A PARALLEL ALGORITHM FOR FLIGHT ROUTE PLANNING ON GPU USING CUDA

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

SEÇKİN SANCI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

APRIL 2010

Approval of the thesis:

A PARALLEL ALGORITHM FOR FLIGHT ROUTE PLANNING ON GPU USING CUDA

submitted by **SEÇKİN SANCI** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Gradute School of **Natural and Applied Sciences** _____

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** _____

Assoc. Prof. Dr. Veysi İşler
Supervisor, **Computer Engineering Dept., METU** _____

Examining Committee Members:

Prof. Dr. Faruk Polat
Computer Engineering Dept., METU _____

Assoc. Prof. Dr. Veysi İşler
Computer Engineering Dept., METU _____

Prof. Dr. Cevdet Aykanat
Computer Engineering Dept., Bilkent University _____

Prof. Dr. İ. Hakkı Toroslu
Computer Engineering Dept., METU _____

Asst. Prof. Dr. Pınar Şenkul
Computer Engineering Dept., METU _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: SEÇKİN SANCI

Signature :

ABSTRACT

A PARALLEL ALGORITHM FOR FLIGHT ROUTE PLANNING ON GPU USING CUDA

Sancı, Seçkin
M.S., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Veysi İşler

April 2010, 96 pages

Aerial surveillance missions require a geographical region known as the area of interest to be inspected. The route that the aerial reconnaissance vehicle will follow is known as the flight route. Flight route planning operation has to be done before the actual mission is executed. A flight route may consist of hundreds of pre-defined geographical positions called waypoints. The optimal flight route planning manages to find a tour passing through all of the waypoints by covering the minimum possible distance. Due to the combinatorial nature of the problem it is impractical to devise a solution using brute force approaches. This study presents a strategy to find a cost effective and near-optimal solution to the flight route planning problem. The proposed approach is implemented on GPU using CUDA.

Keywords: Aerial Surveillance, Flight Route Planning, Waypoint, GPU, CUDA

ÖZ

UÇUŞ ROTASI PLANLAMASI İÇİN GRAFİK İŞLEMCİ ÜZERİNDE ÇALIŞAN VE CUDA KULLANAN PARALEL BİR ALGORİTMA

Sancı, Seçkin
Yüksek Lisans, Bilgisayar Mühendisliği
Tez Yöneticisi : Doç. Dr. Veysi İşler

Nisan 2010, 96 sayfa

Havadan gözetleme görevleri ilgi alanı olarak da bilinen bir coğrafi bölgenin incelenmesini gerektirir. Hava keşif aracının izlediği yol uçuş rotası olarak bilinir. Uçuş rotasının planlanması asıl görevin gerçekleştirilmesinden önce yapılmalıdır. Bir uçuş rotası geçiş noktası olarak bilinen önceden belirlenmiş yüzlerce coğrafi noktadan oluşabilir. İdeal uçuş rotası planlaması bütün geçiş noktalarınından mümkün olan en kısa mesafeyi katederek geçen bir tur bulmayı başarır. Problemin faktöriyel yapısından ötürü her olası çözümün denenmesi pratik değildir. Bu çalışma uçuş rotası planlaması problemine düşük maliyetli ve neredeyse ideal bir çözüm sunmaktadır. Önerilen yaklaşım CUDA kullanılarak grafik işlemci üzerinde uygulanmıştır.

Anahtar Kelimeler: Havadan Gözetleme, Uçuş Rotası Planlaması, Geçiş Noktası, Grafik İşlemci, CUDA

To my family

ACKNOWLEDGMENTS

I am honored to present my special thanks and deepest gratitude to my supervisor Assoc. Prof. Dr. Veysi İşler for all his guidance and support during this study. I would like to thank to STM for providing me time and resources whenever I needed. Finally, I would like to thank my family for all their life-long support.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation of the Study

Actually coming from the French word for “watching over” *Surveillance* can be described as monitoring the activities of entities such as individuals, groups of people or manned / unmanned vehicles often in a surreptitious manner. The act of surveillance is usually carried out legally by governmental bodies. Surveillance has always been an issue in military organizations, also. Types of surveillance are named according to the medium they are carried on. For example while telephone surveillance means collecting data from wired or wireless phone lines, satellite surveillance corresponds to analyzing the transmissions passing through or images taken from satellites orbiting the earth.

Recalling the naming conventions of surveillance types the main concern of this study, aerial surveillance, means the surveillance activity conducted from air. Being a subset of regional surveillance it can be defined as; gathering data from an airborne vehicle. Although it seems as a new technology aerial surveillance has a long history starting from late 1700s in terms of reconnaissance balloons as the leaders of revolutionary French became interested in using the balloons to observe enemy maneuvers. Following the invention of photography, the era of aerial photographs has begun. These photographs were first taken from manned and unmanned balloons, and then from reconnaissance kites (Figure 1.1).

Figure 1.1: Labruguière photographed by kite in 1889

While developing parallel with the current technology, aerial surveillance has taken out of ordinary ways also. On some occasions rockets or even pigeons attached with cameras were used. However the next main step aerial surveillance has come with the advent of airplanes. The first use of airplanes in aerial reconnaissance mission was made during the Italian-Turkish War of 1911–1912 in Libya. The aerial surveillance continued in terms of visual and photographic reconnaissance through the First World War. The Second World War brought the concept of fast, small aircraft which would use their speed and high service ceiling to avoid detection and interception. These new airplanes had no armament but extra fuel cells and were capable of recording video along with photographs. The era of Cold War led to the development of highly specialized and secretive strategic reconnaissance aircraft also known as spy planes which were capable of flying at extreme speed and altitude.

Today's aerial surveillance is defined as gathering of surveillance data in the forms of visual imagery or video by unmanned aerial vehicles or reconnaissance aircraft such as spy planes or helicopters. Advances in digital imaging technology and

hardware capabilities made very high resolution imagery possible (Figure 1.2) which means the ability of identifying objects at extremely long distances.

Figure 1.2: Camera bay of a reconnaissance Mirage III R

Among the aerial surveillance vehicles unmanned aerial vehicles are especially considered in this study. An unmanned aerial vehicle (UAV) can be defined as an aircraft that flies without a human crew on board (Figure 1.3). UAVs may be put in two categories where the vehicle can be controlled from a remote location or fly autonomously based on pre-defined flight plan. UAVs considered in this study fall to the second type of vehicles.

Figure 1.3: Shadow 600, a reconnaissance UAV

Mission plans used in UAVs are produced using software packages called Mission Planning System (MPS). Aim of MPSs is supplying the reconnaissance mission with an application for pre-flight planning, mission rehearsal and post-flight debriefing (Figure 1.4). The ability for military agencies to have comprehensive mission planning system is vital especially when the missions are highly critical. Effective mission planning systems should offer a thorough understanding of each potential situation and any potential items that might put the mission in danger.

Figure 1.4: Sample MPS screen

In terms of aerial surveillance missions carried out with unmanned aerial vehicles, devising a mission plan is both simple and complicated. It is simple since it does not include scenarios such as weapons delivery, airdrop planning and threat analysis. However it is complicated because the main action of the mission plan, flight route planning, is highly critical and should consider fuel consumption while trying to search the maximum part of the area of interest.

This study mainly aims to devise a strategy that tries to reach a near optimal solution to the flight route planning problem while maintaining cost effectiveness in terms of processing time.

1.2 Problem Overview

The aerial surveillance mission of an unmanned aerial vehicle (UAV) is defined simply in three steps: start flying from the initial point, fly through the area of interest (AI) and land on the final point where the final point may as well be the initial point. The AI is defined as the smallest rectangular area that includes the actual land/sea territory that the mission will be flown over. The process of selecting the path to be flown during the mission is called flight route planning. The UAV flies through a pre-planned flight route. The flight route consists of waypoints (Figure 1.5). Waypoints are defined as points that must be visited in the AI in order to ensure that the entire AI is covered during the mission. The search spacing between waypoints is pre-known and is based on the expected optical detection range of the camera of the UAV at a given altitude. Waypoints might be assigned priorities showing their importance for the mission. The waypoints should be flown to in the planned order to complete the mission.

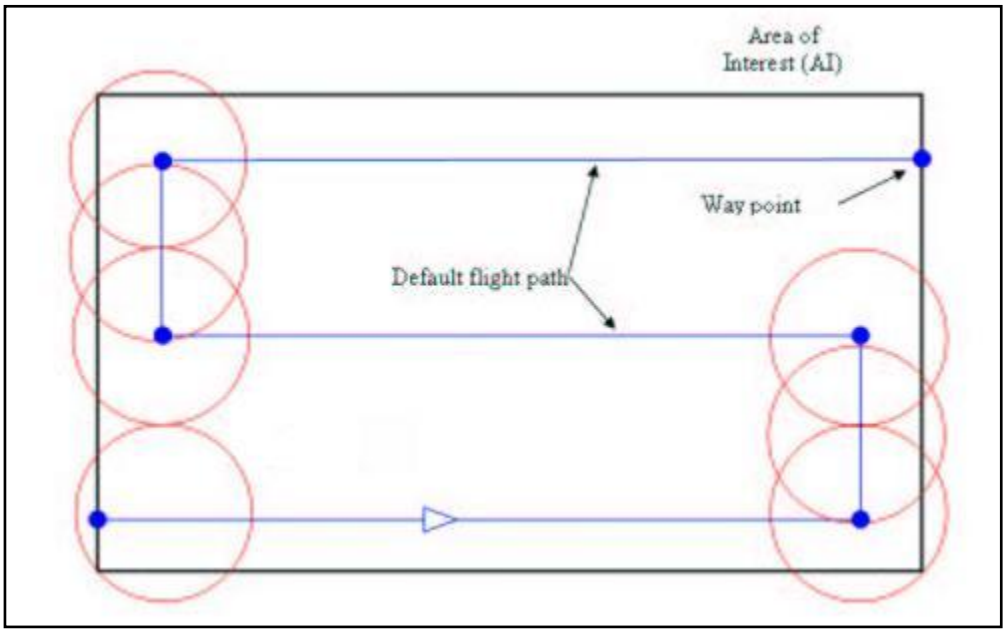


Figure 1.5: Sample Flight Route

The optimal result of the aerial surveillance mission is to search the entire area of interest through the shortest path, in minimum time and with minimum fuel consumption. The practical aim of this study is coming as close as possible to the optimal result however this may not be possible. Depending on time and/or fuel concerns the UAV may not be able to search the entire AI.

In the real world, flight route planning is a 3-dimensional problem consisting of multiple criteria such as time, fuel and mission goals to be concerned. However to find a practical solution it is imperative to simplify the problem to a 2-dimensional shortest path finding problem. Finding the shortest path also means finding the most cost effective path in terms of time and fuel consumption. Despite this simplification calculating the optimal flight route remains a computationally time consuming operation due to the combinatory nature of the problem. Searching through the numerous possible flight routes will demand substantial amount of time especially when the AI and number of waypoints are large. Knowing that surveillance data is valuable and meaningful only for a definite time, this kind of

delay is unacceptable. So that this study aims to devise a strategy to find an almost optimal solution to the flight route planning problem in an acceptable computation time.

1.3 Proposed Solution

This study aims to find a cost effective and practical solution to the flight route planning problem mentioned in the previous section. The first step in the proposed solution is to reduce the 3-dimensional problem to a 2-dimensional shortest path finding problem in order to convert it to Travelling Salesman Problem (TSP). In this study an augmentation of the TSP will be considered where cities in the original problem are replaced by the waypoints defined on the area of interest (AI). The waypoints are assumed to be stationary according to the nature of the flight route planning problem. In the proposed strategy a genetic algorithm is applied to the whole TSP along with a greedy approach for making local decisions. The final algorithm is compiled both on a standard C compiler and CUDA C compiler in order to verify the performance gained using the highly parallel architecture of graphical processing unit (GPU). The components of the proposed algorithm are explained in the following paragraphs.

The **Travelling Salesman Problem** (**TSP**) in computer science is a combinatorial optimization problem in which for a given list of points and their pair wise distances, a shortest possible path that visits all the points is searched (Figure 1.6). For simplicity TSP can be modeled as a weighed graph in which vertices denotes the points and edges the paths, the path distances are shown as edges' length.

Figure 1.6: Original Travelling Salesman Problem

Travelling Salesman Problem has been shown to be NP-hard which means brute force approaches to solve the problem will be fruitless. The direct solution in which all possible paths are tested to be the shortest has a complexity of $O(N!)$ so it doesn't provide any practical solution for input numbers more than 20. With this knowledge at hand, heuristics and approximation algorithms are used for devising practical solutions to the TSP.

A **genetic algorithm (GA)** is an approximation technique used in computer science to find exact or near-exact solutions to searching problems. GAs use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover. Execution of GAs can be summarized as follows:

1. An initial group is selected from candidates set
2. Each individual in the group is tested according to some criteria

3. Following actions are taken repeatedly on the group until termination condition

   i. Select the most appropriate individuals for reproduction
   ii. Produce new individuals through crossover and mutation operations
   iii. Evaluate the new individuals according to the criteria
   iv. Replace least appropriate members of population with new individuals

Applying a GA to NP-hard problems such as the proposed TSP does not ensure an optimal solution. However it usually gives good approximations in a reasonable amount of time and near-perfect results with smaller data sets. In this study the proposed GA will be supported by using greedy heuristics.

A **greedy algorithm** is an algorithm that uses the method of making the decision of selecting the locally optimal choice at each stage of execution in order to find the globally optimum solution. In general greedy algorithms maintain;

1. A candidate set that contains the possible optimal solution
2. A selection function to choose the best candidate and
3. A solution function to decide if the final solution is reached

Applying a greedy algorithm to the proposed TSP will result in going to the nearest point to the current one at each step of execution. For NP-complete cases like the TSP, greedy algorithms do not always find optimum solutions. However, they work quickly and usually give practical approximations to the optimum solution.

The proposed algorithm of this study will be implemented on GPU using CUDA. **Compute Unified Device Architecture** (CUDA) is a general purpose parallel computing architecture developed by NVIDIA. CUDA mainly exploits the parallel compute engine in NVIDIA graphics processing units (GPUs) to solve computational problems. CUDA includes a unique Instruction Set Architecture (ISA) and the parallel compute engine in the GPU. Programming CUDA is through

using 'C for CUDA', a C-like language which has some extensions over the standard C language.

1.4 Organization and Roadmap

Chapter 2 discusses the previous work done on flight route planning problem addressing their different approaches and also the proposed solutions to the TSP and other NP-hard problems in general. This chapter also gives knowledge about improvements achieved using CUDA in different algorithms.

Chapter 3 presents the formal mathematical definition of the flight route planning problem

Chapter 4 explains the proposed algorithm for the problem along with implementation details

Chapter 5 verifies the solution by showing the results of extensive testing done for different situations of the problem

Chapter 6 summarizes the study and concludes it by bringing forward the possible future improvements that can be done over the solution

CHAPTER 2

RELATED WORK

This chapter presents the previous work done on flight route planning problem and the sub-parts of the proposed solution such as: studies on TSP, Genetic Algorithms, Greedy Algorithms, CUDA, Parallel Genetic Algorithms and Computational Geometry. These parts are examined in a comparative manner and according to their relevance to the proposed problem and solution.

2.1 Different Approaches to Flight Route Planning Problem

As discussed in the previous sections, main strategy of this study is reducing the 3-dimensional flight route planning problem to a 2-dimensional shortest path finding problem, namely the Travelling Salesman Problem (TSP). However there are other approaches for solving the flight route planning problem. These will be discussed in the following paragraphs in detail.

One such approach is the Ant Colony Optimization, a meta-heuristic introduced by Dorigo et al. in 1991. For solving the TSP (Dorigo, 1992) the Ant Colony Optimization takes the natural behavior of real ants as a model. In the real world ants communicate through detecting pheromone trails. While an ant passes through a route it leaves a sample amount of pheromone onto the path. These pheromones however, are temporary and will lose their intensity in time due to natural causes such as evaporation, emission by soil etc. Ants show tendency to follow the paths where intensity of pheromone trails is higher which means the higher density of

pheromone trails on a road the more attractive that road will be. The Ant Colony Optimization technique is known to show good performances for the TSP especially for smaller data sets. However it is also known that execution times increase drastically for considerably large domains.

Despite this fact Ant Colony Optimization technique has a high potential for a parallel implementation (Talbi, Roux, Fonlupt, & Robillard, 1999; Stutzle, 1998). Keeping this in mind, You, in his study tried to exploit the highly parallelizable structure of Ant Colony Optimization technique using NVIDIA's CUDA programming model [1]. In this study a slightly altered version of the native Ant Colony Optimization technique was used. The "artificial" ants used in the study are chosen to have list of locations that were visited before in order not to make trivial second visits. Besides keeping the list Ant Colony Optimization technique adds a local search procedure in order to satisfy a more efficient selection of the next path to follow. This approach resembles the use of Greedy Algorithm in the strategy used by this study, again in order to enhance efficiency of the algorithm.

Another approach for solving the flight route planning problem considers an altered version of the problem. In their study (Cross, Marlow & Looker) puts forward a more dynamic approach to the flight route planning problem. They aim to classify the previously detected entities within an Area of Interest (AI) using the radar capabilities of a surveillance aircraft. For this purpose they aim to plan a route that will take entities into the radar range and fly through these entities in a pre-defined order. Another constraint is that the entities are assumed to be non-stationary [2]. They argue that the current methodology to solve this surveillance problem considering only the section of the AI between the current waypoints and stationary entities may not be enough to devise an acceptable solution to the specific case they explained (Figure 2.1). So instead of reducing the flight route planning problem to TSP they consider a Dynamic TSP where the points of interest in the original problem are assumed to be moving.

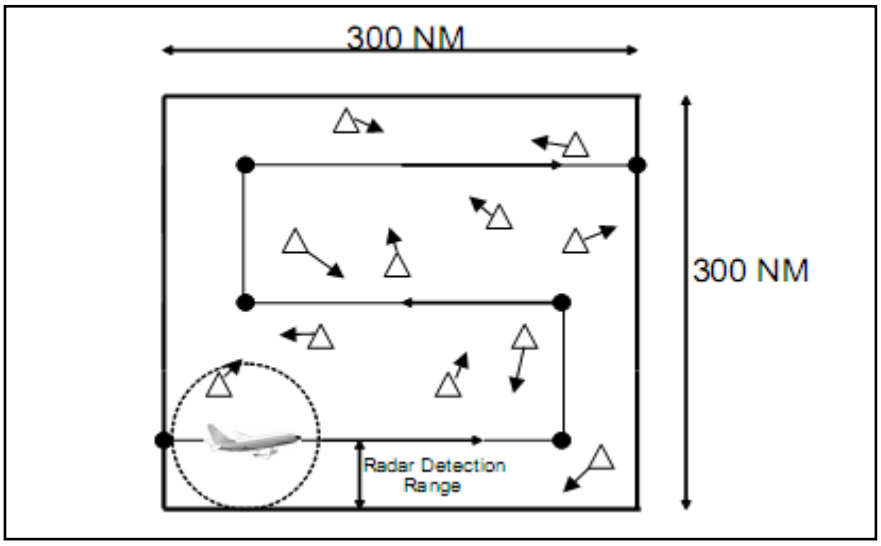


Figure 2.1: Surveillance Problem Defined by Cross, Marlow & Looker

In their algorithm the surveillance aircraft plans to fly to the contacts that have not yet been classified. The contacts should also be flown to in the pre-defined order for completing the mission. However due to the dynamic structure of the problem the aircraft may deviate from the pre-planned flight routes and fly towards entities that need to be classified. The pre-defined list of entities may change as some entities will be able to move in and out of radar detection range [2].

