Multi-UAV Path Planning with Parallel Genetic Algorithms on CUDA Architecture

Ugur Cekmez
Yildiz Technical University
Computer Engineering Department
istanbul, Turkey
ucekmez@yildiz.edu.tr

Mustafa Ozsiginan
Turkish Air Force Academy
Aeronautics and Space Technologies Institute
Istanbul, Turkey
mustafaozsiginan@gmail.com

Ozgur Koray Sahingoz
Turkish Air Force Academy
Computer Engineering Department
Istanbul, Turkey
sahingoz@hho.edu.tr

ABSTRACT

In recent years, the use of Unmanned Aerial Vehicles *(UAVs)* has grown quickly due to its low cost and easily programming for autonomous path following for accomplishing different types of missions. Due to the numerous advantages of multi-UAVs, when comparing with a single powerful one, to perform reconnaissance, monitoring, detection and surveying missions the use of multi-UAVs is generally preferred. While the number of control points and the number of UAVs are increased, the complexity of the problem also increases. This paper presents a solution to the problem of minimum time coverage of ground areas using a number of UAVs. The solution is divided into two parts: Firstly the area is partitioned with K-means clustering and then the problem is solved in each cluster with parallel genetic algorithm approach on CUDA architecture. To illustrate the methodology, the paper presents the experimental results obtained with a multi-UAV system, which has a different number of control points. The results showed the proposed approach produces efficient solutions for these type NP-Hard problems of homeland security applications like wide-area surveillance and site security by using multiple UAVs.

Keywords

Parallel Evolutionary Algorithms; K-Means Clustering; 2-opt; Multi-UAV Path Planning; Genetic Algorithms

1. INTRODUCTION

An Unmanned Aerial Vehicle *(UAV)* is a remotely or autonomously controlled aircraft that can carry different payload types such as cameras, sensors, communications, and electronic warfare equipment. Due to its properties like small size, low cost, low risk for human operator/pilot, increased flight time; UAVs have enormous potential in civilian and military domains.

As a consequence of growing research in robotics and autonomous control systems, the UAV technology has seen a fast ascent in the previous two decades. UAVs have increased usage in firefighting applications, military missions, search and rescue scenarios, surveillance, reconnaissance, target engagement, exploration of unknown environments, etc.

Apart from the commercial planes which fly the predefined trajectories, UAVs have dynamically changing ones that depend on the terrain and the mission dynamics. The path planning is an essential part of the UAV's autonomous control module and it mainly draws the path that a UAV will go over. With the detailed definition, a path planning enables a UAV to find the optimal path from a start point to an end point while visiting all necessary control points *(CPs)*. According to the required mission, there needs to be some criterion such as minimizing the traveled distance, the average altitude, the fuel consumption and the radar exposure. In this case, the path planning process can be formulated as a minimal cost optimization problem.

As the constraints of the problem and the number of control points increases, the complexity of the problem has grown. To cope with this complexity, researchers have gradually moved from using deterministic algorithms to using non-deterministic ones such as evolutionary / swarm algorithms. Usage of Genetic Algorithms *(GAs)*, Particle Swarm Optimization, Ant Colony Optimization, etc. showed an increased performance on path planning type NP-Hard problems. However, while the number of CPs are increased, these algorithms are not sufficient especially in time-constrained conditions. Therefore, parallel implementation of these algorithms is implemented in some studies to solve the problem in a fewer execution time. By this way, acceptable UAV path planning can be achieved in large-scale application areas.

Usage of single UAV is not sufficient to complete the mission in an acceptable time in case of the enlarged mission theatre. The increasing demand on UAVs has brought into focus several challenges, which are associated with the use of multiple UAVs in the operations. Employing UAV teams can reduce the time to accomplish the required mission; however the problem becomes an optimal resource allocation problem, which is NP-hard. Therefore, in recent years, usage of multi-UAVs is preferred, and the number of UAV is added as a new constraint of the path planning problem.

UAV Path planning problem can be seen as a modified form of Travelling Salesman Problem *(TSP)* in its basic representation. TSP type problems can be used in the formulation of different application areas. Especially, in the last decades, researchers have began working on large-scale ones

GECCO'16 Companion, July 20-24, 2016, Denver, CO, USA

DOI: http://dx.doi.org/10.1145/2908961.2931679

and the aim is to have high quality solutions in a reasonable time period. As the hardware systems are improved in terms of computational resource, most of the studies focus on increasing the full use of this resource as well as increasing the throughput of the optimization algorithms' solutions while decreasing the computation time.

One of the key points for autonomous control of a UAV is to find the shortest flying path by fulfilling whole constraints such as threatened area, obstacles, borderlines and the aim of the mission. In recent years, researchers have been studying and proposing a wide variety of solutions for UAV path planning. There exist lots of path planning techniques, which use A* [13], Genetic Algorithms [4], Ant Colony Optimization [5], Particle Swarm Optimization *(PSO)* [2], etc. These techniques mostly find the convergence to the optimal results as they are using known data sets at hand.

