2014 International Conference on Unmanned Aircraft Systems (ICUAS)
May 27-30, 2014. Orlando, FL, USA

A UAV PATH PLANNING WITH PARALLEL ACO ALGORITHM ON CUDA PLATFORM

Ugur Cekmez[1], Mustafa Ozsiginan[2] and Ozgur Koray Sahingoz[3]

***Abstract*— Solving the path planning problem of a *UAV* is a challenging issue especially if there are too many checkpoints to visit. Mainly, the brute force approach is used to find the shortest path in the mission area, which requires too many times to find a solution. Therefore, evolutionary algorithms and swarm intelligence techniques are used to find a feasible solution in an acceptable time. In this study, path planning problem of a *UAV* is solved by using a highly parallelized Ant Colony Optimization *(ACO)* algorithm on *CUDA* platform. The *UAV* path is constructed for disseminating keys and collecting data from a Wireless Sensor Network, which is previously defined.**

Due to its simplicity and effectiveness, *ACO* is selected as a path planning algorithm. However, *ACO* is not satisfactory if the mission area becomes large and there are an excessive number of checkpoints and/or additional constraints. In order to increase the performance, some parallelization techniques must be used in high performance computing platforms. *GPU* architecture has emerged as a powerful and low cost architecture for enabling impressive speedups for scientific calculations. Therefore, the parallel structure is constructed on *CUDA* architecture. The experimental results are compared with the *CPU* performance of the serial algorithm, and they clearly show that the proposed approach have a great potential for acceleration of *ACO* and allow to solve many complex tasks such as *UAV* path planning problem. We also present the execution results with different parameter values to expose the results for the researchers.

I. INTRODUCTION

Unmanned Aerial Vehicles *(UAVs)* have been emerged as one of the most exciting technologies in aeronautics, and this concern increases day by day both in military and civilian areas. A *UAV* is a remotely piloted or self-piloted aerial vehicle, which is able to carry some payloads such as a computing unit, sensors, communication equipments, different types of sensors, etc. *UAVs* have lots of advantages in contrast to standardly piloted aircraft where the *UAV* yields decreased production cost, is small in size and weight, has increased mobility, requires no human casualties, etc. Therefore, they are increasingly used in many application areas.

In this progressive period, *UAVs* have been evolving from being remotely piloted simple vehicles to autonomous robots. The autonomy is defined as the ability of an unmanned system to execute its mission by following a set of preloaded instructions without any human intervention. A *UAV's* autonomy can be constructed in three dimensions: *taking off, flight and landing*. While using an autonomous *UAV*, one of the important research areas is accepted as finding a feasable and effective path, which can be offline, by calculated before the flight, or online, by calculating path on the flight, according to mission type. After constructing this path, *UAVs* are then able to follow the path autonomously.

A *UAV* is required to fly in a complex environments, which may contain some obstacles *(such as mountains, buildings)*, threats *(such as radars, anti-aircraft guns)* and other *UAVs*. At the same time, *UAVs* have some kinematic constraints, and while planning the flight path, they must be taken into consideration.

In the literature, there are many path-planning algorithms. As a simple solution, *Dijkstra*'s algorithm calculates the shortest path in the solution domain by evaluating all possible paths from a single starting point. However, the algorithm does not give the optimum solution of all the visiting points in a single path.

In order to solve the path planning problem, *(by visiting each waypoint exactly once)*, brute force approaches must be used for finding optimal solution. However, this is not feasible if the number of waypoints gets large. Therefore, swarm intelligence and/or evolutionary algorithms are preferred in many researches to find an acceptable solution in a feasible time. Due to its robust structure *(with compared to other existing directed search methods)*, Ant Colony Optimization *(ACO)* algorithm, as a swarm intelligence technique, is preferred for calculating the path of UAVs.

ACO is a well-known swarm intelligence method, which has been successfully applied to several combinatorial optimization problems by imitating social behavior of ant colonies. The idea of *ACO* is based on the behavior of real ants exploring a path between their colony and a source of food. Although, this optimization can be used in solving many real world problems; when facing large and complex problem instances, distributed and/or parallel execution techniques are usually applied to increase the efficiency. These improvements are used to reduce computation time and to improve the solution quality in fewer iterations of *ACO* algorithm, which includes two main operations: *1)* tour construction and *2)* pheromone update. Due to its inherent properties, *ACO* is well-suited for parallel implementation; however, it also poses significant challenges because of its restricted memory access patterns in many core computing platforms.

Although *ACO* brings an acceptable solution in a feasible time, if the number of waypoints increases, solving this problem becomes a challenging issue. Thus, some performance

[1]Computer Engineering Department, Yildiz Technical University, Istanbul, Turkey ucekmez@yildiz.edu.tr
[2]Aeronautics and Space Technologies Institute, Turkish Air Force Academy, Istanbul, Turkey mustafaozsiginan@gmail.com
[3]Computer Engineering Department, Turkish Air Force Academy, Istanbul, Turkey sahingoz@hho.edu.tr

increasing techniques must be used such as distributed and/or parallel execution of codes.