2.2 Travelling Salesman Problem

Reducing the flight route planning problem to the well known Travelling Salesman Problem (TSP) seems to be the natural solution. Even the approaches that does not use the TSP directly, mostly use the ideas behind the TSP to construct their own model for solution or mention it at least as a comparison method. The studies discussed in this section use the TSP as the main approach to solve the flight route planning problem and have significant resemblances to the main strategy devised in this study. However there are some differences which usually stem from the nature of different application areas of the flight route planning problem.

In their study (Marlow, D.O. , P. Kilby and G. N. Mercer, 2009) searches for a strategy for solving an aerial maritime surveillance problem. That study deals with searching for and identifying ships that sail in an Area of Interest (AI). The study expands the concept of TSP by adding some concepts such as ships moving with random velocities (dynamic TSP), different start points for surveillance aircraft (open TSP) and incomplete a pre-knowledge of the AI (on-line TSP) [4]. However in implementation they reduce the problem to an augmentation of the traditional Travelling Salesman Problem (TSP). The study also makes use of new ideas such as alternative default headings and introduction of ghost ships to the problem space.

Introducing the use of alternative default headings to the TSP yields three different possible implementations of the path followed to reach a waypoint (Figure 2.2). The first implementation is direct-to-waypoint default heading which minimizes the travel time; however this implementation may result in not covering the entire AI. The alternative implementation to direct-to-waypoint default heading is perpendicular return to the way line implementation which is likely to increase the percentage of AI to be searched however definitely increasing the distance travelled and as a result amount of fuel consumed. A third implementation is doing a "midway" return which is as its name suggests a middle way move between the direct-to-waypoint heading and the perpendicular return. In this implementation the aircraft aims to reach the midpoint on the way line of the perpendicular intercept point and the waypoint [4]. The strategy described in this study will make use of the direct to the waypoint implementation as fuel consumption is the primary priority.

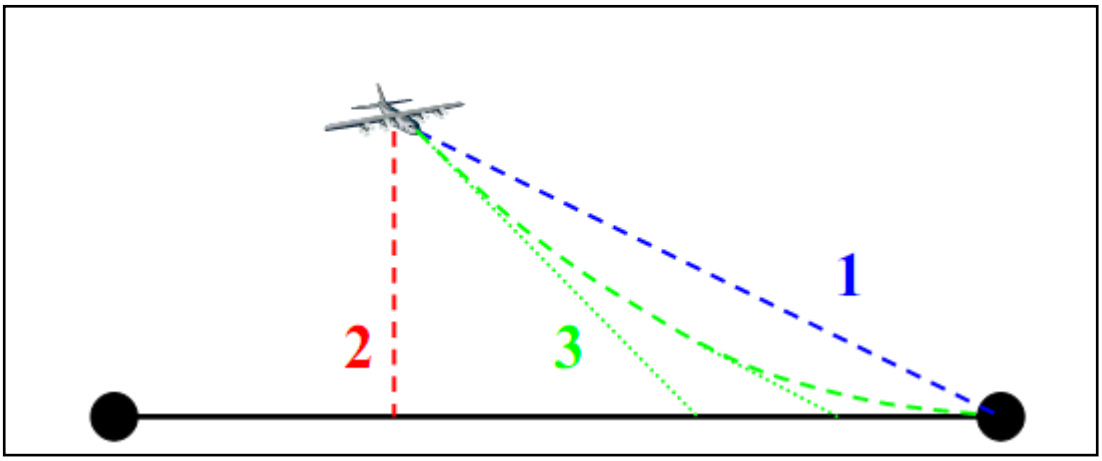


Figure 2.2: Examples of various default heading options

1) Direct to the waypoint

2) Perpendicular return

3) Midway return

The purpose of adding ghost ships to the AI is to make sure that the surveillance aircraft searches the areas that are unlikely to be visited using the pre-defined flight path. In case of an AI with sparsely distributed waypoints a significant percentage of the AI is likely not to be searched (Figure 2.3). This approach is worth to be implemented if the main concern is the amount of region to be searched. Results of the study indicates that the perpendicular default heading and including ghost ships provide improvement in the condition where the waypoints in the AI has low density and/or are concentrated in some part of the AI [4]. For the concept of this thesis the use of a mechanism such as ghost ships may be considered when the main purpose of the mission is searching the highest percentage of the AI as much as possible.

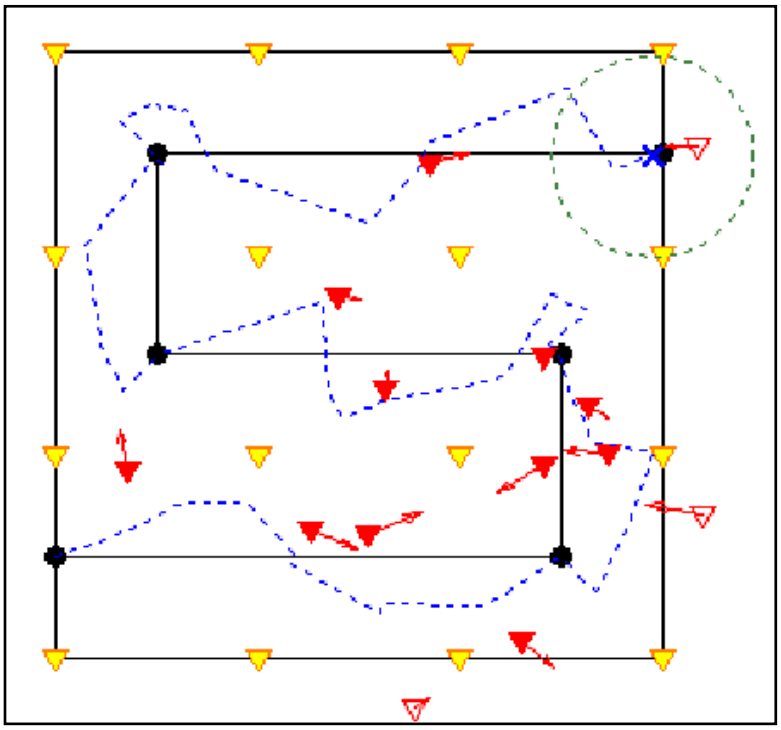


Figure 2.3: Effect of adding ghost ships

Planned flight path without ghost ships (solid line)

Altered flight path due to ghost ships (dashed line)

Ghost Ships (triangles on the edges of the AI)

2.3 Genetic Algorithms

As it was mentioned in the previous sections Travelling Salesman Problem (TSP) seems to be the natural solution to the flight route planning problem. However it is known that due to the combinatorial nature of the TSP it is fruitless to try to solve the problem using exhaustive brute-force methods. Also discussed in the previous sections that genetic algorithms offer practical approximate solutions to NP-Hard problems, such as the TSP discussed in this study.

Genetic algorithms (GA) as an optimization technique reflect principles of the natural evolution. The main principle of behind the GAs is “the survival of the fittest”. This principle provides a mechanism to search for a near optimal solution without going through all possible solutions. Genetic algorithms follow the steps of the natural process of evolution.

As the fittest individuals survive in the nature a GA lets the best guesses to the solution pass to the next generation. The GAs first guess solutions then test them and create a new generation of solutions from the fittest solutions which are expected to be better ones.

The steps of a GA are as follows:

- Evaluation
- Crossover
- Mutation

In the implementation of a GA first an initial population is selected, usually randomly. Then each individual’s fitness is computed. This fitness is used to find which individuals will be considered for crossover. Crossover is the process where two individuals are combined to create new individuals. Then mutation occurs as some individuals are chosen randomly to be mutated. After mutation the next generation is formed. The whole process is repeated until some stopping criteria. At this point the individual which is the fittest according to the criteria proposed is selected as the final solution.

In his study (Bryant, 2000) states that different forms of crossover and mutation techniques can be combined to give various genetic algorithms that can be used to solve the TSP [6].

However, he says that these methods have been tested on different problems and it will therefore be difficult to compare them to each other.

Bryant's study first gives the optimal solutions of some TSP problems for sample number of cities and uses these costs to test the different crossover methods. In order to test only the crossover algorithms the study uses no heuristic information. The compared crossover methods are partially modified crossover, order crossover and matrix crossover. The result of the comparison favors the matrix crossover method over the two other methods. The matrix crossover method gave just %2 more than the optimal case on average for different situations.

The main difference of matrix crossover method is that it uses a matrix representation to hold the city-to-city distance instead of holding the positions of cities. However the study states that the matrix representation used in the method takes more space to store and also more computation time [6].

After testing the non-heuristic methods the study looks at a heuristic crossover also and finds out that the heuristic method sometimes gives the best known solution for that particular problem, and otherwise returns a solution very close to that value [6]. The study concludes by stating that GAs gives good approximations for the TSP especially when matrix or heuristic crossover methods are used. The strategy devised in this thesis considers a heuristic crossover approach.

In their work (Maria John, David Panton and Kevin White, 2001) , like most of the other studies, stated that the flight route planning problem can be reduced to the TSP, citing that it is a computationally time consuming problem and not suitable for an approach using standard optimization techniques [7].

The study focuses on formulating the problem both as an integer programming problem based on grid squares, already knowing that this model will consume significant amount of execution time especially for larger input sets, and a Genetic Algorithm (GA) based on permutations of these grid squares. The main trade off described in the study is the one between the definitely optimal solution of the integer programming method and the shorter execution time of the GA based solution. The two methods are tested for different input sets and the results of the study show that while the integer programming method finds the optimal solution, the GA based approach scored a performance on finding the shortest path problem

only %1.3 to 2.7 outside the optimal solution. However in terms of execution time the GA based approach achieved a speed up of 10 to 30 times. The study is concluded by declaring that for the concept of regional surveillance creating the mission plan in a small amount of time is an operational advantage and in the view of this statement a GA is operationally superior to the integer programming model [7].

2.4 Greedy Algorithms

As it was mentioned in the previous sections Travelling Salesman Problem (TSP) seems to be the natural solution to the flight route planning problem and Genetic Algorithms can be seen as a practical solution to this. While the GAs are quite efficient in terms of execution time, the choice of how to make the local decisions in the algorithm may affect performance. Using a purely random approach may seem natural; however it causes lower performance compared to heuristic methods. The strategy described in this study uses a Greedy approach to make local decisions which arise during the execution of the GA.

In his work (Bryant, 2000) mentions the greedy algorithms citing that since there is no known algorithm that will solve NP-hard problems in polynomial time, the aim of trying to find the optimal solution should be sacrificed in order to devise a solution that needs a shorter execution time [6]. The study puts forward a Greedy Algorithm as a solution method to the TSP.

In the study Greedy Algorithms are described as a method that can find a feasible solution to the TSP. The algorithm defined in the study works as follows:

- Distances between the cities in the TSP is modeled as a graph where the vertices denote the cities and the edges the paths between the cities
- The distance between two cities is assigned as the cost of the edge of the graph which denotes the path between those two cities
- A list of all edges in the graph is created

- Created list is ordered from smallest cost to largest cost
- The edges with smallest cost is selected repeatedly providing they do not create a cycle

The study concludes its discussion about the Greedy algorithms by mentioning that although the greedy algorithms work fast to find a solution, as they are more concerned about selecting the local minimum distance they may lose the notion of finding a global best solution and in practical the cost of final edges that are added to the solution are usually quite large [6].

2.5 CUDA

The processor of a graphics card, namely the GPU is much different than the CPU in terms of internal structure. Instead of having at most four processing cores like a CPU, the GPUs include hundreds of small processing units, called pipelines, which makes them suitable for running parallel applications with great efficiency. The new approach in the graphics card design is implementing the unified shader pipeline which allows the GPU to execute all kinds of operations that the CPU can. This innovation led to the development of CUDA (Compute Unified Device Architecture) by NVIDIA. CUDA is a parallel programming model and software environment that offers the programmers to develop their programs in standard programming languages such as C that will use the capabilities of GPU.

In the study conducted by (Bydal, 2008) the possibility and efficiency of solving the TSP through a GA using the CUDA API is discussed. Also the problem of the performance gain using the GPU instead of the regular CPU is addressed [8].

The study gives information about CUDA citing three of its main characteristics; hierarchy of thread groups, shared memories and barrier synchronization and adds that these are transferred into the C programming language using a set of extensions where these extensions are used by the programmer to create parallelism in the application. The programming model of CUDA is also mentioned in the study shortly, where the execution application covers the following steps:

- The application is started by the CPU
- The execution is passed to the GPU by copying the data from the main memory to GPU's own memory
- After that point GPU makes all operations on the data
- When the GPU finishes execution the results are copied back to main memory

The study ends with presenting the results of the implementation. It is seen from the results that the parallel execution run on the GPU does not provide an increase in the efficiency for smaller data sets. However after number cities in the TSP passes 50 shows its superiority over the serial implementation run on CPU. In terms of execution times while the time consumed by the two implementations is almost equal for input sizes around 50, the CUDA implementation achieves 3 times speed up for an input size of 128 [8].

Although it does not use the TSP directly the work of, You, makes the implementation of Ant Colony Optimization approach using CUDA. Mentioning the importance of memory arrangement of GPU while using CUDA, because of the large access latency difference between the CPU and GPU, the study defines some rules for accessing the memory which are as follows:

- Frequently accessed data is stored on share memory
- Write-once data, is stored on global memory
- Read-only data is stored on GPU's texture cache [1]

The study is concluded by saying that improvement of running time grows as the complexity of problem grows as it was in the previous study. The strategy devised in this thesis uses a similar approach towards implementing a GA using CUDA as in the work of (Bydal, 2008) and some rules for memory management as they were used in, You.

2.6 Parallel Genetic Algorithms

Parallel Genetic Algorithms is a variant of the Genetic Algorithms, where there is more than one initial population and these populations are evolved concurrently. The study of (Nowostawski and Poli, 1999) describes the Parallel Genetic Algorithms in comparison to Serial Genetic algorithms and gives further information about types of Parallel Genetic Algorithms [16].

The study describes Genetic Algorithms as a successful approach for many applications in different domains, but it also says that the approach has some utilization problems that can be overcome with a parallel implementation [16]. The problems addressed for Serial Genetic Algorithms are:

- Large population sizes may require considerable memory to store, such that to run an application efficiently a Parallel Genetic Algorithm becomes necessary.
- Evaluating fitness may require very long time for large input sets making use of a single CPU impractical.
- Serial Genetic Algorithms may converge to a local optimum in the search space making Parallel Genetic Algorithms a solution with a better chance to find a nearer solution to the global optimum [16].

The study goes on with describing various kinds of Parallel Genetic Algorithms and devises taxonomy for differentiating the approaches. The constraints defined in the taxonomy are:

- Method of evaluating fitness and using mutation/migration
- Number of initial populations
- Interaction between populations
- Rules of parent selection

Using the constraints listed above the study divides the Parallel Genetic Algorithms into eight types:

Master-Slave parallelization, Static subpopulations with migration, Static overlapping subpopulations, Massively parallel genetic algorithms, Dynamic demes, Parallel steady-state genetic algorithms, Parallel messy genetic algorithms, Hybrid methods [16].

2.7 Computational Geometry

Computational geometry is the branch of computer science that deals with algorithms which can be expressed in terms of geometrical concepts. The main motivation behind the development of computational geometry is the suitability of problems in the areas of computer graphics, CAD/CAM and GIS applications to be solved by geometric approaches. One of the core problems in computational geometry is the "Euclidean Shortest Path Problem". This problem is analogous to the flight route planning problem discussed in this study.

In their study (Hershberger and Suri, 1997) propose an optimal-time algorithm for computing the shortest path between two points on a plane in the presence of polygonal obstacles [14]. Their algorithm uses an implementation of continuous version of Dijkstra's method and wavefront propagation. The approach uses a preprocessing method which they call conforming subdivision of the plane which divides the plane into cells using horizontal and vertical edges in order to leave each point in a separate cell. The algorithm then inserts the line segments of the obstacles into the cells and uses this structure for propagation algorithm. The study states that passing the plane cell by cell into the wavefront propagation instead of passing it as a whole works more efficiently. The study also adds another idea into the algorithm which is called approximate propagation. This idea is used in order to eliminate the cost of detecting obstacle edges within the cells and uses two wavefronts, instead of one, that approach the edge from opposite directions and estimates the positions of the edges rather than calculating them exactly. At the end of the propagation phase the algorithm collects the results from each cell and uses these results to construct the final shortest path map [14].

Unlike the previous study, (Agarwal, Sharathkumar and Yu, 2009) aim to find an approximate shortest path for the Euclidean Shortest Path Problem in their work [15]. The study aims to devise a linear solution for the proposed problem. The algorithm first tries to create a layout of the plane along with the obstacles, which they call the “sketch”, and then try to find an approximate shortest path using this layout. Creating the sketch works as basically constructing a convex polygon for each obstacle where the polygon contains the obstacle. Then the algorithm travels through the edges of these polygons in order to compute a shortest path through these obstacles [15].

CHAPTER 3

FORMAL PROBLEM DEFINITION

This chapter presents formal mathematical definition of the flight route planning problem along with a description of the main actors of regional surveillance, namely the surveillance mission and the surveillance aircraft. After this, a problem model is constructed first by reducing the flight route planning problem to known computational problems and then introducing regional surveillance specific concepts to the proposed model. The model will be later used for devising the solution in the following chapters.

3.1 The Aerial Surveillance Mission

The basic objective of the aerial surveillance mission is to monitor a given geographic area (Figure 3.1). A particular region is assigned to a surveillance aircraft, which is called the Area of Interest. This study considers a single aircraft and a single rectangular AI.

Type of the aircraft is independent of the problem proposed in this thesis, however using real world values for the aircraft is imperative to get realistic results. For that reason TIHA surveillance aircraft is chosen as the model vehicle for this study. TIHA is an unmanned aerial vehicle being developed for the Turkish Armed Forces by TAI being the main contractor. The reasons for this selection is that; first TIHA

is a national project and second the author of this thesis has a high chance for working in the Mission Planning System that is going to be developed for the TIHA aircraft. Technical details of the aircraft will be given in the following section.

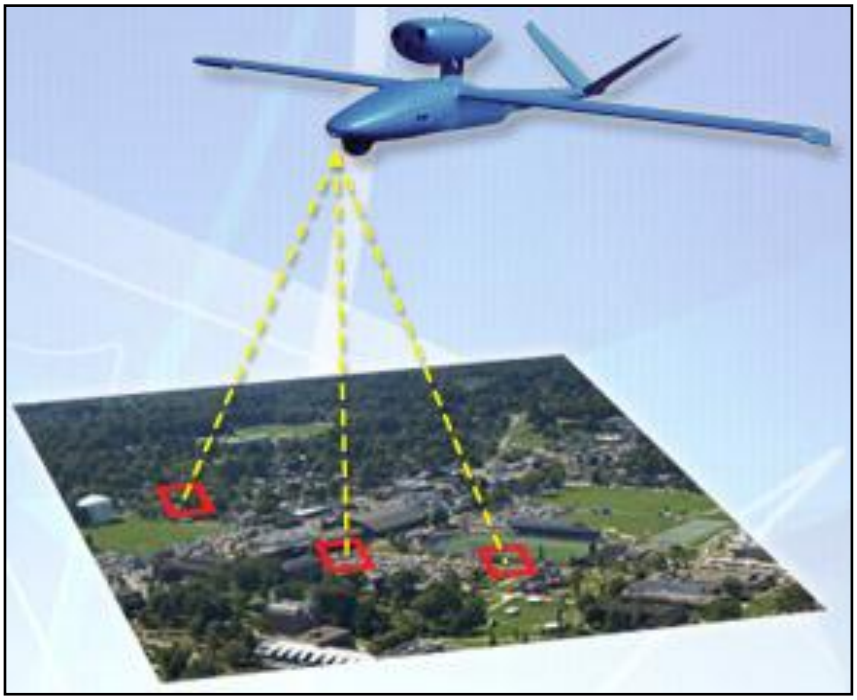


Figure 3.1: Sample Aerial Surveillance Mission

Each aerial surveillance mission has a flight route. The flight route is defined as a list of waypoints, known at the start of a mission. To complete the mission successfully all waypoints must be visited and they should be flown in the pre-defined order. The final waypoint may or may not be the starting location of the mission. The maximum length of the flight route is bounded by the maximum distance that can be travelled by the TIHA aircraft. This maximum length is formulated mathematically in this study.