Sanci and Isler [11] suggest a solution to calculate flight planning for a single UAV. The proposed algorithm uses parallel GA that works on Graphical Processing Units *(GPUs)*. The study indicates that the CPU and the GPU codes vary in terms of the constraints and resource the programming languages use, but both can be thought as logically equivalent. However, they solved only a single UAV path planning problem. On the other hand, Sathyaraj et al. [13] compared the computational times of Dijkstra, Bellman-Ford, Floyd-Warshall and the A* algorithms for multi-UAV path planning. The results in the study show that A* performed better in finding the shortest path for multi-UAV system comparing to other algorithms. Although they tried to solve a multi-UAV path planning problem, the proposed algorithm is not sufficient while the number of CPs is increased.

However, multiple UAVs' path planning problem is similar to multiple TSP problems *(or mTSP problems)*, in its simplest mode. In study [8], the authors propose two methods for partitioning the problem set for multi-UAV path planning. According to the study, K-Means and Gaussian Mixture Model are used to partition the problem and GA as well as ant colony optimization is used to find the optimal paths for the sub problems. The algorithms are implemented to run on CPUs. In [10], Sahingoz proposed another solution of multi-UAV path planning problem by using genetic algorithm. However, due to its computational time constraints, he conducted a test on a relatively small scale problem with 132 control points.

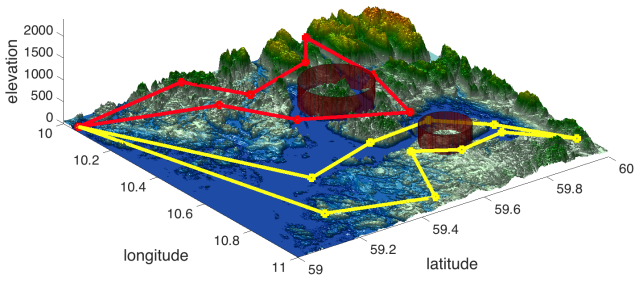

Figure 1: A sample multi-UAV path in 3D Environment

Apart from these studies, we are focused on solving a large scale UAV path planning problem with numerous control points with the help of evolutionary algorithms and with the power of parallel computation on CUDA architecture.

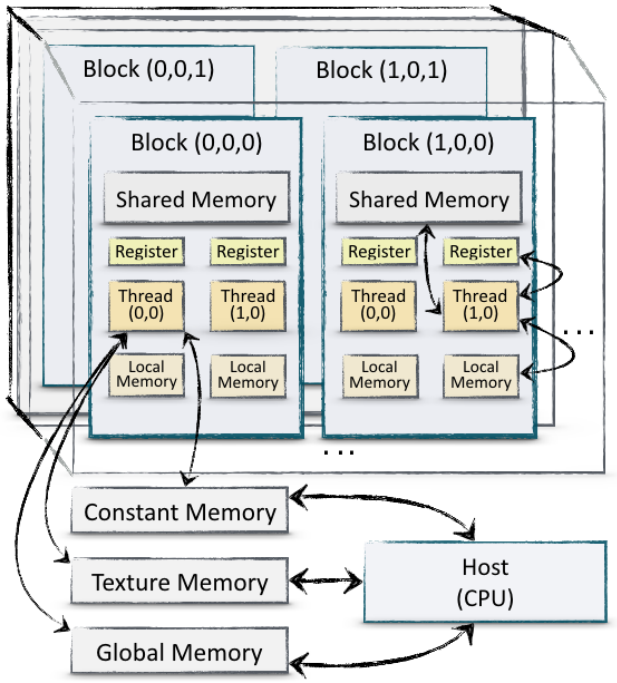

Figure 2: Thread - Memory organization in CUDA.

Therefore, in this paper, it is aimed to check several control points in the mission theater by using multiple UAVs as depicted in Figure 1. Due to the increased number of control points, this problem turns into a more complex form, and it takes a great time to solve with deterministic algorithms. Therefore, a parallel GA is preferred, and the proposed algorithm is tested on a parallel programming and computing platform: CUDA. The experimental results validate the effectiveness of the proposed solution.

This paper is organized as follows. The underlying architecture is presented in Section 2. The methodology of the proposed algorithm including the seed calculation, clustering and solving with genetic algorithm, in which all are implemented on parallel architecture, are detailed in Section 3. In Section 4, the experimental results are depicted and compared with serial execution. Finally, we conclude in Section 5 with some thoughts on future work.

2. UNDERLYING ARCHITECTURE

In recent years, the GPUs are one of the most preferred parallel computing platforms in massive computation with reasonably low cost. Although the GPUs are the easy-to-buy computing devices, they can perform thousands of Giga-FLOPS *(floating point operations)* per second. Since NVIDIA announced its CUDA architecture in 2006, GPUs are in the way of becoming one of the general-purpose computation devices under consideration. Comparing with the conventional CPU architectures, GPUs have a better organization for implementing data intensive parallel algorithms. In the hardware model of CUDA capable GPUs, there are several Symmetric Multiprocessors *(SMs)* including many low-frequency cores on the chip. In the application model of CUDA, there are a number of three-dimensional grids. These grids have the threads located in the thread blocks.

Each thread has its own copy of device function and private data. The basic thread-memory organization model of CUDA is shown in Figure 2.

Considering the basic software model of CUDA, simple algorithms turn into the large problem solvers by simply dividing the problems into parts, solving these parts separately and combining them with a proper data type. Then it becomes a parallel working model of the corresponding algorithm.

