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

# GPGPU Accelerated Real-Time Potential Field Based Formation Control for Unmanned Aerial Vehicles*

Omer Cetin and Guray Yilmaz

*Abstract* — **One of the research topics for the autonomous UAVs is formation flight control that provides two or more aircraft flies together under a discipline. For aerial vehicles the advantages of performing formation flight include fuel saving, improved efficiency in air traffic control and cooperative task allocation. Besides the benefits of the autonomous formation flight it has some difficulties like providing collision avoidance in narrow flight zone. Furthermore an autonomous formation flight must provide dynamic routines like getting formation position or changing formation schemes. Potential field based autonomous control is one of the commonly used control techniques providing dynamic and precise control. The bottleneck for the potential field based control is computation power needs especially for global path planning for dynamic, high resolution and large sized fields. GPGPU is one of the parallel computing architectures that provide programming massively parallel SIMD applications on GPUs. In this work GPGPU accelerated real-time potential field based formation control approach is designed and examined under simulation environment for UAVs. In order to reveal the precise and dynamic control features high-resolution potential field based global path planning models have been developed in real-time successfully.**

## I. INTRODUCTION

Formation flight is a kind of flight technique which is commonly used especially for military aircrafts or acrobatic teams. Military aircrafts flies in a formation especially for mutual defense and concentration of firepower. Also formation flight reduces fuel conception use by minimizing drag and thereby increasing the flight range [1]. The challenge of achieving safe autonomous formation flight control for Unmanned Aerial Vehicles (UAVs) has been extensively investigated in last year's [2]. Besides reducing fuel conception and increasing the flight range, one of the benefits for formation flight for UAVs is carrying more payloads on multiple platforms together [3]. Acting cooperatively together increases the probability of mission success and task completion time will be shortened. Instead of larger class and higher payload capacity type UAV, using multiple relatively smaller platforms is more economical. Distributing necessary sensors or equipment for a specific mission to the multiple platforms will require lower load-capacity UAVs, increase of performance and robustness compared to a single operating vehicle. Keeping the multiple platforms together in a formation will cause additional problems especially in narrow flight positions like collision avoidance. Furthermore during the formation flight obstacle avoidance will become more complex problem. Autonomous formation flight control mechanism must provide obstacle and collision avoidance properties at the same time.

Autonomous formation flight control is accepted as complex problem and it depends on various inputs to maintain the relative positions of the platforms. In particular, formation control as a kind of real-time application requires high processing power and it is directly associated with platforms maneuver capabilities. Thus solving the bottlenecks listed below will determine the effectiveness of the formation flight control approach;

- Formation control mechanism must support various size platforms (different types platforms with heterogeneous or homogeneous characteristics) while they are able to maintain the formation during the flight.
- Formation flight must include an embedded navigation system which is defined by certain rules.
- Formation control mechanism must support different flight formation schemes (narrow, V, echelon, box, line etc.) and it must allow switching between different schemes.
- Formation control mechanism must include collision avoidance future natively; especially while performing formation flight in narrow area.
- Formation control must react to the unexpected natural or artificial obstacles during the navigation; alternative patterns can be discerned during navigation.
- Autonomous formation control approach can generate real-time applicable flight patterns for each platform in the formation. Formation control mechanism must produce flexible and applicable patterns as solution for different type platforms.

According to the different needs like air refueling or manned-unmanned cooperation, formations can be putted forth in various types [4]. Furthermore autonomous formation approach can be used for integration of manned-un

* Research is supported by The Scientific and Technological Research Council of Turkey (TUBITAK) with project number 112E281.

Cetin O., Turkish Air Force Academy Aeronautics and Space Technologies Institute, 34149 Turkey (corresponding author; e-mail: o.cetin@hho.edu.tr).

Yilmaz G., Turkish Air Force Academy Computer Engineering Department 34149 Turkey (e-mail: g.yilmaz@hho.edu.tr).

manned platforms together, for example air refueling formation for unmanned systems or to accompany a manned platform by using unmanned systems. Beyond all of these, autonomous formation control can provide using multiple autonomous UAVs together, instead of using single high class bigger UAV; multiple smaller systems can be used for same mission.

The purpose of this work is developing a formation control mechanism which keeps the platforms in the correct position. In summary, formation flight for the UAVs can be defined as retaining the correct positions of the platforms against to the leader and other formation members during the navigation of multiple air vehicles. As seen from the fig. 1, different formations can be created due to the various purposes like fuel saving, passing between narrow obstacles, getting images of the same target by using different sensors, etc.