While solving the problem using a parallel approach, three main wayouts are taken into account where multi-processor technology, multi-core technology *(in CPU)* and Graphics Processing Units *(GPUs)* are involved. In recent years, *GPUs* have emerged as an exciting hardware which offers intensive computational power. It contains highly parallel and fully programmable architecture and is suitable for executing different algorithms of real world problems. In contrast to standard multi-core *CPUs* which is able to host up to ten independent cores, the *GPUs* can reach their high performance by using their thousands of cores in thousands of *GFLOPS (Giga Floating-point Operations Per Second)*.

In this study, it is aimed to propose an efficient parallelization strategy for the implementation of *ACO* on *GPUs* to solve the described path planning problem of *UAVs* in a large mission area with lots of waypoints in an acceptable time.

The rest of the paper is organized as follows: Problem definition is detailed in the next section. Related Works on the topics and the background information to dive deep into the terms of *ACO*, *GPU* and *CUDA*, are depicted in Section *3* and Section *4* respectively. In Section *5* implementation details of the proposed technique is given. Experimental Results are given and compared with serial implementation on CPU in Section *6*. Finally, conclusions and future works are summarized in the last section.

II. PROBLEM DEFINITION

In this study, it is aimed to plan the path of a *UAV* by using parallel ant colony optimization with the help of *CUDA* platform. In this research, the problem domain is used as shown in Fig. 1, which is described and detailed in [1]. There are lots of sensor nodes *(SNs)* in the mission area, and these *SNs* are organized in a multi-level hierarchy to collect and distribute necessary data. Each node is directly or indirectly connected to the cluster head *(CH)* node and it forwards the collected data to this *CH*. *SNs* are typically deployed in a hostile environment, and they have to use insecure wireless channels. As a result, they are vulnerable to security attacks. In order to enable a secure communication platform, a multi-level dynamic key management mechanism is conducted by using a both pairwise keys and public/private keys according to importance of carried data, and a *UAV* is used as a Mobile Certification Authority [1].

In order to disseminate keys, there is a need for a *UAV*. This *UAV* must visit these *SNs* with certain time intervals, and these nodes are named as the waypoints of the *UAV*, and the path must be planned to go over these nodes. Due to the environmental conditions or the need for deploying new *SNs* to the mission area, the locations of the *SNs* can be changed and a new path must be calculated according to this new circumstance.

In this research, the main aim is to monitor this mission area by using a single *UAV* with minimum cost *(by decreasing the length of the path, fuel consumption and mission time)*. There is only one landing place, therefore, the starting point and the ending point of the path are accepted as same depot.

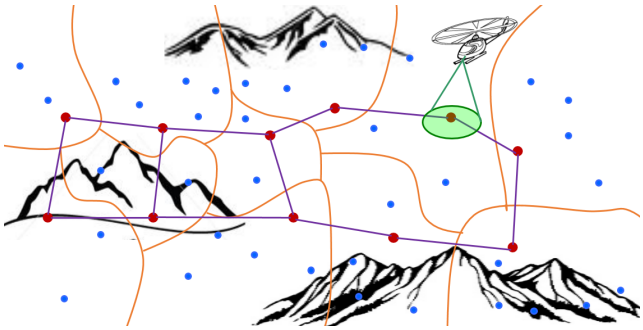

Fig. 1. Multi level data dissemination of the WSN Problem Domain

III. RELATED WORKS

Algorithm of *UAV Path planning* problem is very similar to solving the Traveling Salesman Problem *(TSP)* that is a well-known in combinatorial optimization. Therefore, the focusing area in the literature scan includes this issue and few *UAV* path planning approaches on *CUDA* platform.

Sanci and Isler proposed an approach for finding an acceptable solution to the flight planning problem of a single *UAV* by means of parallel genetic algorithm on *GPUs* [5]. They also compared their approach with alternatives and it showed a remarkable speed up. Authors firstly converted the problem to a standard *TSP* problem and then solve this both with standard *C* programming language on *CPUs* and by using *NVIDIAs CUDA* compiler for aiming to exploit *GPU* processing power as much as possible. However, these codes cannot be same; they were implemented as logically equivalent as possible. They used a modified Genetic Algorithm to meet the needs of the surveillance domain.

Studies related to the parallel ants approach aims to execute the ants' pheromone update and their tour construction phase on many processing elements such as multi core *CPU-based* structures or many core processing units such as *GPUs*. Kugu and Sahingoz implemented a parallel implementation of both *ACO* and Rank based Ant System *(RBAS)* on a multi core *CPU* [6]. In their implementation, they created each ant as a thread and these threads are executed on cores if they are available. The total thread number and so the core number depend on the *CPU* in which the algorithm is run and increasing the number of cores shows that the algorithm execution time decreases.

In [7], the authors proposed a parallel *ACO* to solve the Quadratic Assignment Problems on *CUDA* platform. They also embedded *2-opt* local search to their algorithm for the purpose of optimizing their candidate solutions. They also tried to run this optimization in a parallel approach to gain a speed up the overall timing of finding an acceptable solution. Their algorithm speeds the process up to *25* times faster on average performance comparing to those corresponding serial *CPU* version.