3. PARALLEL SOLUTION OF THE PROBLEM

In this study, it is aimed to solve the multi-UAV path planning problem by using an evolutionary algorithm: GA. It is known that the evolutionary algorithms are one of the most used algorithms in solving NP-hard problems where the complexity increases exponentially as much as the problem domain gets bigger [7]. While the evolutionary algorithms produce a feasible solution in an acceptable time, increasing the problem size still requires more time. Therefore, there is a need for an additional mechanism to solve the problem in a quicker way. Comparing the efficiency of both the conventional and the parallel solutions for the evolutionary algorithms, especially for GA, one can clearly see that it is no longer feasible to use the classical serial models considering the recent advances [4, 6]. Due to the low cost of GPUs and their usage in the field of general-purpose parallel computation, buying and using these GPU cards are getting easier for the end user and for the scientists. As a result, the scientists trend to focus on solving their problems on these parallel computation platforms by implementing new algorithms. Based on such a promising technology, in this study, a parallel GA model is developed and adapted to solve the TSP-like path finding problems. In the scenario of this study, a multi-UAV system is optimized where each city in the TSP is considered as a control point and the UAVs are responsible to fly over these control points. UAVs are then expected to return to the depot point. A sample mission theater as depicted in Figure 1.

The scenario requires the problem domain to be partitioned into K convenient parts and each part is assigned to a single UAV. In this case, partitioning must be in a meaningful structure which satisfies two conditions. The first one is to have each UAV fly over such a cluster that other UAVs do not overlap the control points of others. The second one is that sending multiple UAVs must decrease the mission completion time. For this aim, an easy-to-implement clustering algorithm, K-Means, is chosen and implemented in parallel to take advantage of the underlying GPU to the utmost. After the partitioning is completed, the problem is divided into K sub problems in which each problem will be solved as a single UAV path planning problem. Having many sub problems that are smaller than the whole original problem means that there exist many relatively small search spaces. As the problem domain gets smaller, each UAV is now responsible for traversing smaller number of control points and the time complexity of the problem reduce substantially, too. Knowing that the problem domain is now smaller for each UAV, the GA, running for each sub problem with the same evolutionary operators, will theoretically end up with better fitness values. Even if the population number and the iteration count of each sub GA are decreased in a reasonable amount, the possibility of achieving good fitness values may stay high by gaining the extra computation time. The parallel GA algorithm is explained in section 3.3 in detail.

This approach has outstanding gains since it allows to simplify the path planning problem and needs less computation resource, but there are still a few issues to be considered in the background, such as interpreting the outcomes of using either one UAV for the whole problem domain or multiple UAVs for the partitioned sub problems. One of the main criteria that must be addressed in such an interpreting is whether using multiple UAVs is more useful considering the management resource of all these UAVs, or does it waste these resource. This argument is discussed with the help of experimental results taken from this study in Section 4.

In this study, the proposed algorithm is composed of three main functions which run on GPU: generating random number seeds for the GA work flow, constructing the initial population of size N and evolving it through the generations, and finally solving the problem with parallel genetic algorithm.

3.1 Empowering Probabilistic Operators

It is known that genetic algorithm approach inherently hosts for probabilistic operators. Varying from initializing the first population to selecting the parents for the next generation children, the GA needs high quality of randomness to keep the results from converging poor results.

As of the first step, the algorithm computes N random number seeds by using *cuRAND* defined in CUDA SDK. All these seeds are kept in a 1-D array in the GPU memory. There are two main reasons this library is used. One of them is that the seeds are created by utilizing all the available cores in the GPU, which means that creating the seeds is almost at no cost. After filling in the array of seeds, the probabilistic operations of GA are able to use the corresponding seeds to provide randomness. Each CUDA thread having a unique ID pops its seed and constructs a random number when it needs. The other highlight here is that *cuRAND* provides more realistic randomness comparing to standard *rand()* functions of C++. The proposed algorithm in this study uses N threads to construct a random seed array.

3.2 Parallel K-Means Algorithm

K-Means is a simple clustering algorithm that partitions a given problem set into K clusters. Each entry in the clusters has the nearest mean with respect to their similarity measure. The mean of each cluster is named as *centroid*. The centroids are iteratively updated to maximize the similarity of entries inside the clusters. In this study, it is aimed to reduce the heavy parts of the K-Means algorithm by basically distributing its centroids through a number of threads where each point is assigned to a centroid in parallel.

The basic work flow of how the partitioning is done is defined in the pseudo-code in Algorithm 1.

As a detailed explanation, the inputs to the K-Means algorithm are the coordinates of the control points in this scenario. The parallel algorithm determines K random centroids by going through these points at first. This step is run under a CPU and then all the points including the centroids are sent to a function that runs parallel on the GPU. In this step, each thread represents a point in the problem domain. Each thread takes one point and loops through the centroids. Each thread then assigns a label of the closest centroid to