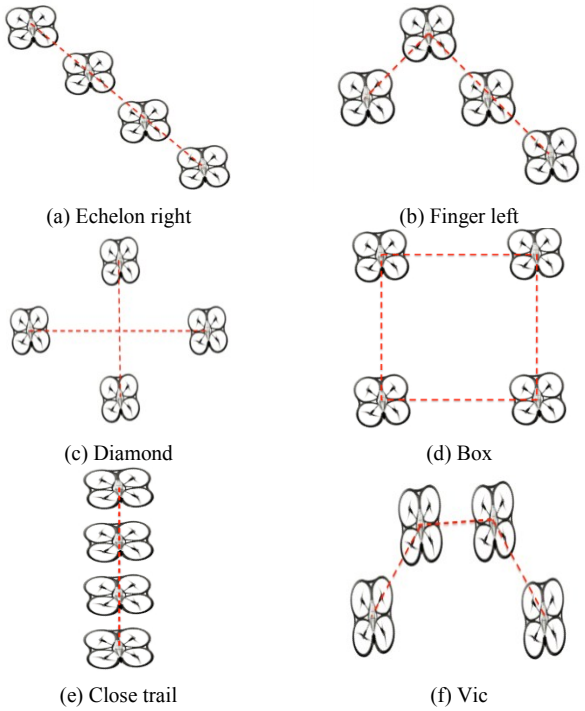



Figure 1. Various flight formation schemes

Formation control is a kind of dynamic problem and it has rapidly changing conditions. Although flight environment is 3D naturally, it can be handled as 2D for the platforms which are flying at the same altitude. There are also many conditions which are affecting the platform behaviors directly during the flight, and that are difficult to model or estimate them like wind, aerodynamic effects etc. Using complex models is hard to implement in real-time especially for the dynamic models. There are number of various approaches in literature for autonomous navigation control. However, each practical realization have some serious problems like imprecision of measured data, absence of detailed knowledge about environment, computational complexity resulting real-time bottlenecks, etc. Basically, in mobile robotic applications there are three main approaches to find the optimal path: heuristic, exact and grid algorithms [5]. Heuristic approaches are successful especially in static environments; they are simple and fast for finding the best solution. But changing the conditions in the problem space as in the dynamic environments, it is highly costly. An exact algorithm is mathematically correct way which ensures finding the best solution. However, they require precise sensing of obstacles and to produce a real-time solution for the complex problems are usually hard to implement. Grid based algorithms are more convenient for practical use because the precision of sensing is limited and they are more suitable for the dynamic environments.

Potential field approach as a kind of grid based algorithm provides finding the shortest and safest path in dynamic environments which may include moving obstacles. Potential filed approach is a result of gradient function. Potential field based control depends on gradient computation of a scalar field to produce vector field which determines the behavior. More precisely, the gradient of a scalar field is a vector field and its magnitude is the rate of change and it points in the direction of the greatest rate of increase of the scalar field. If the vector is resolved, its components represent the rate of change of the scalar field with respect to each directional component. The magnitude of the gradient will determine how fast the variation in that direction. The gradient operation on a scalar function results with the vector field of the scalar field. If the scalar function is differentiable, then the gradient of scalar function dotted with a unit vector gives the slope of the potential in the direction of the vector. More precisely, when scalar function is differentiable, the dot product of the gradient of scalar filed with a given unit vector is equal to the directional derivative of potential in the direction of that unit vector. There are also potential field based approaches which do not require having complete information of environment priori. It can be constructed in steps. It means that potential field based approaches are suitable for the dynamic conditions. It is possible to determine the precision of the grid or to design multi-layered grids, which enable only roughly to navigate a robot in a simple environment with only few obstacles and in the case of a more complicated area to switch to a more detailed grid description. However, also potential fields have some lacks, especially their computational complexity.

Computational complexity is hard to implement especially for dynamic, large-size, complex real-time problems. Besides the performance of the computation is directly related with the scalar model of the problem like dimension, size and complexity. Especially in large size, 3d modeled, dynamic and complex problems the performance of the computation of gradient function is poor to provide a real-time solution with generic processor architectures. Gradient computation of a multi-dimension scalar field is a repetition of an instruction set which is originated from partial derivative form of scalar function for each point in the region. The input data of the same instruction set is different because of the different scalar values in each point of the scalar field. As understood from this description its form is suitable for the Single Instruction Multiple Data (SIMD) parallel computing architecture [6].

In this paper, the performance of the computation of gradient function for a large scale scalar field will be improved by using SIMD form algorithms by using the General Purpose Computing on Graphical Processing Unit

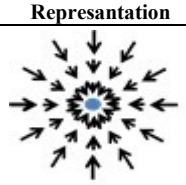
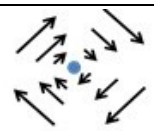
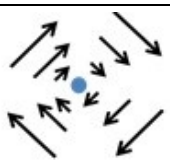
(GPGPU) techniques [7]. With usage of GPU based stream processors instead of single Central Processing Unit (CPU) and converting the scalar values as input data vectors, parallel algorithms for gradient computation is designed and improved by using special enhancements.

## II. POTENTIAL FIELD BASED CONTROL

### A. Basic Potential Field Models

First, in this part, basic potential field models will be discussed to steer the mobile platform as desired. Major movements in the formation can be listed as directing to the correct position, moving away from other platforms, going around other obstacles. These attitudes can be modeled by using four basic forms of vector fields. To produce correct vector fields, first of all, potential field model will be designed. Basic action models can be seen from the Table – 1. As seen from the table each one of the models will be represented with a potential model. By computing gradient of potential model, vector model can be computed as seen from the table 1.

TABLE 1. BASIC POTENTIAL FIELD MODELS

| No | Name | Potential Model | Vector Models | Graphical Represantation |
|---|---|---|---|---|
| 1 | Attractive Potential Field Model of Point Source | Equation (1) | Equations (9 and 13) |  |
| 2 | Repulsive Potential Field Model of Point Source | Equation (2) | Equations (10 and 14) |  |
| 3 | A Tangent Area Point Source Model (clockwise) | Equation (3) | Equations (11 and 15) |  |
| 4 | A Tangent Area Point Source Model (counter-clockwise) | Equation (4) | Equations (12 and 16) |  |

Potential models for the basic potential field structures can be seen fallowing equations (1-4). By using harmonic based functions local minima like potential field problems are solved [3]. Attractive potential field model of point source ($C_{rot}(x,y)$) can be seen form Equation 1 and repulsive potential field model of point source ($I_{rot}(x,y)$) can be seen form Equation 2. The parameter $alphaA$ represents the potential force factor, parameters $(A_x, A_y)$ represents the source point coordinate in 2D field. The parameter ($etaA$) represents the rotation of the field.