On the other hand, to solve the *TSP-like* problems, [8] presented a new data-parallel *GPU* implementation of the Ant System algorithm which parallelizes both the tour construction and the pheromone update stages on *GPU*. Because the structure of the Ant System requires more space on the RAM while the number of control points is increased, after exceeding its limits, the performance of the algorithm decreases. In order to increase the algorithm performance Double-Spin Roulette *(DS-Roulette)*, a highly parallel roulette selection algorithm, is used. *DS-Roulette* reduces the usage of shared memory, the overall instruction count and the processes executed by *GPU*. As a result, this selection speeds up the algorithm performance.

Cecilia et al. [9] also implemented a parallel *ACO* approach for solving *TSP* problem on *CUDA* platform. They used Independent-Roulette *(I-Roulette)* selection mechanism to increase the parallelization of the algorithm.

IV. BACKGROUND

A. Ant Colony Optimization for Route Planning

In this section the Traveling Salesman Problem *(TSP)* for the route planning of *UAVs*, the Ant Colony Optimization *(ACO)* and the way the *TSPs* are solved using *ACO* are briefly described. There are many works related to *TSP* with different variation of *ACO* from the original works of Dorigo et al [2].

ACO is a population based evolutional algorithm that is inspired by the behavior of natural ant colony. It was first applied to *TSP* by Dorigo et al. The reason to choose *TSP* as a base problem is that the *TSP* is an *NP-hard* optimization problem applicable to be solved easily with *ACO* and it is a comprehensible and well-known problem for benchmarking among the evolutionary algorithms [3].

In order to define the *TSP*, consider a *UAV* that is flying among some pre-determined cities and is required to return to home after passing all the given cities exactly once. The aim of *TSP* is to find the shortest tour from given cities, thus producing the minimum length of the Hamiltonian circuit.

In order to solve the route planning problem through the *ACO*, a number of ants working as distributed agents [5] are created and are required to search for the shortest path on the given coordinates of cities. Route planning through *ACO* has some specific constraints where each ant has to visit all the cities and each city has to be visited exactly once [3].

Each ant starts the tour from a randomly selected city and performs distributed moves by applying the random proportional rule on the graph until their tours are completed. In the path planning, the next city is selected by the formula given in the Equation 1 where *i* is the current city the ant *k* is located and *j* is the candidate city the ant may visit next. N_i^k represents the unvisited cities. τ_{ij} is the amount of pheromone the two cities deposited and η_{il} equals $\frac{1}{d_{il}}$ where d_{il} is the distance from city *i* to *l*. α and β are the custom parameters to influence the effects of pheromones and the distances while selecting the next city.

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{il}]^\alpha \cdot [\eta_{il}]^\beta} \tag{1}$$

Thereafter, a tour is constructed by an ant, the pheromone table is updated accordingly so that next ants can perform better selections. Note that the pheromones on the roads are evaporated before the ants start their new tours. ρ is the evaporation rate. This is a technique for prevention of stagnation of the populations Equation 2.

$$\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} \tag{2}$$

The pheromone update stage is demonstrated as in Equation 3 where each edge from city *i* to *j* deposits the total pheromones over the time while ants completing their path constructions.

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k \tag{3}$$

Each ant *k* updates the pheromone table as seen in Equation 4 where C^k is the total distance of the tour T of ant *k*.

$$\Delta\tau_{ij}^k = \begin{cases} \frac{1}{C^k} & edge(i,j) \in T^k \\ 0 & otherwise \end{cases} \tag{4}$$

The overview of the traditional *ACO* algorithm can be seen in Listing. 1. As the pseudo code briefly explains the main stages of the model, the algorithm starts with setting the parameters and initializing some variables *(such as pheromones and distances, etc.)*. The next part is aimed to construct a new tour and to update the pheromone values in a loop. In this stage, each ant works independently.

Listing 1. Overview of the ACO Algorithm

```
Procedure ACO algorithm for TSP
 Set parameters
 initialize pheromone trails
 while (termination not met) do
  ConstructPath
  UpdateTrails
 end
end
```

The tour construction of an ant is shown in Listing 2. The ant is located in a randomly chosen city and it starts the tour by calculating the probabilities for the unvisited cities. Each unvisited city is a candidate to visit. Recall that this probability is shown in Equation 1. After completing the tour, the ant then calculates the fitness *(overall distance)* of the tour then updates the trails.

Listing 2. Tour Construction of an Ant

```
Procedure ConstructPath
 Path[0] <- random city
 for j = 1 to n-1 do
  for k = 1 to n do
   prob[k] <- CalculateProb(Path,k)
```

```
    end
    Path[j] <- SelectCity(prob)
  end
  fitness <- CalculateFitness(Path)
end
```

B. Using Graphics Processing Units

Multicore processors are the popular architectures in recent years since they have higher computational capability and lower energy consumption comparing to those single core processors. Central Processing Units *(CPUs)* have fewer cores that are best optimized for sequential processing where the modern Graphics Processing Units *(GPUs)* are designed to host thousands of low frequency and efficient cores for handling multiple data-parallel tasks by running thousands of lightweight threads concurrently. Having more than hundreds of *GPU* cores and handling millions of lightweight threads significantly accelerate the data-parallel approaches in contrast to the traditional single core general purpose *CPUs*. In this case, if the problem is applicable to solve with the *GPUs*, then taking advantage of this power provides a cost-efficient solutions. The basic structure of of a *GPU* is shown in Fig. 2.

