UAV Path Planning with Parallel Genetic Algorithms on CUDA Architecture

Ugur CEKMEZ, Mustafa OZSIGINAN, Musa AYDIN and Ozgur Koray SAHINGOZ

***Abstract*——In recent years, Unmanned Aerial Vehicles *(UAVs)* have been emerged as an attractive technology for different types of military and civil applications, which have gained importance in academic researches. In these emerging research areas, *UAV* autonomy gets a great part of the study, and mainly it refers the ability for automatic take-off, landing and path planning of *UAVs*. In this paper, we focused of the path planning of *UAVs* for controlling a number of waypoints in the mission area. If the area is large and the number of points that must be checked is greater, then it is not possible to check all possible solutions, therefore, we have to use some efficient algorithms, like genetic algorithms *(GAs)*, to calculate an acceptable path. However, if the number of waypoints exceeds a certain number, then we have to use some additional accelerating mechanisms to speed up the calculation time. Typically two techniques are used for speeding up: parallelization and distribution of calculation. In this paper genetic algorithm is parallelized on *CUDA* architecture by using Graphical Processing Units *(GPUs)*. Experimental results showed that this approach produces efficient solutions in a short time.**

***Index Terms*—Parallel Genetic Algorithm, CUDA, GPU, High Performance, 1T1P, Local Search**

I. Introduction

IN the last few decades, Unmanned Aircraft Systems *(UASs)* represent one of the most interesting technologies due to the advantages of Unmanned Aerial Vehicles *(UAVs)*, such as high mobility, light weight, stealth and zero emission properties, etc. As a result of these advantages, *UAVs* are very useful tools not only in military but also in civilian missions such as surveillance, reconnaissance, targeting, etc. A *UAV* is mainly a remotely controlled or autonomously controlled aircraft, which can carry a specific payload such as cameras; Electro Optical, Infrared, and Synthetic Aperture Radar *(SAR)*; some communication equipment and weapons for combat vehicles.

UAVs can vary in terms of their usage areas, ranges, sizes, and capabilities. Mini and lightweight *UAVs* have limited endurance and flying range, therefore, they are used for close-range missions. However, the larger and heavier ones have higher endurance, therefore, they can have a longer range and can be used in specific missions, which need operational level intelligence. At the same time, *UAVs* can also be divided into three main classes in terms of their design. Two of them are with heavier-than-air technology: rotary-wing aircrafts *(e.g. helicopters)* and fixed-wing aircrafts *(e.g. airplanes)*. One of them is aerostatic aircraft *(e.g. hot air balloons)*.

With their different type of evolution, *UAVs* have been used in lots of different application areas. By increasing the autonomous structure of the system, diversity of the application areas are increased. *UAVs* have increasingly become more autonomous and intelligent by its enhanced decision-making capability, which is programmed and loaded previously. Therefore, in the mission of a *UAV*, autonomous flying plays an important role by using path planning algorithms, which represent a prominent technology for highly independent operations.

In general path planning operation, it is aimed to find the optimal path from the starting point to the ending point subject to go over some waypoints by taking into consideration of some operational constraints. In this operation, several factors should be considered, such as UAV kinematics, terrain information *(e.g. mountains)* and threat information *(e.g. radars)*. In case of large mission area and large number of waypoints, it is hard to find best path due to the exponentially increased search domain. Evolutionary Algorithms *(e.g. genetic algorithms)* are optimization tools for *NP-hard* problems, such as the path-planning problem. Although they do not guarantee the best solution, it is possible to find near optimal solution in an acceptable time in a robust structure with compared to existing directed search methods. They are also easy to implementation both in serial and parallel architecture [1]. Therefore, *GA* is preferred for solving path planning problem of a *UAV*.

Although, *GA* finds a near optimal solution, if the number of constraints is increased, it gets a long time to find a solution. To solve these type problems in real time some additional mechanism must be taken into consideration to speed up the calculation time. In many research parallel solution is preferred by using multi processors, multi-cores, or Graphic Processing Units *(GPUs)*.

GPUs have emerged recently as an exciting new hardware which enables a highly parallel and fully programmable implementation and execution environment. They have excellent performance/price ratio, and they can reach a performance of thousands of giga-FLOPS *(floating point operations)* per second. Due to the inherently parallel nature of Genetic Algorithm, it is relatively easy way to implement on *GPUs*. However, it also brings some significant challenges due to its synchronization points and memory access patterns.

The aim of this paper was to propose efficient parallelization strategies of *UAV* Path planning system, which is implemented with Genetic Algorithms on *CUDA* architecture with Graphics Processing Units. The experimental results showed that, in case of large and complex problem domain, the proposed approach produces a near optimal solution in an acceptable time.