$$C_{rot}(x,y) = -log\left(\begin{pmatrix} alphaA * \\ \begin{pmatrix} sin(fiA)*(y-A_y) \\ -cos(fiA)*(x-A_x) \end{pmatrix}^2 - \\ etaA * \begin{pmatrix} sin(fiA)*(x-A_x)+ \\ cos(fiA)*(y-A_y) \end{pmatrix}^2 \end{pmatrix}\right) \quad (1)$$

$$I_{rot}(x,y) = \log\left(\begin{pmatrix} alphaA * \\ \begin{pmatrix} \cos(fiA)*(y-A_y)- \\ \sin(fiA)*(x-A_x) \end{pmatrix}^2 - \\ etaA * \begin{pmatrix} \sin(fiA)*(x-A_x)+ \\ \cos fiA*(y-A_y) \end{pmatrix}^2 \end{pmatrix}\right) \quad (2)$$

Equation 3 and 4 represents a tangent area point source model clockwise ($\mathrm{R}_p(x,y)$) and counter-clockwise ($\mathrm{R}_n(x,y)$) orderly.

$$\mathrm{R}_p(x,y) = 1 - 2*\left(log\left(-alphaA*\left(\left(\sqrt{(x-A_x)^2+(y-A_y)^2}\right)^2\right)\right)\right) \quad (3)$$

$$\mathrm{R}_n(x,y) = -1*\left(1-2*\left(log\left(-alphaA*\left(\sqrt{(x-A_x)^2+(y-A_y)^2}\right)^2\right)\right)\right) \quad (4)$$

By computing gradient of the potential models 2D vector fields can be obtained as seen from equation (5-8). As seen from the equations gradient operation is a kind of partial derivative for each axis.

$$\nabla C_{rot}(x,y) = \frac{\partial C_{rot}}{\partial x}\hat{\imath} + \frac{\partial C_{rot}}{\partial y}\hat{\jmath} \quad (5)$$

$$\nabla I_{rot}(x,y) = \frac{\partial I_{rot}}{\partial x}\hat{\imath} + \frac{\partial I_{rot}}{\partial y}\hat{\jmath} \quad (6)$$

$$\nabla R_p(x,y) = \frac{\partial R_p}{\partial x}\hat{\imath} + \frac{\partial R_p}{\partial y}\hat{\jmath} \quad (7)$$

$$\nabla R_n(x,y) = \frac{\partial R_n}{\partial x}\hat{\imath} + \frac{\partial R_n}{\partial y}\hat{\jmath} \quad (8)$$

Partial derivatives are computed to determine the change of movement in each axis. In 2D environment, for the x-axis partial derivatives of the basic actions can be computed as seen from equations (9-12). The parameter *k* indicates that each one of similar basic actions type in the same field.

$$DX_{C_{rot(k)}}(x,y) = \frac{\partial C_{rot}(x,y)}{\partial x} = \frac{\left(2*etaA*(cos(fiA)+sin(fiA))* \left(cos(fiA)*(A_y-x)+sin(fiA)*(A_x-x)\right)\right)}{\left(\begin{pmatrix} cos(fiA)*(A-y)-sin(fiA)*(A_y-y) \end{pmatrix}^2 + \\ etaA*\left(cos(fiA)*(A_y-x)+sin(fiA)*(A-x)\right)^2\right)} \quad (9)$$

$$DX_{I_{rot(k)}}(x,y) = \frac{\partial I_{rot}(x,y)}{\partial x} \quad (10)$$

$$= -\frac{\begin{pmatrix} 2 * etaA * \big(cos(fiA) + sin(fiA)\big) * \\ \Big(cos(fiA) * (A_y - x) + sin(fiA) * (A_x - x)\Big) \end{pmatrix}}{\begin{pmatrix} \big(cos(fiA) * (A_x - y) - sin(fiA) * (A_y - y)\big)^2 + \\ etaA * \Big(cos(fiA) * (A_y - x) + sin(fiA) * (A_x - x)\Big)^2 \end{pmatrix}}$$

$$DX_{Rp(k)}(x,y) = \frac{\partial R_p(x,y)}{\partial x} = \frac{\big(4 * (Hx - x)\big)}{((Hx - x)^2 + (Hy - y)^2)} \quad (11)$$

$$DX_{Rn(k)}(x,y) = \frac{\partial R_n(x,y)}{\partial x} = -\frac{\big(4 * (Hx - x)\big)}{((Hx - x)^2 + (Hy - y)^2)} \quad (12)$$

In 2D environment, for the y-axis partial derivatives of the basic actions can be computed as seen from equations (13-16).

$$DY_{C_{rot}(k)}(x,y) = \frac{\partial C_{rot}(x,y)}{\partial y}$$

$$= \frac{\begin{matrix} 2 * \big(cos(fiA) - sin(fiA)\big) * \\ \Big(cos(fiA) * (A_x - y) - sin(fiA) * (A_y - y)\Big) \end{matrix}}{\begin{pmatrix} \big(cos(fiA) * (A - y) - sin(fiA) * (A_y - y)\big)^2 + \\ etaA * \big(cos(fiA) * (A - x) + sin(fiA) * (A_x - x)\big)^2 \end{pmatrix}} \quad (13)$$

$$DY_{I_{rot}(k)}(x,y) = \frac{\partial I_{rot}(x,y)}{\partial y}$$

$$= -\frac{\begin{pmatrix} 2 * \big(cos(fiA) - sin(fiA)\big) * \\ \Big(cos(fiA) * (A_x - y) - sin(fiA) * (A_y - y)\Big) \end{pmatrix}}{\begin{pmatrix} \big(cos(fiA) * (A_x - y) - sin(fiA) * (A_y - y)\big)^2 + \\ etaA * \Big(cos(fiA) * (A_y - x) + sin(fiA) * (A_x - x)\Big)^2 \end{pmatrix}} \quad (14)$$

$$DY_{Rp(k)}(x,y) = \frac{\partial R_p(x,y)}{\partial y} = \frac{\big(4 * (Hy - y)\big)}{((Hx - x)^2 + (Hy - y)^2)} \quad (15)$$

$$DY_{Rn(k)}(x,y) = \frac{\partial R_n(x,y)}{\partial y} = -\frac{\big(4 * (Hy - y)\big)}{((Hx - x)^2 + (Hy - y)^2)} \quad (16)$$