Algorithm 1 Basic work flow of K-Means for partitioning the problem space

```
1: procedure K-Means
2:   for iter in range(100) do
3:     for i in range(K) do
4:       centroids.append(randomPoint)
5:     end for
6:     for each thread as point p do
7:       labels.append(findNearestCentroid(p,centroids))
8:     end for
9:     for i in range(K) do
10:      updateCentroid(centroids(i),labels)
11:    end for
12:  end for
13:  result ← getTheBestPartition
14: end procedure
```

it's point. The function returns a label array where each index in the array corresponds to a label of a point. Each label here corresponds to a sub cluster. The centers of the labeled points represent the new centroids of each cluster. In this step, as the partitioning process can be distributed over K sub problems and K is a relatively low number, solving it with the high-frequency CPU cores in parallel would yield better time performance comparing to low-frequency GPU cores. However, as the problem size and K gets bigger, it is observed that using GPU parallelism turns out to better serve it. The process continues over several iterations aiming to search for better cluster distributions. In this study, K-Means Algorithm has a fixed number of iterations where the iteration count is 100. The algorithm runs 10 times for each problem domain and the best distribution is fetched to better feed the Genetic Algorithm. The basic principle of the parallel K-Means Algorithm is shown in Figure 3.

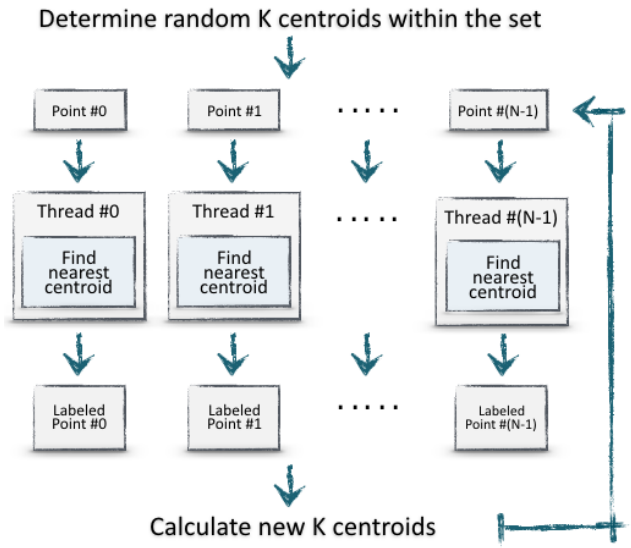

Figure 3: Parallel K-Means Algorithm.

3.3 Parallel Genetic Algorithm

One of the main efforts in search of having better convergence for the big problem space is generally to optimize the conventional approaches with the help of accumulated knowledge. Combining this knowledge with the modern hardware may result in a state of the art solution in return. Considering the constraints of the various types of UAVs in real world, it can be argued that a single UAV can only be actively used in the small-scale missions. It is likely that distributing a task to more than one UAV may yield better performance in a number of cases such as exploring a given area more effectively in less time. By using the advantage of being many, these UAVs can also be used to provide a fault-tolerance at a certain level.

In the scenario of this study, there is a problem space that is exceeding the limits of a single UAV, which the UAV will have difficulties to complete the whole task even if the path finding problem is solved optimally. As the main motivation of this study, it is aimed to distribute the problem domain to a multi-UAV system to apportion the heavy problem into more feasible pieces. In return, the gain varies from having the task completed in a shorter time to less use of the resource.

One of the conventional approaches to find the optimal path for a given set of points in various situations is to use the evolutionary operators of GA [1, 12]. Since it produces acceptable solutions in a limited time, the GA approach is taken into consideration and is optimized to run in parallel on GPUs. Multi-UAV problem requires division of the problem areas into sub domains and this can be done in various ways. In this study, this partitioning process is handled by the help of K-Means Clustering Algorithm and each partition is sent to the GA for producing an acceptable solution.

Initial Population: Creating an initial population for the GA is probably one of the most aspects of the whole since it is the one step before the evolving process starts. All the evolution starts by taking the individuals from initial population and have them evolved by the genetic operators. An example individual is represented in Figure 4. To make a good start, one can take a computational model such as neighborhood metrics. It may be a desired option when the computational resources are limited to be able to handle a big population size or the computational time is restricted for a few iterations. Having a better initial population may return as a fast convergence to the near optimal solution. In this study, it is not indeed concentrated to make calculations to have a better initial population. Instead, the initial population is created randomly by simply shuffling the chromosomes in parallel.

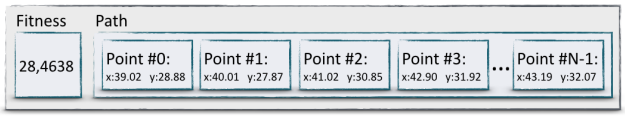

Figure 4: A sample individual in the population.

Comparing with the process of creating random number seeds in the previous section, it is similar to construct and fill in the initial random population array. In this step there are N CUDA threads each using previously constructed random number seeds with their IDs. Each thread is responsible for shuffling N chromosomes and then putting the newly created individual into the 1-D initial population array. The basic principle of creating this population is depicted in Figure 5.