Manuscript received March 14, 2014 revised April 10, 2014

U. Cekmez, Department of Computer Engineering, Yildiz Technical University, Istanbul, Turkey, ucekmez@yildiz.edu.tr

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

M. Aydin, Department of Computer Engineering, Fatih Sultan Mehmet Vakif University, Istanbul, Turkey, maydin@fsm.edu.tr

O.K. Sahingoz, Department of Computer Engineering, Turkish Air Force Academy, Istanbul, Turkey, sahingoz@hho.edu.tr

ISBN: 978-988-19252-7-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

The rest of the paper is organized as follows: Section 2 presents the background information about *UAV* Path Planning, *CUDA* Architecture and Genetic Algorithms. Then Implementation details of the Parallel structure of the proposed algorithms are depicted in Section 3. Section 4. Gives the experimental result of the parallel approach and finally conclusion and future works are explained in Section 5.

II. Related Works

The path planning problem is similar to the traveling salesman problem that is one of the well-known and practiced combinatorial optimization problems in computer science. Many researchers planning to study on the *UAV* path planning starts to solve the *TSP* first, then adapts it to solve the path planning by considering the limitations of the *UAV* as well as the environment.

Sanci and Isler suggest an approach to solve the path planning problem by using parallel genetic algorithm on *GPU* architecture. The approach is converted to *TSP* in order to simulate it easily and it is compared to the *CPU* version and the experimental results show that the parallel version has a promising speedup rates. Although the *CPU* and the *GPU* versions of the algorithm are almost same, some hardware-dependent limitations make minor changes in the either codes [2].

In the survey that Knysh ve Kureichik has studied, the parallel approaches for the genetic algorithm are examined. The study shows that many of the works take the synchronization and the interaction between the populations into account so that it affects the distributed structure of the parallel *GA*. As a result, the problem-specific criteria is touched upon to see whether the parallelization of the *GA* speeds the process up[3].

Another study on implementing the *GA* to solve the *TSP* tries to develop order crossover with *2-opt* mutation. The approach is to parallelize some of the individual operators in addition to computing the consecutive parts. The individual parallelization is made for the crossover and for the mutation. In order to parallelize the crossover, the approach here is to produce only one child from two parents, as our proposed approach will also suggest. The experiments in this study show that the results reach up to *24* times speedup comparing to the *CPU* version of the algorithm [4].

III. CUDA Architecture

NVIDIA announced *CUDA* architecture in *2006* to enable parallel computations on *GPUs*, in preference to using graphics-purpose. *CUDA* is one of the parallel computing platform providing both high computation power and comparatively low cost.

Inside of the *CUDA* architecture, organization of the threads is located as a group in blocks and every block is located as a group in a grid.

Every thread is liable for copy of their functions and data part. In the runtime, each thread is created and executed concurrently as warps those are groups of *32* threads in a grid in the *GPU*. If the thread is in the same block with others, then it is able to communicate with each other by using their shared memory in the block they are assigned and they can be synchronized in the same block, too. In the *CUDA* architecture, each thread has control its own register and read/write data to its local memory. The thread-memory organisation is seen in Figure 1 as well as the *CUDA* hardware and the software model in Figure 2.

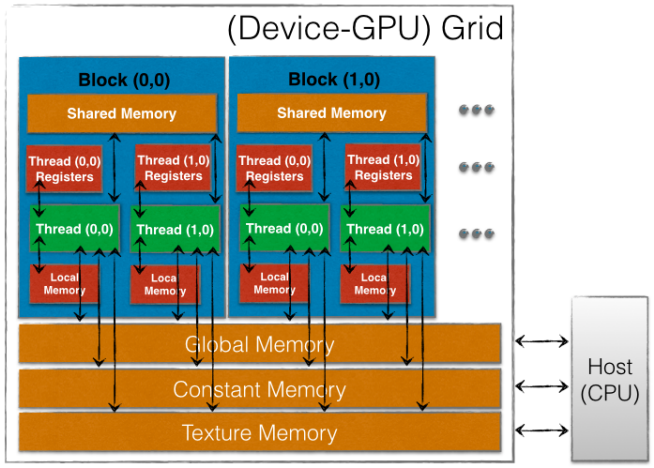

Fig. 1. GPU thread-memory organisation

In this study, the algorithm is developed on *NVIDIAs GeForce GTX 680* graphics card that has Kepler *GK104* architecture with *3.54* billion transistors, *1536* CUDA Cores and *3090* GFLOPs. The Graphics Clock speed is *1006* MHz, Memory Clock is *6008* MHz, Memory Bandwidth is *192.26 GB/sec* and *TDP* is *195W*.