### *B. Determination of the Potential Fields Boundaries*

Each basic potential field model which is used to obtain total field, must be limited with a kind of dynamic boundary function. The purpose of limiting process is obtaining a suitable pattern for the UAVs in formation. Dynamic limiting functions can be created by using different functions like logical equations, sigmoid functions etc. [8]. Logical equations are easy to implement and fast calculable. But they can produce hard implementation solutions for the UAV maneuvers like sharp turn commands. In this work to obtain more suitable commands for UAVs, sigmoid functions are used as boundary functions. More smooth commands are obtained especially while passing between the basic sub-zones, especially in the regions which are near the zone's borders. Basic decreasing and increasing sigmoid boundary functions can be seen from the fig. 2.

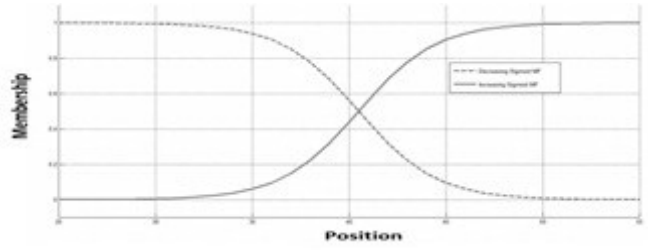



Figure 2. Basic decreasing and increasing sigmoid boundary functions

Decreasing sigmoidal boundary function can be obtained by using equation 17, and the increasing one can be produced by using the equation 18. The parameter *Slope* represents the breaking point and the parameter ρ provides the coverage of the function.

$$BF(k) = \frac{1}{1 + e^{-Slope*\rho}} \quad (17)$$

$$BF(k) = 1 - \left(\frac{1}{1 + e^{-Slope*\rho}}\right) \quad (18)$$

If we assume that there are "*m*" times same basic attractive field model, "*n*" times same basic repulsive field model, "*p*" times same basic clockwise tangential model and "*r*" times same basic counter-clockwise model in the same total field to model desired action by using as sub components as seen from equation 19, by summing each similar action type partial derivatives with other basic total forms, total change of movement can be computed for each axis.

$$SX(x,y) = \sum_{k=0}^{m} DX_{C_{rot}(k)}(x,y)\,\mathrm{BF(k)} + \sum_{k=0}^{n} DX_{I_{rot}(k)}(x,y)\,\mathrm{BF(k)} + \sum_{k=0}^{p} DX_{Rp(k)}(x,y)BF(k) + \sum_{k=0}^{r} DX_{Rn(k)}(x,y)BF(k)$$

$$SY(x,y) = \sum_{k=0}^{m} DY_{C_{rot}(k)}(x,y)BF(k) + \sum_{k=0}^{n} DY_{I_{rot}(k)}(x,y)\,\mathrm{BF(k)} + \sum_{k=0}^{p} DY_{Rp(k)}(x,y)\mathrm{BF(k)} + \sum_{k=0}^{r} DY_{Rn(k)}(x,y)\,BF(k) \quad (19)$$

### *C. Generation of Flight Commands*

To direct the platform in the flight area which is modeled by potential field, flight commands must be generated by using total vector field. To demonstrate generation process of flight commands, an example is used in this part. Example flight area is demonstrated with the parameters which are shown in Table 2.

TABLE 2. VALUE OF SIMULATION PARAMETERS

| Parameter | Value |
|---|---|
| $size\ (Area\ Dimension)$ | 500x500 |
| $Res\ (Area\ Resolution)$ | 20 |
| $alphaA$ | 1.0 |
| $etaA$ | 1.0 |
| $A_x, A_y$ | 30,250 |
| $fiA$ | $\pi$ |
| $alphaA1$ | 1.0 |
| $etaA1$ | 1.0 |
| $A1_x, A1_y$ | 150,400 |
| $fiA1$ | $\pi$ |
| $alphaA2$ | 1.0 |
| $etaA2$ | 1.0 |
| $A2_x, A2_y$ | 120,70 |
| $fiA2$ | $\pi$ |

The vector field that is calculated by parameter values in table 2 can be seen from fig. 3.