In this study, the proposed model is implemented on *NVIDIA's GeForce GTX 680* graphics card that has *Kepler GK104* architecture, having *3.54 billion* transistors, *1536 CUDA Cores* and *3090 GFLOPs*. Its Graphics Clock speed is *1006 MHz*, Memory Clock is *6008 MHz*, Memory Bandwidth is *192.26 GB/sec* and *TDP* is *195W*. It has *4* Graphical Processing Clusters *(GPC)*, *8* next-generation Streaming Multiprocessor *(SMX)* and *4* memory controllers.

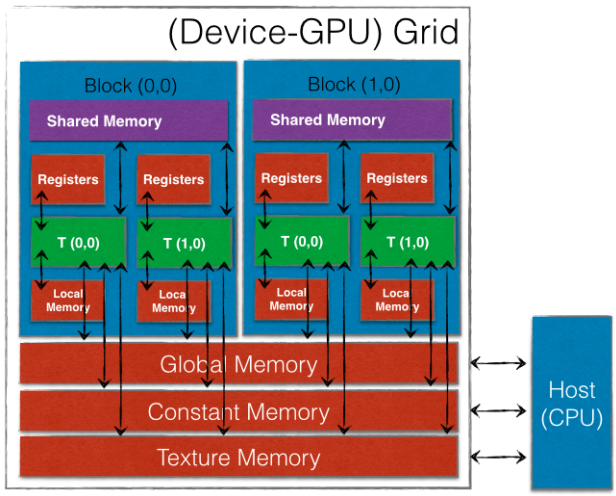

Fig. 2. GPU Thread-Memory Organization

C. CUDA Architecture

NVIDIA introduced *CUDA* in *2006* to allow parallel computations on *GPUs*, rather than only graphics-purpose usage. *CUDA* is a parallel computing platform for the software developers and researchers who intent to take the advantage of the *GPUs* to yield high computation power with a relatively low cost.

In *CUDA* architecture, every thread is organized as a group in thread blocks and every thread block is organized as a group in a grid. Each thread is responsible for the copy of their kernels *(functions)* and their data part. Every thread is created and executed at runtime where the threads run concurrently as warps *(group of 32 threads)*. Threads in the same thread block are able to communicate with other threads through their shared memory and synchronization among the threads is as easy as one line simple command. Every thread has their own registers and local memory to keep the most accessed data at hand. The principles of hardware and the software model as well as how the threads are organized are shown in Fig. 3.

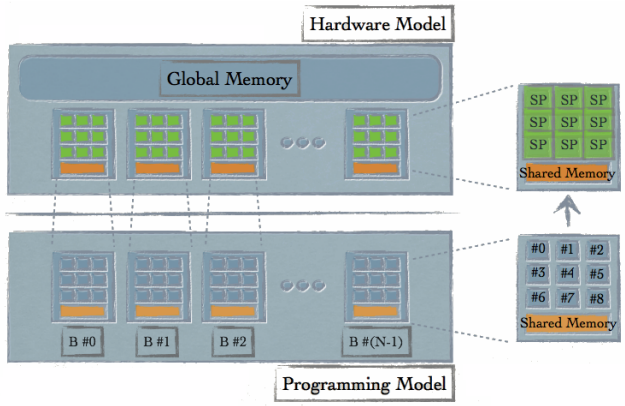

Fig. 3. Hardware and the Programming model of CUDA. The right upper box is the Streaming Multi Processor and the lower one is a grid with a block of threads. *SP: Streaming Processor, B: Block and numbers represent the thread IDs.*

V. PARALLEL IMPLEMENTATION OF ACO FOR THE ROUTE PLANNING

In this section, the proposed algorithm for calculating a UAV path (as shown in Fig. 4) is described and its steps are briefly explained. The algorithm has four main steps where the first three are calculated once and the last step is repeated until a termination criteria has met.

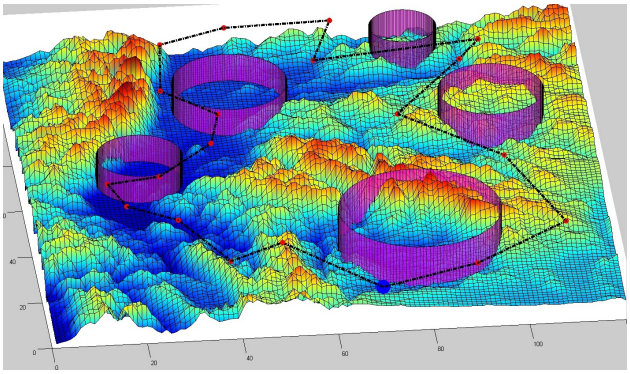

Fig. 4. A sample UAV path in a constrained area

In the proposed approach, each ant is represented as a thread block and there are *N* threads running in the corresponding block where *N* is equal to the number of cities each ant is required to visit [4]. For instance, considering a problem including *52* points and *52* ants, there are *52* thread blocks and *52* threads in each. Thus, *2704* threads are involved in solving this problem. If the problem includes *1002* points and the ant number increases to *4096*, then the total number of threads required to solve this problem becomes *4096 x 1002 = 4.104.192.*

A. Random Number Generation

In the tour construction step, each ant selects the next city using a probabilistic model shown in Equation 1. Since there are a number of cities to visit, the ant decides it partly random. It produces a random number by using the *RNG* seeder taken from its current city index [Fig. 5]. *N* piece of seeders are put into a *1-D* array by *N* threads. For the random number generation, the *cuRAND* library of *CUDA* is used rather than standard *C++ rand()* function. *cuRAND* uses the dynamic utilization for using the *GPU's* cores to yield maximum throughput.