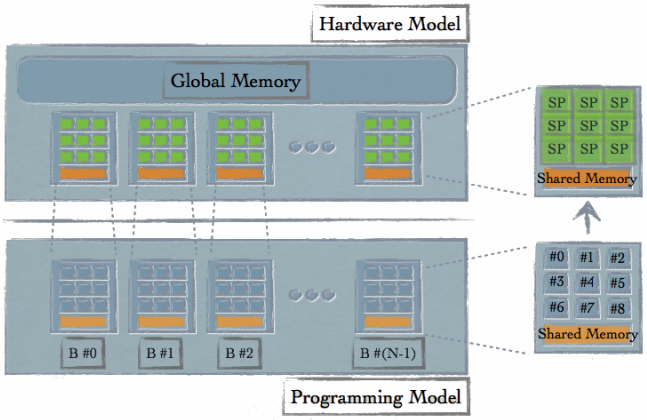

Fig. 2. Hardware and the programming model of CUDA. Streaming multi processor is in the above right box, the below is the block of threads. SP: Streaming Processor, B:Block

IV. Parallel Implementation of Genetic Algorithm for the UAV Path Planning

The aim of the *UAV* is designed to collect some necessary data from sensor nodes in Wireless Sensor Network, or disseminate them some important information such as public keys of group keys, as detailed in [5], [6]. Therefore, the proposed system mainly calculates a feasible path, which is depicted in Figure 3 over 3D environment. Firstly, the control points are converted to 2D space and then necessary calculations are executed accordingly.

The proposed approach for computing the path planning has three initial computations as well as two main functions

ISBN: 978-988-19252-7-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

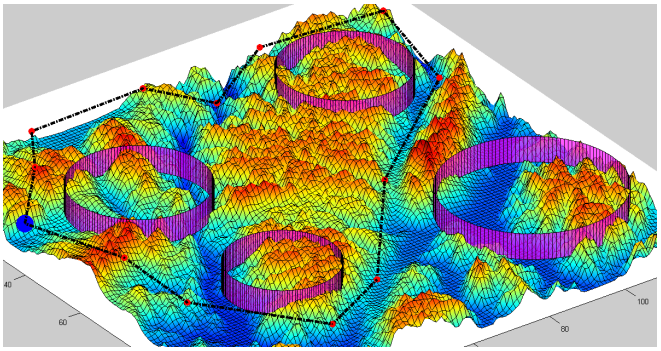

Fig. 3. A Sample UAV Path in 3D Environment

that are iterated over times. All the functions run in parallel and have some synchronization points when necessary. The initial and one-time computations are *1)* generating the random number seeders for further use, *2)* calculating the distances among all the cities in the task and then *3)* preparing an initial random population in order to start the evolving process after, respectively.

After the initialization part, the population is evolved iteratively until the stopping criteria have met. While evolving the population, the conventional steps of the genetic algorithm are included in the approach. These steps are *1)* choosing two parents over the current population, then *2)* creating a child by using the crossover method. In this study, the selection involves the tournament selection method and two parents are inter-crossed to produce one child instead of two. This way is more suitable when each thread is responsible for producing and evolving one individual in the whole population. New children of the generation are then *3)* mutated by a given probability rate. The threads satisfying the probability use the swap method of mutation. Each child, including the mutated ones, is sent to *4)* the local optimization function. The *2-opt* local optimization is used to overcome the possible crosses in the calculated routes, namely chromosomes. It is seen that applying the local optimization yields promising results in finding the optimum or near-optimum routes when there is enough power of calculation. The *GPUs* are proven to be very suitable when calculation is needed. So, since there is the opportunity of having these computation power over the *GPUs*, the local optimization is applied all of the children in the generation. As soon as the local search to optimize the chromosome is completed, the new fitness values of the new children are then calculated by the help of the distance table which have previously been filled in one of the one-time calculations. One of the evolving processes is thus completed by the fitness calculation. Before starting to compute the new generations of the evolution, it must be guaranteed that *5)* the elitism logic is satisfied. In the sense of elitism, here it means that the number of pre-determined children from the previously calculated generation remain same without encountering any operation, thus being transferred to the new generation of chromosomes. Those elites are the ones having the best fitness values. In this study, the number of elite chromosomes is set to be a constant value, *32*. Recall that *CUDA* sets the threads as a group of warps so that they run the same kernel code in parallel where the number of threads in a warp is *32*. So this number keeps the threads away of being divergent from the others when a condition is occurred in the code, hereby it keeps the parallelism high. Along the generations are evolved and the results are optimized, there occurs the need for *6)* a stop. In order to terminate the algorithm, several conditions may be appropriate to be stipulated. For instance, if the error rate of the best child from the last generation gives the exact best results or is sufficient to be the answer for the problem, then there is no need to continue the process. Other than the error rate condition, if there is a time limitation for the algorithm to be able to continue, then bounding the iterations is an option as a stopping criteria. In this study, the latter option is chosen to equally compare the time of both *CPU* and *GPU* versions where it takes to finish the algorithm [7].