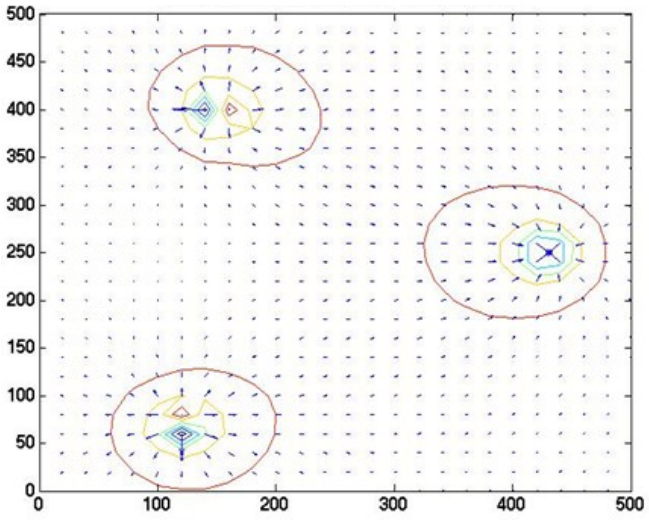


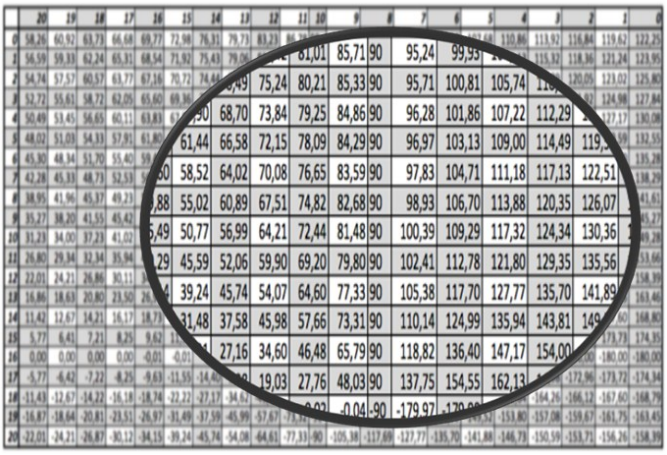

Figure 3. Example Vector Field

As shown in fig. 3, there are two repulsive and one attractive point as example potential sources which are determining actions of the platform in the field. Parameter *m* is equal to 1 and parameter *n* is equal to 2. First each basic potential model is computed by using related derivative functions to compute the vector field. After computing each vector field, total vector field is computed. To direct the platform, vector values must convert heading and speed commands. Heading command is direction of the vector and speed command is magnitude of the vector in each *(x, y)* point. Command values can be computed by using equations 20 and 21.

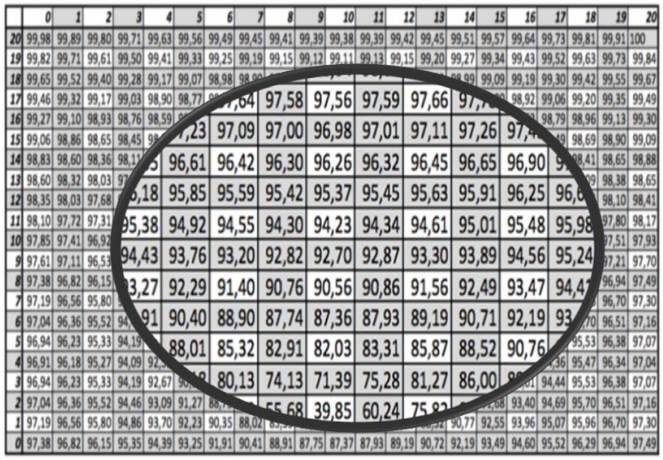
$$\text{heading}(x,y) = \arctan\left(\frac{SY(x,y)}{SX(x,y)}\right) \tag{20}$$

$$\text{speed}(x,y) = 1 - \sqrt{SX(x,y)^2 + SY(x,y)^2} \tag{21}$$

For each *(x, y)* point in the flight area heading and speed command can be computed by using same procedure and heading and speed command tables can be obtained as seen from fig. 4.



(a) Heading command table



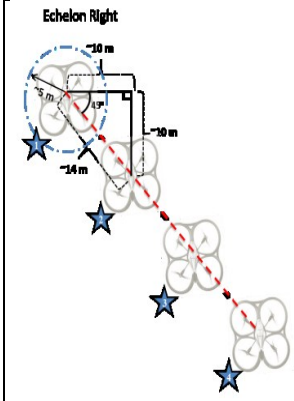
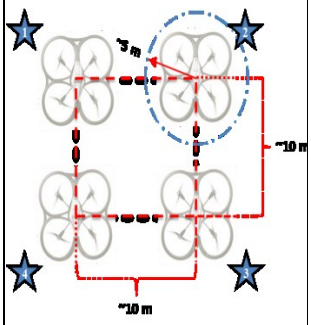
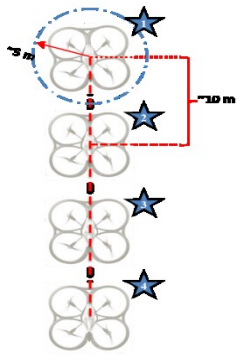
(b) Speed command table

Figure 4. Platform Command Tables in Flight Area

## III. REAL-TIME FORMATION CONTROL APPROACH

In this work, the echelon right formation (fig.1-a) is used for fuel saving, the close trail formation (fig.1-e) is used for passing between narrow gap and the box formation (fig.1-d) is used for getting the images of the same target by using different sensors. Furthermore in generally the reasons for the formation flight for the UAVs are keeping the platforms in a narrow distance to communicate between each other. Each platform tries to retain their correct position during the flight as shown in Table 3 and the leader continues the navigation during mission. Besides, each platform applies necessary maneuvers to avoid the collision during the flight. Formation control mechanism produce necessary commands for the each platform to retain the correct position and avoid collision in this work. Each platform in the formation is a kind of obstacle for the others and the position that the correct place as defined in the table 3 is the attractive source point. If we assume that there are 4 platforms in the formation, potential field for each platform totally four different potential fields must be calculated. Four different potential field and each one includes four different basic potential model must be calculated dynamically in real-time to provide a kind of collision avoidance supported formation control. The resolution is a kind of parameter which is directly affecting computation performance and precision of the maneuvers.