In order to present a mathematical formulation of the flight route the following definitions are necessary:

Velocity (*v*): The speed that the surveillance aircraft flies during the mission, it is assumed to be constant.

Number of waypoints (*n*): Total number of points to be visited during the mission.

Path ($x_{(ij)}$): The distance between the i^{th} and j^{th} waypoint. For *n* waypoints there exist (n-1) paths in a flight route.

Total Route Length (*R*): Total distance of the paths considered for the flight route. As the waypoints are sorted while constructing the flight path, it equals the total length of the paths between consecutive waypoints. The total route length can be formulated as:

$$\sum_{i=1}^{n-1} x_{i(i+1)} \tag{3.1}$$

Total mission time (*T*): Time to complete the mission, equals to the ratio of total route length to aircraft's velocity. The total mission time can be formulated as:

$$T = R / v \tag{3.2}$$

Max Range (R_{max}): The maximum distance that the aircraft is capable of flying during mission.

Max mission duration (T_{max}): The maximum duration that the aircraft is capable of flying during mission.

3.2 The Surveillance Aircraft

TIHA, coming from the abbreviation of Turkish words *Türk İnsansız Hava Aracı* (Turkish Unmanned Aerial Vehicle) is a Medium Altitude Long Endurance (MALE) Unmanned Aerial Vehicle (UAV). The formal name of the aircraft is TURKISH INDIGENOUS MALE UAV. The contract concerning the development of the Turkish Indigenous Medium Altitude Long Endurance (MALE) Unmanned Aerial Vehicle (UAV) – TIHA Program was signed between the Under secretariat for Defense Industries (SSM) and TAI on December 24, 2004, to meet the requirements of Turkish Armed Forces (TAF). The program covers design, development, production, test and delivery of three TIHA air vehicle prototypes, all related support and ground systems and technical documentation. The program is composed of four steps:

- Conceptual Design
- Preliminary Design
- Detail Design and Development
- Tests and Evaluation

Following the final integration of the system, the maiden flight of TIHA is planned to be in 2009.

The TIHA system consists of:

- Aerial Platform
- Ground Control Station (GCS)
- Transportable, Image Exploitation Station (TIES)
- Portable Video Terminal (PVT)
- Ground Support Equipment (GSE)

The system composition is shown in the figure given in Figure 3.2.

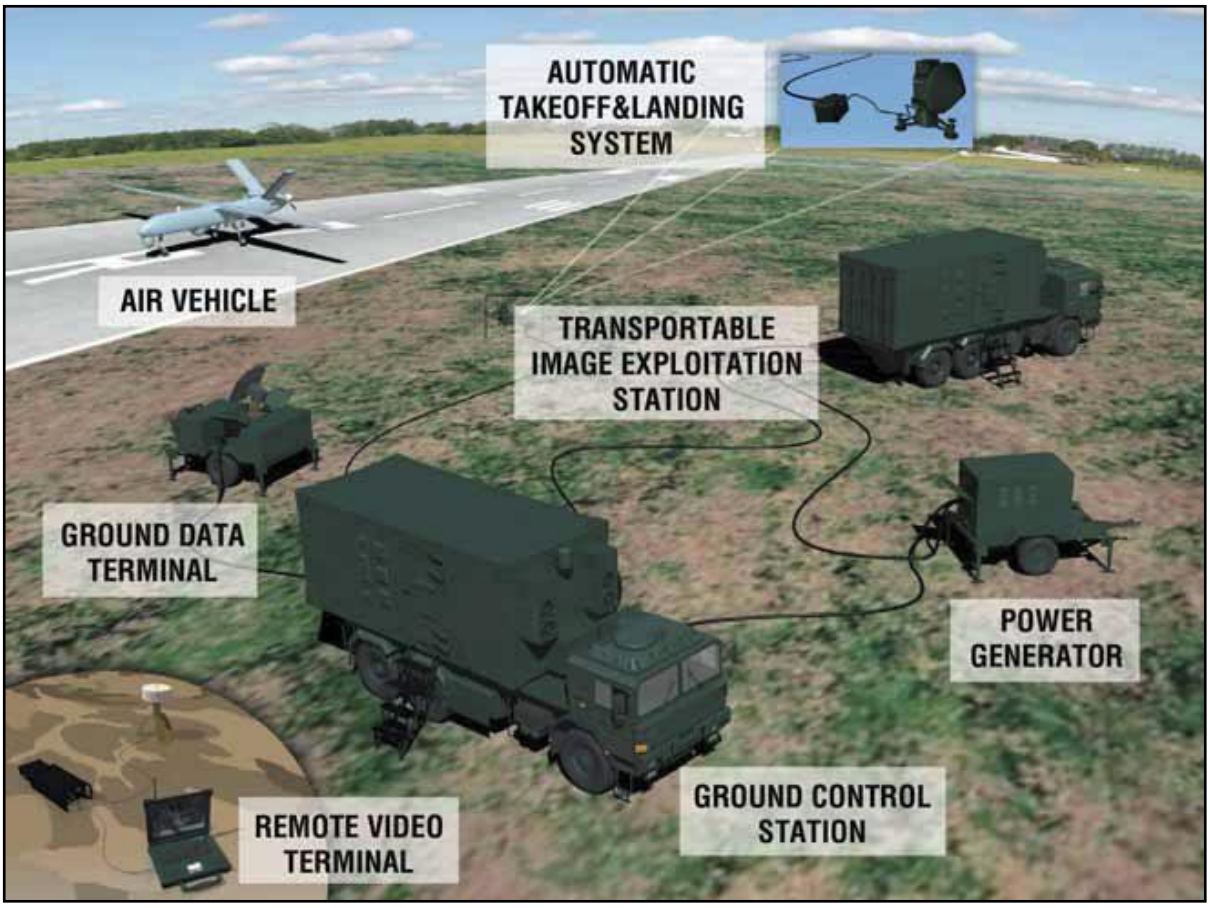


Figure 3.2: TIHA System Composition

The TIHA system is developed for day and night real time image intelligence for surveillance, reconnaissance, fixed/moving target detection, identification and tracking missions. To accomplish these capabilities TIHA aircraft will be equipped with the following payload:

- Electro-Optical Day Camera (EO Day TV),
- Day Camera (EO-Electro Optic) / Thermal Camera (IR-Infrared) / LRF-Laser Range Finder & LD-Laser Designator and Spotter,
- SAR-Synthetic Aperture Radar / MTI- Moving Target Indicator & ISAR-Inverse SAR

Figure 3.3: TIHA Aircraft on a Mission

Full composite airframe is composed of monocock fuselage, detachable wing and V-Tail, retractable landing gear, equipment bays, service doors and other structural components. The air vehicle is powered by a pusher type piston-prop propulsion system. The airframe is equipped with miscellaneous sub systems like fuel system; de/anti-ice devices; environmental conditioning system for cooling/heating requirements of the compartments (Figure 3.3).

The avionics system includes a Flight Management System (FMS); integrated to FMS, flight sensors (pitot-static sensor, embedded GPS/INS, magnetic heading, displacement, temperature, pressure transducers), actuators; dedicated

communication and identification devices; mission control, record; and other control and interface units.

TIHA system basic performance parameters are as follows:

- Service Ceiling: 30,000 ft
- Endurance: 24 hrs
- Cruise Speed: >75 kts
- Environmental Conditions: 15 kts side wind, 20 kts head wind; temperature, humidity, rain and icing limits as defined in MIL-HDBK-310

Air vehicle's basic specifications are given in Table 3.1 [9]:

Table 3.1: TIHA Aircraft Basic Specifications

Property	**Unit**	**Value**
Fuselage Length	m	10
Wing Span	m	17
Wing Area	m^2	13.6
Wing Aspect Ratio	-	22
Wing Sweep (quarter chord)	°	0
MTOW-Maximum Take-off Weight	kg	1,500
Fuel Weight	kg	250
Payload Weight	kg	200

3.3 Problem Definition

There is no data structure that directly maps the explained real world flight route planning problem to the proposed algorithm. That is why the input space should first be mapped into a more suitable data structure. In order to do this pair wise distances between the waypoints on the Area of Interest are calculated and stored in an (n x n) symmetrical matrix. The distances stored in and accessed from the matrix

are in such a format; where d_{ij} stands for the distance between the i^{th} and j^{th} waypoints. The cost of mapping the inputs is $O(N^2)$, while the a operation of accessing the distances through waypoint indexes is $O(1)$.

Upon the reduction of the 3-dimensional input space of the flight route planning problem, it is natural to view it as a 2-dimensional shortest path finding problem. The proposed solution to this problem is an implementation of a slightly augmented Travelling Salesman Problem where the real difference is that a route created as the result of the algorithm should not finish in the initial start point. TSP is a combinatorial problem which can be stated simply as; a salesman spends his time visiting n cities. In one tour he visits each city just once, and finishes up where he started. In what order should he visit them to minimize the distance traveled? In this study the final constraint of "finishes up where he started" is not exercised due to the reasons specific to the real world problem. Definition of the TSP continues as follows;

- It is assumed that each city is reachable from all other cities.
- There exists a route between every pair of cities with a defined cost of travel between these cities.
- For only two cities then the problem is trivial, since only one tour is possible.

The TSP can also be formulated as a graph problem. For a given graph G with a set of N vertices, a list of edges in G which passes through each vertex of G once and only once is called a Hamiltonian Cycle. Assuming G is a complete weighted graph with N vertices; TSP becomes the problem of finding the shortest Hamiltonian Cycle of G.

As the input space of the of the problem is mapped to an ($n \times n$) matrix, the formal definition of the TSP considered in this study can be stated as; Given a ($n \times n$) distance matrix $C = (c_{ij})$ find a permutation (π), that is a member of all solutions ($\pi \in Sn$) that minimizes the sum [10];

$$\sum_{i=1}^{n-1} c\,\pi(i)\,\pi(i+1) + c\,\pi(n)\,\pi(1) \qquad (3.3)$$

In more detail TSP considered in this study falls to the category of symmetrical TSPs; where $c_{ij} = c_{ij}$ in the constructed distance matrix. This means that the distance between the i^{th} and j^{th} waypoints are equal in both forward and backward directions. This constraint makes the distance matrix symmetrical about the diagonal. Further the TSP considered is also an Euclidian TSP. Euclidian TSPs, although they constitute a small portion of all TSPs, can form a solution to many problems as most real-world problems can be mapped to them. The Euclidian TSPs are defined as;

$$\forall i,j \in C : c_{ij} = c_{ji} = \sqrt{a^2 + b^2}, \text{ where } a = (a_i - a_j) \text{ and } b = (b_i - b_j) \qquad (3.4)$$

Euclidian TSPs are symmetric by definition, however all symmetrical TSPs do not fall to the category of Euclidian. For the Euclidian TSPs, as they are also symmetrical TSPs, it is faster to find the solutions since only half of the solution space needs to be searched. However there is no polynomial algorithm to find the optimal solution.

In terms of complexity it is intuitive to view the TSP as a determination of a specific permutation of waypoints, numbered from 1 to N, which is a non-repeating sequence and represents the visiting order of the waypoints. The permutation is specific as it represents the visiting order for which the weight sum is minimized [11].

The search space contains N! permutations and since TSP is NP complete, problems dealing with optimizing the TSP are NP-hard. In any case the number of solutions becomes extremely large for large N, so that an exhaustive search is impractical. The best known algorithms for the solution yield exponential run time complexity. These combinatorial optimization problems are a part of the Genetic Algorithms domain [11].

To summarize the problem defined in this section the following table is constructed (Table 3.2). The table actually lists the requirements of the real world flight path

planning problem. It should be noted that the domain of the problem is a list of geographical coordinates instead of the distance matrix in order to reflect the actual real world problem. Assuming complete weighted graph G which is described above;

Table 3.2: Formalized Table of Problem Domain

Size of Solution Space	O(N!) for N geographical coordinates of waypoints
Problem Domain	D_{input}: (G) – a list of coordinates (x, y) D_{output}: (G') – an ordered permutation of G such that the total distance of the Hamiltonian cycle is minimized
Objective Function	Min g = $\sum d_{ij}$ where d_{ij} is the distance between adjacent elements of G'
Candidate Solution	G' which is an ordered permutation of G
Selection Function	This function is used to determine if G' is an actual Hamiltonian cycle. However since the TIHA is an aerial surveillance vehicle the graph G is totally connected, making all of its permutations Hamiltonian cycles. So this function is assumed to be always true.
Feasibility Function	This function is used to determine if a Hamiltonian cycle exists. However since the TIHA is an aerial surveillance vehicle the graph G is totally connected, making all of its permutations Hamiltonian cycles. So this function is assumed to be always true.
Solution Function	G' is an ordered permutation of G and Min g = $\sum d_{ij}$ where d_{ij} is the distance between adjacent elements of G'

CHAPTER 4

IMPLEMENTATION

In this chapter, implementation details of the proposed solution to the outlined flight route planning problem are described. The chapter is organized as follows: Section 4.1 gives an overview of the implementation along with the inputs and outputs. The other sections give implementation details about subparts of the proposed solution. Section 4.2 describes the Genetic Algorithm implementation for solving the Travelling Salesman Problem. Section 4.3 provides the details of Greedy Heuristic used inside the Genetic Algorithm and finally Section 4.4 gives the implementation details specific to CUDA.

4.1. Overview

The flight route planning problem, in the concept of this study, is reduced to the Travelling Salesman Problem as addressed in Chapter 3. An approach based on using a Genetic Algorithm is proposed as a solution to this problem. In the implementation a Greedy Approach is utilized in certain steps of the GA. The flight route planning solution proposed in this study is implemented as an application with two versions. In the first version the algorithm is implemented using standard C language and works on the CPU. The second version of implementation however uses NVIDIA's CUDA compiler and aims to exploit GPU processing power as much as possible. The reason for implementing two versions of the algorithm is to

demonstrate the feasibility of the parallel structure of the GPU for solving computationally time consuming problems. The two versions are implemented as logically equivalent as possible.

Although there are two versions of the application the input and outputs of these versions are exactly the same. The applications will take a list of waypoints in the XML format, as illustrated in Figure 4.1, which consists of Id and geographical coordinates for each waypoint. The other inputs for the applications will be the start and end waypoint Ids and some parameters for the execution of the Genetic Algorithm. These parameters will be discussed in the following sections.

```
<?xml version="1.0" encoding="utf-8" ?>

<WayPointList>

<Waypoint Id="1" Lat="39.12" Lon="30.25" />

< Waypoint Id="2" Lat ="38.89" Lon ="30.16" />

< Waypoint Id="3" Lat ="39.32" Lon ="32.11" />

< Waypoint Id="4" Lat ="39.11" Lon="29.92" />

< Waypoint Id="5" Lat ="40.25" Lon="30.45" />

</WayPointList>
```

Figure 4.1: Sample waypoint list

The outputs of the application are the final flight route and the total length of this route. The flight route is represented as a list of waypoint Ids.

4.2. Solution Approximation Using Genetic Algorithm

4.2.1. Approach

The strategy proposed in this study uses a Genetic Algorithm (GA) to solve the Traveling Salesman Problem (TSP). Testing every solution for the TSP means *N!* operations for *N* cities and adding another city increases the number by the factor of *(N + 1).* Keeping this in mind brutal force approaches seem to be impractical.

GAs otherwise, are sure to find a solution to the TSP in shorter time but the goal of finding the optimal solution is sacrificed most of the time. Although the solution, a GA finds for the TSP, is often not the optimal one, it can find near perfect solutions for an input size of 100 in a few minutes.

Survival of the Fittest is the main idea behind GAs since they imitate natural evolution. As described previously in Section 1.3 the steps of execution of a GA in general can be summarized as (Figure 4.2):

- An initial candidate set of solutions is selected from the whole solution space
- Each candidate in this set is tested according to some fitness criteria
- The candidates that pass the test are used to produce new and better solutions through crossover and mutation until a termination condition is reached.

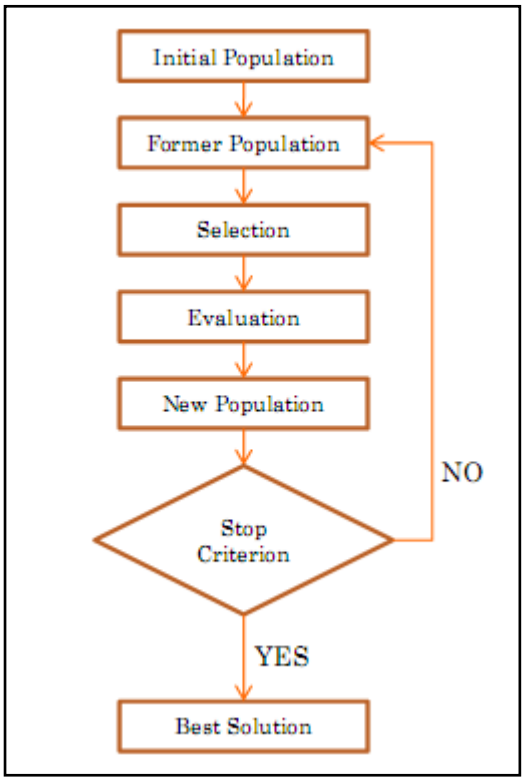


Figure 4.2: Operation of Genetic Algorithm

This study uses a GA that is slightly modified to meet the needs of the surveillance domain. In more detail the GA proposed in this study works as follows:

- A defined number of flight routes are created, this is called a **population**.
- A **greedy** approach is used to create the initial population such that; the waypoints closer to each other are preferred for creating flight routes.
- The flight routes are tested according to a criterion which is being shorter.
- Two of the better flight routes are picked and named as **parents**.
- The parents are combined through **crossover** to reproduce two **child** flight routes which are hopefully better in terms of shortness.
- In order to prevent the future generations of flight routes become too similar to each other, the child flight routes are **mutated** in a defined percentage.
- The finalized child flight routes are inserted back to the population replacing two of the longer flight routes. By this method the population size is always constant.

- Reproduction of new flight routes are continued until the pre-defined number generations are created.
- After termination the best flight route in the population is presented as the solution of the flight route planning problem.

The main differences between the algorithm proposed in this study and GAs in general, is the greedy approach used in constructing the initial population and the **crossover** method used for the reproduction of new generations. The greedy approach is explained in the next section in detail. The main reason for not using a standard crossover method is because the surveillance domain has some of its own characteristics thus need to be attended in a different fashion. These constraints are explained in the following paragraphs.

In a standard GA, where a candidate is represented by a string of letters or numbers, the crossover operation is performed simply by deciding a point in the string and exchanging the parts of two strings after that point. Assuming waypoints are represented as w_n; where w_i stands for the i^{th} waypoint and a sample flight route is represented as a list of waypoints such as; $w_i\ w_j w_k$, where i, j and k are integers Table 4.1 shows a sample crossover operation.

Table 4.1: Sample Crossover Operation

Parent 1	$w_6\ w_1 w_2\ w_5 \mid w_3 w_7 w_4$
Parent 2	$w_4\ w_5\ w_1\ w_3 \mid w_7\ w_2\ w_6$
Child 1	$w_6\ w_1\ w_2\ w_5 \mid w_7\ w_2\ w_6$
Child 2	$w_4\ w_5\ w_1\ w_3 \mid w_3\ w_7\ w_4$

In this example above, the selected point is between the 4^{th} and 5^{th} items of the strings.

In order to create children the remaining parts of the strings after the crossover points is replaced with each other.

