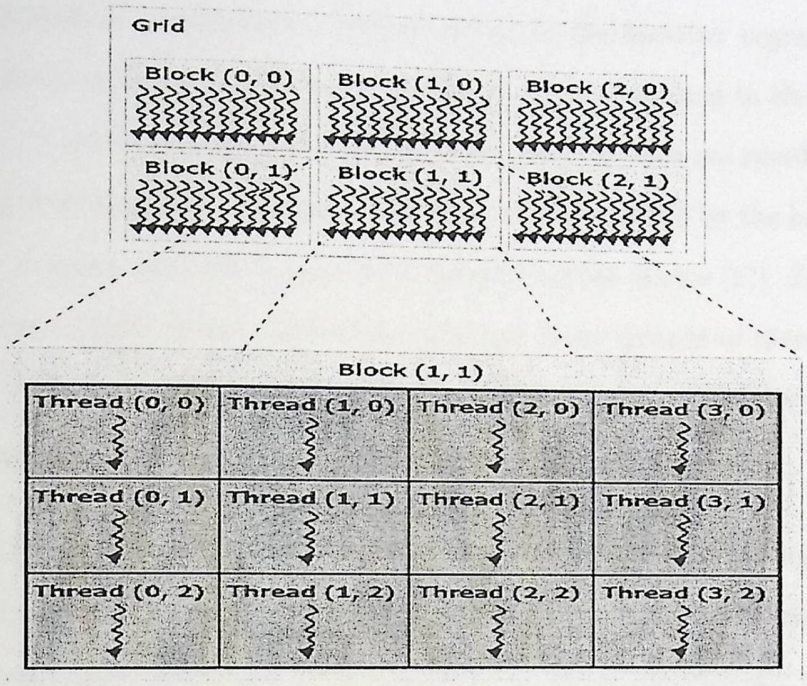


Figure 2.11: Grid of threads Blocks [17]

- Thread $\longrightarrow$ SP: Each MP is further divided into a number of stream processors (SPs), with each SP handling one or more threads in a block.

2. Executing the code From the host point of view, kernel invocations are asynchronous function calls. Synchronization is done explicitly by calling a synchronization function, or implicitly when the host tries to access memory on the device. In both cases, synchronization takes the form of a barrier that blocks the calling host thread until all previously called kernels have been finished [1].

   When the CUDA device is idle, the kernel immediately starts running based

on the execution configuration and according to the function arguments. At this moment and after the kernel launched, the host continues to the next line of code. So we can say that the CUDA device and the host are simultaneously running their separate programs. If another kernel is called by the host immediately, it waits until all threads have finished on the device [17]. Each block is split into SIMD (Single Instruction Multiple Data) groups of threads called warps. Each warp contains the same number of threads, called the warp size, which are executed by the multiprocessor in a SIMD fashion.

A programming example to show the different styles of programming [7] could be found in Figure 2.12, it shows some basic features of straightforward implementations (sequential) and parallel programming with CUDA for computing $y = a*x+y$. in both sequential and parallel [7]. Given vectors x and y containing n floating point numbers, it performs the update $y = a*x+y$. The serial implementation is a simple loop that computes one element of y in each iteration. The parallel kernel effectively executes each of these independent iterations in parallel, assigning a separate thread to compute each element of y [7].

The *__global__* modifier indicates that the procedure is a CUDA kernel, and the extended function call syntax *saxpy* $<<< B; T >>>$ (...) is used to launch the kernel *saxpy*( ) in parallel across B blocks of T threads each. Each thread of the kernel determines which element it should process from its integer thread block index (*blockIdx:x*), its index within its block (*threadIdx:x*), and the total number of threads per block (*blockDim:x*). The other parameters between parenthesis are

considered to be an extra parameter and we can ignore it [17].

```
void saxpy(uint n, float a,
           float *x, float *y)
{
   for (uint i = 0; i<n; ++i)
      y[i] = a*x[i] + y[i];
}

void serial_sample ()
{
   // Call serial SAXPY function
   saxpy (n, 2.0, x,y);
}
```

(a)

```
__global__ void saxpy(uint n, float a,
                      float *x, float *y)
{
   uint i = blockIdx.x*blockDim.x
          + threadIdx.x;

   if( i<n ) y[i] = a*x[i] + y[i];
}

void parallel_sample()
{
   // Launch parallel SAXPY kernel
   // using [n/256] blocks of 256
   // threads each
   saxpy<<<ceil(n/256),256>>>(n, 2, x, y);
}
```

(b)

Figure 2.12: Serial (a) and Parallel (b) kernels for computing $y = a * x + y$

2.2.6 Memory Hierarchy

As mentioned before, the CUDA-enabled GPU has its own memory, and the CUDA-enabled GPU memory has a different hierarchy from the usually computer have. The CUDA-enabled GPU memory hierarchy is described in Figure 2.13 [17] and summarized in the following points [17, 16]:

- Global memory: This memory is built from a bank of SDRAM chips connected to the GPU chip. Any thread in any MP can read or write to any location in the global memory. Sometimes it is called device memory. Potentially 150x slower than register or shared memory.

- Texture cache: It is a memory within each MP that can be filled with data from the global memory so it acts like a cache. Threads running in the MP are restricted to read-only access of this memory.
- Constant cache: It is a read-only memory within each MP.
- Shared memory: It is a small memory within each MP that can be read/written by any thread in a block assigned to that MP, and can be as fast as a register when there are no bank conflicts or when reading from the same address.
- Registers: Each MP has a number of registers that are shared between its SPs, and it is the fastest form of memory on the multi-processor.
- Local memory: It implies "local in the scope of each thread". It is a memory abstraction, not an actual hardware component of the multi-processor. In actuality, local memory gets allocated in global memory by the compiler and delivers the same performance as any other global memory region. Local memory is basically used by the compiler to keep anything the programmer considers local to the thread but does not fit in faster memory for some reason.

2.2.7 Programming in CUDA

With the CUDA architecture and tools, developers are achieving speedups in many fields such as medical imaging, natural resource exploration, and cryptography. One of the major benefits of CUDA as compared to other GPU programming systems is its use of a C dialect, such that, original C function for the CPU can often be

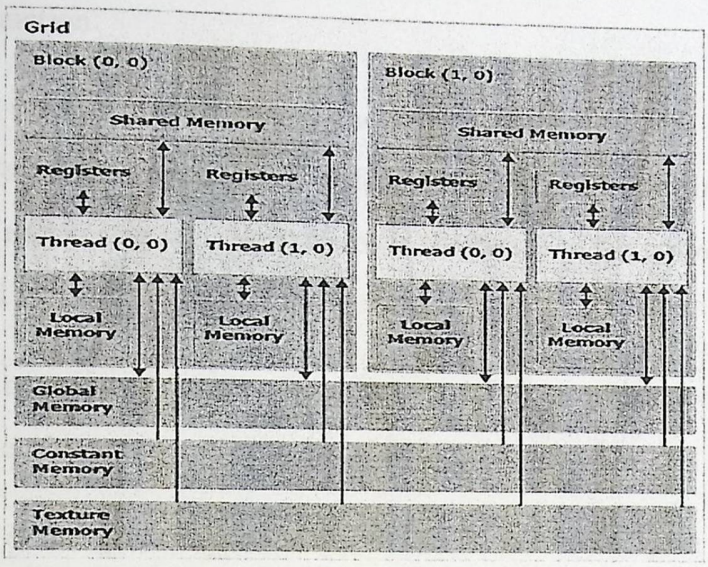


Figure 2.13: CUDA memory [17]

transformed into a CUDA kernel with only slight modifications. CUDA provides developers with C libraries that expose all device functionalities needed to integrate CUDA into a C program. Furthermore, CUDA enables this unprecedented performance via standard APIs as OpenCL, DirectX Compute, and high level programming languages such as C/C++, Fortran, Java, Python, and the Microsoft.NET Framework.

In order to write a CUDA program, the programmer, normally begins from a sequential version and proceeds through the following steps [11]:

1. Identify a kernel, and package it as a separate function.

2. Specify the grid of GPU threads that execute it, and partition the kernel computation among these threads, by using blockIdx and threadIdx inside the

kernel function.

3. Manage data transfer between the host memory and the GPU memories (global, constant and texture), before and after the kernel invocation. This includes redirecting variable accesses in the kernel to the corresponding copies allocated in the GPU memories.

4. Perform memory optimizations in the kernel, such as utilizing the shared memory and coalescing accesses to the global memory.

5. Perform other optimizations in the kernel in order to achieve an optimal balance between single-thread performance and the level of parallelism.

In addition a CUDA program may include multiple kernels, thus the above mentioned procedure needs to be applied to each of them.

2.3 Speedup

Make the common case faster; dont spend time on improving a piece of code that is only used once. It is not here you will gain any great performance improvements, on the other hand if a loop is being used a 100 times or a 1000 times in the program, then we improve that piece of code to gain a optimal amount of performance. Two reasons for making the common case fast

- It helps performance.

- It is simpler and faster to do.

Amdahls law can be used for this principle. Amdahl's law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used. Also known as speedup, which can be defined as the maximum expected improvement to an overall system when only part of the system, is improved [10].

$$S_p = \frac{Executiontimefortaskwithoutenhancement}{Executiontimefortaskwithenhancement}. \tag{2.1}$$

There exist many cases with different speedup equations to calculate the speedup. In our cases, only a few parts will be executed on parallel so we will use the simple formula of speedup to apply it. Speedup is defined by the following formula:

$$S_p = \frac{T_{CPU}}{T_{GPU}}. \tag{2.2}$$

where:

T_{CPU} is the execution time of the sequential algorithm.

T_{GPU} is the execution time of the algorithm runs on CPU and GPU (sequantial and parallel).

- It is simpler and faster to do.

Amdahls law can be used for this principle. Amdahl's law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used. Also known as speedup, which can be defined as the maximum expected improvement to an overall system when only part of the system, is improved [10].

$$S_p = \frac{Executiontimefortaskwithoutenhancement}{Executiontimefortaskwithenhancement}. \tag{2.1}$$

There exist many cases with different speedup equations to calculate the speedup. In our cases, only a few parts will be executed on parallel so we will use the simple formula of speedup to apply it. Speedup is defined by the following formula:

$$S_p = \frac{T_{CPU}}{T_{GPU}}. \tag{2.2}$$

where:

T_{CPU} is the execution time of the sequential algorithm.

T_{GPU} is the execution time of the algorithm runs on CPU and GPU (sequantial and parallel).

Chapter 3

E-MOGA

In this chapter, implementation details of the proposed platform to solve the Multi Objective optimization using Genetic Algorithm running on CUDA are described. The chapter is organized as follows: Section 3.1 describes the used methodologies during the implementation process. Section 3.2 highlights the used tools and software development kits. Section 3.3 overviews the structure of E-MOGA platform. Section 3.4 overviews the implementation works. Section 3.5 describes the analysis work to guarantee that we achieved and solved the main objectives. Section 3.6 highlights the benchmarks that used to validate the results.

The outcomes of this chapter are: creating a new design for MOGA architecture that depends on the given criteria to make MOGA ready to be executed on CUDA. Proposing different techniques to enhance the performance of MOGA, developing the new design by using MATLAB IDE and proposing a way to enhance the quality of the result.

3.1 E-MOGA Design

To realize the objective of our research, we propose the structure shown in Figure 3.1, the design shows the relation between system components as layers. In this figure, E-MOGA will split the execution between CPU and GPU. MATLAB will host the E-MOGA execution. Since MATLAB is not designed to be executed on GPU, additional layers were used to make the MATLAB be able to execute on GPU. The additional layer consists of two SDKs, C for CUDA and GPUmat. GPUmat is a software development kite that allows standard MATLAB code to run on GPU.