TABLE 3. RELATIVE POSITIONS OF THE PLATFORMS IN FORMATION FLIGHT

| Formation | UAV Positions |
|---|---|
|   **Echelon Right** | ***UAV_1***: $x_1, y_1$<br>***UAV_2***:<br>$x_2 = x_1 + \left(\frac{size}{10 * res}\right)$<br>$y_2 = y_1 - \left(\frac{size}{10 * res}\right)$<br>***UAV_3***:<br>$x_3 = x_1 + 2 * \left(\frac{size}{10 * res}\right)$<br>$y_3 = y_1 - 2 * \left(\frac{size}{10 * res}\right)$<br>***UAV_4***:<br>$x_4 = x_1 + 3 * \left(\frac{size}{10 * res}\right)$<br>$y_4 = y_1 - 3 * \left(\frac{size}{10 * res}\right)$ |
|  **Box** | ***UAV_1***: $x_1, y_1$<br>***UAV_2***:<br>$x_2 = x_1 + \left(\frac{size}{10 * res}\right)$<br>$y_2 = y_1$<br>***UAV_3***:<br>$x_3 = x_1 + \left(\frac{size}{10 * res}\right)$<br>$y_3 = y_1 - \left(\frac{size}{10 * res}\right)$<br>***UAV_4***:<br>$x_4 = x_1$<br>$y_4 = y_1 - \left(\frac{size}{10 * res}\right)$ |
|  **Close Trail** | ***UAV_1***: $x_1, y_1$<br>***UAV_2***:<br>$x_2 = x_1$<br>$y_2 = y_1 - \left(\frac{size}{10 * res}\right)$<br>***UAV_3***:<br>$x_3 = x_1$<br>$y_3 = y_1 - 2 * \left(\frac{size}{10 * res}\right)$<br>***UAV_4***:<br>$x_4 = x_1$<br>$y_4 = y_1 - 3 * \left(\frac{size}{10 * res}\right)$ |

### *A. GPGPU Computing Architecture*

In November 2006, NVIDIA introduced Compute Unified Device Architecture (CUDA), a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU [9]. CUDA comes with a software environment that allows developers to use C as a high-level programming language which provides a simple path for users familiar with the C programming language to easily write programs for execution by the device. CUDA C extends C by allowing the programmer to define C functions as group of instruction sets, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. CUDA threads may access data from multiple memory spaces during their execution. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps as SIMD parallel computing architecture [10]. An instruction that accesses addressable memory (i.e., global, local, shared, constant, or texture memory) might need to be re-issued multiple times depending on the distribution of the memory addresses across the threads within the warp. As shown in Table 4, stream multiprocessor (SM) is named as SMX with the Kepler architecture [10]. In Kepler GK110 architecture each SMX includes six memory controllers (64 Bit) as it is applied on K20c. Each SMX has got 192 single-precision CUDA cores, 64 double-precision units and 32 load/store units as shown in table 4 [10].

TABLE 4. KEPLER GK110 ARCHITECTURE WITH TESLA K20C

| Graphic Card | Tesla K20c |
|---|---|
| Architecture | Kepler GK110 |
| Number of SM/SMX | 13 SMX |
| Total CUDA core | 13 x 192 = 2496 |
| Number of SFU | 32 (on each SMX) |
| Number of LD/ST | 32 (on each SMX) |
| Memory Architecture | GDDR5 320 bit |
| Shared Memory Configuration (smem/cache – L1) | 16/32/48 Totally 64 KB |
| Compute Capability | 3.5 |

With CUDA, which will be held during the development of parallel software, depending on the hardware capabilities need to focus on are the two main elements;

- How many threads will be used simultaneously in parallel,
- Which memory architectures for thread communication, synchronization and data processing for the threads and their groups will be used?

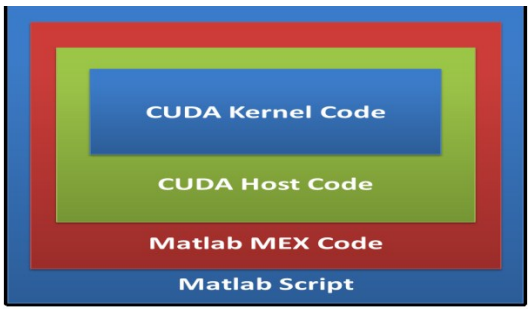



Figure 5. Software development environment

Furthermore it is clear that the performance of the GPGPU programming depends on hardware architecture directly. Although the CUDA is kind of C-based programming language, it is hard to produce visual outputs. Furthermore it is not easy to analyze math functions by producing visual graphics.

As shown in Fig. 5, the CUDA kernel code developed in this work, developed in C language called CUDA host code and natively supported by the CUDA architecture is surrounded by an upper-level code in this work. Prepared in the C language program called MATLAB MEX interface to access software was utilized, and therefore will create interfaces MEX codes and functions are used. MEX functions developed in Matlab scripts directly access is provided.

### B. GPGPU Accelerated Parallel Potential Field Computation Algorithm

In this part of the work, computation of the potential field based vector fields on NVIDIA GPU hardware with CUDA support which are compatible with GPGPU will be discussed. For this purpose, parallel software development environment will be summarized and the SIMD based algorithm will be designed.

*Matlab Script:* First, as outermost layer in the structure of software development Matlab script will be addressed. As shown in the following pseudo code block, firstly parameters of the potential field are defined and then MEX function is called with these parameters. Return values are used for computation of visual outputs by using Matlab standard functions.

```
1  Begin
2      Clear workspace and variable space

3      Define and initialize potential field parameters
       //area_dimension, area_res, x1, y1, alfa1, etc.

4      Call MEX Function with input parameters
       //[A, X, ...] = Potential_field([area_dimension,
   area_size], [x1,y1,alfa1,...],...]

5      Get return values and assign to output variables

6      Compute visual outputs
       //quiver (A, X, Y, SX, SY)
7  End
```

As shown in the code block, MEX function is named as "*Potential_field*" and it has input and output parameters. It is written in C++ programming language and it includes CUDA host and kernel code blocks.

*Matlab MEX Code:* In fact Matlab MEX code is a kind of C++ functions with its input and output parameters. But it additionally supports calls from Matlab scripts. After compiling MEX code with CUDA compiler PTX code is achieved. The compiler generates PTX code which is also not hardware specific. At runtime the PTX is compiled for a specific target GPU - this is the responsibility of the GPU driver. PTX code can contain many functions that can run on the CPU (as host code) as well as on the GPU (as device code).