Fitness Function: The fitness function in GA is what

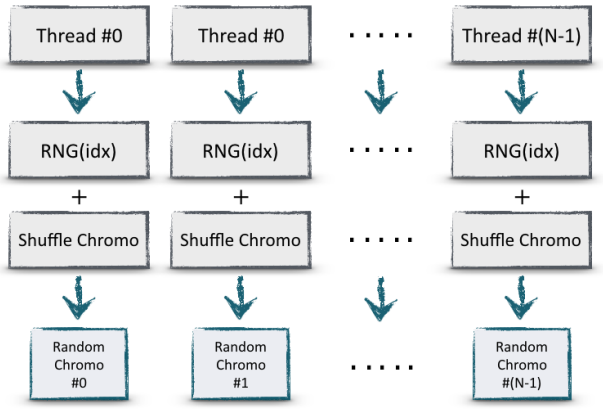

Figure 5: Constructing a random initial population.

defines the solution space of the problem. It is very crucial in the aim of solving the problem properly and being a guiding factor of evolving new solutions. In this study, the fitness function is as simple as calculating the total Euclidean distance between the points through the path the UAV follows.

Evolving the Population: After the initialization parts, all the necessities the GA requires has met. Now it is time to evolve the initial population until a stopping criterion is reached. In the parallel approach, all the conventional steps of the GA such as parent selection, crossover, mutation and elitism are used as they are. In addition, the experiments showed that using a local optimization technique *(2-opt)* increased the solution quality significantly at each iteration.

The evolutionary operators work as follows. The first step is to choose two individuals to match as the parents. The parents are put into the crossover step to combine their specific parts to create one new child. The newly generated child is then mutated, by a given probability rate. Before the child is ready to find a place in the new generation, the 2-opt local optimization technique is applied to that child to overcome the possible crosses in the calculated path. The basic work flow of evolving process is depicted in Figure 6. Detailed information about the evolving process is explained in the following subsections.

Elitism: The term elitism in the evolving process is simply to migrate some of the best individuals directly to the next generation without any genetic operators affected to those individuals. What the term best means is that these individuals, namely chromosomes, have the lowest fitness value in terms of the total distances of their solution paths. Keeping these elites from being affected by the genetic operators will guarantee that the overall best solution for each generation will not be worse than the previous generation. So, if the new generation produces worse children, the elite individuals are kept. In this study, the best 32 individuals of each generation in the iterations are kept as elites. Why the number stays 32 for each problem set lies in the structure of how CUDA threads handle the conditional statements in the code. The CUDA threads work truly parallel in the thread groups called warps. Each warp has 32 threads in the SMs *(Symmetric Multiprocessors)*. Each warp works concurrently in the whole system. It is known that if there are conditional statements in the CUDA code such as "*if, else*".

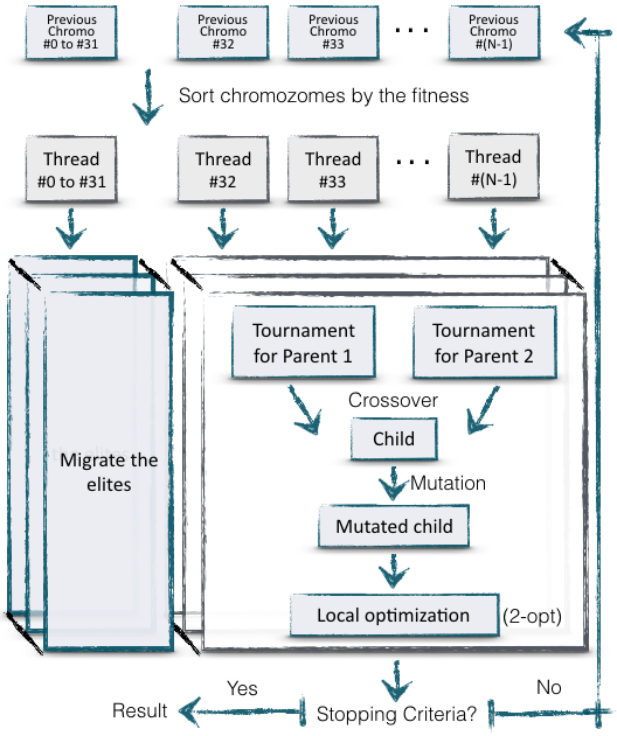

Figure 6: Evolving step for each individual.

These statements are handled by warps and the warps met the "*if*" condition works first and after that the "*else*" statement works. This means that if there are 32 *(or its multiples)* threads that met the condition, then all these threads work in parallel as warps. Similarly, if the number of threads is fewer than 32 *(or its multiples)*, then there are free slots in the warps while the "*if*" condition is processed. This is the basic definition of why elitism is taken as a fixed number.

Selection of Parents: The individuals not in the elite list are replaced in the evolving process with the help of genetic operators. According to a selection rule, in this case it is tournament selection, four individuals are randomly selected. After selecting the individuals, the best individual of four is selected as the first parent. The same process is followed to choose the second parent. Then these two parents are collected by one thread, which is responsible for the i_{th} index in the individual chromosome in the population array.

Crossover: Creating a new child by combining the parents with a selected technique is called crossover. It is one of the core concepts in the GA. The principle here is to inherit the characteristics of the parents to the new child. As one can create two children from a crossover, in this study the crossover produces one child. It is aimed for one thread to be responsible for one index in the population and replace it with the new generation individuals through the iterations. For the simplicity of its implementation, the 1-point crossover technique is selected. In 1-point crossover, starting with the first field, a randomly selected portion of the first parent is directly put to the new child and the remaining blank fields are then filled in by the second parent. To prevent the duplications in the newly created child chromo-

some, only the unselected parts of the first parent is chosen from the second one.