The problem arising from this kind of crossover is that; the flight routes that are constructed may not be valid flight rules. Especially violating the constraints that each flight route should:

- Pass through each waypoint
- Do not pass through a waypoint more than once

Concerning the table above and using the direct crossover approach, Child 1 passes through w_2 and w_6 twice and it fails to pass through w_3 and w_4. So such a flight route is unacceptable and should be refused. The proposed crossover mechanism should eliminate such invalid routes however it is computationally more efficient not to let them to be created in the first place.

4.2.2. Crossover Operation

The crossover method considered in this study is an implementation of cycle crossover. This method solves the problems that can arise because of using an inconvenient crossover method. These possible problems are described as “failing to visit a waypoint” and “visiting a waypoint more than once” in the previous paragraphs. Cycle crossover method solves these problems by definition because it does not let any waypoint previously used in the construction of a child flight route, to be added again into the same child flight route providing that both parent flight routes are valid ones (Figure 4.3). Crossover methods in general start by selecting a number of potential parents from the created initial population. The number of potential parents is called group size. The decision of the group size affects the performance of the algorithm in terms of running time and solution optimality. This

study considers a parametric group size that can be changed before the flight route planning application is run.

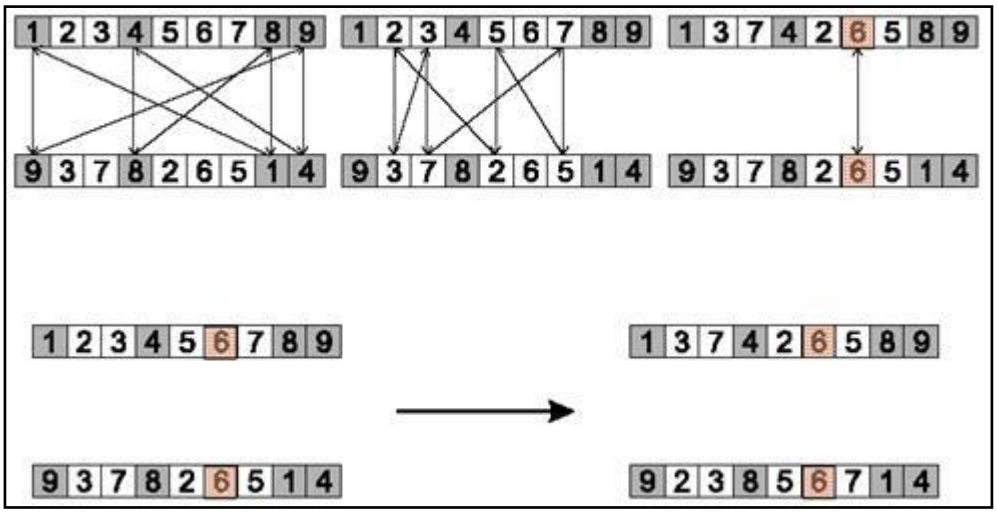


Figure 4.3: Sample Cycle Crossover Method

Cycle crossover method, unlike the direct crossover method described above, does not pick a crossover point. The method basically works as follows;

- Start form a parent.
- Try to place each element to a position in the first child where that place is the same as in either one of the parents'.
- Construct the second child as the complement of the first child.

Below there is an illustration of the method. Starting with the following parents Parent 1 and Parent 2:

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8

(Parent 1)

w_8	w_5	w_2	w_1	w_3	w_6	w_4	w_7

(Parent 2)

Let us first pick w_1 from Parent 1 and add it to Child 1:

w_1	*	*	*	*	*	*	*

(Child 1)

As it is customary to pick every element from one of the parents and place it in the same position to the child, and since the first position in the child is filled by w_1, the waypoint w_8 cannot be taken from parent2. So w_8 is also picked from Parent1 and added to Child 1.

w_1	*	*	*	*	*	*	w_8

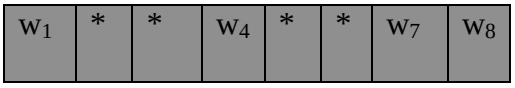
(Child 1)

Again as the 8^{th} slot in Child 1 is filled, w_7 of Parent 2 cannot be used for Child 1, so w_7 is taken form Parent 1 and added to the 7^{th} slot of Child 1.

w_1	*	*	*	*	*	w_7	w_8

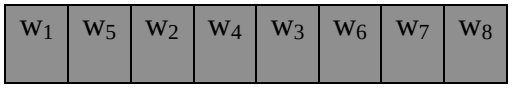
(Child 1)

After putting w_7 into the 7^{th} slot the method forces us to put w_4 to the 4^{th} slot of Child 1 as Parent 2 has w_4 in its 7^{th} slot which is not usable as the slot is already filled in Child 1. Then Child 1 looks like:



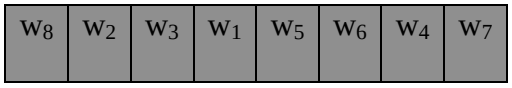
(Child 1)

Note that upon adding w4 to Child 1 the not usable waypoint becomes w1 however as it is inserted before inserting it again will cause a cycle to be formed. At this point the method fills the remaining slots of Child 1 from Parent 2 which makes Child 1:



(Child 1)

As explained before the second child, Child 2 is constructed as the complement of the first child. This means that while filling in its slots the waypoints that are not used in Child 1 are used. So Child 2 becomes:



(Child 2)

The crossover method described above is guaranteed to create valid flight routes providing that both the parent flight routes are valid. However it should be noted that there is a possibility that the produced children can be the same as the parents. Although this may seem as a problem at first, as the parents are selected from the shorter flight routes the children are still good solutions.

4.2.3 Mutation Operation

The GAs eventually start to produce similar even identical results. This situation is undesirable since many possible good solutions may be overlooked; moreover after some point the GA will not be able to find any better solutions. In order to get around this problem, this study proposed three techniques each of which applies to different steps of the GA.

The first two approaches focus on the selection of the initial population. The first method is about the initial size of the population. Selecting a larger population will cause the GA converge to identical results later than the one working on a smaller population. However selecting a larger initial population increases the total time elapsed to find the solution. In order to deal with this trade off it is decided that selection of the first population will be a parametric value which can be changed before the flight route planning application is run. By this way it would be possible to select an appropriate initial population size value for specific cases.

The second method concerns about the usage of the greedy approach while selecting the initial population. As it is clear that the GA, while using the greedy approach, will prefer to make links between cities that are close to each other, so that the initial flight routes will be similar to each other. To overcome this case the greedy approach will be limited with two parameters. Namely the percentage of the population selected using the greedy approach and the number of cities that will be considered nearer. Again like the case of initial population size these two parameters will be able to be changed before the flight route planning application is run.

The final method used for creating randomness in the proposed solutions to the flight route is the **mutation**, which is an essential part of any GA. Mutation can be defined as random alteration of possible solution. GAs use mutation in order not to get stuck at local optimum values. The nature of GAs usually takes them near local optimums instead of the global optimum. So that in each step the solutions that are

nearer to the local optimums will be selected for crossover as their fitness values are better and the final result will be probably near the local optimum but will have little chance in reaching the global optimum. So that mutation comes forward as a random way of getting to possible solutions that would unlikely to be found.

In the concept of flight route planning it is imperative to give extra attention to mutation, as it was done in the crossover step, in order not to let invalid flight routes to be created. So that the mutation proposed in this study is not a completely random one. The proposed strategy of this study also uses a parametric mutation which can be set as a percentage before the flight route planning application is executed.

Recalling the sample flight route created in the crossover step, Child 1, mutation operation can be further demonstrated as follows:

w_1	w_5	w_2	w_4	w_3	w_6	w_7	w_8

(Child 1)

Implementation of a completely random mutation method to the proposed example will work as changing a waypoint on a random position with another randomly selected waypoint. It should be noted that this operation is not swapping as only the value at the selected position of the flight route changes. So that an example mutation is assigning the 3rd position of the flight route the waypoint w_7, making Child 1 look like;

w_1	w_5	w_7	w_4	w_3	w_6	w_7	w_8

(Child 1)

The output of this mutation operation is clearly invalid as it fails to pass through w_2 and visits w_7 twice. In order to eliminate invalid mutations the strategy described in this study uses a special kind of mutation.

The mutation method used in this study introduces the concept of **sub routes**. A sub route is any subset of a flight route consisting of consecutive waypoints. However the flight route itself is not considered a sub route and the reason for this will be discussed in the following paragraphs. The method mutates the flight route by simply reversing the selected sub route. Again considering, Child 1, the output of the crossover operation a mutation concerning the sub route w_2 - w_6 will cause Child 1 to become:

w_1	w_5	w_6	w_3	w_4	w_2	w_7	w_8

(Child 1)

Notice that this mutation operation always yields valid flight route outputs for valid inputs. Using this mutation operation will make the final check of flight routes against the constraints proposed obsolete. So no final verification of the flight routes is necessary.

4.2.4 Genetic Algorithm Parameters

As discussed in the previous section some values used in the calculation of the flight route will be parametric and will be able to be changed before the flight route planning application is run. This choice is more a necessity than a preference as different parameters may perform better in specific conditions. Default values will be supplied for the following parameters however these default values are also not constant as they will be decided according to the input space size. The following six parameters are used to control the GA proposed in this study:

Initial Population Size: As discussed before a defined number of flight routes are created at the start of the GA and this is called a population. The size of this initial

population affects the results of the algorithm since while a larger population increases the running time of the application, a smaller initial population lowers the chance to find a solution near global optimum.

Greedy Selection Percentage: It was discussed in the previous section that although the greedy approach used in constructing the initial population yields a faster algorithm it eventually leads to flight routes that look similar to each other hence decreasing the chance of reaching the global optimum. The Greedy Selection Percentage is used while creating a route to decide if a closer waypoint or a random waypoint will be used as the next one.

Number of Closer Waypoints: This parameter decides the number of waypoints that would be considered close to a specific waypoint. The decision of closeness to a waypoint is done in the preprocess step and the list of decided number of waypoints is kept for each waypoint. Closeness is calculated according to the pair wise distances between waypoints. While using greedy approach to create the initial population the GA prefers to link waypoints that are close to each other. It should be noted there is no priority among the waypoints selected as being closer and there is an equally random chance that any one of these waypoints will be selected.

Group Size: As discussed in the previous sections, at the start of each crossover step a number of flight routes are selected and the best two of them are used as parents in the crossover operation. The group size parameter is very effective on the GA since a large group size causes the GA run faster however may result starvation such that some flight routes may never be used as parents. So that reducing the chance to reach the global optimum.

Mutation Percentage: Mutation is used in order to protect the flight routes to look identical after a number of crossover steps. The mutation percentage decides the probability of a child flight route created as a result of crossover to undergo mutation or not.

Termination Step: The GA is clearly a repetitive operation, so there needs to be a termination condition on which the algorithm stops and presents the best solution it has as the final one. Although it may seem plausible to stop the algorithm after the solutions become stable this is both hard to implement and may be impractical for very large input spaces. So that in this study running time of the algorithm is limited with the number of crossover operations and named as Termination Step.

Table 4.2 shows a sample list of configurable parameters for the GA:

Table 4.2: Sample Configuration Parameters for the GA

Parameter	Sample Value
Initial Population Size	**1000**
Greedy Selection Percentage	**80 %**
Number of Closer Waypoints	**5**
Group Size	**10**
Mutation Percentage	**5 %**
Termination Step	**100**

4.2.5 Selection Development Using Greedy Heuristic

The pros and cons of using the greedy approach are discussed in the previous sections. The proposed strategy of this study considers a greedy approach consisting of two different versions. During the construction of the initial flight routes, the Greedy Selection Percentage defined in the previous section is used to decide which version of the greedy approach is used. In the first version of the approach, the selection of the next waypoint to be added to the flight route is done randomly among the waypoints that are listed as being closer to the current waypoint. The second version of the approach, which is definitely less greedy, works as selecting two random waypoints which are not among the waypoints that are listed as being closer to the current waypoint and adds the one which brings less cost to the route.

The details of the two versions of the greedy approach are given as pseudo codes below:

Greedy Approach Version I

Choose a random waypoint;

Mark it visited;

While (all the cities not visited) {

Select a waypoint defined as being closer to the selected waypoint randomly;

Mark it visited;

Connect it to the previous visited waypoint;

}

Greedy Approach Version II

Choose a random waypoint;

Mark it visited;

While (all the cities not visited) {

Choose a random waypoint w_1;

Choose a random waypoint w_2;

Calculate the cost of addition of each waypoint;

Mark it visited;

Connect it to the previous visited waypoint;

}

4.3 Mapping the Genetic Algorithm into CUDA

In the concepts of this study two versions of the proposed application is developed. The first version which is called the serial implementation is written in C language. The second version called the parallel implementation is written in C language using CUDA extensions and compiled with the CUDA compiler. These two versions are used to compare the respective performances of parallel and serial implementations also the feasibility of a GPU solution for the proposed problem. In order to have a suitable comparison, two versions of the application are designed as logically equal as possible. However there are small differences mostly due to the differences between the architectures of GPU and CPU. In order to further test the genetic algorithm implementation in the GPU a control version using a distributed

approach is also implemented. The distributed genetic algorithm has been developed in the same environment with the GPU version of the genetic algorithm.

The following sections describe the implementation details of the GPU implementations. Section 4.3.1 presents the extra implementation constraints that arise because of using CUDA. Section 4.3.2 explains the steps of genetic algorithm implementation and Section 4.3.3 describes the distributed genetic algorithm implementation.

4.3.1 CUDA Specifics

Implementation of an algorithm on CUDA requires some special measures to be taken in order not to turn the advantage gained by parallel processing into a disadvantage. This section mentions the key points of CUDA programming along with the solutions proposed by this study. The CUDA specific implementation details are explained in three sections namely; threads, conditional statements and memory management.

4.3.1.1 CUDA Threads

CUDA handles the thread structure through a hierarchical order. Up to maximum 512 threads form a block and a grid of blocks make up a kernel. This structure is illustrated in Figure 4.4. The key point in this hierarchy is that threads in a single block can communicate through a fast local cache that is called the shared memory. Communication among the blocks is maintained through the main memory which is much slower compared to the shared memory. The necessity of accessing the main memory for block-to-block communication yields to a design principle that; threads that will run in different blocks should be designed to be totally independent of each other.

To control the described hierarchy CUDA presents three variables to the user:

blockDim; is a 3D vector that describes how many threads each block contains

blockIdx; is a 2D vector that contains the placement of a block in the grid

threadIdx; is a 3D vector that describes the placement of a thread in the block

These three variables are used in the given order in order to let the programmer access the threads that are kept in an array like structure.

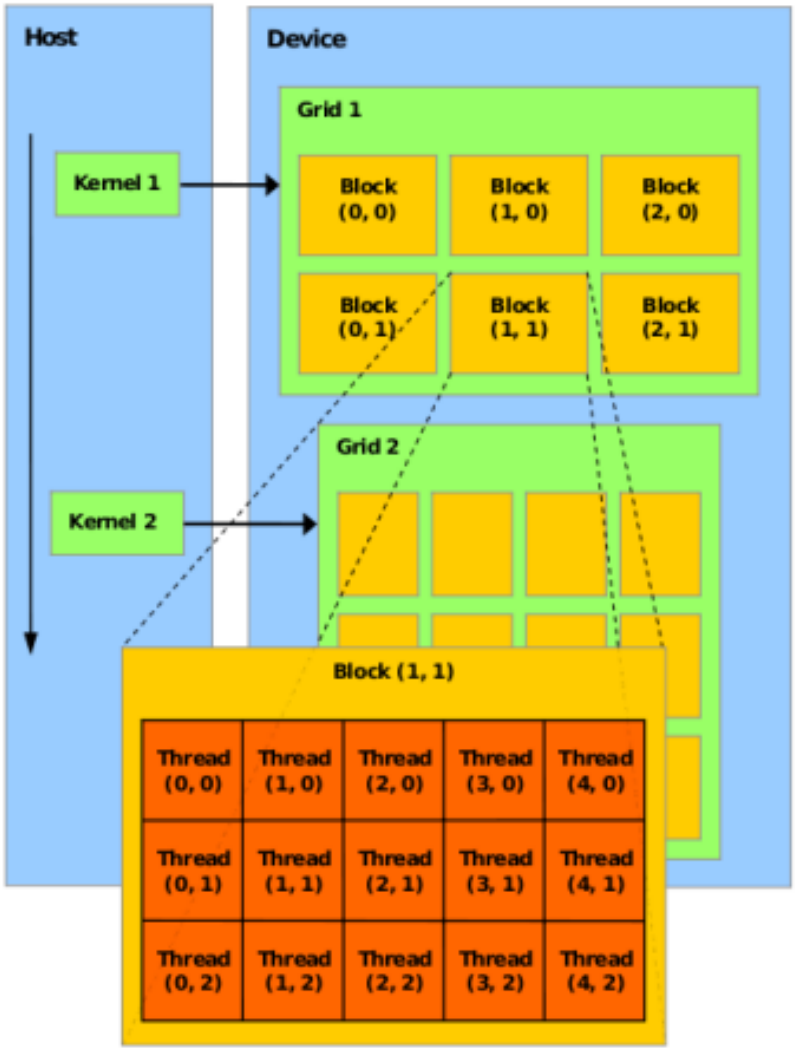


Figure 4.4: The relationship between threads, thread blocks and grids in CUDA

4.3.1.2 Conditional Statements

Another consideration for CUDA implementation is the usage of conditional statements such as if and else. Such branching operations may yield to a loss in performance. As threads in CUDA, unlike CPU threads which are executed independently, are scheduled in groups, they start running together at the same program address and execute one common instruction at a time. If threads of a group diverge according to a data dependent conditional statement, the group should serially execute each branch path taken, until all paths are complete and the threads converge back to the same execution path. [13]

4.3.1.3 Memory Management

As discussed in the section concerning CUDA threads, main memory access is a costly operation for CUDA applications. Special care has to be taken when developing a CUDA application while considering memory management. The latency difference between the shared memory and the main memory requires the frequently used data to be in the shared memory. However CUDA also provides some caching structures in order to speed up the access to global memory.

As discussed in Chapter 2 this study some rules for accessing the memory which are as follows:

- Frequently accessed data is stored on share memory
- Write-once data, is stored on global memory
- Read-only data is stored on GPU's texture cache

In terms of implementation frequently accessed data is the structures that hold the flight paths, write-once data is the final flight route and read-only data are the coordinates of the waypoints and GA parameters.

4.3.2 Genetic Algorithm Steps on CUDA

This section describes the implementation details of the parallel version of the genetic algorithm on CUDA. The parallel implementation is actually similar to the serial implementation; however there are small differences and these are addressed in the following paragraphs.

```
// CPU  Implementation

CreatePopulation ( populationSize )

For ( numberOfGenerations )

        SelectGroup ( groupSize )

        CreateChildren ( ) // Crossover and mutation is executed here

End
```

Figure 4.5: Pseudo Code for CPU version of the Genetic Algorithm

Figures 4.5 and 4.6 summarize the CPU and GPU implementations of the genetic algorithm respectively. Both implementations follow the same logical steps generally; however the GPU implementation uses threads for creating initial population and creating children, thus doing multiple operations at each iteration.

```
// GPU  Implementation

 CreateThreads ( maximumNumberOfThreads )

Foreach (thread)

        CreateFlightRoute ()

End

For ( numberOfGenerations )

        CreateThreads ( maximumNumberOfThreads )

        Foreach (thread)

                SelectGroup ( groupSize )

                CreateChildren ( ) // Crossover and mutation is executed here

        End

End
```

Figure 4.6: Pseudo Code for GPU version of the Genetic Algorithm

Initial Population Generation: The initial population of candidate flight routes considered for the genetic algorithm is constructed using the same greedy approach in both serial and parallel versions of the algorithm. However a randomization function such as the “rand()” of C is necessary for both deciding what method of greedy approach is selected and for selecting waypoints randomly. CUDA does not provide such a function but it is possible to implement a random number generator in CUDA. For this study the Linear Congruential Generator implementation described in the GPU GEMS 3 will be used [12].