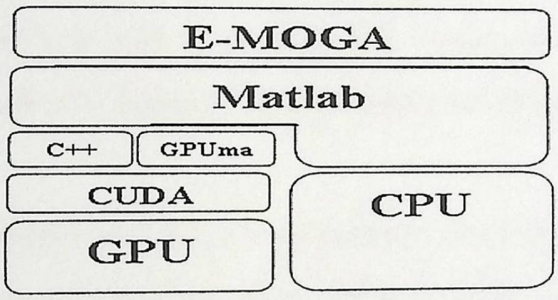


Figure 3.1: System structure

3.2 Tools and SDK

As shown if Figure 3.1, many tools were used in our implementation to achieved the proposed design; all used tools are considered as open source software development kit. They are summarized as follow:

3.2.3 GPUmat

GPUmat is a framework that enables MATLAB code to run on the Graphical Processing Unit (GPU) directly. The following is a summary of GPUmat most important features [9]:

- GPU computational power can be easily be accessed from MATLAB without any GPU knowledge.
- GPUmat speeds up MATLAB functions by using the GPU multi-processor architecture.
- GPU operations can be easily recorded into new functions using the GPUmat compiler.

3.3 E-MOGA Platform

The designed platform can work on any computing environment containing the CUDA enabled device and a pre-installed software as explained previously.

As stated previously, CUDA is NVIDIA parallel computing architecture, but it differs from the other parallel platform because it runs on GPU that was developed as SIMD (see Chapter 2). This architecture is designed to execute an arithmetic or logic operation on huge amount of data. On the other hand, this architecture is not designed for looping, branching or if-statement instructions.

Since the Multi-Objective Genetic Algorithm is designed and based on many loops, branching, sorting and computation; the implementation phases will split the program into two parts of sub-programs:

1. Sub-program that runs on GPU; high computation without the looping or branching style.

2. Sub-program that runs on CPU; the main subprogram, all the branching, looping and conditional (the main part).

The implementation was started by preparing, analyzing, enhancing and testing for MOGA to get the E-MOGA. Analyzing MOGA is a very important step, in order to figure out how to split it into many parts as stated before, a lot of modifications on these parts were done.

To convert MOGA into E-MOGA, many modification should take place. The modifications that related to performance enhancement can be summarized as follow:

1. Transferring the huge mathematical computations to the GPU instead of CPU.

2. Exploring the loops as vectors; parallelizing loops.

3. Mixed implementation; C/C++ and MATLAB script.

4. Converting from MATLAB script to compiled files.

ification steps will be explained in details in the next section.

The modifications that summarized in the previous points, aim to increase the performance of the system; first, it transfer the huge computation after parallelizing its operation to be executed on GPU instead of CPU. Second, to parallelizing the loops and the conditional statements by replacing it with parallel exploration. Third, using a mixed implementation between different languages to get integrity between performance and ease of use. Finally, converting the MATLAB script to a compiled file; this will reduce the execution time.

The parallelizations almost affect the whole components in MOGA implementation, Most of the parts were partially parallelized, and a few of them were fully parallelized. Since the MOGA applied single operation on the whole population each time, so, it can easily converted to become in Single Instruction Multiple Data (SIMD) form. The psuedo code of parallelized MOGA is shown in Algorithm 4:

Algorithm 4 MOGA psuedo code

Begin
STEP 1: Random initial population.
Start parallelization (perform any operation on the whole population in parallel form)
STEP 2: Perform GA operations.
STEP 3: Fitness assignments (Pareto Rank).
STEP 4: Diversity (Kernel).
STEP 5: Elitism (Archive).
Stop parallelization
STEP 6: Selection
STEP 7: If stopping criteria met then go to step 8, else go to step 2.
STEP 8: Return Pareto Front.
End

3.4 Implementation Phases

We choose to work on a previous implemented version of MOGA, which is available and commonly used by researchers and developers. Many implementations founds for the multi objective genetic algorithm as an open source or closed source. After reading many researches and implementation we found that the most popular and common used MOGA tools are: NSGA-II, evMOGA and gamultiobj.

A well-coded sequential version found in MATLAB optimization toolbox, that used as graphical user interface for gamultiobj, makes a good challenge to start from, the selected MOGA can be found in Optimization Tool (optimtool) in MATLAB, as shown in Figure 3.2, the tool implements all the configurations and properties that can be found in MOGA techniques.

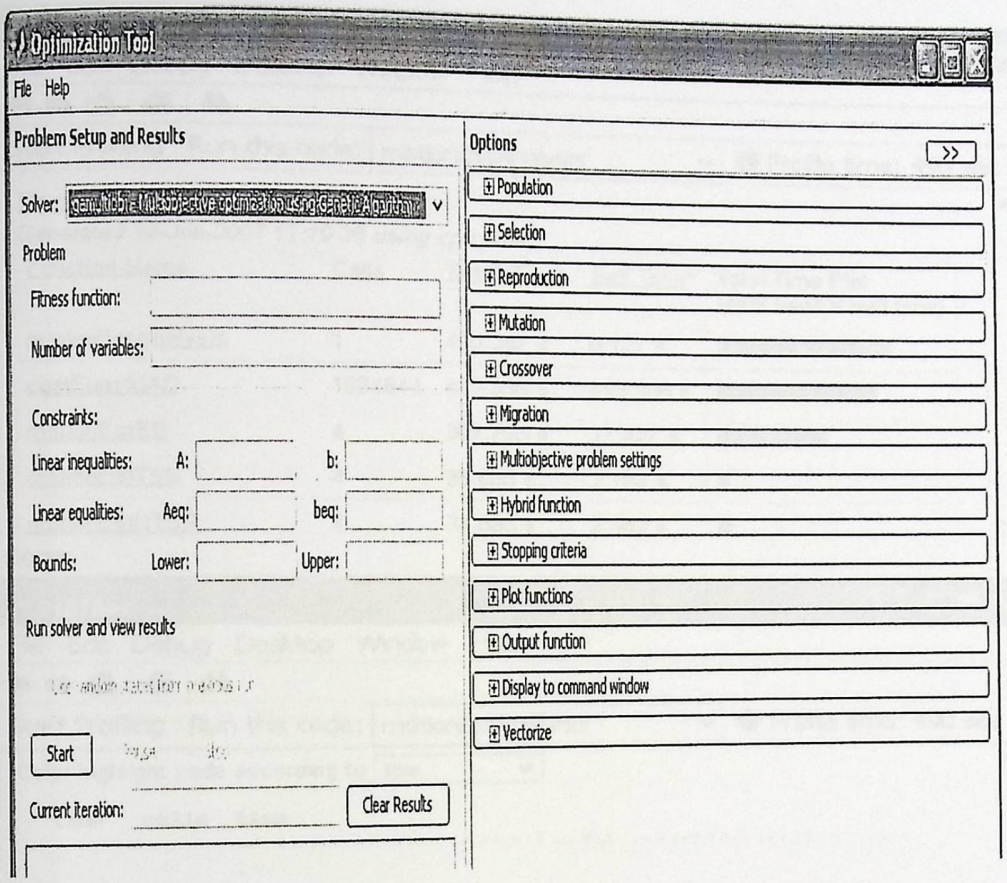


Figure 3.2: Optimization tool interface

3.5 Analysis

The first and most important step is to analyze MOGA and find out how and where to parallelize. A MATLAB tool, called Profiler was used in analyzing MOGA, it helps to improve the performance of MOGA. When running a MATLAB statement or an M-file in the Profiler, it produces a report of where the time is being spent as in Figure 3.3.

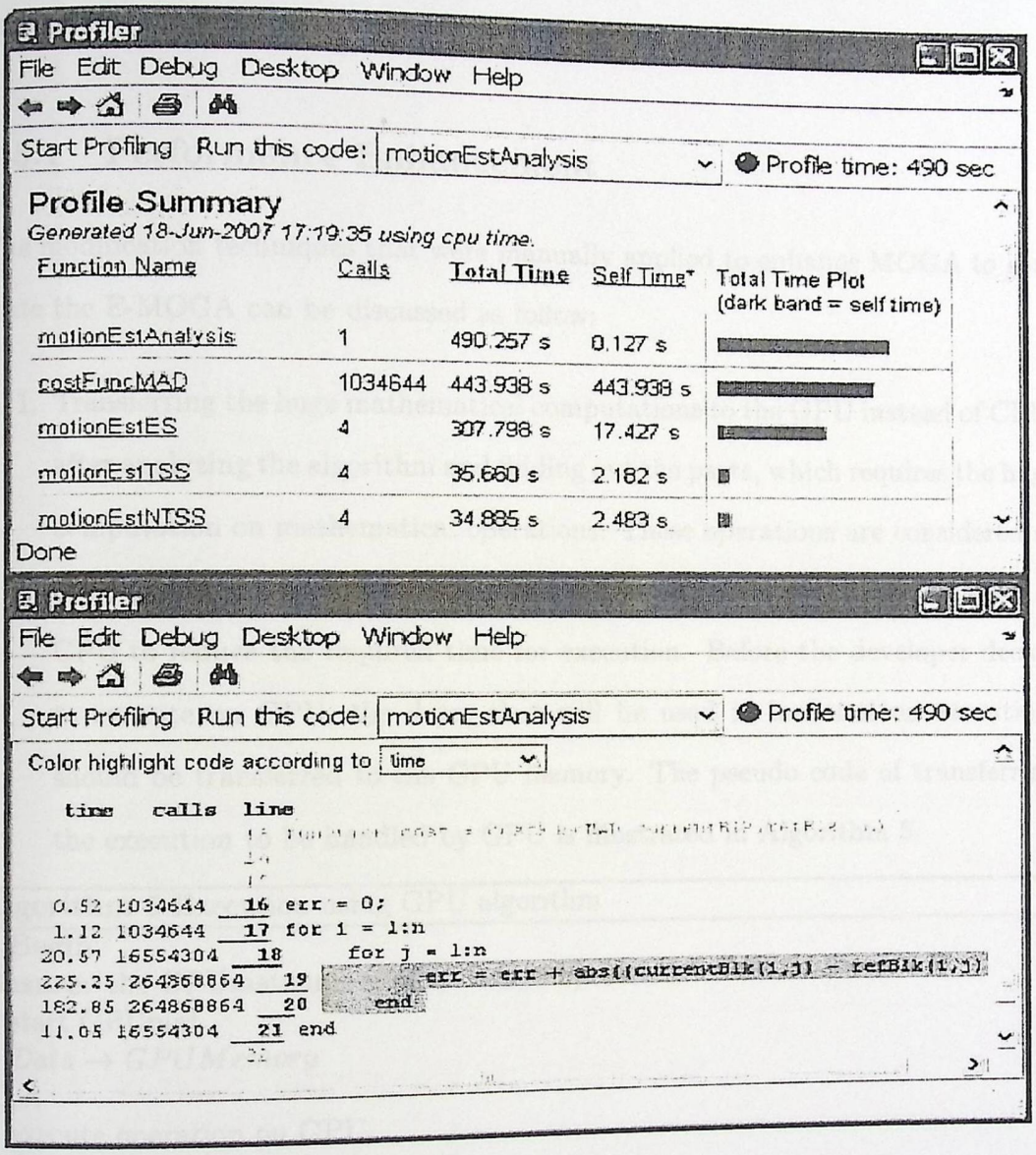


Figure 3.3: Profile Tool

In our case we analyze each function used in MOGA. After the analysis, we know which parts we should work on. These parts are considered to be the most time consuming in MOGA, and are located in many functions and subroutines of

MOGA.

3.5.1 Performance Enhancement

The modification techniques that were manually applied to enhance MOGA to generate the E-MOGA can be discussed as follow:

1. Transferring the huge mathematical computations to the GPU instead of CPU; after analyzing the algorithm and finding out the parts, which requires the huge computation on mathematical operations. These operations are considered as time consuming parts. So, these parts will be executed by the GPU instead of CPU to reduce the required time for execution. Before the developer decide to execute on GPU, the data, that will be used in instructions execution, should be transferred to the GPU memory. The pseudo code of transferring the execution to be handled by GPU is illustrated in Algorithm 5.