Mutation: As the iterations continue, there is a possibility for the chromosomes to converge to a local optimum value. This happens when a poor quality of the initial population is used or when the random selections are not random enough as explained in the previous sections. Even the structure of the problem or the point sequence in the problem may be inclined to be converged to a local optimum. In such cases *(including the scenarios where everything goes well, as it is supposed to be)* a jumping operator, which the individuals are changed according to a technique is used. This step is called mutation. In mutation step, the individual is mutated by a possibility. By mutating some individuals, the diversity between the generations tends to be high. In this study, an individual is simply mutated by randomly selecting its two points and swap them. Swapping the points may result in a small or a big change in the fitness of the individual. The individual may have better or worse fitness after the mutation. However, there is a local optimization to prevent the generations getting worse.

Local Optimization: The conventional GA saves the elites of each iteration and it guarantees that the best solutions stay there, thus it keeps the diversity of the generations. The steps explained above are very promising with the aim of optimizing the problems. However, the experiments showed that sticking with the conventional approach requires so many iterations to reach a feasible error rate and it is likely to converge to a sub optimal solution. To keep from converging immediately, a local optimization technique can be applied to a few chromosomes.

In this study, 2-opt local optimization technique is adapted to fine-tune the newly generated chromosomes from the GA operators. 2-opt takes a chromosome that has a path crosses over itself and re-constructs it to delete the cross. It does the job by iterating over the path of chromosome N times and checking whether there exists a hidden pair of nodes that is better than an actual one [9]. The procedure behind the 2-opt is straightforward. If $distance(i, i+1)+distance(j, j+1)$ is smaller than $distance(i, j+1)+distance(j, i+1)$ then remove the connection between i and $j+1$ and connect i with $i+1$. Similar connection is made on j and $j+1$.

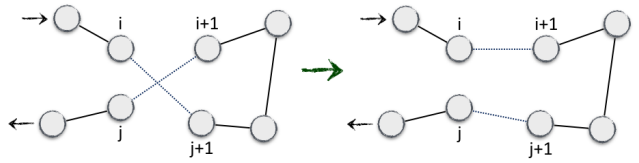

Figure 7: 2-opt local optimization technique.

Just to be able to use the underlying massively parallel architecture to the utmost, we applied a 2-opt local optimization technique for the first 32 generations in the whole iterations. It is experienced that having 2-opt enabled for the generations turns into a time consuming ($BigO(N^2)$) event and no significant quality difference is observed comparing to its usage on the first 32 generations. The number 32 comes from the GPU warps that are explained in the earlier sub sections. So, applying 2-opt for a few generations turns each of those generations better. So from where it left off, continuing the conventional approach without local optimization still brings better results but in a shorter time.

Sorting Mechanism: Recall that the elite individuals are kept from the ongoing evolving process. To apply the elitism step in GA, there is a need for a sorting mechanism that moves the best individuals to a specific location. For this aim, a library called *Thrust* in the CUDA SDK is used. *Thrust* library uses the GPU power to sort custom vector types such as vector of chromosomes as they are the individuals in the current population. Since the new individuals may be better than the previous elite ones, the sorting mechanism works at the end of each evolution to pick the new elites.

Stopping Criteria: One of the important factors is to decide when the GA should be stopped. In this study when there are 100 iterations *(for both parallel and serial versions of the algorithm)* the GA stops and the best-so-far individual is considered as the result.

4. EXPERIMENTAL RESULTS

Simple UAV path planning problem is a TSP-like problem in case of its limitation. Both the serial and parallel equivalent versions of the proposed algorithm used the same problems. The results are compared and discussed by their speedups and solution qualities as well as the gains that having a multi-UAV system has brought. The problem sets and their parameters are shown in Table 1.

Table 1: Problem sets and experimental parameters

Parameters	Values
# of visiting CPs (+ 1 airport)	100 / 255 / 439 575 / 1002 / 2392
Population Size	1024
K (partition size)	1 / 2 / 3 / 4 (UAVs)
Elitism	First 32 chromosomes in each generation
Parent Selection	Tournament (select 4 and get the best)
Crossover type	1-point
Mutation type	Swap
Mutation rate	0.08
Local Search Type	2-opt
Local Search Applied	First 32 generations
# of Iteration	100

In this study, the parallel model of K-Means and GA is implemented on NVIDIA GeForce GTX 970 graphics card. The Operating System is Ubuntu 15.10 where it runs CUDA SDK 7.5 and the programming language is CUDA C. The serial versions of the algorithms are run on an Intel i5 2.7 GHz CPU and the algorithms are written in C++. Table 2 shows the detailed underlying CPU and GPU architectures.

In the parallel algorithm, the first step is to apply K-Means to partition the problem set into K solvable pieces. The 100-CPs problem, as an example, is partitioned into 2 sub problems after the K-Means applied. After partitioning the data, each partition is solved by the GA separately. At this point, each UAV takes one portion of the problem and is required to complete its task. The UAVs take off from an airport point and to return to the depot point (they are assumed to be same point). Figure 8 shows the routed version of the previously partitioned problem.