MEX calls and in compliance with the parameters must be defined in a special format. This structure is shown in the following code block.

```
1  void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
   mxArray const *prhs[])
2  Begin
3   Initialize GPU device
     //mxInitGPU()
     ...
```

The parameter "*plhs*" is an array of right-hand input arguments, the parameter "*plhs*" is an array left-hand output arguments. The parameter "*nrhs*" represents the number of right-hand arguments, or the size of the "*phrs*" array and the parameter "*nlhs*" is the number of left-hand arguments, or the size of the "*plhs*" array. The access of the input and output parameters of the MEX function are provided through these pointers.

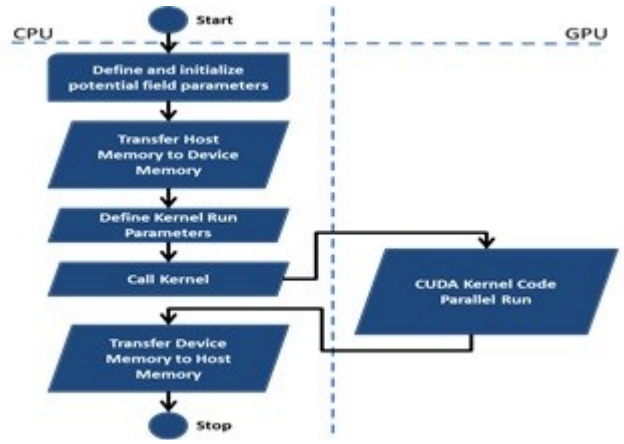



Figure 6. Basic CUDA parallel code flow chart

*CUDA Host Code:* Basically a CUDA program is a kind of process as shown in Fig. 6. It consists of six main routine and five of them runs on the host side. The kernel code which is a kind of SIMD structure is the only one that performed on GPU processor. The following pseudo code block shows the appropriate definitions with MEX structure and CUDA.

```
4      Define host variables
       // alpha_CPU  = mxCreateDoubleMatrix(nrhs-1, 1,
   mxREAL);
       ...
5      Get input parameters values and assign
       // in_area = mxGetPr(prhs[0]);
    // double a_size=in_area[0];
         ...
6      Define and initialize GPU variables
       // Dp_alpha_GPU  =
   mxGPUCreateFromMxArray(alpha_CPU);
       ...
       Get pointers of the CPU variables
7      // Hp_alpha_CPU   = mxGetPr(alpha_CPU);
    // Hp_ptype_CPU   = mxGetPr(ptype_CPU);
       ...
8      Get pointers of the GPU variables
       // double *p_kX_GPU , *p_kY_GPU, *p_eta_GPU,
   *p_fi_GPU;
       // p_kX_GPU = (double
   *)(mxGPUGetDataReadOnly(Dp_kX_GPU));...
```

The values which are specified with MEX structure as input data are assigned to the CPU memory variables. Memory address values of these variables are represented by different variables. Later by making the variable definition on the GPU in the same way, memory data were transferred from system memory to GPU memory.

As seen in the following code block to the next process defined in the GPU memory, but not receiving an external input parameter values from the variable definitions have been carried out. On GPU inherently variable definitions SIMD array is defined as usually. Core routines will handle similar data were collected on the same arrays. GPU created in memory as the initial value for each element of the array of data "0" is assigned.

```
9    Determination of the size of n x m dimensional array
     on GPU memory
       // mwSize element_size1[1];
            element_size1[0] = size;
            ...
10   Define and initialize GPU arrays
       // mxGPUArray *A_GPU, *X_GPU, *Y_GPU,
     *DX_GPU, ... ;
        A_GPU  = mxGPUCreateGPUArray(1,
     element_size1, mxDOUBLE_CLASS, mxREAL,
     MX_GPU_INITIALIZE_VALUES);
           ...
11   Get the pointers of GPU arrays
          // Dp_A_GPU  = (double
     *)(mxGPUGetData(A_GPU));

          Dp_X_GPU  = (double
     *)(mxGPUGetData(X_GPU));

     ...
```

The next operation which is performed on the host code is determining kernel call parameters and implementation of the kernel call as shown in fallowing pseudo code block. To compute the potential field, in the other words to compute the vector field on GPU by using partial derivatives to implement gradient operation, three different kernel code block is designed and used together. First two kernel codes block implements “meshgrid” function which is defined in Matlab standard dictionary as a kind of function. The last one computes the partial derivative of the scalar field function to obtain vector field.

```
12     Definition of the Stepper kernel function parallel
       computing parameters and kernel call
       // dim3 threadsPerBlock (size, 1, 1);
         int blockx = (size+threadsPerBlock.x-
       1)/threadsPerBlock.x;
         dim3 blocksPerGrid(blockx,1,1);
         stepper<<<blocksPerGrid,
       threadsPerBlock>>>(d_A, Dp_A_GPU);

       Definition of the grid kernel function parallel
13     computing parameters and kernel call
       // dim3 blocksPerGrid2(size,1,1);
         Grid_builder<<<blocksPerGrid2,
       threadsPerBlock>>>(Dp_X_GPU, Dp_Y_GPU,
       Dp_A_GPU);

       Definition of the potential kernel function parallel
       computing parameters and kernel call
14     // dim3 threadsPerBlock3 (size, 1, 1);
         dim3 blocksPerGrid3(size,1,1);
          potential<<<blocksPerGrid3,
       threadsPerBlock3>>> (p_kX_GPU, p_kY_GPU,
       p_fi_GPU, p_eta_GPU, Dp_X_GPU, Dp_Y_GPU,
       Dp_DX_GPU, Dp_DY_GPU, Dp_SX_GPU,
       Dp_SY_GPU);
```

One of the CUDA structures is grid, block and thread organization. This structure can be explain briefly as threads that are created by kernel function, access to data in regulated order and also they are separated each other by using this notation. Threads which are derived from the core code, constitute blocks by grouping and the blocks come together to constitute the grid structure. The maximum number of the blocks in a grid and the maximum number of the threads in a block is kind of parameter which is defined by the hardware specific feature. Usually grids are produced by using different core functions but blocks and threads which are in the same grid are produced by using same kernel code. Another difference between these structures is the memory architectures. For example threads communicate with each other by using shared memory but blocks use global memory. Besides each thread has own register space.