Algorithm 5 Execution using GPU algorithm

Begin
Ensure: the GPUmat support the desired operation
start GPUmat
$Data \rightarrow GPUMemory$
...
execute operation on GPU
...
$result \leftarrow HostMemory$
...
Continue execution sequentially on CPU
End

An example on how to execute on GPU using MATLAB and GPUmat is

illustrated here:

```
GPUstart;
Ah = single(rand(100,100));    % Ah in on CPU memory
A = GPUsingle(Ah);    % A is on GPU memory
C = exp(A);    % exp(A) performed on GPU
Ch = single(C);  % convert C (GPU) to Ch (CPU)
```

2. Exploring the loops as vectors; the GPU as discussed before, is not well optimized to work with loops and branching, so the loops and branching in the parts executed on the GPU should be converted to the form that the GPU can handle without slowing the system. By converting the loops into vectors of operations and converting the branching to a simple mathematical operations, they can be easily handled by the GPU. The pseudo code for vectorizing the loop is shown in Algorithm 6

Algorithm 6 Loop vectorization process

```
  Begin
Ensure: the loop contain a possible vectored operation
  select loop to vectorized
  for each instruction in the loop do
    if the instruction can be vectorized then
      convert it to a vector operation
    end if
Ensure:    validate the loop execution
  end for
  End
```

An example for vectorizing a loop as shown next, the first is the sequential loop:

```
FOR{i=0 to 10}
 {
 Data1[i]= Data[i] * i
 Data2[i]=Data1[i] * 5
 }
ENDFOR
```

Will be converted to become in the following form:

```
i=(1:10)
Data1= Data * i
Data2=Data1 * 5
```

3. Mixed implementation; the different programming languages differ in specification, the easiest programming language may not be able to deliver the best performance, the MATLAB is the easiest development tool that enables the programmer to develop, modify, test and implement a complex algorithm very fast, but the MATLAB performance is not the best. Using the scientific programming languages such as C/C++ can deliver a better performance. So, hybrid systems between the MATLAB and C/C++ are generated by developing some functions on C/C++ and calling it from MATLAB.

By default MATLAB can executes it is code on CPU and can not transfer the execution to GPU. The GPUmat which acts as an intermediary between MATLAB and CUDA, is used to provide the developer with the ability to select where the instructions will be executed. The GPUmat is a limited implementation and can not map all the instructions to be executed on GPU.

This option is related to the first point, since the GPUmat can not map all desired operation, the developer can write his own operation and execute it directly on GPU using C. This step can be divided into two operations as follow:

(a) Developing the code in C and compiling it using MATLAB as in the following Algorithm 7:

Algorithm 7 developing the code in C and compiling it using MATLAB

Begin
write the operation in C
select the desired compiler from MATLAB (such as VC++)
compile the source code
generate the .MEX (executable) file
End

(b) Call the operation from MATLAB as shown in Algorithm 8

4. Converting from MATLAB script to compiled files; the MATLAB programs usually written by using the script language that are interpreted by MATLAB. Since the performance of the compiled files is much better than the interpreter, the final version of the tool contains many compiled functions.

Algorithm 8 call .MEX file form MATLAB

Begin
start GPUmat
$Data \rightarrow GPUMemory$
call the MEX and send the parameters
$result \leftarrow HostMemory$
End

These four options may partially be implemented on each sub-program. After implementing each option, a test on the system should take place to ensure that a good change was made. Then the changes integrated to make the new sub-program.

After each modification, the enhanced system should be examined to ensure that there is no side effect (slowing down the system) on both the performance and the result quality, in this respect, mathematical benchmarks which is considered as real life problems were used.

3.5.2 Quality Enhancement

MOGA always returns a Pareto front (solution set) with constant dimensionality equal to the number of individuals in the population (population size) on each iteration. The algorithm will decide which solutions -considered as non promising solutions- to be replaced with a better solutions (see Chapter 2).

To improve the quality of solution, we use the idea in the GA which states: if you explore more solutions you can find a better one, unless you reach the optimal one. The tool (E-MOGA) will explore more solutions than MOGA by eliminating more none promising solution from Pareto front on each iteration.

What to change? as mentioned previously, Any individuals chance of being selected is decreased the greater the density of individuals in its neighborhood, so, we modified the selection probability value by ignoring the crowded solutions from the getting value and by this we increase the eliminated solutions from population and we have to explore more solutions. The fitness assignment algorithm will be modified than Algorithm 2 to become in the following form as in Algorithm 9:

Algorithm 9 Pareto Ranking-modified

Begin
STEP1: Set $i = 1$ and $TP = P$.
STEP2: Identify non-dominated solutions in TP and assigned them set to F_i.
STEP3: Calculate the crowding distance C_d for all solutions in P
STEP4: If $C_d > Threshold$ set $i = i + 1$ and go to Step 2, else Set $TP = TP * F_i$.
If $TP = \phi$ go to Step 4, else set $i = i + 1$ and go to Step 2.
STEP5: For every solution $x \in P$ at generation t, assign rank $r_1(x,t) = i$ if $x \in F_i$.
End

So, we will explore more solutions each time by increasing the number of eliminated solutions from Pareto front. An easy way to do it by modifying the threshold in Algorithm 9 each time and we the following formula to calculate the Threshold by selecting a partial value from summation of crowded distance each time. The value of $z = 3$ is selected by examination.

$$Threshold = \sum_{i=1}^{n} C_{di} - \frac{z}{100} * \sum_{i=1}^{n} C_{di} \tag{3.1}$$

3.6 Benchmarks

To test the enhanced version of MOGA, different benchmarks with different specifications were selected in order to compare the result with the existing MOGA. The benchmarks reflect mathematical model for real life problems. The benchmarks were taken from many previous implementation for MOGA [42, 19] are summarized as followed.

3.6.1 Griewank

The Griewank function is a function widely used to test the convergence of global optimization functions and to test the ability of different solution procedures to find local optima [42]. The Griewank function [2] of order n is defined by:

$$f(x_1, x_2, ..., x_n) = 1 + (0.00025) \times \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos(\frac{x_i}{\sqrt{i}}) \tag{3.2}$$

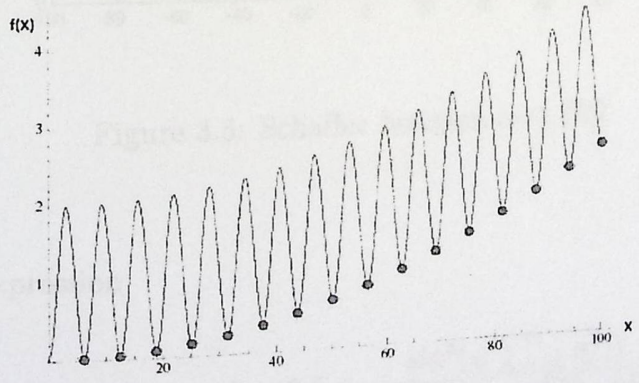


Figure 3.4: Griewank function n=1 [42]

Where n indicates the dimensions of the problem and in our case the number of objectives. Figure 3.4 [42] shows the function f plotted for $x_i \in [0, 100]$ and $n = 1$.

3.6.2 Schaffer F6

This parametric optimization problem is multi modal. It is difficult to solve by most usual methods due to circular local maxima [36].

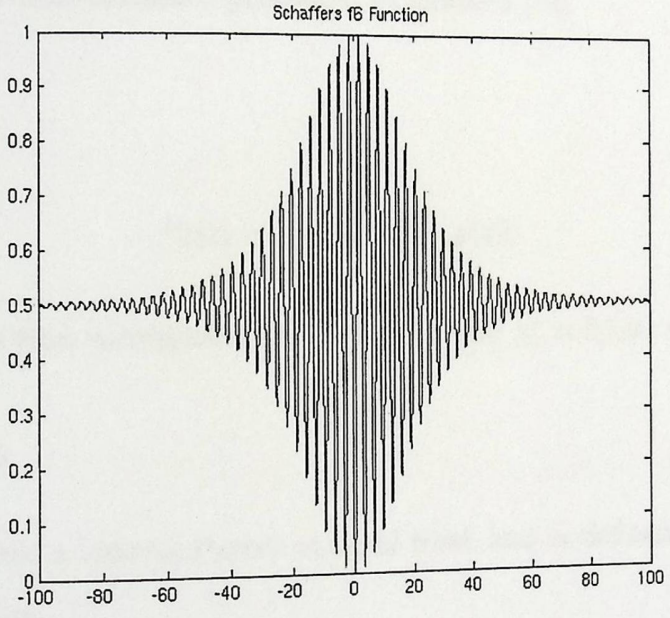


Figure 3.5: Schaffer function n=1 [36]

It has the expression

$$f(x_1, x_2, ..., x_n) = 0.5 + \frac{\sin^2(\sqrt{\sum_{i=1}^{n} x_i^2})}{1 + 0.001 \times (\sum_{i=1}^{n} x_i^2)^2} \tag{3.3}$$

Where n indicates the dimensions of the problem and in our case the number of objectives. Figure 3.5 [36] shows the function f plotted for $x_i \in [-100, 100]$ and $n = 1$.

3.6.3 Zitzler-Deb-Thiele (ZDT) Test Problems

The ZDT test problems are two objective problems framed by Zitzler [32]. ZDT test problems are unconstrained problems, as followed [54].

$$f1(x). \tag{3.4}$$

$$f2(x) = g(x)\dot{h}(f1(x), g(x)). \tag{3.5}$$

The Pareto region corresponds to $0 \leq x^* \leq 1$ and $x_i^* = 0 for i = 2, 3, ..., n$.

ZDT1 Problem

Problem ZDT1 has a convex Pareto optimal front and is defined as shown in the following equations:

$$f_1(x) = x_1. \tag{3.6}$$

$$g(x) = 1 + \frac{9}{n-1}\sum_{i=2}^{n} x_i. \tag{3.7}$$

$$h(f_1, g) = 1 - \sqrt{\frac{f_1}{g}}. \tag{3.8}$$

ZDT2 Problem

Problem ZDT2 has a concave Pareto optimal front. The problem is presented in in the following equations:

$$f_1(x) = x_1. \tag{3.9}$$

$$g(x) = 1 + \frac{9}{n-1}\sum_{i=2}^{n} x_i. \tag{3.10}$$

$$h(f_1, g) = 1 - (\frac{f_1}{g})^2. \tag{3.11}$$

ZDT3 Problem

Problem ZDT3 has a number of disconnected Pareto fronts. The problem is presented in in the following equations:

$$f_1(x) = x_1. \tag{3.12}$$

$$g(x) = 1 + \frac{9}{n-1}\sum_{i=2}^{n} x_i. \tag{3.13}$$

$$h(f_1, g) = 1 - \sqrt{\frac{f_1}{g}}. \tag{3.8}$$

ZDT2 Problem

Problem ZDT2 has a concave Pareto optimal front. The problem is presented in in the following equations:

$$f_1(x) = x_1. \tag{3.9}$$

$$g(x) = 1 + \frac{9}{n-1}\sum_{i=2}^{n} x_i. \tag{3.10}$$

$$h(f_1, g) = 1 - (\frac{f_1}{g})^2. \tag{3.11}$$

ZDT3 Problem

Problem ZDT3 has a number of disconnected Pareto fronts. The problem is presented in in the following equations:

$$f_1(x) = x_1. \tag{3.12}$$

$$g(x) = 1 + \frac{9}{n-1}\sum_{i=2}^{n} x_i. \tag{3.13}$$

$$h(f_1, g) = 1 - \sqrt{\frac{f_1}{g}} - \frac{f_1}{g} \times \sin(10\pi f_1). \tag{3.14}$$

Chapter 4

Experimental Results and Analysis

This chapter presents the experiments that used to validate the new platform and the analysis of the results of the study. The chapter is organized as follows: Section 4.1 describes the hardware and software configuration that the platform tested on it. Section 4.2 describes the experiments on each benchmark using MOGA and E-MOGA separately. Section 4.3 shows the analysis of the achieved performance. Section 4.4 shows the analysis of the solutions quality.