Table 2: Hardware Features for the CPU and the GPU used in this study

	CPU	GPU
Manufacturer	Intel	NVIDIA
Model	i5	Geforce GTX 970
Architecture	Haswell	Maxwell
Clock-frequency	2700 MHz	1228 MHz
Cores	4	1664
DRAM Memory	8 GB DDR3	4 GB DDR5

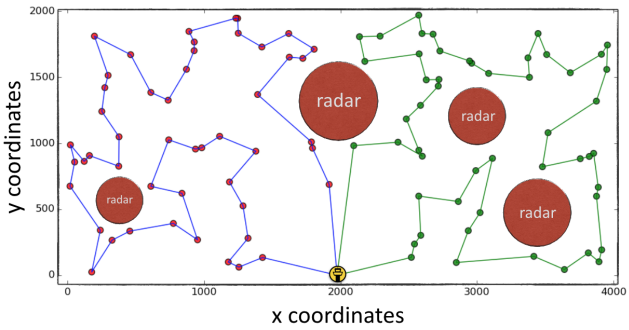

Figure 8: The result of 100-CPs problem when there are obstacles

In the experiments, there are 4 scenarios taken into account. The first one is when only one UAV gets the mission. The other scenarios are when there are 2, 3 and 4 UAVs respectively. In each scenario, the number of UAV is equal to K in the K-Means algorithm.

The experiments show that increasing K yields 1) less computation time of running GA in total and 2) shorter duration of completing the problem assigned to the UAVs. As an example, Figure 9 shows that one UAV with the speed of 40 km/h completes the 2392-point problem in 277 minutes. Using one UAV for this type of mission seems infeasible. As of the year 2016, small UAVs reach the capability of speed up to 60 km/h with approximately 60 to 80 minutes of flight time [3]. If we try to solve the problem with 2 UAVs, then the problem traversed in 135 minutes. Three UAVs traverse it in 115 minutes and 4 UAVs in 66 minutes. The traversal time is determined by the least recently completed path. Considering that 2392-CPs problem is located in a large space, 1 UAV may not finish visiting all the CPs in such a long time. Instead, letting 4 UAVs visit all the partitioned CPs is more feasible and the time it takes to finish the whole area has 4.20x speed up. The other problem sets show similar results where increasing K decreases the path completion time. As an overall view, the detailed computation times for different number of control points are shown in Figure 10.

In the experiments, two versions of the same multi-UAV path finding algorithm are designed and implemented. These are the parallel and the equivalent serial versions. They are compared in terms of the execution times where K-Means partitions the problems and the GA solves them. In this scenario, the computation time comparison is made with respect to the Equation 1.

$$Speedup(solution) = \frac{T_{exec}(serial)}{T_{exec}(parallel)} \quad (1)$$

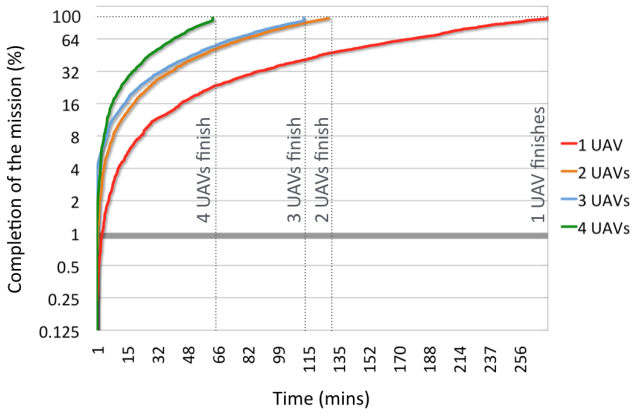

Figure 9: Flight time comparison for 1-2-3-4 UAVs to finish the 2392-CPs problem.

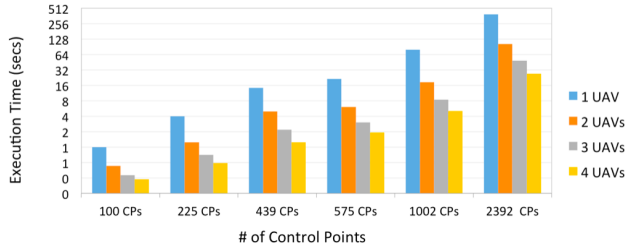

Figure 10: GPU computation time comparison of different number of control points.

According to the Equation 1, the speedup gains of parallel implementation are calculated. The experiments show that the speedup of parallel version, varying from 1 UAV to 4 UAVs is very promising. The gains differ from 100x to 558x for 1 UAV, 67x to 358x for 2 UAVs, 79x to 335x for 3 UAVs and 185x to 223x. The computation time difference depends on the problem domain where the points are located in specific locations. So, K-means Algorithm may produce an undesirable partition that makes one part of the problem very big comparing to other parts. As an example for this, there is a speedup loose in 100-CPs problem as the K increases, because the sub problems K-Means generated have a big difference in size. However, in 225-CPs problem, as the K increases, it gains speedup. That is because the sub problems are almost in equal size. On the other hand, considering the 439-CPs problem, increasing K gives an irregular behavior in the result. The reason is having the sub problems in different portion in size as the K changes. As another aspect, there also seems a general decrease in the speedup when the problem size gets bigger. Because as they get bigger, they require much more computation power to be solved, so the free slots of the GPU are fulfilled and the process is continued serially at some point. This exposes the limits of the underlying hardware.