Population Set Organization: It was mentioned in the previous sections that, upon the creation of the initial population, the candidate flight routes are tested according to a criterion, which is being shorter. In order to operate on the population set, the GA should be able to access these flight routes in an indexed fashion. Testing the flight routes according to the criterion of shortness, namely calculating their length is done simply by staring from the initial waypoint and adding the distances between two adjacent waypoints until the final one is reached. The parallel version uses the advantage of the GPU and assigns each flight route length calculation to a separate thread so that makes the operation concurrently.

To work efficiently the GA needs to work on a list of flight routes sorted according to their length. The sorting of flight routes is done by using two arrays where one holds the unique identifiers for the flight routes and the other holds their respective lengths. The sorting is done using quicksort. The parallel version is able to make the sorting of both arrays simultaneously.

Crossover: The cycle crossover operation is implemented similarly in both versions. In order to prevent invalid flight routes to be constructed both the parallel and serial algorithms follow the same steps. This type of approach is also a necessity to provide logical equality since different results at this step of the GA may result in different runtime behavior in the two implementation versions. The main difference in serial and parallel versions is that, while the serial version works on one group at a time thus does the crossover operations one by one, the parallel version executes the multiple crossover operations on different groups simultaneously.

Mutation: The mutation operation is quite similar in both parallel and serial versions. In both implementations two waypoints on the flight route are selected and the sub-route between them is reversed. The only difference is that while the serial implementation makes mutation one by one for each flight route, the parallel version works simultaneously by assigning each mutation operation to a different thread.

4.3.3 Distributed Genetic Algorithm Steps on CUDA

This section describes the implementation details of the distributed genetic algorithm and on CUDA. The distributed genetic algorithm implementation has many common properties with the genetic algorithm implementation addressed in the previous section. The similarities and differences between the two implementations are described in the following paragraphs.

Figure 4.7 summarizes the distributed genetic algorithm implementation on GPU. The difference of distributed version is that instead of creating a single population which consists of all parents, a determined number of subpopulations are created. The creation of subpopulations is done concurrently using GPU threads.

Another point that separates the distributed genetic algorithm form the genetic algorithm implementation of the previous section is the usage of migration instead of mutation at the end of each iteration of the child creation process. Migration is also executed using threads and thus concurrently on the memory of the graphics card.

```
// Distributed GPU Implementation

CreateThreads ( maximumNumberOfThreads )

Foreach (thread)

        CreateSubpopulations ( numberOfSubpopulations )

End

For ( numberOfGenerations )

        CreateThreads (numberOfSubpopulations )

        Foreach (thread)

                SelectGroup ( groupSize )

                CreateChildren ( ) // Crossover is executed here

                DoMigration ( ) //  Successful children are distributed to other subpopulations

        End

End
```

Figure 4.7: Pseudo Code for Distributed Genetic Algorithm

As explained earlier, distributed genetic algorithms have different types which are mainly determined by the types and numbers of populations that the algorithm operates on. The distributed genetic algorithm used for this study falls to the class of static subpopulations with migration or in other words genetic algorithm with multiple demes. The term “deme” means population. In this implementation the initial population is divided into subpopulations (demes) and these demes are isolated during execution. However there is some sort of communication between these demes.

The communication between the demes is provided using a new operation called “migration”. Migration operation carries the children created during crossover operation of each subpopulation to the other subpopulations. Crossover is executed same in both genetic algorithm for the previous section and the distributed genetic algorithm of this section with the exception that mutation is not exercised for distributed genetic algorithm implementation. The condition for a child to be carried to other subpopulations is that the children should be better solutions than both of their parents. This condition makes the migration a greedy operation since it does not let worse solutions to be inserted into subpopulations. The type of migration described here is known as “Island Model” (Figure 4.8). In this model the children from each subpopulation is carried to all other subpopulations.

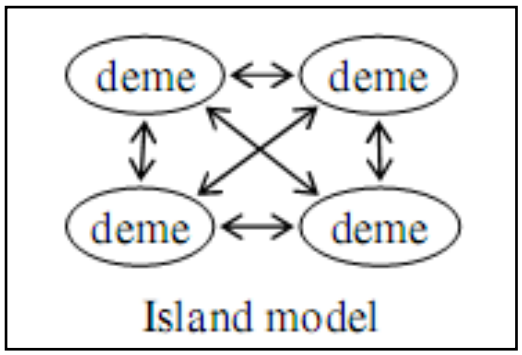


Figure 4.8: Island Migration Model

CHAPTER 5

RESULTS

This chapter presents the results of the study acquired by running four different programs, namely the serial and parallel versions of the genetic algorithm and the serial and parallel versions of random search algorithm. The chapter is organized as follows: Section 5.1 describes the hardware and software configuration that the tests are run on. Section 5.2 gives the results of comparison of parallel and serial versions of the genetic algorithm. Section 5.3 explains the results of parallel version of the genetic algorithm with a GPU point of view. Section 5.4 further gives detail about the results of parallel version of the genetic algorithm by emphasizing the effects of genetic algorithm parameters. Section 5.5 adds the results obtained by the serial and parallel versions of the random search program. Section 5.6 compares the genetic and random search algorithms. Finally Section 5.7 concludes the chapter by comparing the genetic, random search and distributed genetic algorithms.

5.1 Test Environment

The following tests are performed on a system with the following configuration:

- Intel Core2Duo P8400 processor with a cloak frequency of 2.13 Ghz
- 4 GBs of memory with a cloak frequency of 800 Mhz
- NVIDIA GeForce 9800M graphics card
- Windows Vista SP1 operating system
- NVIDIA Forceware CUDA enabled drivers version 195.94

- CUDA development SDK version 2.3
- Microsoft Visual Studio 2008

All the results presented in the following sections are obtained by running the program to be tested, five times successively, omitting the highest and lowest results and averaging the remaining three. This care has been taken in order to eliminate the risk of taking exceptional values that may have resulted from a hardware or software fault.

5.2 Comparison of Parallel and Serial Versions of the Genetic Algorithm

5.2.1 Running Time Comparison

First test run for the comparision of the parallel and serial versions of the genetic algorithm measures the running time of the two programs for a sample number (100000) of generations and for different number of waypoints (Table 5.1). The results of the test are displayed in Figure 5.1.

Table 5.1: Running Time Test Results for Genetic Algorithm

Number of WPs	**32**	**64**	**128**
CPU Running Time (sec)	3,721	8,775	18,341
GPU Running Time (sec)	0,202	0,436	1,123

It can be easily deducted that running time for sample number of generations for the GPU implementation is much more faster than the CPU implementation. However it should also be noted that the increase proportion in running time in the two implementations is almost the same. This situation is expected since no algorithmic approach effects the running time for a sample number of generations. The important point to be kept in mind from this test is that running time increases drastically with the number of waypoints. This is due to two reasons; first, creating

generations for larger number of waypoints take a larger time. Second, the time to create initial populations and sorting inputs/outputs is a more time consuming operation for larger input sets.

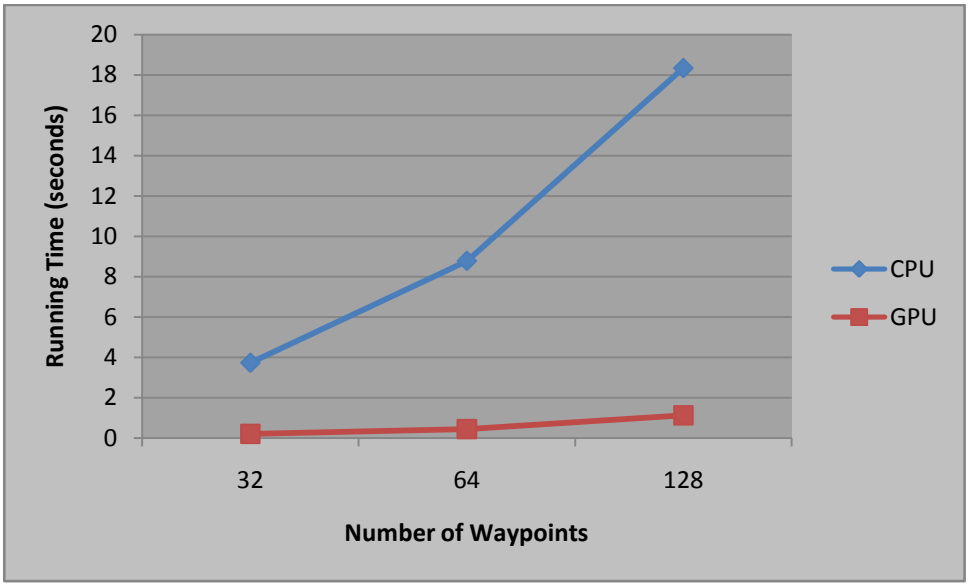


Figure 5.1: Running Time Comparison for Parallel and Serial Versions of the Genetic Algorithm

5.2.2 Convergence Time Comparison

Second test run for the comparision of the parallel and serial versions of the genetic algorithm measures the convergence time of the two programs for different number of waypoints (Table 5.2). The results of the test are displayed in Figure 5.2. For testing convergence time, the programs are run for a relatively large number of generations. Algorithm is assumed to have converged if the flight route with the lowest length has not changed for a long time and the time of the generation when the best flight route is found is recorded as the convergence time.

Table 5.2: Convergence Time Test Results for Genetic Algorithm

Number of WPs	**32**	**64**	**128**
CPU Convergence Time (sec)	1,178	14,871	168,819
GPU Convergence Time (sec)	0,687	3,188	34,158

The results of the convergence time test is useful for understanding the behaviour of the genetic algorithm. While the running time was inspected to be increasing linearly with the increasing number of waypoints, the convergence time increases exponentially. This situation is due to the fact that the number of alternative flight routes increases in a factorial order. So that there are much many flight routes to test. The exponential increase applies to both parallel and serial versions of the implementation as they follow the same logical path. The GPU implementation is still faster with respect to the CPU implementation, however the proportion of speed up between the two implementations seems to be lower than the running time comparision. The comparison of speed up values for the running time and convergence time is explained in the next section.

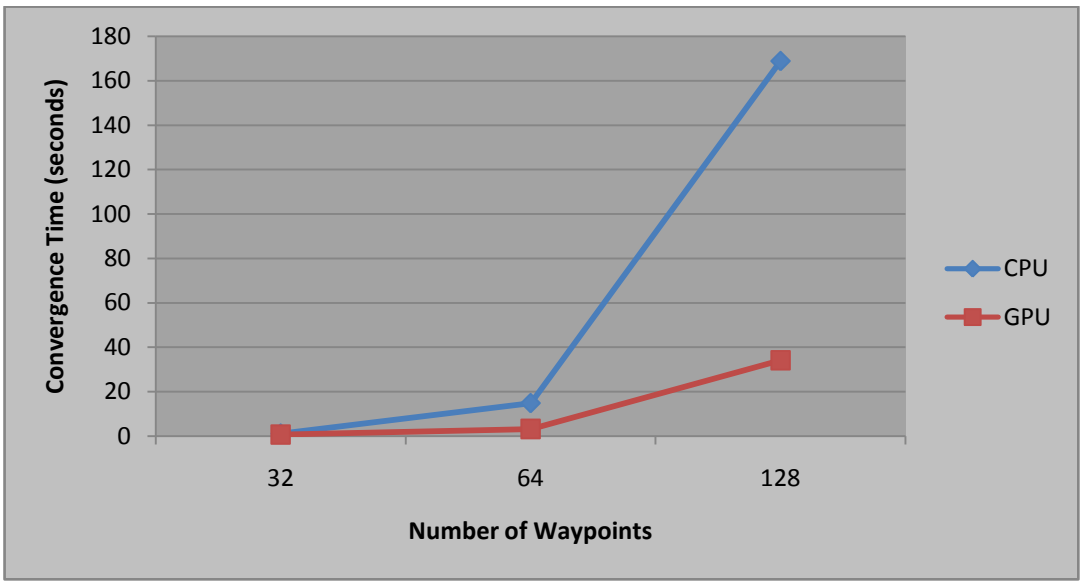


Figure 5.2: Convergence Time Comparison for Parallel and Serial Versions of the Genetic Algorithm

5.2.3 Speedup Comparision

Third test run for the comparision of the parallel and serial versions of the genetic algorithm measures the speed up values between the parallel and serial implementations for the running time and convergence time tests (Table 5.3). The speed up results are displayed in Figure 5.3. These results are obtained directly by dividing the results of running and convergence times of the CPU implementation to the respective results from the GPU implementation.

Table 5.3: Speed Up Test Results for Genetic Algorithm

Number of WPs	**32**	**64**	**128**
Convergence Speed Up	1,7	4,7	4,9
Execution Speed Up	18,4	20,1	16,3

The results of the speed up comparison reflect the algorithmic performance of the two implementatons. While the speed up in running time reaches up to 25, the one for convergence time can only reach to 5. After some inspection on these results it can be deduced that the parallel implementation does not work as efficient as the serial one in terms of number of generations. This result is expected since the evolutionary model used for the foundation of the genetic algorithm depends on the quality of previous generations and this quality may be lower for the parallel case where evolution of individual groups happen simultaneously.

Another point that can be deducted from the results is that the increase in speed up values slows down for increasing number of waypoints for convergence and even falls down for running time. This situation can be explained as an implementation specific case for the running time test, however for the convergence time test it results from the increasing number of generations needed for the algorithm to converge. The issue about generation numbers is discussed in the next section.

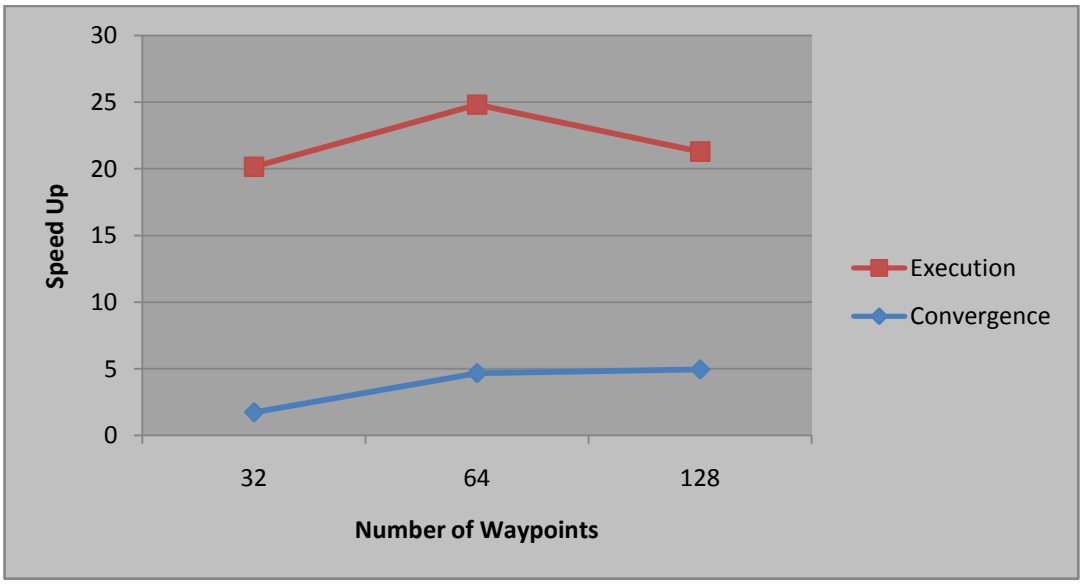


Figure 5.3: Speed Up Coefficient Comparison for Parallel and Serial Versions of the Genetic Algorithm

5.2.4 Number of Generations Comparison

Fourth test run for the comparision of the parallel and serial versions of the genetic algorithm measures the number of generations needed for the convergence of the genetic algorithm (Table 5.4). The results of the test are displayed in Figure 5.4. The results are taken along with the convergence time test and denotes the number of generation when the best flight route is found.

Table 5.4: Number of Generations Test Results for Genetic Algorithm

Number of WPs	**32**	**64**	**128**
CPU	31597	182129	990427
GPU	102400	1048576	5242880

The first obvious deduction obtained from the results of the tests is that the GPU implementation of the genetic algorithm needs much more number of generations to converge than the CPU one. This situation is a direct result of the fact that the parallel implementation has to work on lower quality previous generations than the serial one. Although the much smaller execution time needed for one generation makes the GPU implementation still faster, the GPU implementation works algorithmically worse than the CPU one.

Another important point extracted from the test results is that in both parallel and serial implementations the number of generations to converge increases. This increase seems to be linear and can be applied to both implementations, although there is no direct proportion between them. As a result it can be concluded that, in the concept of convergence, despite the CPU and GPU imlementations behave differently in runtime they are effected in the same way from the increse in the number of waypoints.

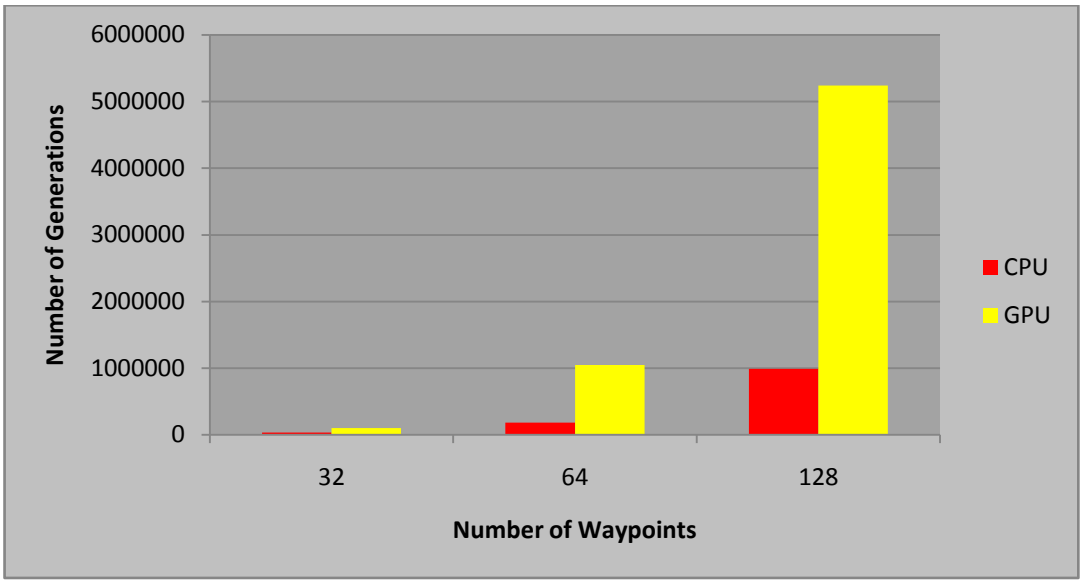


Figure 5.4: Number of Generations Comparison for Parallel and Serial Versions of the Genetic Algorithm

5.3 GPU Time Consumption

The second part of the tests includes only the parallel version of the genetic algorithm implementation. These tests aim to further investigate the behavior of the parallel version of the genetic algorithm and the GPU.

5.3.1 GPU Time Usage of Individual Functions

The first test is run for assessing the GPU time consumption of different steps of the algorithm. The steps are determined with respect to their functionality. The individual operations to be tested are defined as; GPU overhead, preparing inputs, creating the initial population, running the TSP function and copying back the results.

GPU overhead means initializing the CUDA execution and copying the input values from the main memory to graphic card's memory. Preparing inputs implies taking the array like inputs and arranging them into structures that are more efficiently used by the GPU. Creating the initial population covers randomly constructing a defined number of flight routes that are to be used as parents for the later generations. Running the TSP function includes selecting groups of flight routes form the population, creating children from these groups and applying mutation. Finally copying back the results contains taking results back from the graphics card's memory to the main memory and freeing the memory used by CUDA.