A. Random Number Generation

The genetic algorithm approach inherently includes probabilistic operations at many parts of it, thus providing a high quality random numbers keep the output off from the results of poor quality.

In order to calculate the random numbers, the *CPU* version of the algorithm uses the *C++ rand()* function with a time seeder where the *GPU* version of the algorithm uses the open source *cuRAND* library which is in the *CUDA SDK*. *cuRAND* utilizes all the available cores dynamically in the *GPU* so that the throughput is maximized and the process is parallelized to the utmost as depicted in Figure 4. As the library providers indicate as well as the results it produces, the *cuRAND* has more realistic randomness at hand. The Random Number Seeds are calculated and kept into a *1-D* array for the further use. The array length is equal to the population size. When a thread needs a random number in an operation, it uses its thread *ID* to produce a random number with the appropriate seeder from the array and the produced number is of an uniform distribution. The proposed algorithm here uses N threads in parallel to produce random number seeders array of length N.

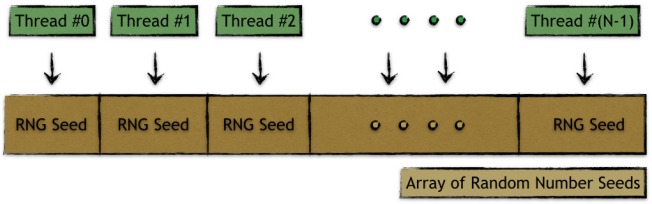

Fig. 4. Parallel Random Number Generation in RNG array

B. Distance Table Calculation

There exist two basic logical options when calculating the fitness value of a chromosome. If the fitness value is needed to be calculated a few times, then it is done each time by using the coordinates of the points over again. The point-to-point distances are calculated on GPU with respect to the euclidean method and the action is iterated among the points in the route as depicted in Figure 5. Since this is a duplicate calculated over and over again, the second option arises with some advantages compared to the first one. Distance table is used to fetch the point-to-point distances when necessary. In this case, the euclidean distance function is run $N * N$ times

ISBN: 978-988-19252-7-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

where N is the number of points. Each time the thread is required to calculate the fitness value of its chromosome, it only needs to loop over the points in the distance table and sums them to find the total cost. The proposed algorithm uses $N * N$ threads to fill in the distance table. Each thread is responsible for calculating the euclidean distance from one city to another. As an example of thread and the memory usage, assume that there are *1002* points in the task. In this case, there should be *(1002 ∗ 1002) 1.004.004* threads running for the calculation. The table *(1-D array)* includes *1.004.004* distances. Each distance is of a *float* data type. The float has a *4* bytes space in the memory. The total memory it requires is then approximately *3.82 MB*.

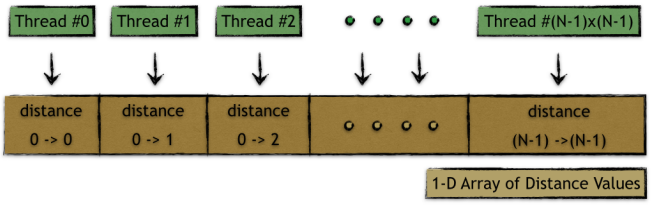

Fig. 5. Parallel Distance Table Calculation

C. Initial Population

Creating the initial population is a crucial step. Since it is the one step behind the *GA* evolution progress starts, having the initial population by a computational model *(such as neighborhood metrics)* to make a good start is a desired option in many cases. In this study, the initial population is created randomly in order to simplify the progress. Since the fine results can be achieved through the new generations using the power of many cores in the *GPUs*, the initialization step is not a big deal. There are N threads shuffling N chromosomes and putting them into the population as depicted in Figure 6.

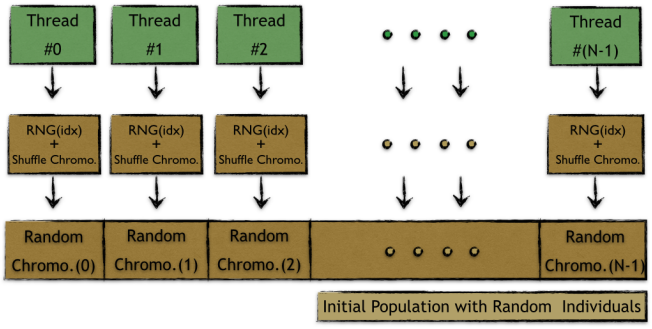

Fig. 6. Creating an initial population. Each thread shuffles a chromosome and puts it into the 1-D population array.

D. Fitness Function

Determining what kind of problem is going to be solved through the *GA* is equal to determine the fitness function. It is a crucial course in developing the *GA* to a particular problem as well as it is a guidance factor of evolving new solutions. In this study, since the basic problem is a *TSP-like* task, the fitness value is the sum of the full path distances where each point is visited exactly once and the aim is to traverse the points in the shortest way.