To depict a clear distinction between the GPU and CPU implementations of the proposed algorithm, execution times of the serial runs are shown in Figure 11 and the parallel runs are shown in Figure 12 It is clearly seen that in the proposed parallel model there is a substantial improvement on the execution time.

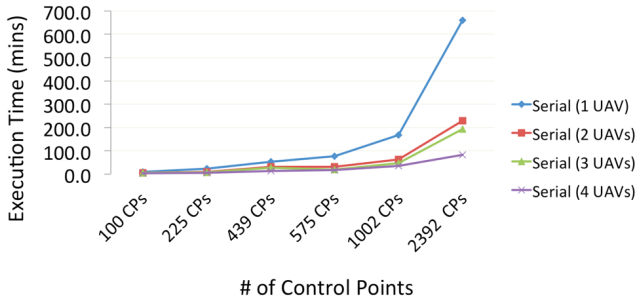

Figure 11: Serial computation times for different number of control points.

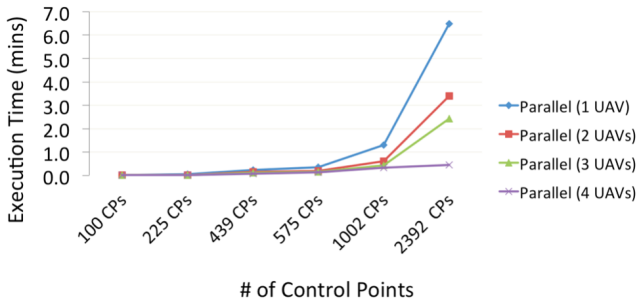

Figure 12: Parallel computation times for different number of control points.

5. CONCLUSION

In conclusion, this paper mainly discusses how to plan feasible paths for multi-UAVs by using a parallel Genetic Algorithm on CUDA architecture. While the number of control points and constraints are increased, the path planning for a single UAV is a trivial issue. According to mission needs, the use of multi-UAVs is inevitable in many cases, and this also increases the complexity of the problem. So, it is aimed to solve the problem in a parallel programming and computing platform: CUDA. In the proposed algorithm, we first use a clustering approach to find the subsets of control points. And then, a parallel genetic algorithm is used to solve each cluster. The proposed approach remarkably reduces the computational time and gives much better results than serial algorithms.

6. ACKNOWLEDGMENTS

We would like to thank the Aeronautics and Space Technologies Institute, Turkish Air Force Academy for letting us to use their GPUs in the Parallel Programming Lab.

7. REFERENCES

[1] J. d. S. Arantes, M. d. S. Arantes, C. F. M. Toledo, and B. C. Williams. A multi-population genetic algorithm for uav path re-planning under critical situation. In *Tools with Artificial Intelligence (ICTAI), 27th International Conference on*, pages 486–493. IEEE, 2015.

[2] Y. Bao, X. Fu, and X. Gao. Path planning for reconnaissance uav based on particle swarm optimization. In *Computational Intelligence and Natural Computing, Second International Conference on*, volume 2, pages 28–32. IEEE, 2010.

[3] Baykar. Technical Features of Bayraktar Mini UAS. http://baykarmakina.com/en/sistemler-2/bayraktar-mini-iha/#1458634622425-7f20b8f9-b28c. Accessed: 2016-04-03.

[4] U. Cekmez, M. Ozsiginan, and O. K. Sahingoz. Adapting the ga approach to solve traveling salesman problems on cuda. In *Computational Intelligence and Informatics (CINTI), 14th International Symposium on*, pages 423–428. IEEE, 2013.

[5] U. Cekmez, M. Ozsiginan, and O. K. Sahingoz. A uav path planning with parallel aco algorithm on cuda. In *Unmanned Aircraft Systems, International Conference on*, pages 347–354. IEEE, 2014.

[6] Y. Lu, L. Zheng, L. Li, and M. Guo. Parallelism vs. speculation: exploiting speculative genetic algorithm on gpu. In *Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 68–74. ACM, 2015.

[7] F. Neumann and C. Witt. Bioinspired computation in combinatorial optimization: algorithms and their computational complexity. In *15th annual conference companion on Genetic and evolutionary computation*, pages 567–590. ACM, 2013.

[8] T. Phienthrakul. Clustering evolutionary computation for solving travelling salesman problems. *International Journal of Advanced Computer Science and Information Technology*, 3(3):243–262, 2014.

[9] K. Rocki and R. Suda. Accelerating 2-opt and 3-opt local search using gpu in the travelling salesman problem. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 489–495. IEEE, 2012.

[10] O. K. Sahingoz. Flyable path planning for a multi-uav system with genetic algorithms and bezier curves. In *Unmanned Aircraft Systems (ICUAS), International Conference on*, pages 41–48, May 2013.

[11] S. Sancı and V. İşler. A parallel algorithm for uav flight route planning on gpu. *International Journal of Parallel Programming*, 39(6):809–837, 2011.

[12] A. Sathyan, N. Boone, and K. Cohen. Comparison of approximate approaches to solving the travelling salesman problem & its application to uav swarming. *Int. J. Unmanned Syst. Eng*, 3(1):1–16, 2015.

[13] B. M. Sathyaraj, L. C. Jain, A. Finn, and S. Drake. Multiple uavs path planning algorithms: a comparative study. *Fuzzy Optimization and Decision Making*, 7(3):257–267, 2008.