The following graphs are constructed according to the GPU time usage for the individual functions described above using different number of waypoints:

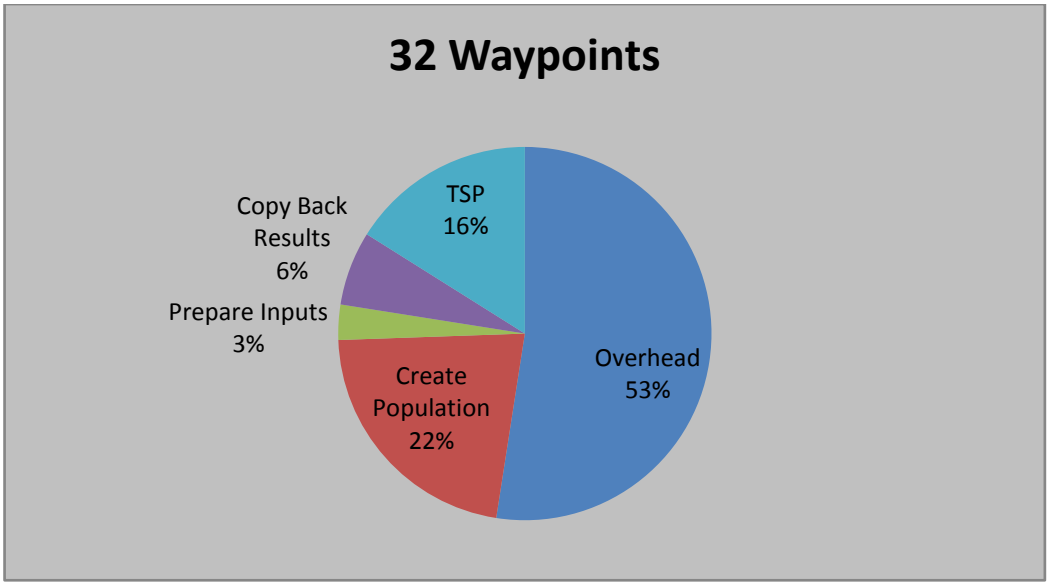


Figure 5.5: Percentage of Operations according to GPU Time Consumption for the 32 Waypoint Test

Figure 5.5 displays the GPU Time consumption of individual functions for the case where the flight route contains 32 waypoints. The most interesting result of this test is that the CPU to GPU overhead takes more than half of the execution time. There are two reasons of this situation; first the overhead is really large and second the convergence for the 32 waypoint case takes considerably short time. It should also be noted that creating the initial population also takes an important percentage of the GPU time.

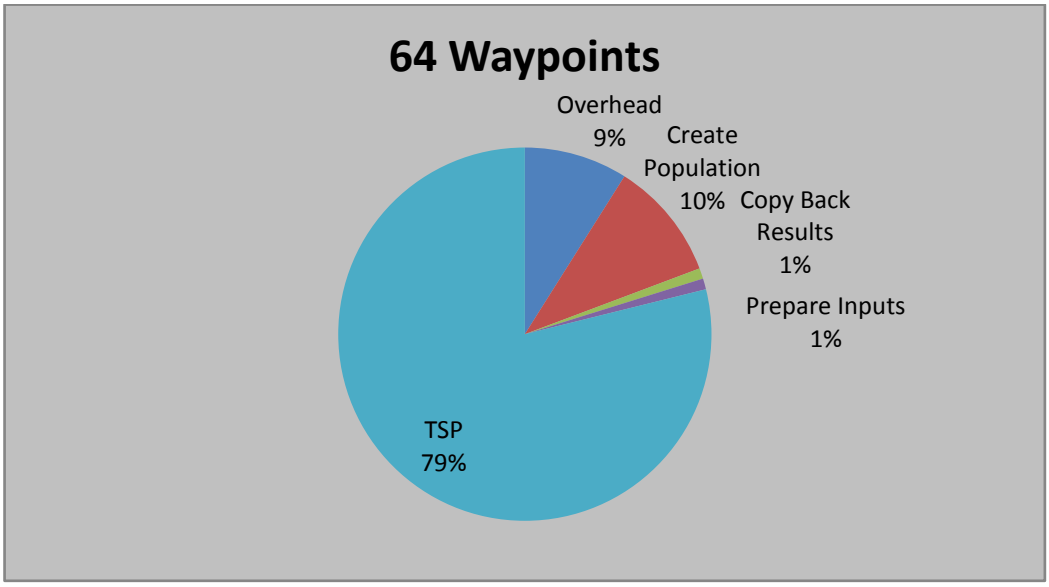


Figure 5.6: Percentage of Operations according to GPU Time Consumption for the 64 Waypoint Test

Figure 5.6 displays the GPU Time consumption of individual functions for the case where the flight route contains 64 waypoints. In this graph the most significant

change is in TSP execution time. This situation is expected since the generations needed for the execution of the algorithm increases exponentially. Another interesting point is that while creating the initial population took less than half of the time of the CPU to GPU overhead for the 32 waypoint case in takes a bit more time in than the overhead for the 64 waypoint test. This situation can be explained such that creating initial population takes more time when there are more waypoints while the overhead stays the same.

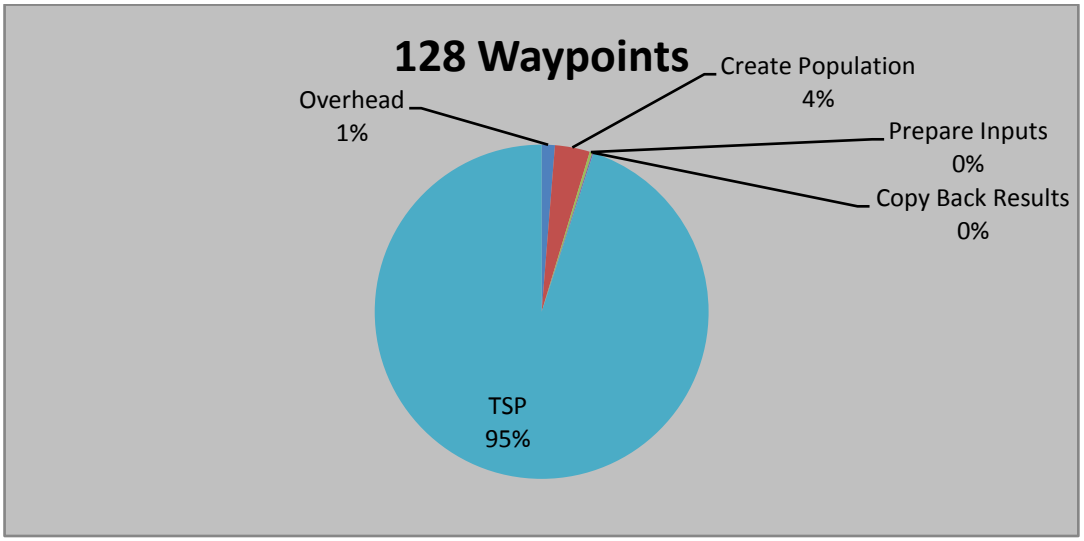


Figure 5.7: Percentage of Operations according to GPU Time Consumption for the 128 Waypoint Test

Figure 5.7 displays the GPU Time consumption of individual functions for the case where the flight route contains 128 waypoints. On this test TSP execution time takes almost all of the GPU time. All other operations, including the CPU to GPU overhead, other than creating the initial population are insignificant. Besides the percentage of creating the initial population has shrunk to 4 percent.

5.3.2 Effects of Thread Usage

Thread usage is one of the main issues in GPU programming. To get maximum efficiency from the GPU and executing programs in the smallest time possible as many as possible threads were used for the execution of programs. The limits of number of threads are selected according to two criterion; first the hardware limitations and second the requirements of the algorithm. The current CUDA version allows 512 threads to be run as a block so that this number of threads can be used without losing any efficiency. However as the algorithms were inspected to be running stable for at most 256 threads it was decided to use 256 as number of threads whenever possible.

For testing the effect of using different number of threads the algotirhm was run to converge using 16, 32, 64, 128 and 256 threads (Table 5.5). The results of the test are displayed in Figure 5.8.

Table 5.5: Number of Threads Test Results for Parallel Genetic Algorithm

Number of Threads	**16**	**32**	**64**	**128**	**256**
GPU Time (sec)	47,786	23,888	12,110	6,140	3,182

As expected, using more threads has a direct effect on execution time of the program. Actually multiplying the number of threads by two reduces the execution time to almost 50% of the original. This test gives a result about the internal structure of the GPU, such that when not assigned to threads the pipelines of the hardware do nothing. So it is a direct deduction that using as many threads as allowed by the GPU and the algorithm will give the optimal results.

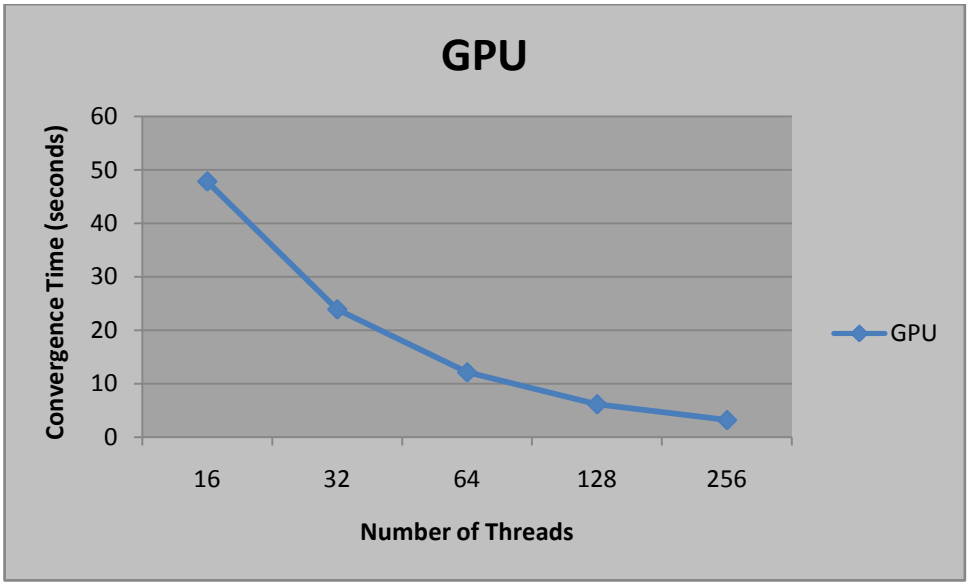


Figure 5.8: Effect of Using Different Number of Threads for the Genetic Algorithm in the GPU

5.4 Effects of TSP Parameters

Third set of tests run in order to examine the effects of different parameters that are passed to the genetic algorithm. Each parameter is tested in an isolated manner such that, while testing a parameter the other ones are kept fixed. The results are evaluated according to two criterion namely convergence time and flight route length. The tests of this section aim to further investigate the behavior of the parallel implementation of the genetic algorithm. The results of this section will be compared with the assumptions made for the genetic algorithm parameters that are described in section 4.2.4.

5.4.1 Initial Population Size

The first test about genetic algorithm parameters considers the effect of different initial population sizes. Initial population means the randomly created set of flight routes that will likely become parents while creating the future generations in the

genetic algorithm. The tests were run for 100, 1000 and 10000 number of initial flight routes for the parallel version of the genetic algorithm implementation and the results are acquired for the convergence case (Table 5.6, Table 5.7). These results are displayed in figures 5.9 and 5.10.

Table 5.6: Effect of Initial Population Size to Convergence Time

Initial Population Size	**100**	**1000**	**10000**
Convergence Time (sec)	3,003	3,446	6,305

Table 5.7: Effect of Initial Population Size to Fitness

Initial Population Size	**100**	**1000**	**10000**
Fitness	3984	3954	3538

As described in Section 4.2.4 a larger initial population is expected to find a better solution while suffering from a larger convergence time. The results justify these assumptions. The test using the largest initial population (10000) finds a 10% better result while converging in more than twice the time of the test using the smallest initial population (100).

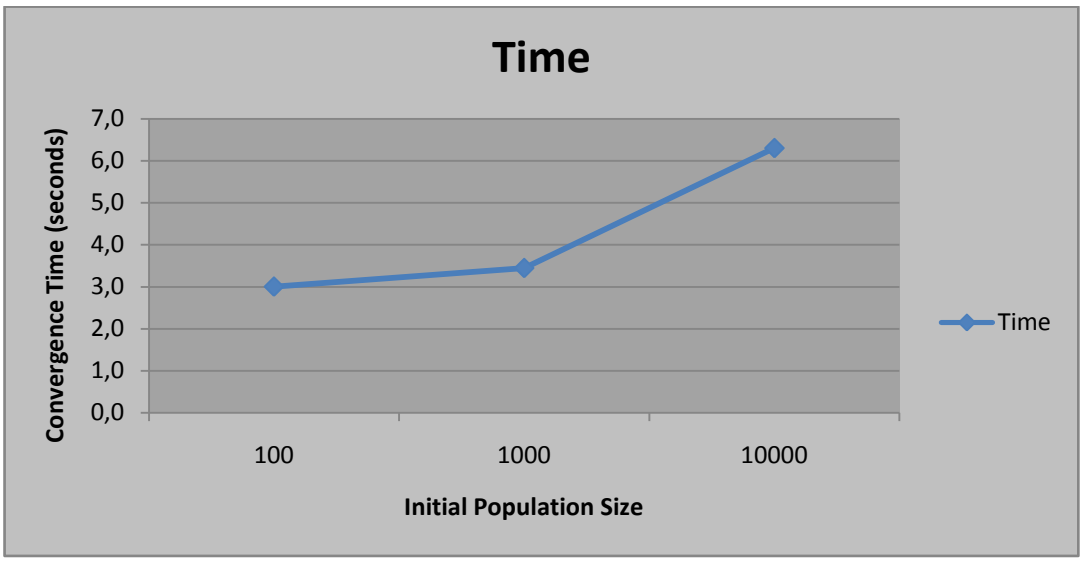


Figure 5.9: Effect of Using Different Initial Population Sizes on Running Time for the Genetic Algorithm

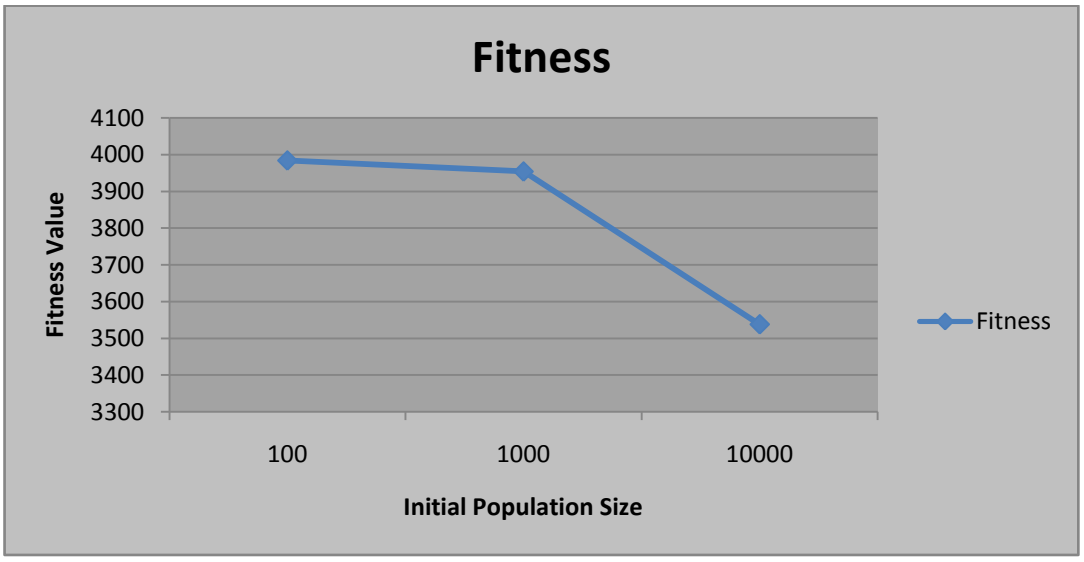


Figure 5.10: Effect of Using Different Initial Population Sizes on Flight Route Length for the Genetic Algorithm

5.4.2 Greedy Selection Percentage

The second test about genetic algorithm parameters is about the utilization of a greedy approach during the execution of the genetic algorithm. The greedy approach is used for reducing the convergence time. However there is a risk of sticking to a local optimum than the global one while using a greedy approach as described in previous chapters. The tests for measuring the effect of greedy approach usage were run for 0, 30, 60 and 90 percentages and the results are collected for the convergence case (Table 5.8, Table 5.9). These results are displayed in figures 5.11 and 5.12.

Table 5.8: Effect of Greedy Selection Percentage to Convergence Time

Greedy Selection Percentage	**0**	**30**	**60**	**90**
Convergence Time (sec)	4,391	2,476	2,955	3,597

Table 5.9: Effect of Greedy Selection Percentage to Fitness

Greedy Selection Percentage	**0**	**30**	**60**	**90**
Fitness	3724	3782	3760	3820

Using a greedy approach indeed lowers the convergence time. However its effect is not as direct as the initial population size since over-using it may increase the convergence time. The effect of greedy selection percentage seems insignificant for this test case and a percentage between 30 and 60 seems plausible for achieving shortest convergence time.

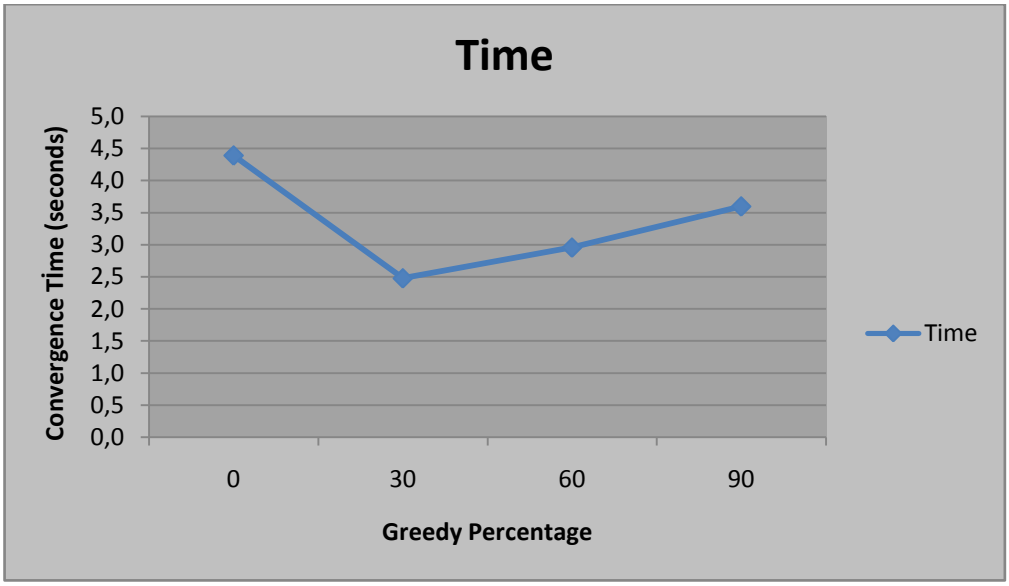


Figure 5.11: Effect of Using Different Greedy Percentages on Running Time for the Genetic Algorithm

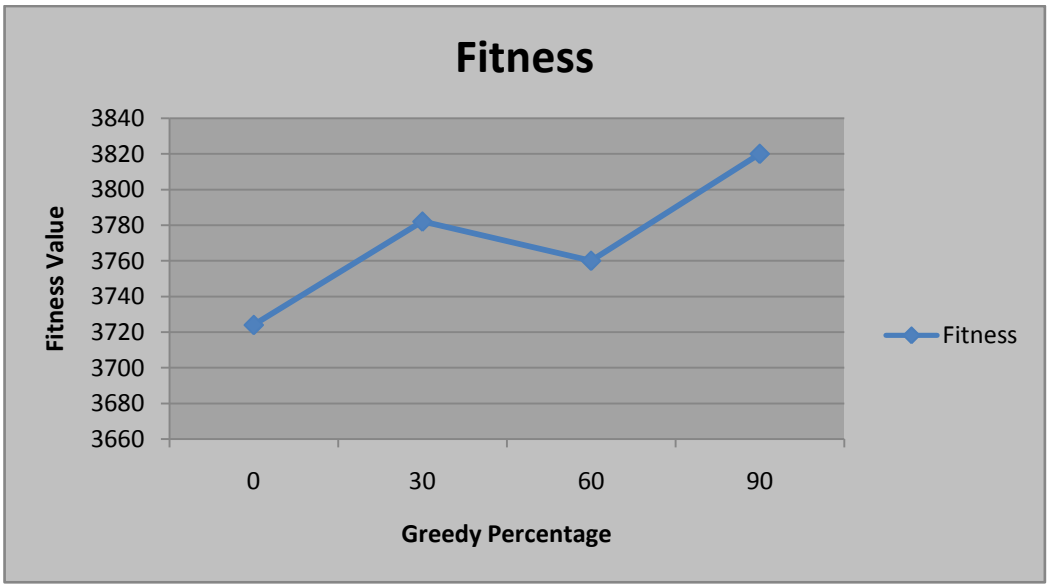


Figure 5.12: Effect of Using Different Greedy Percentages on Flight Route Length for the Genetic Algorithm

5.4.3 Number of Closer Waypoints

The third test about genetic algorithm parameters examines the usage of closer waypoints while constructing flight routes. As described in the previous chapters close waypoints are decided according to the pair wise distances between two waypoints. Using the notion of closer waypoints aims to achieve better results than the case of using completely random waypoints while creating the initial population. The tests for measuring the effect of closer waypoints usage were run for 3, 10 and 20 closer waypoints and the results are collected for the convergence case (Table 5.10, Table 5.11). These results are displayed in figures 5.13 and 5.14.