4.1 Experimental Environment

Since our research is a special type of software that requires a special type of hardware; certain requirements must be available. Testing and development were made on the following environment specifications:

- Personal computer (no matter what are the specifications of it) the used PC for testing is Pentium 4 with 512 MB RAM.
- Operating system Windows xp or higher.
- MATLAB version 2008.
- CUDA enable device (Graphical card) the used device is nVidia 9500 GT with 1GB memory.
- Tools and SDK.

4.2 Experimental Results

The obtained results of the enhanced version of MOGA were analyzed, tested and compared to the performance of the old system to ensure the quality of the results. The tests as described in the implementation steps are done on selected benchmarks. Table 4.1 shows the selected benchmarks with the used constrains.

The experiments performed done on the selected benchmarks as separated cases, each of which will be executed ten times to find the results in average. In each experiment, we selected the optimal solution, constrains, number of objective, stopping criteria and we specified a suitable fitness function. To make the testing process easy and fair, the selected regions(constrains), dimensionality and the optimal solution vector were set the same for all benchmarks as shown in Table 4.2.

Table 4.1: The mathematical formulation for benchmarks

benchmark	mathematical formula
ZDT1	$f_1(x) = x_1$ $g(x) = 1 + \frac{9}{n-1}\sum_{i=2}^{n} x_i$ $h(f_1, g) = 1 - \sqrt{\frac{f_1}{g}}$
ZDT2	$f_1(x) = x_1$ $g(x) = 1 + \frac{9}{n-1}\sum_{i=2}^{n} x_i$ $h(f_1, g) = 1 - (\frac{f_1}{g})^2$
ZDT3	$f_1(x) = x_1$ $g(x) = 1 + \frac{9}{n-1}\sum_{i=2}^{n} x_i$ $h(f_1, g) = 1 - \sqrt{\frac{f_1}{g}} - \frac{f_1}{g} \times \sin(10\pi f_1)$
ZDT4	$f_1(x) = x_1$ $g(x) = 1 + 10 \times (n-1) + \sum_{i=2}^{n} [x_i^2 - 10 \times cos(4\pi x_i))]$ $h(f_1, g) = g(x)\left[1 - \sqrt{\frac{f_1}{g(x)}} - \frac{f_1}{g(x)} sin(10\pi f_1)\right]$
ZDT6	$f_1(x) = 1 - exp(-4x) sin^6(6\pi x_1)$ $g(x) = 1 + 9\left[\frac{\sum_{i=2}^{n} x_i}{n-1}\right]^{0.25}$ $h(f_1, g) = g(x)\left[1 - (\frac{f(x_1)}{g(x)})^2\right]$
Schaffer F6	$f(x_1, x_2, ..., x_n) = 0.5 + \frac{\sin^2(\sqrt{\sum_{i=1}^{n} x^2}))}{1+0.001\times(\sum_{i=1}^{n} x^2)^2}$
Griewank	$f(x_1, x_2, ..., x_n) = 1 + (0.00025) \times \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos(\frac{x_i}{\sqrt{i}})$

Table 4.2: Benchmarks properties

property	value
constrains	$-100 \leq x_i \leq 100$
dimensionality	$10, 20, 30, 40, 50$
optimal solution	$x^* = \{0, ..., 0\}$

Stopping criteria determines what causes the algorithm to terminate. MOGA can be implemented using one or more stopping criteria from the following [4]:

1. Generations specifies the maximum number of iterations the genetic algorithm performs.

2. Time limit specifies the maximum time in seconds the genetic algorithm runs before stopping.

3. Fitness limit; If the best fitness value is less than or equal to the value of Fitness limit, the algorithm stops.

4. Stall generations; if the weighted average change in the fitness function value over Stall generations is less than function tolerance, the algorithm stops.

5. Stall time limit; if there is no improvement in the best fitness value for an interval of time in seconds specified by Stall time limit, the algorithm stops.

6. Function tolerance; if the cumulative change in the fitness function value over Stall generations is less than function tolerance, the algorithm stops.

In testing process, the stopping criteria selected as follow:

1. Since the complexity of the problem will increase if the number of objectives increased, the maximum number of generation that will be selected will be in a variable form to reflect the complexity of the problem, so, by examination, we select the maximum number of generation to be equal to 400 * number of objectives .

2. Function tolerance and we select a small value $\epsilon = 10e - 3$.

The experiments are summarized as follow:

Experiment 1:

In this experiment, the performance and the quality of ZDT1 problem will be tested on MOGA and on E-MOGA using the same conditions and properties to compare the results.

Regarding the performance, the results in Table 4.3 shows a significant speedup between 3X and 19X, the results in Figure 4.1 shows the relation between number of objectives and the resulted speedup gain for this experiment.

Table 4.3: Execution time for ZDT1 benchmark on the MOGA and E-MOGA

Dimension	CPU-time (s)	GPU-time (s)	Speedup
10	217	77	3
20	785	178	5
30	3218	224	15
40	4233	269	16
50	6281	340	19

Regarding the quality, the results in Table 4.4 shows better solutions closer to the optimal solution.

Table 4.4: Solution quality for ZDT1 benchmark on the MOGA and E-MOGA

Dimension	CPU-Quality	GPU-Quality	Quality difference
10	64.5	28.5	36
20	112.9	34.9	78.0
30	202.3	69.6	132.7
40	164.2	69.9	94.3
50	186.0	97.8	88.2

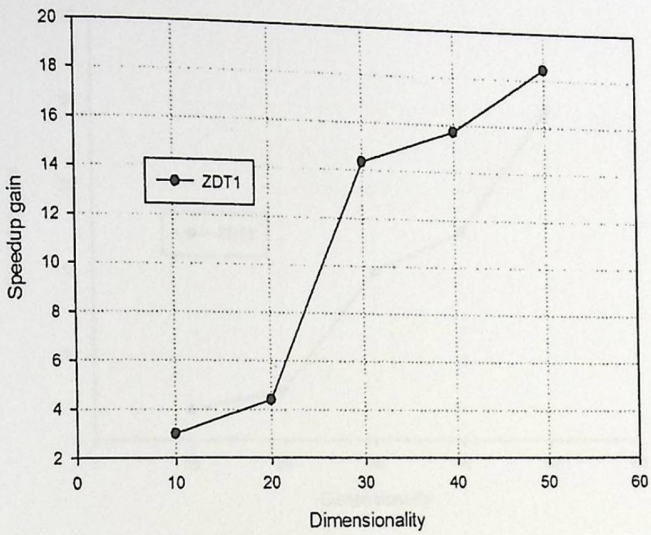


Figure 4.1: Performance enhanced on ZDT1

Experiment 2:

In this experiment, the performance and the quality of ZDT2 problem will be tested and compared as ZDT1 using the same conditions and properties.

Regarding the performance, the results in Table 4.5 shows a significant speedup between 7X and 26X, the results in Figure 4.2 shows the relation between number of objectives and the resulted speedup gain for this experiment.

Table 4.5: Execution time for ZDT2 benchmark on the MOGA and E-MOGA

Dimension	CPU-time (s)	GPU-time (s)	Speedup
10	379	56	7
20	548	69	8
30	3177	210	16
40	4216	239	18
50	4979	197	26

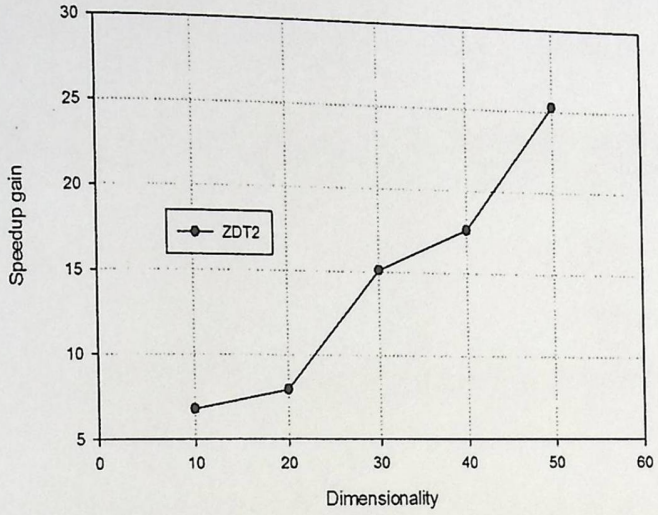


Figure 4.2: Performance enhanced on ZDT2

Regarding the quality, the results in Table 4.6 shows better solutions closer to the optimal solution.

Table 4.6: Solution quality for ZDT2 benchmark on the MOGA and E-MOGA

Dimension	CPU-Quality	GPU-Quality	Quality difference
10	408	17.7	390.4
20	390.4	115.8	274.7
30	573.9	54.7	519.1
40	613.3	74.8	538.5
50	152.9	93.0	59.9

Experiment 3:

In this experiment, the performance and the quality of ZDT3 problem will be tested and compared as ZDT1 using the same conditions and properties.

Regarding the performance, the results in Table 4.7 shows a significant speedup

between 4X and 54X, the results in Figure 4.3 shows the relation between number of objectives and the resulted speedup gain for this experiment.

Table 4.7: Execution time for ZDT3 benchmark on the MOGA and E-MOGA

Dimension	CPU-time (s)	GPU-time (s)	Speedup
10	504	135	4
20	1085	144	8
30	3129	188	17
40	4238	214	20
50	14035	260	54

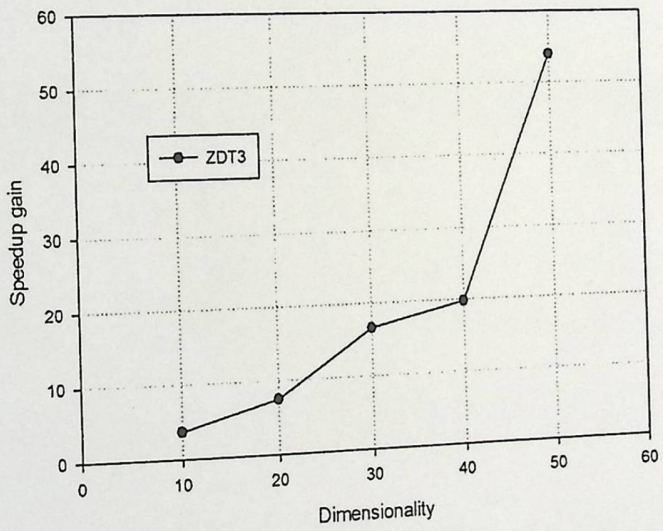


Figure 4.3: Performance enhanced on ZDT3

Regarding the quality, the results in Table 4.8 shows better solutions closer to the optimal solution.

Experiment 4:

In this experiment, the performance and the quality of ZDT4 problem will be tested

Table 4.8: Solution quality for ZDT3 benchmark on the MOGA and E-MOGA

Dimension	CPU-Quality	GPU-Quality	Quality difference
10	201.2	119.2	82
20	69.1	41.3	27.9
30	299.1	58.5	240.7
40	337.5	94.2	243.3
50	167.6	102.7	64.9

and compared as ZDT1 using the same conditions and properties.

Regarding the performance, the results in Table 4.9 shows a significant speedup between 3X and 37X, the results in Figure 4.4 shows the relation between number of objectives and the resulted speedup gain for this experiment.

Table 4.9: Execution time for ZDT4 benchmark on the MOGA and E-MOGA

Dimension	CPU-time (s)	GPU-time (s)	Speedup
10	133	46	3
20	970	133	8
30	1190	150	9
40	2318	182	13
50	8263	227	37

Regarding the quality, the results in Table 4.10 shows better solutions closer to the optimal solution.