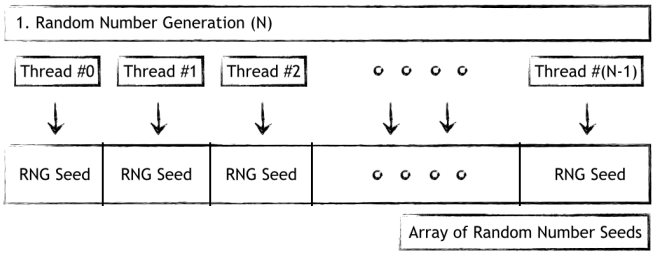

Fig. 5. N threads fill in the array with Random Number Generation (RNG) Seeder. *N:Number of cities.*

B. Distance Table Calculation

The next step is to produce the distance table for the cities. Distance table is calculated by N^2 threads where *N* is equal to the number of cities. Each thread #(i * n + j) is responsible for calculating the distance between the city *i* and *j*, then filling the corresponding position of the *1-D* array, as shown in Fig. 6. For instance, if the problem is of *1002* cities, then the size of distance table is *1.004.004* and it is filled by the same number of threads.

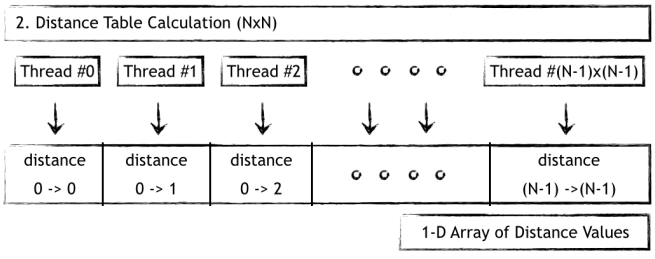

Fig. 6. N^2 threads calculate the distances from city *i* to *j*. *N:Number of cities.*

C. Initialization

Before the tour construction processes of the ants, the initial pheromones must be assigned the vertices among the cities so that an ant can decide which city to visit next. This process is similar to calculating the distance values in case of work flow of the threads. N^2 threads are involved in initializing the pheromones, as shown in Fig. 7. The initial value of the pheromones are selected to be 10^{-6}.

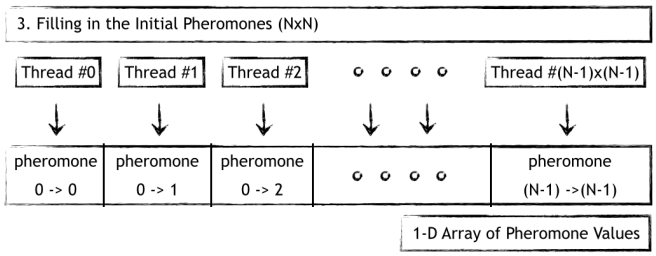

Fig. 7. Initial pheromones are inserted into the pheromone table by N^2 threads. *N:Number of cities.*

D. Tour Construction

The last step for an ant is to construct its tour. Recall that an ant is represented by a thread block and each thread block includes number of threads equal to the number of cities. In such a case, assume that each ant has *N* feet where each feet *(thread)* makes some computations in parallel and after all, one foot takes the results, compares them all and selects the best choice to visit. This process is continued until there is no choice left. As shown in Fig. 8, each foot calculates the probability of visiting from the ant's current city to city *j*, where *j* is all the cities that are not yet visited. In the corresponding figure, the Block #1 is represented. As it is seen that the Thread #1 and its calculation are crossed, which means that the probability of visiting this point is *0* since it is already visited at hand.

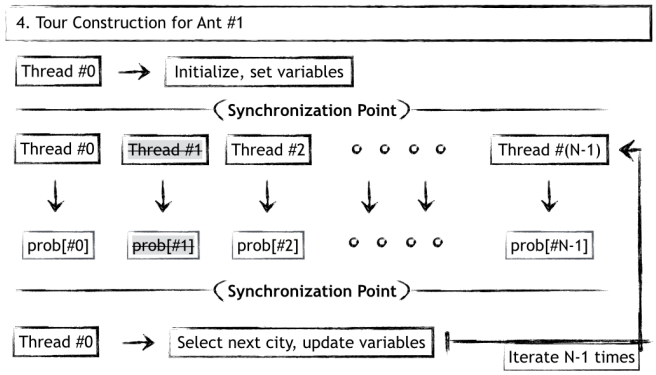

Fig. 8. Tour construction for the Ant #1. Probability calculations are processed in parallel by *N* threads. The thread #1 is crossed since it represents the block #1 itself *N:Number of cities.*

While an ant calculates its tour with its feet, the other ants are also working on constructing their tours as well [Fig. 9]. After constructing all tours, each ant calculates its tour's fitness values and updates the shared pheromone table

so that the next generation of ants take shorter paths into account when new tours are searched for.