Table 5.10: Effect of Number of Closer Waypoints to Convergence Time

Number of Closer Waypoints	**3**	**10**	**20**
Convergence Time (sec)	3,138	3,423	4,002

Table 5.11: Effect of Number of Closer Waypoints to Fitness

Number of Closer Waypoints	**3**	**10**	**20**
Fitness	4184	3658	3842

Using closer waypoints helps the genetic algorithm to find a better solution. However the number of waypoints that are defined to be close to a waypoint should be limited since each waypoint may not have that many waypoints actually near to it. So this parameter becomes somehow input dependent. For the test run in this case a number of 10 closer waypoints scored the best result in terms of flight route length but it is worth to keep in mind that this value of the parameter may not be optimal for another input set.

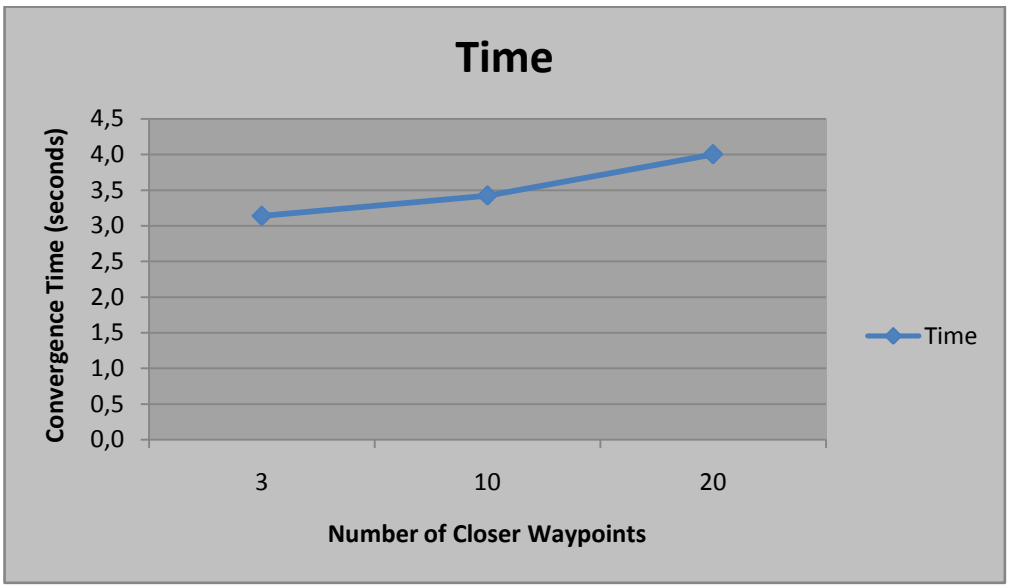


Figure 5.13: Effect of Using Different Number of Closer Waypoints on Running Time for the Genetic Algorithm

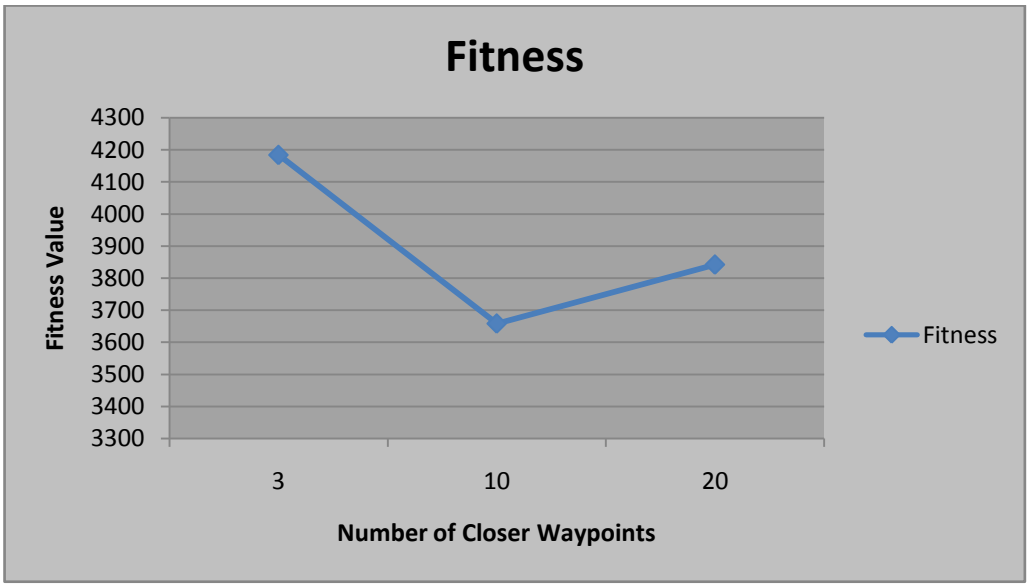


Figure 5.14: Effect of Using Different Number of Closer Waypoints on Flight Route Length for the Genetic Algorithm

5.4.4 Group Size

The fourth test about genetic algorithm parameters deals with using different group sizes while creating generations. As mentioned in Section 4.2.4 larger group sizes makes the genetic algorithm run faster however they may yield to a result which is not close to the global best. The tests for determining the effect of using different group sizes were run for the values of 5, 10 and 20 and the results are collected for the convergence case (Table 5.12, Table 5.13). These results are displayed in figures 5.15 and 5.16.

Table 5.12: Effect of Group Size to Convergence Time

Group Size	**5**	**10**	**20**
Convergence Time (sec)	4,069	3,101	2,594

Table 5.13: Effect of Group Size to Fitness

Group Size	**5**	**10**	**20**
Fitness	3821	3725	3823

Benefit of using a larger group size in terms of convergence time is obvious. Selecting a larger group size makes the algorithm converge in fewer generations. The reason is that, the parents chosen for each generation are the best of a larger group and they are usually good candidate solutions themselves. However using a large group size also means that, some flight routes may never be chosen as parents even if their children will be good choices. Using the test results the worse solution obtained using 20 sized groups, than the one with 10 elements can be explained

with that fact. This situation presents a trade-off between using a larger group for a faster solution or a smaller one for a better solution. The choice should probably be made due to the requirements of the application.

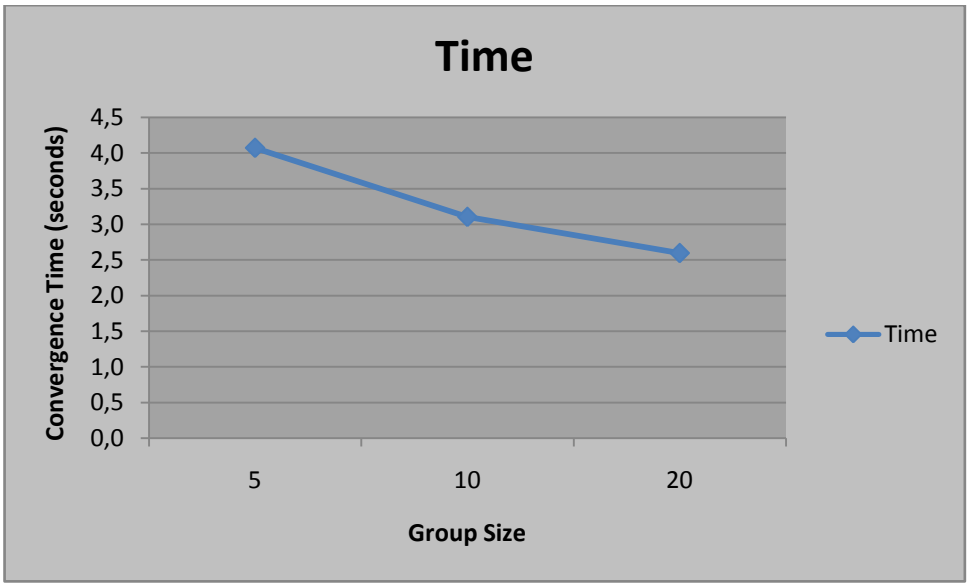


Figure 5.15: Effect of Using Different Group Sizes on Running Time for the Genetic Algorithm

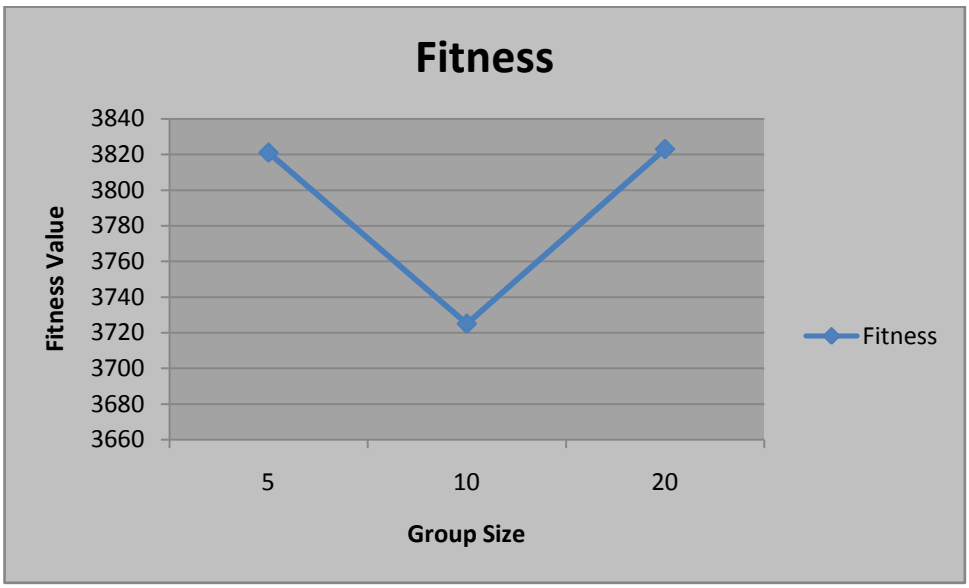


Figure 5.16: Effect of Using Different Group Sizes on Flight Route Length for the Genetic Algorithm

5.4.5 Mutation Percentage

The fifth test about genetic algorithm parameters examines the effect of using various mutation percentages during the crossover operation of the genetic algorithm. Mutation is used to create flight routes that are unlikely to be created using the selected parents in the crossover phase and arriving at possible other good solutions. Using mutation is expected to achieve at better results however may lower the performance in terms of execution time. The tests for determining the effect of using different mutation percentages were run for the values of 0, 3 and 10 and the results are collected for the convergence case (Table 5.14, Table 5.15). These results are displayed in figures 5.17 and 5.18.

Table 5.14: Effect of Mutation Percentage to Convergence Time

Mutation Percentage	**0**	**3**	**10**
Convergence Time (sec)	2,721	3,160	3,023

Table 5.15: Effect of Mutation Percentage to Fitness

Mutation Percentage	**0**	**3**	**10**
Fitness	3870	3548	3607

As in the process of natural evolution, mutation is not a phenomenon that is experienced frequently. So that it might be expected that using it too much will not do any good. The results of the tests justify this claim. Using a high mutation percentage such as 10 lowers the performance in both convergence time and fitness. However a more plausible percentage like 3 yields a 10% better result in fitness than the case of not using mutation, however suffering a 15% penalty in the convergence time.

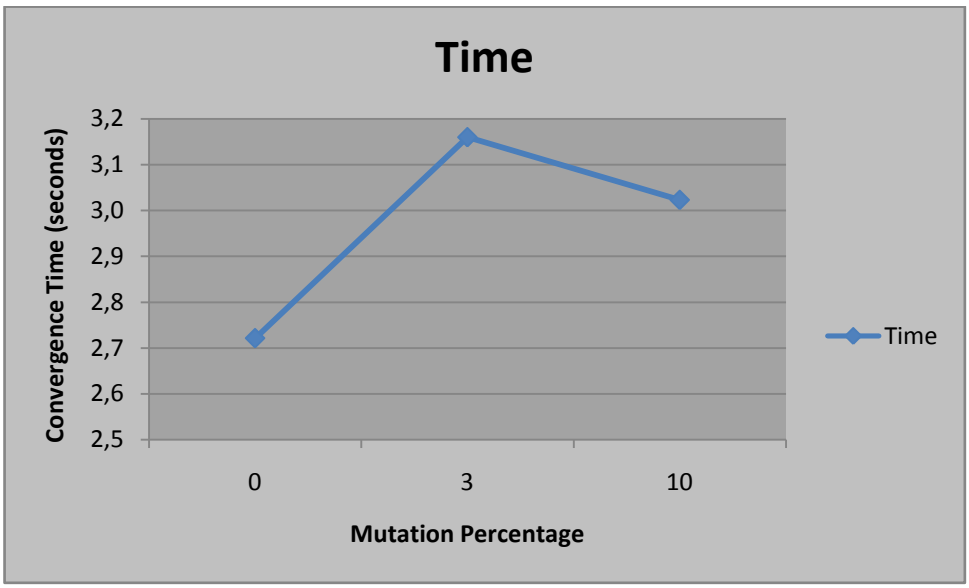


Figure 5.17: Effect of Using Different Mutation Percentages on Running Time for the Genetic Algorithm

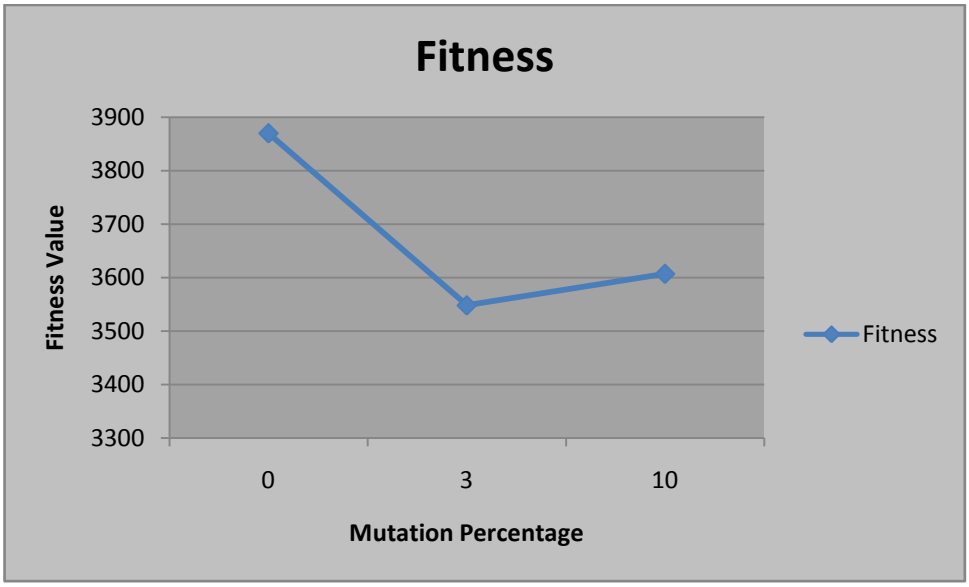


Figure 5.18: Effect of Using Different Mutation Percentages on Flight Route Length for the Genetic Algorithm

5.5 Comparison of Parallel and Serial Versions of the Random Search Algorithm

Implementation of the random search algorithm has been done in order to have an alternative method that can be compared with the genetic algorithm for the solution of the genetic algorithm. Like the genetic algorithm's implementation, both parallel and serial versions of the algorithm have been developed.

5.5.1 Running Time Comparison

First test run for the comparision of the parallel and serial versions of the random search algorithm measures the running time of the two programs for different number of iterations (Table 5.16). The results of the test are displayed in Figure 5.19.

Table 5.16: Convergence Time Test Results for Random Search Algorithm

Number of Iterations	**1000**	**10000**	**100000**	**1000000**
CPU Running Time (sec)	0,091	0,750	7,583	77,250
GPU Running Time (sec)	0,038	0,068	0,383	3,594

Using the power of its parallel pipelines the GPU implementation is much faster than its CPU counterpart. As there is no sequential operation in the random search algorithm, the threads can be run without any algorithmic overhead. The only extra time consuming event is starting the CUDA execution and copying the inputs to the graphic card's memory.

Another important point about the random search algorithm is that, unlike the genetic algorithm, it does not get more complex with increasing number of iterations such that the increase in running time in both CPU and GPU

implementations is linear with respect to increasing number of random flight routes that are created and tested.

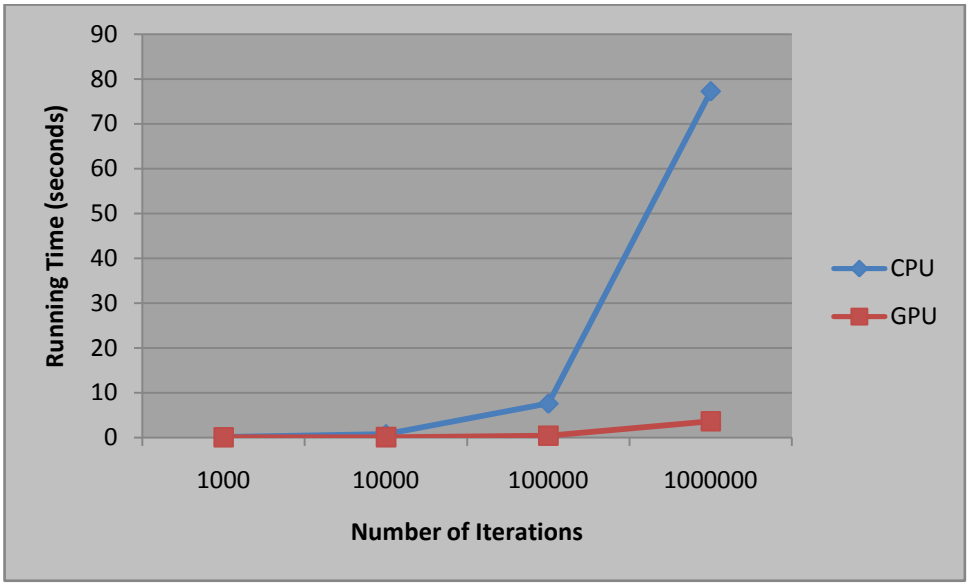


Figure 5.19: Running Time Comparison for Parallel and Serial Versions of the Random Search Algorithm for Different Number of Iterations

5.5.2 Speedup Comparision

Second test run for the comparision of the parallel and serial versions of the random search algorithm measures the speed up values between the parallel and serial implementations for different number of iterations (Table 5.17). The speed up results are displayed in Figure 5.20. These results are obtained directly by dividing the results of running times of the CPU implementation to the respective results from the GPU implementation.