Table 4.10: Solution quality for ZDT4 benchmark on the MOGA and E-MOGA

Dimension	CPU-Quality	GPU-Quality	Quality difference
10	121.8	16.2	105.6
20	81.0	30.3	50.7
30	53.5	49.4	4.1
40	98.6	74.2	24.3
50	98.8	83.8	15.0

Experiment 5:

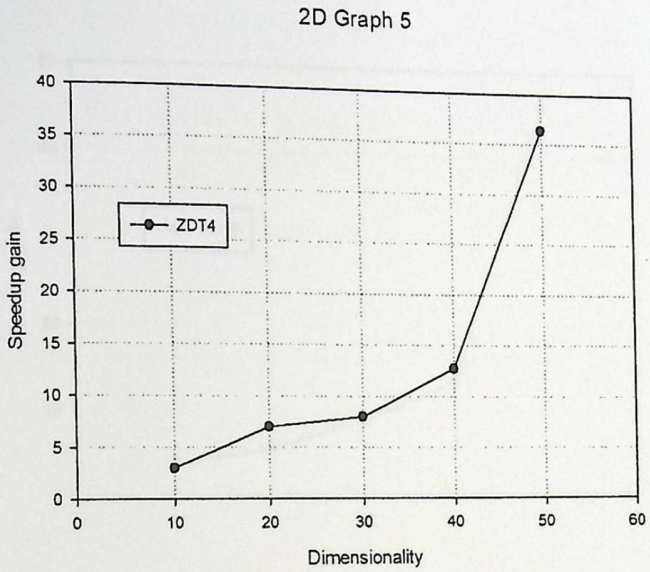


Figure 4.4: Performance enhanced on ZDT4

In this experiment, the performance and the quality of ZDT6 problem will be tested and compared as ZDT1 using the same conditions and properties.

Regarding the performance, the results in Table 4.11 shows a significant speedup between 4X and 46X, the results in Figure 4.5 shows the relation between number of objectives and the resulted speedup gain for this experiment.

Table 4.11: Execution time for ZDT6 benchmark on the MOGA and E-MOGA

Dimension	CPU-time (s)	GPU-time (s)	Speedup
10	123	32	4
20	382	68	6
30	1010	113	9
40	2118	159	14
50	9820	214	46

Regarding the quality, the results in Table 4.12 shows better solutions closer to

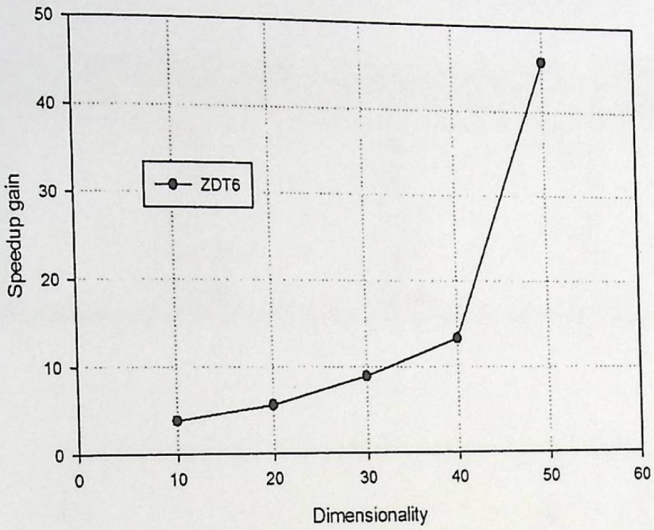


Figure 4.5: Performance enhanced on ZDT6

the optimal solution.

Table 4.12: Solution quality for ZDT6 benchmark on the MOGA and E-MOGA

Dimension	CPU-Quality	GPU-Quality	Quality difference
10	27.2	22.7	4.5
20	122.2	45.0	77.1
30	224.5	64.3	160.2
40	105.5	86.6	18.8
50	489.9	108.9	381.0

Experiment 6:

In this experiment, the performance and the quality of Schaffer F6 problem will be tested and compared as ZDT1 using the same conditions and properties.

Regarding the performance, the results in Table 4.13 shows a significant speedup between 4X and 11X, the results in Figure 4.6 shows the relation between number

of objectives and the resulted speedup gain for this experiment.

Table 4.13: Execution time for Schaffer F6 benchmark on the MOGA and E-MOGA

Dimension	CPU-time (s)	GPU-time (s)	Speedup
10	100	32	4
20	197	60	5
30	572	100	6
40	1331	167	8
50	1413	162	11

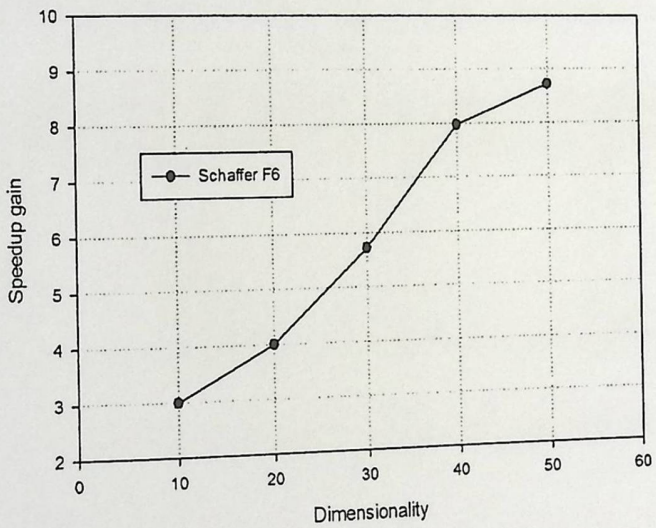


Figure 4.6: Performance enhanced on Schaffer F6

Regarding the quality, the results in Table 4.14 shows better solutions closer to the optimal solution.

Experiment 7:

In this experiment, the performance and the quality of Grewink problem will be tested and compared as ZDT1 using the same conditions and properties.

Table 4.14: Solution quality for Schaffer F6 benchmark on the MOGA and E-MOGA

Dimension	CPU-Quality	GPU-Quality	Quality difference
10	24.1	4	20.1
20	58.4	5.0	53.4
30	81.6	8.0	73.6
40	83.6	7.0	76.6
50	129.7	11.0	118.7

Regarding the performance, the results in Table 4.15 shows a significant speedup between 4X and 46X, the results in Figure 4.7 shows the relation between number of objectives and the resulted speedup gain for this experiment.

Table 4.15: Execution time for Grewink benchmark on the MOGA and E-MOGA

Dimension	CPU-time (s)	GPU-time (s)	Speedup
10	77	28	3
20	363	61	6
30	874	97	9
40	1600	158	11
50	9820	214	46

Regarding the quality, the results in Table 4.16 shows better solutions closer to the optimal solution.

Table 4.16: Solution quality for Grewink benchmark on the MOGA and E-MOGA

Dimension	CPU-Quality	GPU-Quality	Quality difference
10	17	3	14
20	36.0	5.0	31.0
30	64.4	6.0	58.4
40	86.0	8.0	78.0
50	94.0	11.0	83.0

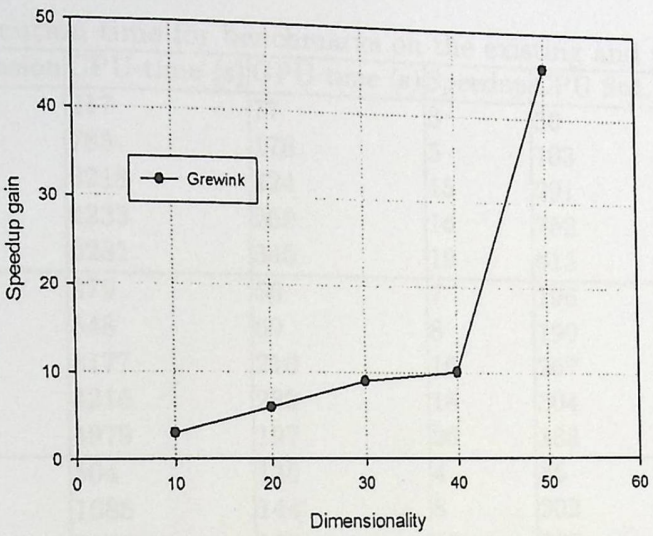


Figure 4.7: Performance enhanced on Grewink

4.3 Performance Analysis

Depending on the stopping criteria, and, since MOGA starts the population randomly, the tool does not reach the Pareto front in a fixed time for any benchmark each run time, the average time should be taken for (10) executions in order to have reasonable results.

The tests were taken for both systems; MOGA and E-MOGA with different number of objectives on each benchmark as shown in the previous section. The tests results are summarized in Table 4.17:

The statistical analysis was applied for the results achieved from E-MOGA and MOGA for the different benchmarks, and the results of the same dimensionality for each benchmark was compared together. The statistical analysis showed in the

Table 4.17: Execution time for benchmarks on the existing and proposed systems

Benchmark	Dimension	CPU-time (s)	GPU-time (s)	Speedup	CPU Std. deviation	GPU Std. deviation
ZDT1	10	217	77	3	36	10
	20	785	178	5	103	9
	30	3218	224	15	291	12
	40	4233	269	16	752	24
	50	6281	340	19	615	28
ZDT2	10	379	56	7	196	7
	20	548	69	8	190	5
	30	3177	210	16	267	14
	40	4216	239	18	304	65
	50	4979	197	26	188	13
ZDT3	10	504	135	4	35	8
	20	1085	144	8	202	18
	30	3129	188	17	707	8
	40	4238	214	20	448	31
	50	14035	260	54	1467	19
ZDT4	10	133	46	3	38	7
	20	970	133	8	84	14
	30	1190	150	9	577	15
	40	2318	182	13	352	21
	50	8263	227	37	1170	40
ZDT6	10	123	32	4	32	3
	20	382	68	6	71	1
	30	1010	113	9	396	3
	40	2118	159	14	196	2
	50	9820	214	46	1463	6
Schaffer F6	10	100	32	4	28	2
	20	197	60	5	30	2
	30	572	100	6	122	2
	40	1331	167	8	335	3
	50	1413	162	11	271	12
Grewink	10	77	28	3	13	2
	20	363	61	6	86	3
	30	874	97	9	96	5
	40	1600	158	11	74	8
	50	9820	214	46	632	5

following charts:

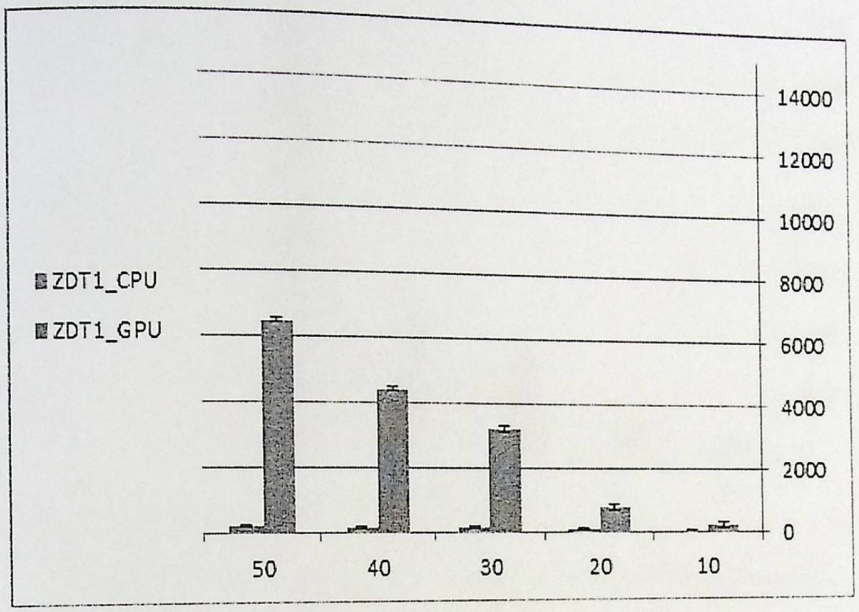


Figure 4.8: Statistical analysis for ZDT1 Benchmark

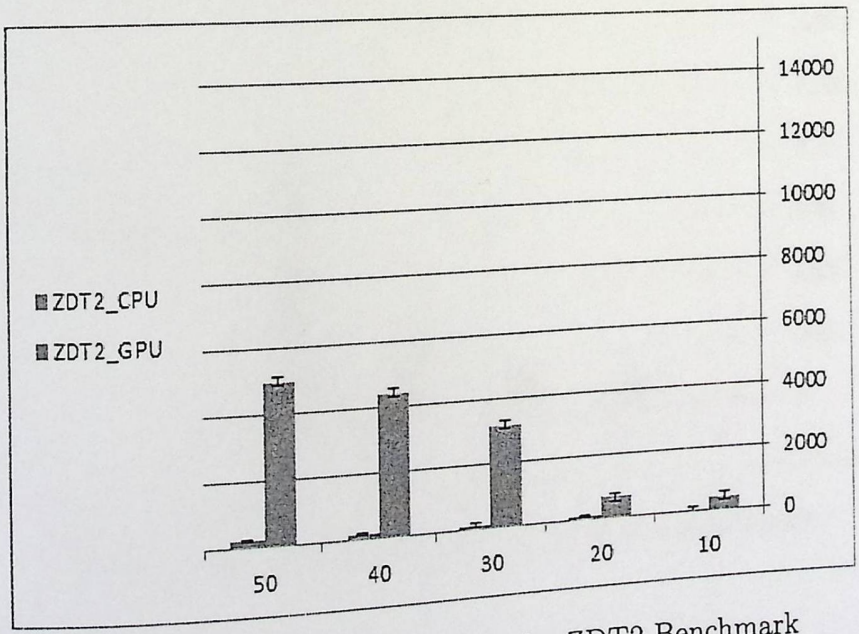


Figure 4.9: Statistical analysis for ZDT2 Benchmark

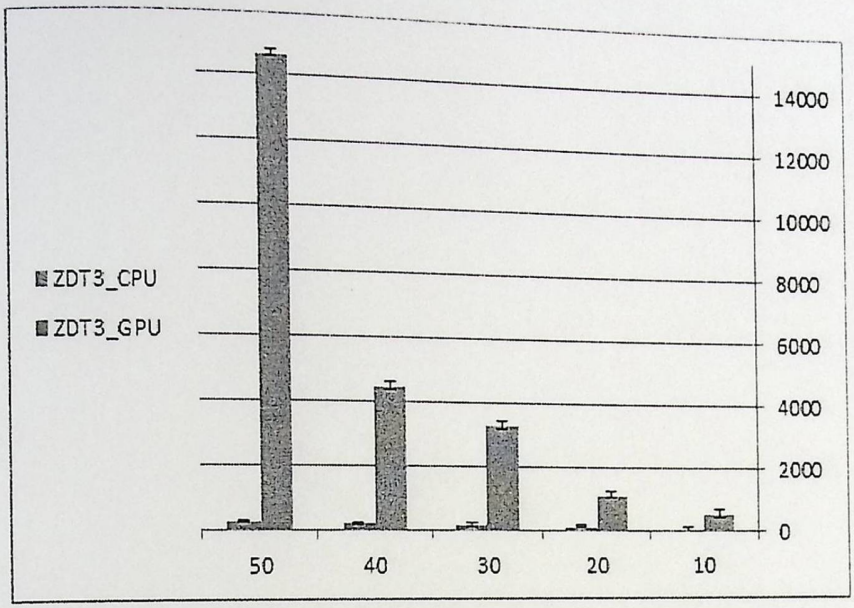


Figure 4.10: Statistical analysis for ZDT3 Benchmark

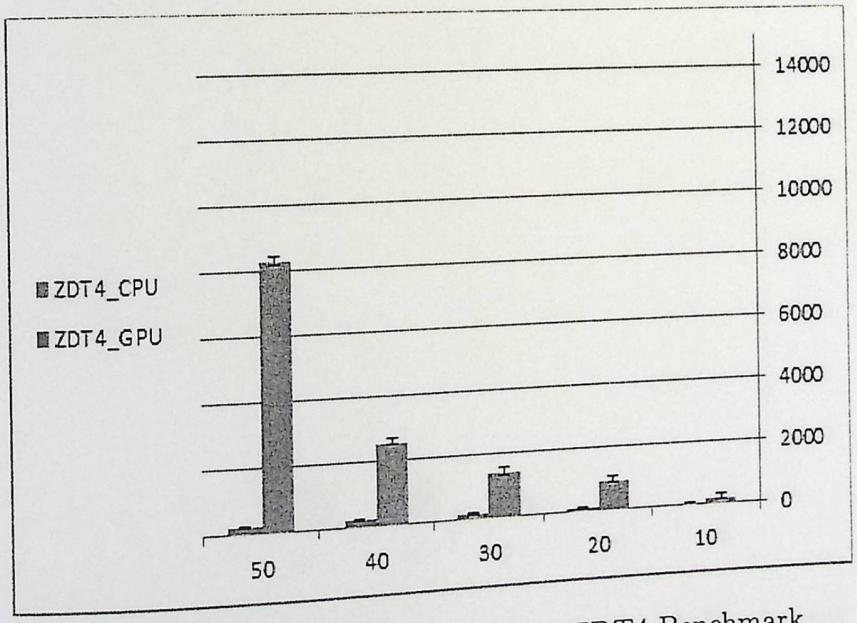


Figure 4.11: Statistical analysis for ZDT4 Benchmark

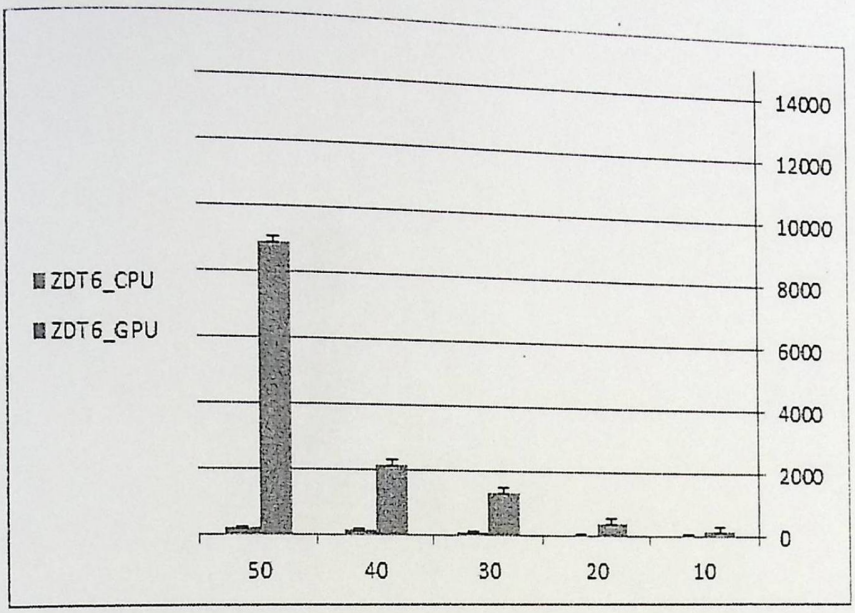


Figure 4.12: Statistical analysis for ZDT6 Benchmark

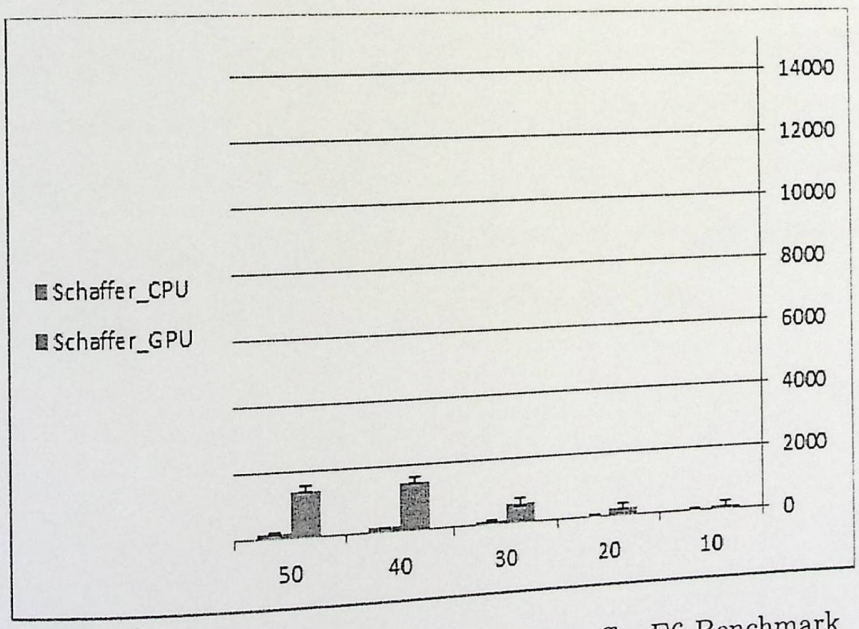


Figure 4.13: Statistical analysis for Schaffer F6 Benchmark

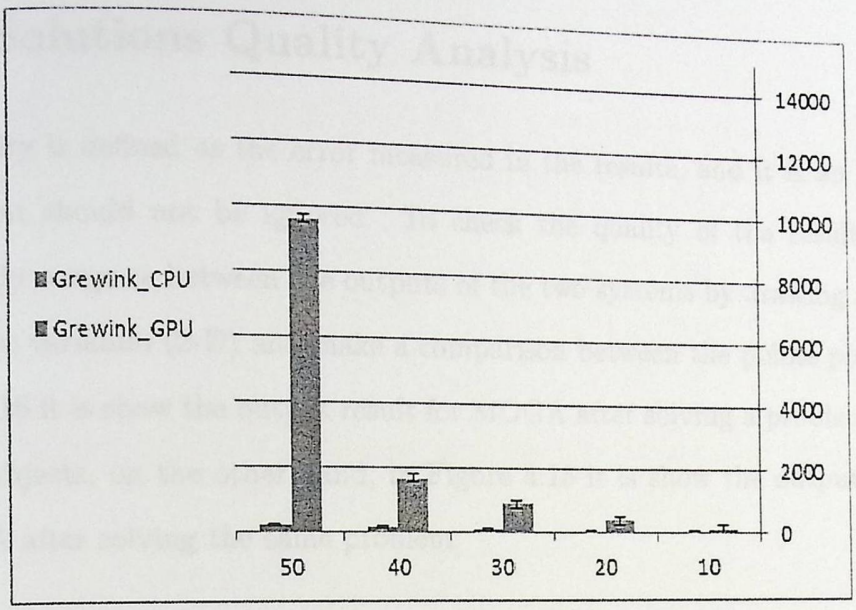


Figure 4.14: Statistical analysis for Grewink Benchmark

From the previous charts that shows the error bars for the experiments means with +/- standard error (shown as intervals on the head of the bar) that known as confidence interval and since there are no overlap between the results for each experiment of the same dimensionality of each experiment, we become sure that the difference between the two means is statistically significant.

4.4 Solutions Quality Analysis

The quality is defined as the error measured in the results, and it is an important factor that should not be ignored. To check the quality of the results, we can graphically compare between the outputs of the two systems by drawing the results of any two variables (2-D) and make a comparison between the points positions. In Figure 4.16 it is show the output result for MOGA after solving a problem with two conflict objects, on the other hand, in Figure 4.15 it is show the output result for E-MOGA after solving the same problem.