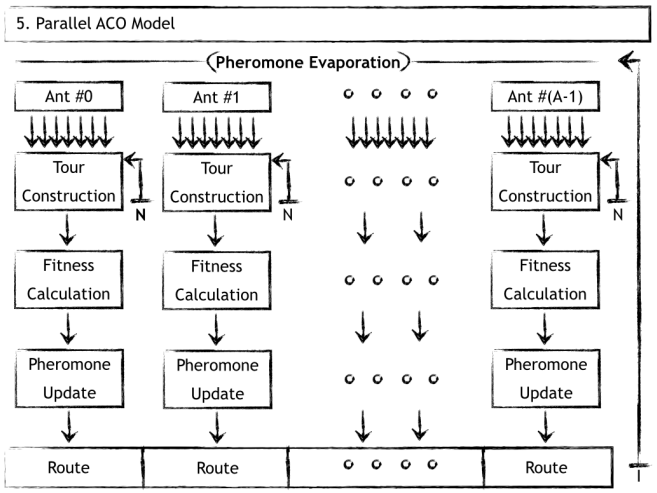

Fig. 9. Overall process for the parallel algorithm. Each ant constructs a tour. *N:Number of cities, I: Iteration.*

As can be seen from Fig. 9 there is not any synchronization line, Therefore, there is no waiting time for an ant to calculate the fitness of its tour and update the trails. Since all ants are distributed and work asynchronously, reaching to one of the bottlenecks of the algorithm, where the shared trails are updated, is a bit different in contrast to a standard shared variable rules. There is a chance for ants so that they can update the corresponding indexes without going into race conditions; but there is no guarantee that it happens all the time. If all ants finish their fitness calculation and start to update the pheromone table at the same time, then the access for the same pointers must be serialized, which is an undesirable condition. That is why, this action is thought to be asynchronous to make it possible for some ants to reach the shared variable and to update it while there is no other ant waiting to update it, since they are either already done or they are still in some actions and have not yet made it through.

Although these calculations produce a path for a single *UAV*, for converting this path to a flyable path some smoothing operations can be done by as detailed in [10]. This operation is left as the future work of the study.

VI. EXPERIMENTS AND ANALYSIS

Since the simple *UAV* route planning problem is similar to the *TSP* when considering the limitations, and the optimal solutions of the *TSP* is already at hand, in order to make the tests more realistic and accurate, the known *TSP* libraries are used. There are *5 TSP* libraries in this study where *52, 76, 100, 225, 439 and 1002* points exist. The *GPU* is *NVIDIA Geforce GTX 680*, having *1536* graphics core and each has *1002 MHz*. The serial tests are made on an Intel i5 *3.10 GHz CPU*. The operating system these experiments are run is Ubuntu *13.10*, and the compiler is *CUDA SDK 5.0* where the programming language is *CUDA C*. Table III shows the brief specifications for the serial and parallel hardware.

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

Each problem is tested with the number of ants where the number of ants is equal to *1) the number of cities in the problem, 2) 1024, 3) 2048 and 4096*. For the CPU version, the first *2* property is used. The computation time of the rest is assumed since the CPU shows a linear increase in time when the ant number increases by *2*.

All the parameters used for the experiments are shown in Table II.

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

Parameters	Values
# of visiting points	52, 76, 100, 225, 439, 1002
Ants	(52, 76, 100, 225, 439, 1002), 1024, 2048, 4096
α	1.0
β	2.0
ρ *(evaporation rate)*	0.5
Initial Pheromone	0.000001
Iteration	10,20,...,90,100

The Fig. 10 shows the total *GPU* calculation times of creating and filling in the distance table, setting the initial pheromones and the evaporating the existing pheromones.

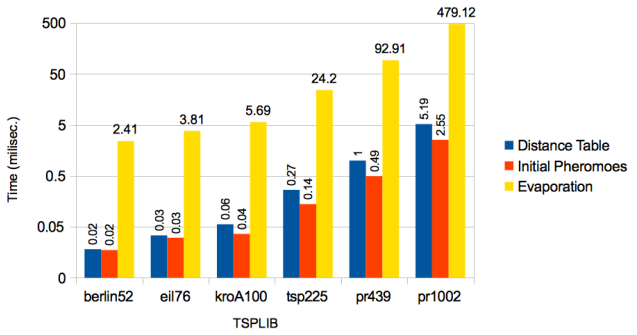

Fig. 10. Distance table creation, initial pheromone set and pheromone evaporation comparisons of all libraries examined in this study.

The Fig. 11 shows the overall calculation times of the parallel *ACO* through the iterations when the ant number is *4096*. As its shown in the graph, the calculation times increase linearly. Do not miss that the graph is shown in a logaritmic scale to be able to see the view wider.

The Fig. 12 shows the comparison of *pr1002* with both serial *CPU* and parallel *GPU* versions of the proposed *ACO* algorithm. It is clearly seen in the graph that the results are always better in the parallel version and increasing the ant

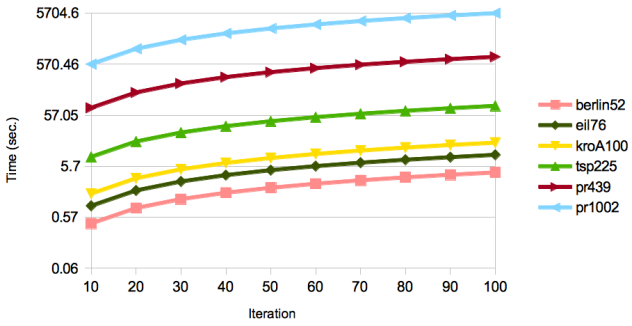

Fig. 11. Calculation times of all libraries examined in this study.