Table 5.17: Speed Up Test Results for Random Search Algorithm

Number of Iterations	**1000**	**10000**	**100000**	**1000000**
Speed Up	2,395	11,029	19,799	21,494

The speed up values show the efficiency of the GPU implementation more profoundly. It is seen from the results that the speed up values increase with the increasing number of iterations, so that yielding to an increasing efficiency. However a question may arise here; why does the speed up values are lower with lower number of iterations.

The question put forward in the paragraph above is closely connected to another result that can be obtained from Table 5.17 and Figure 5.20. Although there is an increase in the speed up values for increasing number of iterations, the acceleration of this increase seems to be decreasing.

The answer to the first question is the overhead time between CPU and GPU. When there are fewer random search iterations the overhead takes a larger percentage of execution time making the speed up value lower. The answer to the question about the decreasing acceleration of the increase of speed up values is that, when the overhead starts to become less significant with the increasing number of operations the speed up becomes purely the proportion of execution times of a single random search operation in the GPU and CPU. So, it can be deducted from these results that for considerably large number of iterations the speed up values will become constant.

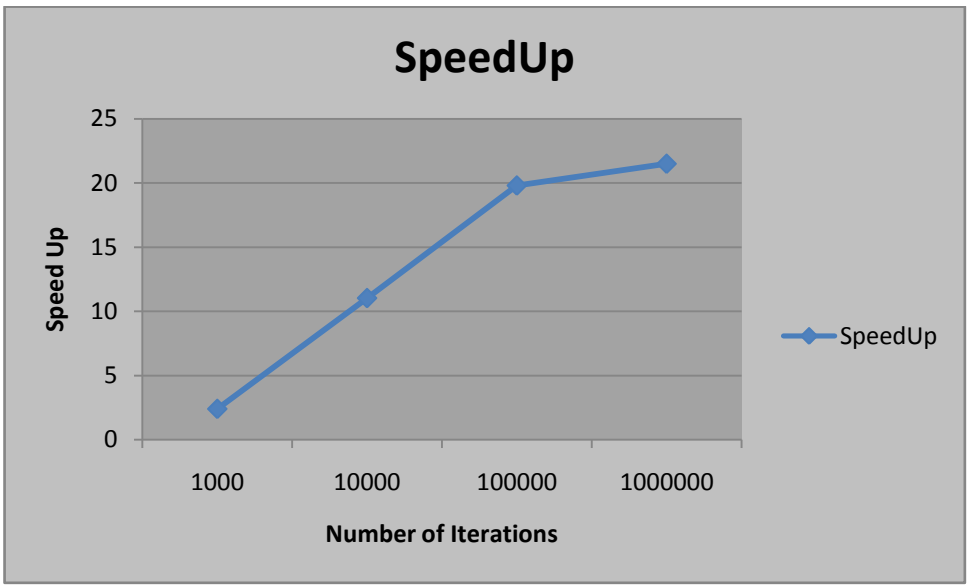


Figure 5.20: Speed Up Coefficient Comparison for Parallel and Serial Versions of the Random Search Algorithm for Different Number of Iterations

5.6 Comparison of Genetic and Random Search Algorithms

At this step the tests it is worth to make a comparison between Genetic and Random Search Algorithms that are implemented. In the following tests both CPU and GPU versions of the algorithms are compared.

As genetic and random search algorithms run with different parameters and mechanisms the only way to compare them is using the domain specific parameters of the flight route planning problem. These parameters are the running times of the programs and the fitness values of the flight routes. For the tests the algorithms are run for some time and the fitness of the best flight route for each algorithm is listed (Table 5.18, Table 5.19). The running time of the algorithms is depicted as the approximate running time since it was not possible to stop the algorithms at an exact time, however the difference between the exact and depicted running times is negligible. The results of the test are displayed in Figures 5.21 and 5.22.

Table 5.18: Comparison of Approximate Running Times and Fitness values between CPU versions of Genetic and Random Search Algorithms

Approximate Running Time (sec)	**3,75**	**7,5**	**30**	**75**
Genetic Algorithm Fitness	5053	4162	3958	3929
Random Search Fitness	9126	7337	4592	4471

Table 5.19: Comparison of Approximate Running Times and Fitness values between GPU versions of Genetic and Random Search Algorithms

Approximate Running Time (sec)	**0,375**	**1,500**	**3,350**	**16,000**
Genetic Algorithm Fitness	5071	4054	3936	3820
Random Search Fitness	7337	4670	4369	3957

The first result obtained from the Running Time/Fitness comparison is that the genetic algorithm converges much faster than the random search in both CPU and GPU versions. On the other hand, although having worse fitness results at each step, the random search algorithm closes the gap and it has a higher speed in reducing the fitness value for each step. For the GPU version the gap between fitness values fall to an acceptable %3.5, however it stays about at %14 for the CPU version at the final step of the tests. At this point it can be deducted that the random search algorithm is more suitable for a GPU implementation.

It is easy to deduce from the main mechanisms of the algorithms that the random search is more likely to find the global optimum solution if it is run for a very large amount of time, while the genetic algorithm will probably converge at a local optimum value. Inspecting the results partially justify this claim. Such that while it is seen from Figure 5.22 that the random search algorithm will likely provide better results for larger running times in the GPU implementation, its result for 16,000 seconds is worse than the genetic algorithm's result for 3,350 seconds. On the other

hand Figure 5.22 shows that the CPU implementation of the random search algorithm works worse than the corresponding genetic algorithm in terms of both fitness and running time.

At the end of the comparison it is not easy to say that one algorithm is better than the other for the GPU version as the results may change for different input sets. But the CPU tests show that the genetic algorithm is superior to the random search algorithm. The important result is that, as the previous tests proved, both GPU implementations are superior to their respective CPU implementations and this result is the point where this study aims to reach. Finally it can be concluded that while the genetic algorithm runs more efficiently than the random search for cases where reaching an optimum solution is more important the random search algorithm may be a better choice for finding the optimum solution in a GPU implementation.

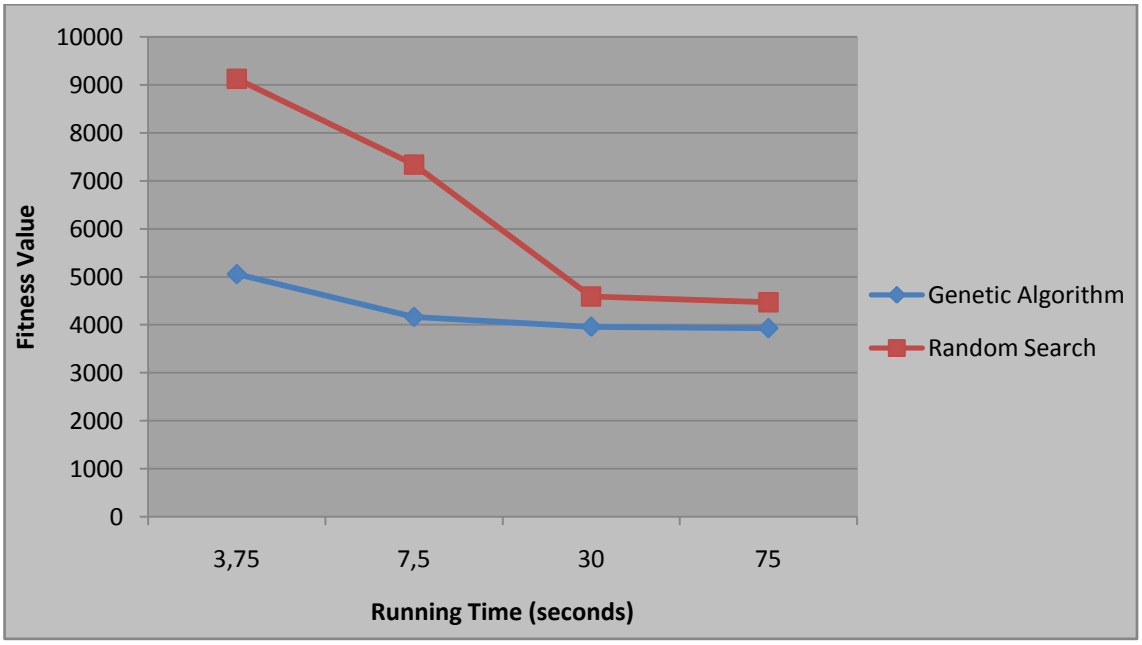


Figure 5.21: Running Time and Fitness Comparison between CPU versions of Genetic and Random Search Algorithms

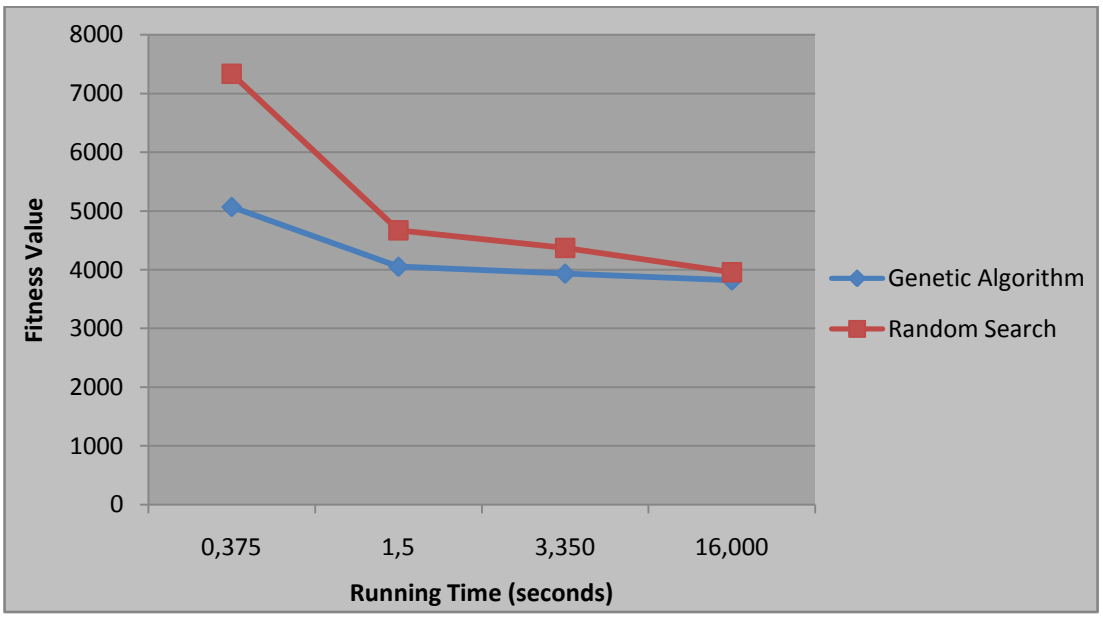


Figure 5.22: Running Time and Fitness Comparison between GPU versions of Genetic and Random Search Algorithms

5.7 Comparison of Distributed Genetic Algorithm with Genetic and Random Search Algorithms

To conclude the tests it is imperative to make a comparison between all the three algorithms that are implemented. As the main aim of this study is to demonstrate the benefits of CUDA based GPU implementations, only parallel versions of the algorithms are used for the final step of the tests.

Like the tests of the previous section the tests of this section are run with respect to running times of the programs and the fitness values of the flight routes. Again for the test the algorithms are run for some time and the fitness of the best flight route for each algorithm is listed (Table 5.20). The results of the tests are displayed in Figure 5.23.

Table 5.20: Comparison of Approximate Running Times and Fitness values between Genetic, Random Search and Distributed Genetic Algorithms

Approximate Running Time (sec)	**0,375**	**1,500**	**3,350**	**16,000**
Genetic Algorithm Fitness	5071	4054	3936	3820
Random Search Fitness	7337	4670	4369	3957
Distributed Genetic Algorithm Fitness	4765	3844	3793	3781

Unlike the tests of the previous section the results of this section are much clearer about the best algorithm for the flight route planning problem. The distributed version of the genetic algorithm proves itself to be much more faster than the other two algorithms in terms of both convergence speed and fitness value. The results on Table 5.20 shows that the distributed genetic algorithm finds a very good solution at 1,5 th second, which is more than twice faster than the traditional genetic algorithm and nearly 10 times faster than the random search algorithm for a similar solution.

Also Figure 5.23 shows that the curve of distributed genetic algorithm is much flatter than the other two algorithms, which means that the distributed genetic algorithm converges much faster than the other two algorithms. Keeping in mind that the distributed genetic algorithm converges to better results it can be deduced that the migration approach used in distributed genetic algorithm is superior to the mutation used in genetic algorithm.

Final conclusion of the tests is that, for a GPU implementation the distributed genetic algorithm proves to be a better solution than both traditional genetic and random search algorithms. This situation is due to the fact that the distributed genetic algorithm is already suitable for parallel execution and its main problem of communication latency is overcome by the efficient shared memory usage in the GPU.

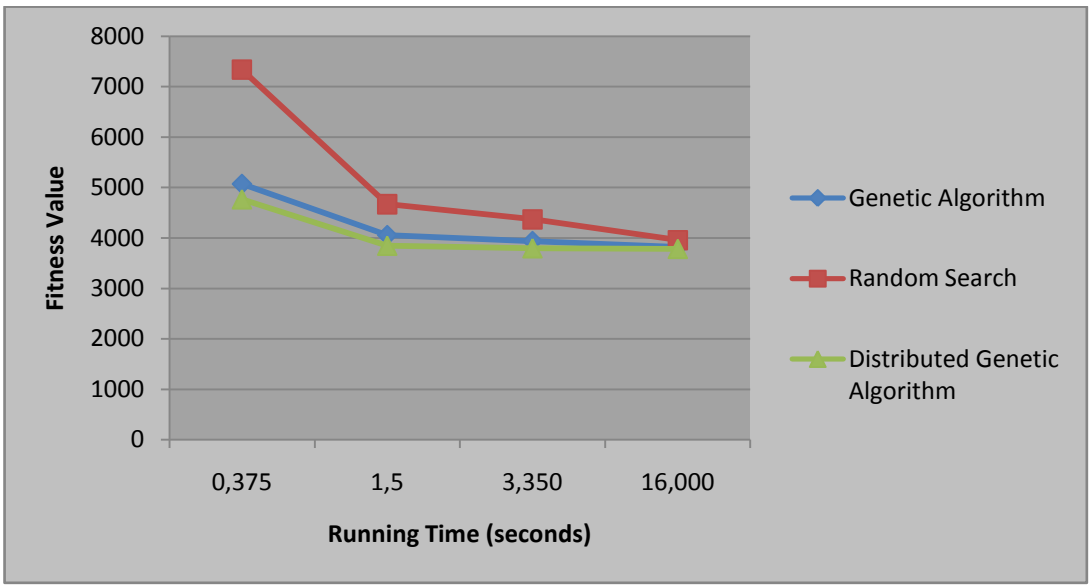


Figure 5.23: Running Time and Fitness Comparison between Genetic, Random Search and Distributed Genetic Algorithms

CHAPTER 6

CONCLUSION

The flight route planning problem is one of the substantial challenges in the concept of mission planning systems. The main trade off in flight route planning is between finding the optimal solution and finding the solution in a reasonable amount of time. The concern about time originates from the nature of reconnaissance. The results of reconnaissance missions are often meaningful and valuable for a certain amount of time.

Although many solution models are presented Travelling Salesman Problem suits the flight route planning problem the best. This study uses an approach that utilizes a Genetic Algorithm to solve the TSP. GAs give good approximations to the TSP in short amount of time. The GA is further supported with a Greedy Approach in order to converge to a solution even faster.

The devised algorithms have been implemented in two versions where they use serial and parallel execution methods respectively. The parallel version of the algorithm is developed using NVIDIA's CUDA compiler. The second implementation demonstrates both the modern GPU's processing power and CUDA compiler's capabilities. As two versions of implementation follow the same logical steps, the results of the study purely demonstrates the improvement maintained by this new technology.

At the end of the study it is shown that GPU processing capabilities are more than a match for the classical CPU based approaches 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.

6.1 Future Work

Future work related to flight route planning and CUDA programming are discussed separately in the following sections.

6.1.1 Flight Route Planning

Future work related to flight route planning is summarized as follows:

- The proposed flight route planning solution of this study does not take into account the effect of wind on the flight route. So that for a more realistic flight route, a model which includes aerodynamic calculations may be introduced.
- There may be zones which may be defined as unavailable on the Area of Interest resulting from real world situations.
- Other kinds of payload than the optical camera can be introduced to the solution model such as radar systems.

6.1.2 CUDA Programming

Future work related to CUDA Programming is summarized as follows:

- An Object Oriented (OO) version of the application can be developed using the probable future compiler of CUDA which supports OO languages such as C++ and Java.

REFERENCES

[1] Ying-Shiuan You. Parallel Ant System for Traveling Salesman Problem on GPUs. In GECCO 2009 - GPUs for Genetic and Evolutionary Computation. Pages: 1-2, 2009

[2] Martin Cross, Dr David Marlow and Dr Jason Looker. Application of the Non-stationary Travelling Salesman Problem to Maritime Surveillance. Proceedings of MISG 2007. Pages: 1-4, 2007

[3] Philip Kilby, Patrick Tobin, Ruth Luscombe. The Maritime Surveillance Problem. Proceedings of MISG 2007. Pages: 33-35, 2007

[4] Marlow, D.O. , P. Kilby and G. N. Mercer. Examining Methods for Maximizing Ship Classifications in Maritime Surveillance. 18th World IMACS / MODSIM Congress, Cairns, Australia. Pages: 1630-1632, 2009

[5] Barry Secrest. Travelling Salesman Problem for Surveillance Mission Using Particle Swarm Optimization. School of Engineering and Management of the Air Force Institue of Technology, Air University. Pages: 10-13 , 2001

[6] Kylie Bryant. Genetic Algorithms and the Traveling Salesman Problem. Department of Mathematics, Harvey Mudd College. Pages: 10-12, 20-23, 2000

[7] Maria John, David Panton and Kevin White. Mission Planning for Regional Surveillance, Annals of Operations Research, pp. 108, 157–173, 2001

[8] Asbjorn Bydal. Implementation of Genetic Algorithm on CUDA. http://fag.grm.hia.no. (last accessed date: 18 December 2009)

[9] TURKISH INDIGENOUS MALE UAV (TIHA). http://www.tai.com.tr. (last accessed date: 20 December 2009)

[10] Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., Shmoys, D.B. (1985), The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, Wiley, Chichester.

[11] G. Üçoluk. Genetic Algorithm Solution of the TSP Avoiding Special Crossover and Mutation. Intelligent Automation and Soft Computing, 3(8), TSI Press, Pages: 1-3, 2002.

[12] NVIDIA. GPU Gems 3. Addison-Wesley Pearson education, 2008.

[13] NVIDIA Compute PTX ISA 1.2 manual pp.9

[14] John Hershberger, Subhash Suri. An Optimal Algorithm for Euclidean Shortest Paths in the Plane. SIAM Journal on Computing 28 (6), Pages: 1-12, 1997

[15] Pankaj K.Agarwal, R.Sharathkumar, HaiYu .Approximate Euclidean Shortest Paths amid Convex Obstacles. Proceedings of the nineteenth annual ACM symposium on Theory of computing, Pages: 1-10, 2009

[16] Mariusz Nowostawski, Riccardo Poli. Parallel Genetic Algorithm Taxonomy. KES'99 Pages: 1-5, 1999