E. Elitism

Elitism simply refers to migrate some of the best chromosomes to the new generation. With the term "*best*", those having the lowest fitness values, thus the shortest path, are meant to be the elites. Keeping the elites of the previous generation in the current one guarantees that the solution does not go much worse through the new generations evolved. THis approach is implemented as depicted in Figure 7. In this study there are several pieces of *TSP* libraries used where the number of population differs from *1024* to *8192*. Although the population size changes over the experiments, the elitism number stayed same to provide the simplicity during the tests. The best *32* chromosomes are chosen to be the elites and they are kept and transferred to the new generations at each iteration as they are optimized and updated, too.

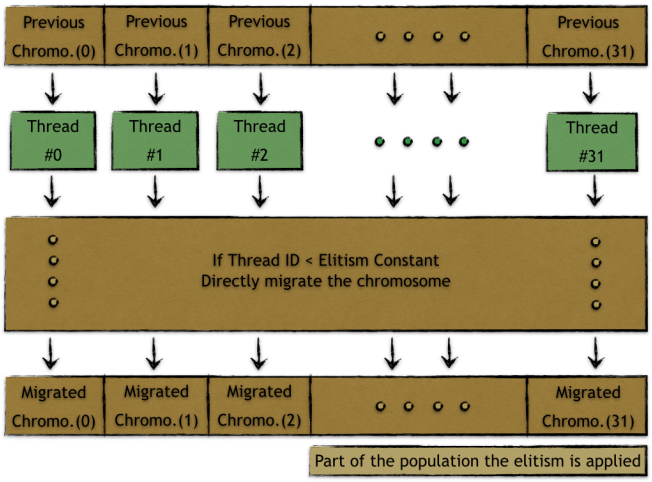

Fig. 7. Applying the elitism to the population

F. Parent Selection

The chromosomes that elitism does not apply are passed through the other conventional *GA* steps. The first step for a thread is to choose *2* parents from the previous generation to process and produce a new child chromosome of the parents. The selection method of a parent is the tournament model where the chromosome having the lowest fitness value of the two randomly chosen chromosomes is set to be one of the parents of the child, and the other parent is also selected with the same approach. Considering that all N threads are responsible for the population index i where i is the *ID* of the threads, then it is obvious that the selection is operated by N threads in parallel at each iteration.

G. Crossover

Crossover is one of the core concepts in the *GA* approach. The aim is basically producing a new chromosome while inheriting its parents' characteristics. In order to enhance the throughput, each thread produces one child and puts it into the relevant index of the population for the later construction steps. In this approach, the crossover method is *1-point* where the child has the first N elements of the first parent and the remaining parts are filled by the second parent respectively. N is a random number between *0* and the number of points in the task.

ISBN: 978-988-19252-7-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

H. Mutation

Through the generations iterated, not only the inter-crossing between the parents produces the children in the population; but, with some probability, there occurs a mutation on some of the children. As the mutation may occur at any time on any child, it is capable of both making small differences on the children as well as keeping the individuals converging to sub optimal results. In this study, the swap mutation model is used to keep the diversity between children over the generations. The swap model basically chooses two random points in the chromosome and swaps them. Swapping may yield a small changes in the route as it may be a big change, according to the chosen points to be swapped. The ratio used in this study is *0.015*, meaning that there may be *15* chromosomes get mutated over one thousand at each iteration. The ratio is kept same in the process of experiments.

I. Local Search

The simple *GA* approach with the steps defined above is a very promising technique to have the problems optimized. Because the algorithm, as a basis, keeps the best solutions and transfers them to the new generations while protecting the individuals by trying to increase the diversity. But the experiments suggest that it requires so many generations evolved to be able to reach high optimality whereas, at some points, it gets stuck with the sub optimal result because of the crossed paths that are taking place with a high probability as the problem domain gets bigger. In order to boost the performance to get the desired results in a short period of time, the local search method, namely local optimization, technique can be used. In this study, the local search tries to find the crossed paths between the points in the route. The *2-opt* local optimization is a $BigO(N^2)$ complexity search technique where it requires a high computational power when the search space get bigger. In the algorithm, each child is sent to the *2-opt* local optimization function to get rid of undesired crosses in their route. Hence, fewer iterations are needed to find the valuable results.