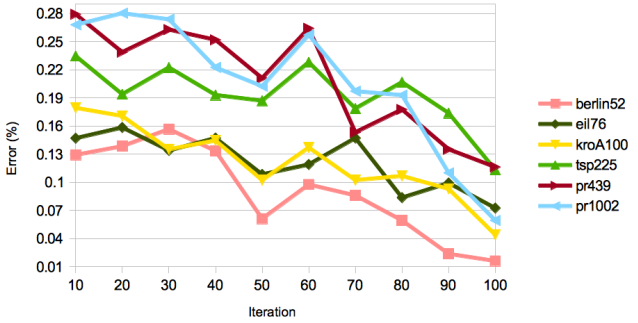

Fig. 14. Error rates through the iterations of all libraries examined in this study, for 4096 ants.

number that are visiting the points provides better acceleration in time comparing to the equivalent *CPU* version

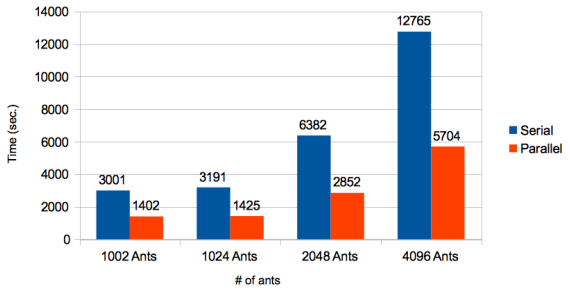

Fig. 12. Comparison of the execution times of serial and parallel versions for the *1002* points.

In the Fig. 13, it is seen that the same ant number provides different error rates comparing to the city number. In this experiment, *1024* ants are used for all of the libraries and the error rates through iterations are shown. In the next experiment, the ant number is increased up to *4096* and the results are shown in Fig. 14.

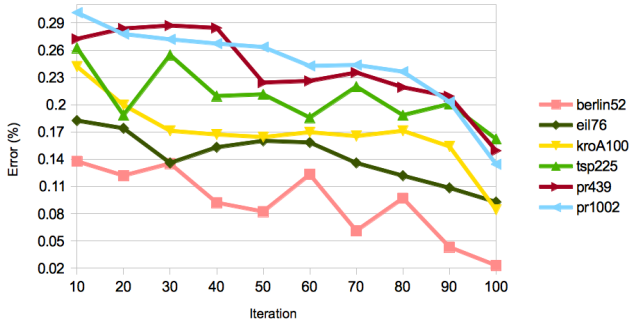

Fig. 13. Error rates through the iterations of all libraries examined in this study, for 1024 ants.

Comparing these two experiments; increasing the ant number up to *4* times will not make a significant difference in case of improving the results but clearly shows that there is an achievement.

In the Table III, the overall results and comparisons of both serial *CPU* and parallel *GPU* versions of all libraries are shown. The table includes total computation times after *100* iterations.

TABLE III

OVERALL RESULTS AND COMPARISONS OF BOTH CPU AND GPU VERSIONS OF ALL LIBRARIES EXAMINED IN THIS STUDY. **: estimated approximate time values for CPU under normal conditions.*

# of points	# of ants	CPU error (%)	CPU time (sec)	GPU error (%)	GPU time (sec)	Speedup rate (%)
52	52	0.1407	0.41	0.0564	0.09	340.37
52	1024	0.0670	15.75	0.0231	1.09	1341.39
52	2048	x	31.5*	0.0215	1.94	1523.27
52	4096	x	63*	0.0161	4.3	1364.73
76	76	0.1809	1.24	0.0952	0.21	465.31
76	1024	0.1527	9.93	0.0928	2.42	308.88
76	2048	x	19.87*	0.0914	4.5	340.7
76	4096	x	39.74*	0.0725	9.54	316.28
100	100	0.2055	2.98	0.1114	0.52	464.78
100	1024	0.1691	58.39	0.0843	3.59	1525.29
100	2048	x	116.78*	0.0695	8.19	1325.08
100	4096	x	233.56*	0.0441	16.51	1314.48
225	225	0.3261	31.01	0.1656	5.14	502.21
225	1024	0.2843	269.72	0.1622	22.07	1121.67
225	2048	x	539.44*	0.1533	43.77	1132.38
225	4096	x	1078.88*	0.1132	86.86	1142.05
439	439	0.2135	245.28	0.1558	87.81	179.32
439	1024	0.2094	999.25	0.1494	199.23	401.55
439	2048	x	1998.5*	0.1423	397.86	402.3
439	4096	x	3997*	0.1163	795.31	402.56
1002	1002	0.2685	3001.79	0.1953	1402.84	113.97
1002	1024	0.2536	3191.32	0.1344	1425.98	123.79
1002	2048	x	6382.64*	0.0741	2852.25	123.77
1002	4096	x	12765.28*	0.5935	5704.6	123.77

As the problem domain gets bigger as well as its complexity, the achievement in case of the produced path lengths is somewhat decreased; but it is seen that increasing the ant number may yield better error ratios, too. As the ant number increases, the computation time goes up in a linear fashion.