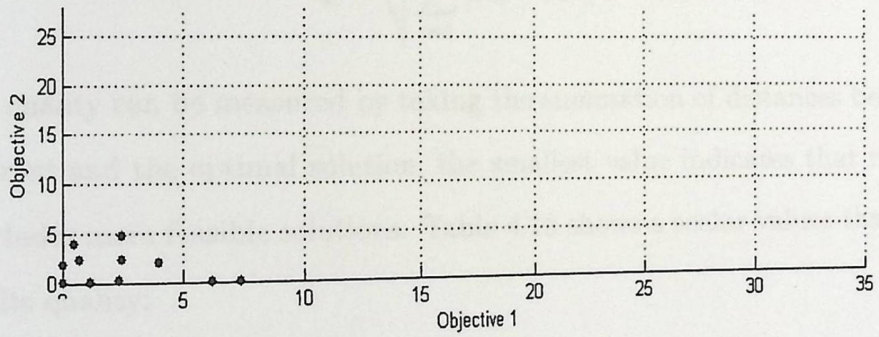


Figure 4.15: The Pareto front after the final iteration on E-MOGA

It is clear that both Figures almost contains the same number of points, but a deep look at the E-MOGA result, the result distribution is very close to the optimal solution which is (0,0) in this case, on the other hand, the CPU solution take a larger range of distribution.

The graphical comparison is not a sufficient indicator of the results quality, so, a new idea was implemented in Equation 4.1.

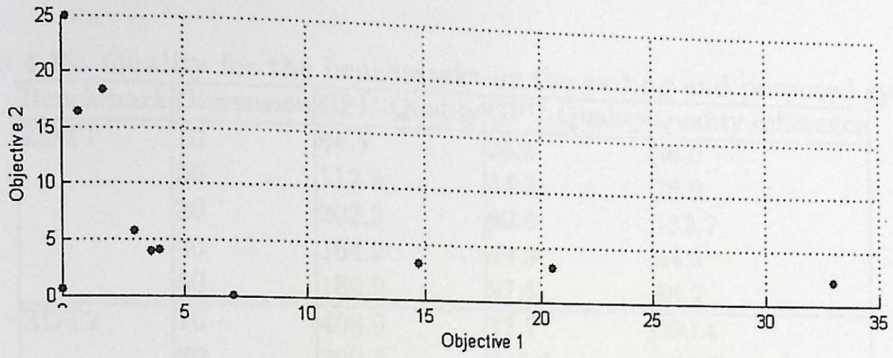


Figure 4.16: The Pareto front after the final iteration on MOGA

$$Q = \sqrt{\sum_{i=1}^{n} (\mathbf{X}_i - \mathbf{X}^*)^2}. \tag{4.1}$$

The quality can be measured by taking the summation of distances between the Pareto front and the optimal solution; the smallest value indicates that the Pareto front includes more feasible solutions. Table 4.18 shows a scalar values that indicate the results quality:

Table 4.18: Quality for the benchmarks on the exiting and proposed systems

Benchmark	Dimension	CPU-Quality	GPU-Quality	Quality difference
ZDT1	10	64.5	28.5	36.0
	20	112.9	34.9	78.0
	30	202.3	69.6	132.7
	40	164.2	69.9	94.3
	50	186.0	97.8	88.2
ZDT2	10	408.0	17.7	390.4
	20	390.4	115.8	274.7
	30	573.9	54.7	519.1
	40	613.3	74.8	538.5
	50	152.9	93.0	59.9
ZDT3	10	201.2	119.2	82.0
	20	69.1	41.3	27.9
	30	299.1	58.5	240.7
	40	337.5	94.2	243.3
	50	167.6	102.7	64.9
ZDT4	10	121.8	16.2	105.6
	20	81.0	30.3	50.7
	30	53.5	49.4	4.1
	40	98.6	74.2	24.3
	50	98.8	83.8	15.0
ZDT6	10	27.2	22.7	4.5
	20	122.2	45.0	77.1
	30	224.5	64.3	160.2
	40	105.5	86.6	18.8
	50	489.9	108.9	381.0
Schaffer F6	10	24.1	4.0	20.1
	20	58.4	5.0	53.4
	30	81.6	8.0	73.6
	40	83.6	7.0	76.6
	50	129.7	11.0	118.7
Grewink	10	17.0	3.0	14.0
	20	36.0	5.0	31.0
	30	64.4	6.0	58.4
	40	86.0	8.0	78.0
	50	94.0	11.0	83.0

4.5 Analysis of the Results

From Table 4.17 it is clearly shown that the new enhanced MOGA is faster than the old MOGA in all cases and when the dimensionality (number of variables) increases, the power of the GPU rises to handle the complex computation on its parallel architecture.

The Speedup varies due to the complexity of the problem; as the number of objectives increase, the system becomes more complex to be solved and needs more time. The GPU performance can beat the CPU performance all the time. The minimum Speedup can not be less than 2.7X and the Speedup can increase to become more than 53X in some cases. Figure 4.17 shows the relation between the number of objectives and Speedup.

Again, it is clear in Table4.18 that the new system does not only beat the old one when comparing the execution time,but also the quality of the result is better in all cases; that we get a closer solutions to the optimal one with less time.

As stated previously, the performance results from paralyzing the time consuming parts that founds within the Multi Objective Genetic Algorithm. The parallelization by itself can't lead to a better performance but we have to focus on two main polices during our work with parallelization process as follow:

- Utilize GPU as possible: As many local threads as possible should work in an SIMD fashion where the same instructions are applied on multiple data.
- Reduce memory transfer : The memory transfers should be kept at a minimum.

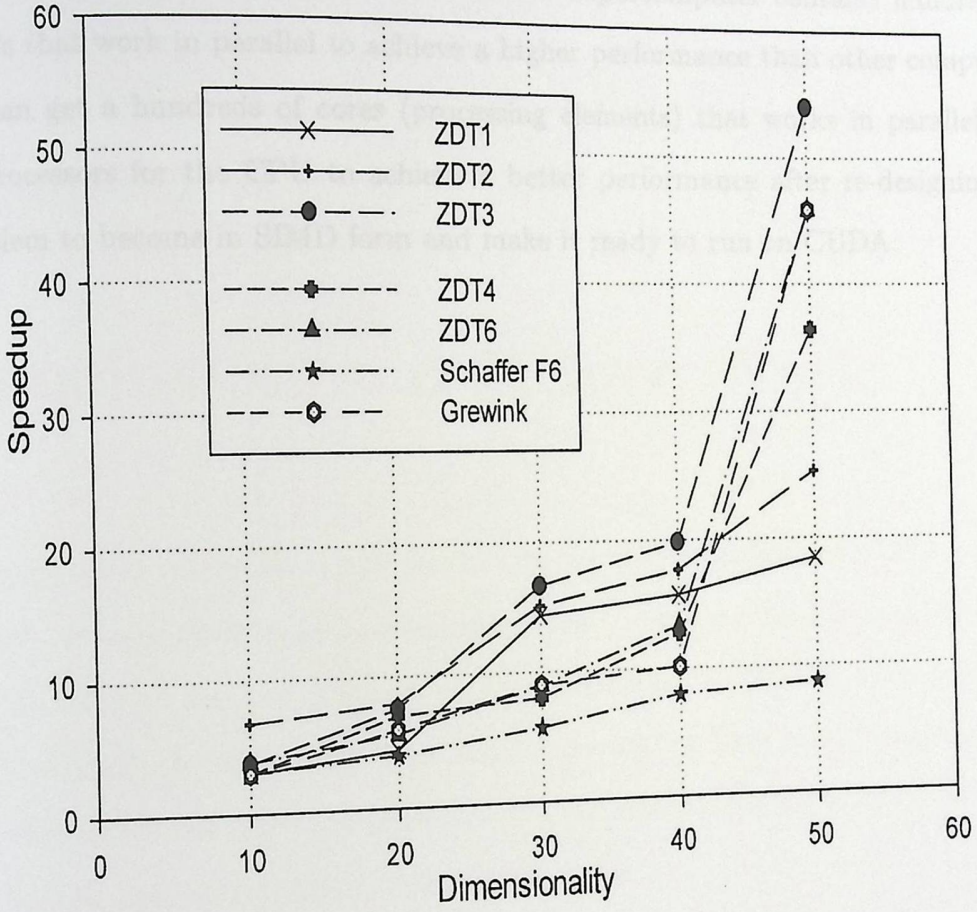


Figure 4.17: The relation between number of objective and the speedup

The tests were performed on Pentium four computer with CUDA enabled device that work as a co-processor with CPU, and the performance of such computer is not considered to be high in these days, so, we made an extra test using a high performance computer that work on Intel i7 and we got almost the same results with ignorable changes in the resulted performance. So, we prove the idea that

we mentioned before that the researchers may have their own supercomputer with low price when they using CUDA; since the supercomputer contains hundreds of CPUs that work in parallel to achieve a higher performance than other computers, we can get a hundreds of cores (processing elements) that works in parallel as a co-processors for the CPU to achieve a better performance after re-designing the problem to become in SIMD form and make it ready to run on CUDA.

Chapter 5

Conclusion and Future Work

The purpose of this master thesis was to enhance the performance of MOGA by using CUDA, propose criteria for measuring the error in the results, enhance the results quality and test the new platform on many benchmarks. During these operations we faced many problems and after finish the work we have many conclusions. In this chapter we will show the problems that we faced during the work on thesis, the conclusions that we reached and the future works.

5.1 Limitations

The work on previous implementation of Multi Objective Genetic Algorithm (MOGA) is not an easy work; to analyze and understand what actually happened within the program is one of the obstacles that we faced during our work. Reading and analyzing thousands of lines of code help us to solve the problem. CUDA as

stated previously can be accessed directly by using the CUDA-C language that is very similar to the standard ANSI C, we did not found any problem to learn it and start the work using it. Converting codes from ANSI C to become ready to execute on CUDA it is a simple operation that required re-designing the code to get the best performance, the codes that should be executed on CUDA called kernel. Re-designing the code is the most important operation that may lead to a bad performance if the code designed in wrong way. The data that is required by the kernels should be transferred to CUDA device before kernel execution, and after execution we should transfer the result back. This operation can lead to a bad performance since the memory access can add extra time for the whole execution time. The good design try to reduce the number of memory access can solve the problem.

5.2 Conclusions

The main question that we may ask, which better, CUDA or parallelization?

The parallelization is the process to perform many tasks simultaneously and this technique requires many physical processors that may found within the same computer or in separated computers. The use of parallelization to solve problems may lead to some problems that solved by using CUDA, such as, cost, size, watt per process and less implementation complexity.

The researchers need powerful computers to do their research, and since the computer specification increases, the price would increase as well. So the trend to explore all the resources that are found inside the computer case especially the GPU,

is the way for each researcher to own a supercomputer within the lowest cost.

The Multi Objective Genetic Algorithm optimization problem is one of the important algorithm used in many field for finding the tradeoff between conflict objectives, MOGA in many applications needs to be finished in reasonable time with accurate result that is closer to the optimal solution, many implementations of MOGA was done by making a tradeoff between finding the optimal solution and finding the solution in a reasonable amount of time.

The Enhanced Multi Objective Genetic Algorithm (E-MOGA) and Multi Objective Genetic Algorithm (MOGA) have been implemented and tested in two versions; we used MOGA implementation found in MATLAB optimization toolbox that execute by CPU, on the other hand, E-MOGA is an enhanced version of MOGA, it use serial and parallel execution methods. The enhanced version demonstrates both the modern CPU processing power and CUDA enable device capabilities. As two versions of implementation follow the same logical steps, the results of the study purely demonstrates the improvement in performance and less error rate maintained by this new version that we called it E-MOGA.

All the results reported show the efficiency of our E-MOGA as compared to a well-coded sequential version found in MATLAB optimization toolbox. The tool demonstrates significant speed gains using affordable, scalable and commercially available hardware. A comparison with a generic MOGA library work in sequential, would result in much greater speedup. The error rate that we define as the sum of distances of the results from the optimal solution was improved in E-MOGA too

and we show that E-MOGA have a high degree of accuracy.