Usage of these GA operator (crossover, mutation and local search, as depicted in Figure 8

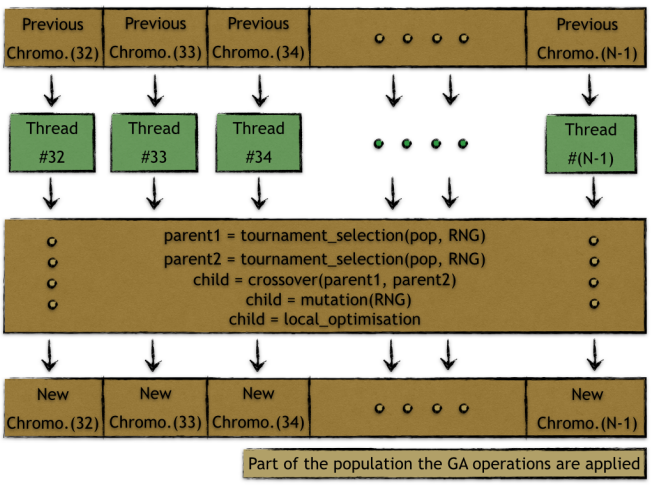

Fig. 8. Applying the GA operators to the population

J. Sorting Individuals

After completing the evolving phase of the *GA*, the newly generated population is then required to get ready for the next iteration. For this reason, the population is sorted according to the fitness values of the individuals. The sorting mechanism is needed where the elitism step takes place. Recall that the elitism refers to transferring the best individuals to the next generation to keep the solution quality high. Except the elite ones, the individuals are changed with the genetic operators. These new individuals may have better fitness values comparing the elites so each sorting phase may change the order of elites or completely discard the previous elites so that they are joined the non-elite ones to be processed via the genetic operators. In this study, the Thrust library in the *CUDA SDK* is used to keep the population in a vector and sort it when necessary. The sorting function run in parallel where the Thrust library handles the sorting algorithm and maximizes the throughput.

K. Stopping Criteria

The stopping criteria generally varies as the problem to be solved via the *GA* changes. It can be limiting the iterations, stopping the algorithm when a desired result is found or a stop time is reached. For the comparisons of both *CPU* and *GPU* versions of this study, the same criterion is set as it is to limit the iteration count. Because it is aimed to present that the *GPU* version takes much less time when the similar conditions occurs comparing to the *CPU* version. After *100* iterations, the algorithm stops.

According to these parallel operations the algorithm is executed as depicted in Figure 9

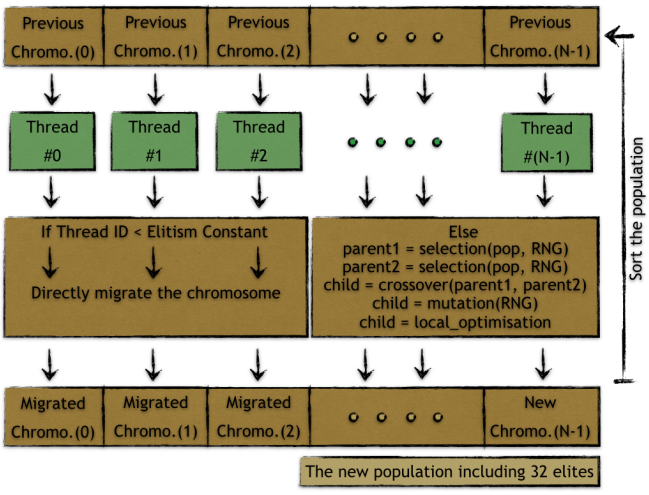

Fig. 9. The overall representation of the proposed algorithm.

V. Experiments and Analysis

To make a realistic comparison with real-world scenarios, some particular problems from *TSPLIB* [8] are tested in the proposed algorithm, which is explained throughout the paper. This parallel algorithm is implemented by using both standard C for the serial version and *CUDA C* for the parallel version of the algorithm. The parallel version is compiled via *CUDA* compiler. Either versions have some minor differences that are related to the hardware limitations. Other than that,

ISBN: 978-988-19252-7-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

there is no major differences in the approach. The test libraries include *52, 76, 100, 225, 439, 575, 1002* and *2392* waypoints. Each problem is solved with 1024, 2048, 4096 and 8192 populations. The *GPU* in this study is the *NVIDIA GeForce GTX 680* which has *1536* graphics cores and each of them has a *1002* MHz frequency. The serial version is run on an Intel *i5 3.10 GHz CPU*. The operating system for this experiment is Ubuntu *13.10*. The compiler for the parallel version is *CUDA SDK 5.0* and the language is *CUDA C*. Table I shows the details of the underlying system.

TABLE I
HARDWARE FEATURES FOR THE CPU AND THE GPU USED IN THIS EXPERIMENT

	CPU	GPU
Manufacturer	Intel	NVIDIA
Model	i5	Geforce GTX 680
Architecture	Sandy Bridge	Kepler
Clock Frequency	3100 MHz	1002 MHz
Cores	4	1536
DRAM Memory	4 GB DDR3	4 GB DDR5

The solution quality as well as the execution time of the *GA* mainly depends on the parameter decisions. The parameters of the algorithm chosen in this study are as shown in Table II.