The speed up column is defined as it is shown in Equation 5. This equation gives a percent ratio of the gained speed.

$$Speedup(tsp) = \frac{T_{exec}(serial) - T_{exec}(parallel)}{T_{exec}(parallel)} \quad (5)$$

Another interesting factor here is the solution quality. The parallel model produces better solutions comparing to the serial version of the algorithm. Since there are quite a few differences in the serial and parallel versions of the algorithm in terms of codification, such as the way the random numbers are generated, and considering that the used *cuRAND* library provides better randomness by using the parallel processing power of the *GPU*, the ants in the parallel algorithm are seen that they are more likely to choose better cities while constructing their tours [Fig. 15].

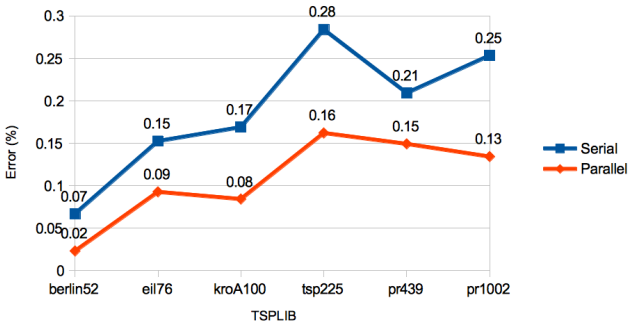

Fig. 15. The CPU and GPU solutions for all the libraries after 100 iterations. 1024 ants are involved in this experiment for both versions.

VII. CONCLUSIONS

In this paper, it is aimed to calculate the path of a *UAV* by using a parallelized Ant Colony Optimization Algorithms. To increase the performance of the system, implementation is done on *CUDA* architecture. Because of its inherently parallel nature, *ACO* is well-suited to *GPU* implementation, however it also poses significant challenges due to irregular memory access patterns of the algorithm. Experimental results are compared the *GPU* performance with *CPU* performance and the obtained results indicate that proposed parallel *ACO* approach clearly shows that this model has a great potential for acceleration of *ACO* and allow to solve much complex tasks.

Future work will concentrate on improving the performance of parallel *ACO* algorithm, by taking the pop-up threats, obstacles and real-time demand into the path planning process and on using *ACO* to calculate the routing path of messages between UAVs for constructing different types of networking models as depicted in [11]. At the same time, it is planned to use multi-colony *ACO* to calculate the paths of *multiple UAVs*. Because of the high performance power of *GPUs* it is expected this parallel approach will increase the performance of the system.

REFERENCES

[1] O. K. Sahingoz, "Large scale wireless sensor networks with multi-level dynamic key management scheme", Journal of Systems Architecture, vol. 59, no. 9, pp. 801-807, 2013.

[2] M. Dorigo, M. Birattari, & T. Stutzle, "Ant colony optimization", Computational Intelligence Magazine, IEEE, 1(4), 28-39, 2006

[3] T. Sttzle and M. Dorigo, "ACO Algorithms for the Traveling Salesman Problem", In K. Miettinen, M. Makela, P. Neittaanmaki and J. Periaux (eds.), Evolutionary Algorithms in Engineering and Computer Science, Chichester: Wiley, 1999.

[4] U. Cekmez, M. Ozsiginan, O. K. Sahingoz, "Adapting the GA approach to solve Traveling Salesman Problems on CUDA architecture", Computational Intelligence and Informatics (CINTI), 2013 IEEE 14th International Symposium on. IEEE, 2013.

[5] S. Sanci, & V. Isler, "A parallel algorithm for UAV flight route planning on GPU". International Journal of Parallel Programming, 39(6), 809-837, 2011.

[6] E. Kugu, O.K. Sahingoz, "ACO algorithms with multi-core implementation", 2013 7th International Conference on Application of Information and Communication Technologies (AICT), pp. 1-5, Oct. 2013.

[7] S. Tsutsui, N. Fujimoto, "Fast QAP solving by ACO with 2-opt local search on a GPU", 2011 IEEE Congress on Evolutionary Computation (CEC), pp.812-819, 5-8, June 2011.

[8] L. Dawson, I. Stewart, "Improving Ant Colony Optimization performance on the GPU using CUDA", 2013 IEEE Congress on Evolutionary Computation (CEC), pp.1901-1908, June 2013.

[9] J. M. Cecilia, J. M. Garcia, A. Nisbet, M. Amos, and M. Ujaldon, "Enhancing data parallelism for ant colony optimization on GPUs", Journal of Parallel Distributed Computing, vol. 73, no. 1, pp. 42-51, 2013.

[10] O.K. Sahingoz, "Generation of bezier curve-based flyable trajectories for multi-UAV systems with parallel genetic algorithm", Journal of Intelligent and Robotic Systems: Theory and Applications, 74 (1-2), pp. 499-511, 2014.

[11] O.K. Sahingoz, "Networking models in flying Ad-hoc networks (FANETs): Concepts and challenges", Journal of Intelligent and Robotic Systems: Theory and Applications, 74 (1-2), pp. 513-527, 2014.