Even for variable number of objectives, the E-MOGA can work well with simple changes in parameters. The results show that, the more the complexity of the problem is, the more speedup gain will introduce, this will highlight the power computation of GPU in highly computation and complex problems.

Referring to the research objectives we can say that we reach our objectives and we can summarize the results and contributions as followed:

- The research outcome with a new tool that used to solve Multi Objective Genetic Algorithm optimization problem that developed in serial and parallel form; the serially executed parts will handled by CPU and the parallel executed parts will handled by the parallel architecture founds in GPU (CUDA enabled device) and we state the specialized operations of each.
- The new tool (E-MOGA) shows a speedup gains when compared with a will coded version founds in MATLAB optimization toolbox. The speedup gained was vary from 3X-54X refer to the change of the problem and number of objectives defined.
- E-MOGA can run only on systems contains CUDA enabled device.
- Not only the performance of MOGA was improved, but also the quality of the results was improved also; The idea for replacing more solutions in each iteration leads us to a great results.

- The E-MOGA was tested using many benchmarks with different number of objectives; the results shows that our tool is general enough to handle any problem that considered as multi objective and we test it up to 50 objectives only.

- By comparing our work with similar work found in [20] they used CUDA for solving Multi Objective Parallel Genetic Algorithm (MOPGA) in dedicated problem that is documents searching problems and they achieved a speedup gain up to 53X, we can see that we got almost the same result for solving general purpose multi objective optimization problem and we guarantee that we will have the better solutions quality.

At the end of the study it is shown that GPU processing capabilities are more than the classical CPU especially when an algorithm can be parallelized. CUDA offers ease of usage and some other optimizations for GPU programming. Although limited to a specific kind of hardware and programming languages, CUDA has important potential that can be utilized in solving complex computational problems.

Finally, we want to highlight that our system is versatile and modular and, even in its present version, can be general enough to be used to optimize a vast classes of conflict optimization problems, and the user has to provide a proper fitness function kernel when changing the problem.

5.3 Future Works

In future work we wish to investigate and examine the behavior of this system on a cluster containing more capable GPUs. We think a few changes and modifications can take place in E-MOGA to make it run on multiple GPUs. We can also try to work on parallel MOGA and test it on GPU in order to enhance it. Other researchers could find the way to use the power of GPU in their application; performance enhancement, low cost computing architecture, real time application....etc.

Many applications and algorithms could be re-designed to become ready to execute on CUDA if the parallelization is possible such as security algorithms, image processing, medical application, and robotics.

One of the great ideas that may found the space in the next few years is the mobile phone application; NVIDIA developed a new parallel architecture for mobile phones called TEGRA that contains up to 8 cores and we can find the way to develop a real-time and complex application to be executed easily on mobiles by using such architecture.

Bibliography

[1] Jens Breitbart. Cupp - a framework for easy cuda integration. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[2] Huidae Cho, Francisco Olivera, and Seth D. Guikema. A derivation of the number of minima of the griewank function. *Applied Mathematics and Computation*, 204(2):694–701, 2008.

[3] O. Cordon, F. Moya, and C. Zarco. A new evolutionary algorithm combining simulated annealing and genetic programming for relevance feedback in fuzzy information retrieval systems. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 6(5):308–319, August 2002.

[4] Kalyanmoy Deb and Deb Kalyanmoy. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[5] Kusum Deep, Krishna Pratap Singh, M. L. Kansal, and C. Mohan. An interactive method using genetic algorithm for multi-objective optimization problems

modeled in fuzzy environment. *Expert Syst. Appl.*, 38:1659–1667, March 2011.

[6] E.P.Ephzibah. Cost effective approach on feature selection using genetic algorithms and ls-svm classifier. *IJCA Special Issue on Evolutionary Computation*, (1):16–20, 2010. Published by Foundation of Computer Science.

[7] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE Micro*, 28:13–27, 2008.

[8] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[9] GPUmat. Gpumat user guide. Technical Report Version 0.27, December 2010.

[10] John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31:532–533, 1988.

[11] Tianyi David Han and Tarek S. Abdelrahman. hicuda: a high-level directive-based language for gpu programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 52–61, New York, NY, USA, 2009. ACM.

[12] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, Cambridge, MA, USA, 1992.

[13] R. Sophia Porchelvi i. An algorithmic approach to multi objective fuzzy linear programming problem international journal of algorithms. *International Journal of Algorithms, Computing and Mathematics*, 3:61–66, November 2010.

[14] NIVIDIA INC. Nividia cuda tutorial no nda. 2010.

[15] NVIDIA INC. Nvidia cuda architecture introduction and overview. 2009.

[16] NVIDIA INC. Nvidia cuda c programming best practices guide. 2009.

[17] NVIDIA INC. Nvidia cuda programming guide, version 2. 2009.

[18] VIRYANET INC. The balancing act of mobile workforce planning fulfilling multi-service objectives with priority based optimization. *International Journal of Algorithms, Computing and Mathematics*, 2010.

[19] Omar Al Jadaan, C. R. Rao, and Lakshmi Rajamani. Solving constrained multi-objective optimization problems using non-dominated ranked genetic algorithm. In *Proceedings of the 2009 Third Asia International Conference on Modelling & Simulation*, pages 113–118, Washington, DC, USA, 2009. IEEE Computer Society.

[20] Sathish AP Kumar Jason P. Duran. Cuda based multi objective parallel genetic algorithms: Adapting evolutionary algorithms for document searches. *EEE 11 conferance*, July 2011.

[21] D. F. Jones, S. K. Mirrazavi, and M. Tamiz. Multi-objective meta-heuristics: An overview of the current state-of-the-art. *European Journal of Operational Research*, 137(1):1–9, February 2002.

[22] Sarnath Kannan and Raghavendra Ganji. Porting autodock to cuda. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.

[23] GURLEEN KAUR. Design of recursive digital filters using multiobjective genetic algorithm. *International Journal of Engineering Science and Technology (IJEST)*, 3(7), Juky 2011.

[24] J Knox. Tabu search performance on the symmetric traveling salesman problem. *Computers and Operations Research*, 21(8):867–876, 1994.

[25] Abdullah Konak, David W. Coit, and Alice E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, September 2006.

[26] Marco Laumanns and Nando Laumanns. Evolutionary multiobjective design in automotive development. *Applied Intelligence*, 23:55–70, July 2005.

[27] Aaron Lefohn, Ian Buck, John D. Owens, and Robert Strzodka. Gpgpu: General-purpose computation on graphics processors. In *Tutorial 3 at IEEE Visualization*, October 2004.

[28] Ivana Ljubi and Gnther R. Raidl. An evolutionary algorithm with stochastic hill-climbing for the edge-biconnectivity augmentation problem. In *Applications*

of Evolutionary Computation, volume 2037 of LNCS, pages 20–29. Springer, 2001.

[29] Manuel López-Ibáñez and Thomas Stützle. An analysis of algorithmic components for multiobjective ant colony optimization: A case study on the biobjective TSP. pages 134–145.

[30] Philipp Lucas. *CGiS: High-Level Data-Parallel GPU Programming.* PhD thesis, Universität des Saarlandes, August 2007.

[31] Nipen M Shah, Hubert Thieriot, Franois Marchal, and Andrew Hoadley. *A comparison of multi-objective optimisation approaches for a gas-phase refrigeration process.* 2008.

[32] Nipen M Shah, Hubert Thieriot, Franois Marchal, and Andrew Hoadley. *A comparison of multi-objective optimisation approaches for a gas-phase refrigeration process.* 2008.

[33] Volker Mehrmann, Vasile Sima, Andras Varga, and Hongguo Xu. A matlab mex-file environment of slicot, 1999.

[34] Melanie Mitchell. *An introduction to genetic algorithms.* A Bradford book. MIT Press, Cambridge, Mass. [u.a.], 1996.

[35] Melanie Mitchell. *An Introduction to Genetic Algorithms (Complex Adaptive Systems).* A Bradford Book, third printing edition, February 1998.

[36] Endusa Muhando, Hiroshi Kinjo, and Tetsuhiko Yamamoto. Enhanced performance for multivariable optimization problems by use of gas with recessive gene structure. *Artificial Life and Robotics*, 10(1):11–17, 2006.

[37] Asim Munawar, Mohamed Wahib, Masaharu Munetomo, and Kiyoshi Akama. Hybrid of genetic algorithm and local search to solve max-sat problem using nvidia cuda framework. *Genetic Programming and Evolvable Machines*, 10:391–415, December 2009.

[38] L. Muniglia, L. Kiss, C. Fonteix, and I. Marc. Multicriteria optimization of a single-cell oil production. *European Journal of Operational Research*, 153(2):360–369, 2004.

[39] J. Novo, M. G. Penedo, and J. Santos. Evolutionary multiobjective optimization of topological active nets. *Pattern Recogn. Lett.*, 31:1781–1794, October 2010.

[40] M. Oiso, Y. Matsumura, T. Yasuda, and K. Ohkura. Implementing genetic algorithms to cuda environment using data parallelization. *Technical Gazette*, 18(4):511–517, 2011.

[41] Ian C. Parmee, Dragan C. Cvetković, Andrew H. Watson, and Christopher R. Bonham. Multiobjective satisfaction within an interactive evolutionary design environment. *Evol. Comput.*, 8:197–222, June 2000.

[42] Silvia Poles, Enrico Rigoni, and Tea Robič. MOGA-II performance on noisy optimiztion problems. In Bogdan Filipič and Jurij Šilc, editors, *Proceedings of*

the International Conference on Bioinspired Optimization Methods and their Applications - BIOMA 2004, pages 51–62. Jožef Stefan Institute, 2004.

[43] Petr Pospchal. Gpu-based acceleration of the genetic algorithm. In *Potaov architektury a diagnostika 2010*, pages 75–80. Faculty of Information Technology BUT, 2010.

[44] Petr Pospchal, Ji Jaro, and Josef Schwarz. Parallel genetic algorithm on the cuda architecture. In *6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 210–2011. Masaryk University, 2010.

[45] Petr Pospchal, Josef Schwarz, and Ji Jaro. Parallel genetic algorithm solving 0/1 knapsack problem running on the gpu. In *16th International Conference on Soft Computing MENDEL 2010*, pages 64–70. Brno University of Technology, 2010.

[46] Xavier Prats. Trajectory optimization for aircraft noise mitigation. In *Transportation Seminar Series. University of California, Berkeley.*, Berkeley, California (USA), Apr 2010. Department of Civil and Environmental Engineering. Institute of Transportation Studies.

[47] M. Srinivas and Lalit M. Patnaik. Genetic algorithms: A survey. *IEEE Computer*, 27(6):17–26, 1994.

[48] TEIMOUR TAJDARI. Gpu implementation using cuda, master thesis report. 2009.

[49] Thomas Weise. *Global Optimization Algorithms – Theory and Application.* it-weise.de (self-published): Germany, 2009.

[50] wikipedia.org. http://en.wikipedia.org/wiki/gene , http://en.wikipedia.org/wiki/dna, 2011.

[51] Jialong Wu and Alice M. Agogino. Automating keyphrase extraction with multi-objective genetic algorithms. In *HICSS*, 2004.

[52] www. wordpress.com. http://llpanorama.wordpress.com/2008/06/11/threads-and-blocks-and-grids-oh-my/, 2011.

[53] Yong Yan and Xiaodong Zhang. Profit-effective parallel computing. *IEEE Concurrency*, 7:65–69, April 1999.

[54] E. Zitzler, K. Deb, and L. Thiele. Comparison of Multiobjective Evolutionary Algorithms: Empirical Results (Revised Version). TIK Report 70, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, December 1999.

[55] E. Zitzler, M. Laumanns, and S. Bleuler. A tutorial on evolutionary multi-objective optimization. In X. Gandibleux et al., editors, *Metaheuristics for Multiobjective Optimisation*, volume 535 of *Lecture Notes in Economics and Mathematical Systems*. Springer, 2004.