TABLE II
EXPERIMENTAL PARAMETERS OF GENETIC ALGORITHM

Parameters	Values
# of visiting points	52 / 76 / 100 / 255 / 439 / 575 / 1002 / 2392
Population Size	1024/ 2048 /4096 / 8192
Elitism	First 32 chromosomes in each generation
Parent Selection	Tournament (select 2 and get the best)
Crossover type	1-point
Mutation type	Swap
Mutation rate	0.015
Local Search Type	2-opt
Local Search Rate	1
# of Iteration (as ending criteria)	100

The main performance metric of the proposed algorithm is the speed up of the solution which is calculated according to Equation 1.

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

The proposed parallel algorithms is executed on different number of waypoints. The speed up performance is more expressive when the number of chromosome in the population is increased. Therefore, speed up of two different scenarios are depicted in Figure 10 according to the population with 8192 chromosomes. A Detailed comparison according to the population size is listed in Table III and Table IV.

Although, large number of populations result greater error with respect to less population size, Due to the realistic comparison with other scenarios we iterate the algorithms for 100 times. In case of increasing the number of iterations,

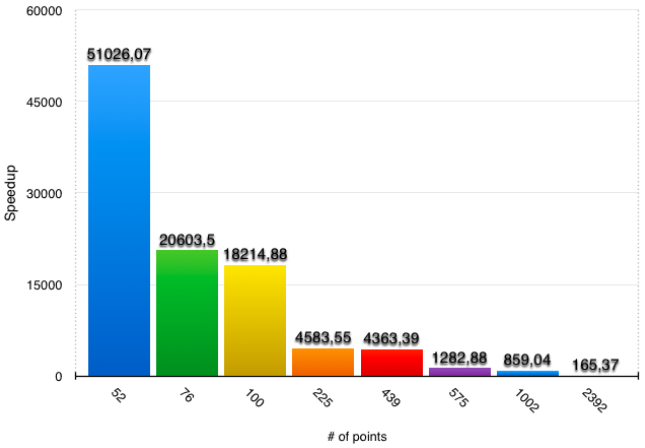

Fig. 10. Total speedups of the used libraries. The population size is of 8192.

the proposed algorithm produces better solutions as depicted in Figure 11.

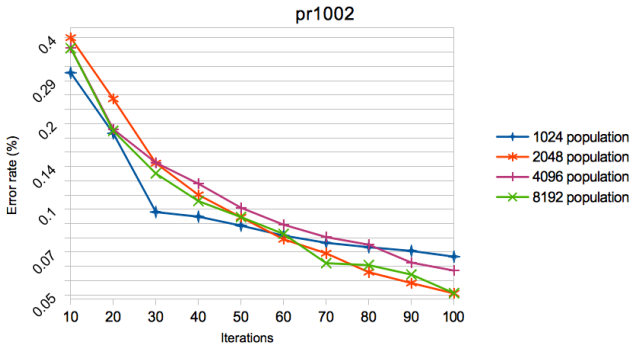

Fig. 11. The GPU error rates through the iterations of all populations examined for 1002 points.

To depict a clear distinction between the *CPU* and *GPU* implementation of the proposed algorithm is presented for comparing both algorithm with *100* waypoints in Figure 12.

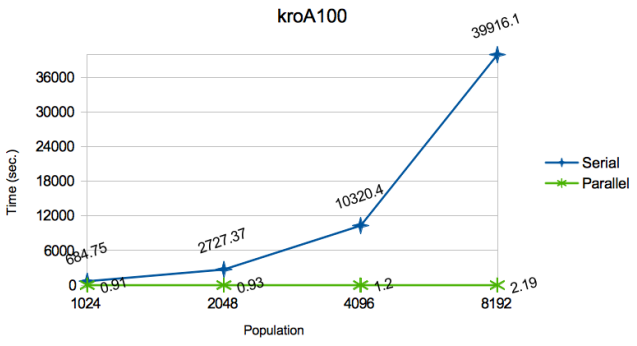

Fig. 12. CPU and GPU time comparison of a problem including 100 points.

VI. CONCLUSION

UAVs have many advantages especially about not risking a human pilot. Therefore, there is an increasing need about *UAVs* to become more independent and to have more intelligent decision-making capability especially typical surveillance operations. The path planning is a prominent element

ISBN: 978-988-19252-7-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE III
OVERALL RESULTS AFTER 100 ITERATIONS.

#	pop.	CPU error	CPU time (sec.)	GPU error	GPU time (sec.)	Speed up rate
52	1024	0.0361	335.04	0.0003	0.2217	1510.57
52	2048	0.0237	1388.26	0.0003	0.2315	5996.25
52	4096	0.0003	5511.51	0.0003	0.2704	20381.67
52	8192	0.0003	21823.8	0.0003	0.4276	51026.07
76	1024	0.0620	524.07	0.0253	0.7562	692.96
76	2048	0.0637	2127.63	0.0199	0.8157	2608.15
76	4096	0.0562	8246.41	0.0118	1.0902	7563.64
76	8192	0.0527	31869.5	0.0118	1.5468	20603.5
100	1024	0.0123	684.75	0.0001	0.9078	754.27
100	2048	0.0049	2727.37	0.0001	0.9331	2922.71
100	4096	0.0025	10320.4	0.0001	1.2029	8579.39
100	8192	0.0025	39916.1	0.0001	2.1914	18214.88
225	1024	0.1221	1890.76	0.0146	6.9822	270.79
225	2048	0.1128	6758.55	0.0123	5.8850	1148.42
225	4096	0.1024	23796.6	0.0102	7.0600	3370.59
225	8192	0.1069	89000.3	0.0212	19.4173	4583.55
439	1024	0.1111	4895.49	0.0137	17.3561	282.06
439	2048	0.0953	15314.3	0.0150	16.8677	907.9
439	4096	0.0882	50195.7	0.0035	24.1392	2079.42
439	8192	0.0722	178671	0.0024	40.9477	4363.39
575	1024	0.0645	7430.67	0.0669	61.1384	121.53
575	2048	0.0384	22016.2	0.0843	74.5503	295.32
575	4096	0.0595	69633	0.0836	79.4542	876.39
575	8192	0.0510	241785	0.1093	188.47	1282.88

TABLE IV
OVERALL RESULTS AFTER 100 ITERATIONS.

#	pop.	CPU error	CPU time (min.)	GPU error	GPU time (min.)	Speed up rate
1002	1024	0.0974	308.3	0.0692	2.2	139.85
1002	2048	0.0773	830.9	0.0515	3.1	265.98
1002	4096	0.0692	2397.9	0.0619	5.0	476.73
1002	8192	0.0692	7743.1	0.0516	9.0	859.04
2392	1024	0.4546	1519.4	0.5046	26.4	57.52
2392	2048	0.3217	3516.4	0.3946	36.5	96.19
2392	4096	0.2827	8793.4	0.3127	68.9	127.49
2392	8192	0.2295	22771.7	0.2466	137.7	165.37

in *UAVs*' autonomous control module. It allows the *UAV* to autonomously compute the best *(or near to the best)* trajectory over waypoints in the mission area. However, if the number of waypoints increases then there is a need to speed up the computing process by using parallel processing. There in this paper it is aimed to calculate the path of a *UAV* by using highly parallel genetic algorithm on *CUDA* platform with *GPUs*. The experimental results showed that the proposed approach is very efficient and can also be used in real time path calculation even in large areas with lots of waypoints and constraints. As a future work, it is planned to improve the performance of the parallel genetic algorithm, and modify the approach to enable the use of multi *UAVs*, and decrease the total mission time in an efficient way.

ACKNOWLEDGEMENT

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

REFERENCES

[1] I.K. Nikolos, K.P. Valavanis, N.C. Tsourveloudis, and A.N. Kostaras. Evolutionary algorithm based offline/online path planner for uav navigation. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 33(6):898–912, Dec 2003.

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

[3] DS Knysh and VM Kureichik. Parallel genetic algorithms: a survey and problem state of the art. *Journal of Computer and Systems Sciences International*, 49(4):579–589, 2010.

[4] Noriyuki Fujimoto and Shigeyoshi Tsutsui. A highly-parallel tsp solver for a gpu computing platform. In *Numerical Methods and Applications*, pages 264–271. Springer, 2011.

[5] Ozgur Koray Sahingoz. Large scale wireless sensor networks with multi-level dynamic key management scheme. *Journal of Systems Architecture*, 59(9):801 – 807, 2013.

[6] Ozgur Koray Sahingoz. Multi-level dynamic key management for scalable wireless sensor networks with uav. In Youn-Hee Han, Doo-Soon Park, Weijia Jia, and Sang-Soo Yeo, editors, *Ubiquitous Information Technologies and Applications*, volume 214 of *Lecture Notes in Electrical Engineering*, pages 11–19. Springer Netherlands, 2013.

[7] Ugur Cekmez, Mustafa Ozsiginan, and Ozgur Koray Sahingoz. Adapting the ga approach to solve traveling salesman problems on cuda architecture. In *Computational Intelligence and Informatics (CINTI), 2013 IEEE 14th International Symposium on*, pages 423–428. IEEE, 2013.

[8] G. Reinelt. TSPLIB - A t.s.p. library. Technical Report 250, Universität Augsburg, Institut für Mathematik, Augsburg, 1990.

ISBN: 978-988-19252-7-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)