TABLE 5. NVIDIA GEFORCE GTX 670MX DEVICE INFO

| | |
|---|---|
| ComputeCapability | 3.0 |
| Driver Version | 5 |
| Toolkit Version | 5 |
| Max Threads Per Block | 1024 |
| Max Shmem Per Block | 49152 |
| Max Thread Block Size | [1024 1024 64] |
| Max Grid Size | [2.1475e+09 65535 65535] |
| SIMD Width | 32 |
| Total Memory | 3.2212e+09 |
| Multiprocessor Count | 5 |
| Clock Rate KHz | 601000 |

In Matlab programming environment “gpuDevice” command output can be seen from table 5 for NVIDIA GeForce GTX 670MX device as an example. These parameters are hardware specific values and they determine thread, block and grid parameter sizes.

```
15     Memory transfer from device to host
       //plhs[0] = mxGPUCreateMxArrayOnCPU (A_GPU);
         plhs[1] = mxGPUCreateMxArrayOnCPU
       (X_GPU);
         plhs[2] = mxGPUCreateMxArrayOnCPU
       (Y_GPU);
          ...

16     Free up allocated GPU memory
       //mxGPUDestroyGPUArray(Dp_alpha_GPU);
         mxGPUDestroyGPUArray(Dp_ptype_GPU);
         ...

17   End
```

After kernel functions are invoked, return values must be copied from device to system memory as shown in Fig. 6. After transfer operation is completed there is no need to allocate GPU memory anymore, so that GPU allocated memory space is released.

*CUDA Kernel Codes:* As seen from Fig. 5, the inner layer of the software development environment is kernel codes. Kernel codes are the same code blocks which runs on the different data as defined in SIMD structure. The purpose is increasing computing performance by using multiple GPU processors on different data in parallel. By using the kernel code different threads are created and each one of the threads has a unique thread and block number. Each thread access different data and after process it writes results on different memory spaces. All of this process is implemented on GPU memory.

First kernel code to compute potential field is stepper function which is used for creating a meshgrid.

```
1  void __global__ stepper(double const * const area, double
   * const A)
2  Begin

3      int const i = blockDim.x * blockIdx.x + threadIdx.x;

4      if (i < ((area[0] / area[1]) + 1))
5      Begin
6        A[i] = i * area[1];
7      End
8  End
```

As seen the above pseudo code, each thread computes a member of array "A" according as thread, block and grid number of thread. The "*area[0]*" variable keeps area size value, "*area[1]*" variable keeps area resolution value.

```
1  void __global__ grid_builder (double * X, double * Y,
   double * const A)
2  Begin

3      int const i = blockDim.x * blockIdx.x + threadIdx.x;

4      X[i] = A [threadIdx.x];
5      Y[i] = A [blockIdx.x];
6  End
```

The upper code block shows the Grid_builder kernel function. Stepper and Grid_builder kernel codes do the same job with the "meshgrid" function which is defined in Matlab programming environment. But they run on GPU processors parallel. As seen from the code block [X, Y] is kind of pixel grid that is computed by using thread and block id parameters.

```
1  void __global__ potential(double * Ax, double * Ay, double
   * fiA, double * etaA,  double * X, double * Y, ...  double *
   SX, double * SY)
2  Begin

3      int const i = blockDim.x * blockIdx.x + threadIdx.x;
       int const k = 0;
4      Compute A_DY[i]
5      Compute A_DX[i]
6      Synchronize
7      Compute SX[i]
8      Compute   SY[i]
9      Synchronize
10     Compute Heading[i]
11     Compute Speed[i]
12 End
```

The "*potential*" kernel code block which is defined above, calculates the vector field in parallel. Each *(x, y)* point in the potential field is computed by one thread which is running on GPU stream processor. Each thread accesses one memory location which is defined as input stream by using thread and block id parameters. Step 4 and 5 in the algorithm compute the partial derivative of the potential field by using appropriate functions which are defined in table 1. This structure is a kind of SIMD type parallel GPGPU based process. After computing different point's partial derivatives, all of the threads must synchronize as seen step 5. Then total value of the vector field can be computed as shown with step 6 and 7. After computing all total values for each *(x, y)* point with synchronize operation as shown with step 8, platform commands (heading and speed) can be computed parallel to generate command tables as shown with Fig. 4.

### *C. GPGPU Accelerated Parallel Potential Field Computation Simulation*

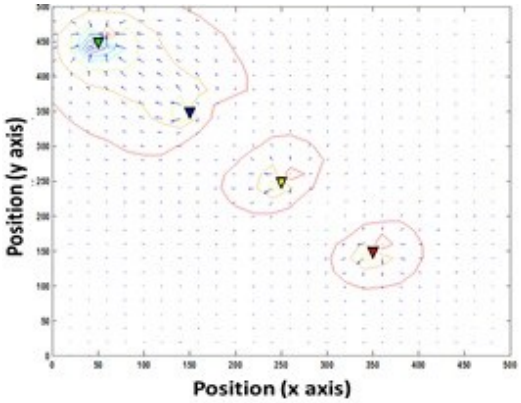
In this part of the work, the success of the GPGPU accelerated parallel potential field computation algorithm will be evaluated with a simulation for different formation schemes. Only the leader platform position will be given manually as shown in table 6 and the other platforms will calculate their positions by using the equations which are defined with table 3.

TABLE 6. SIMULATION PARAMETERS

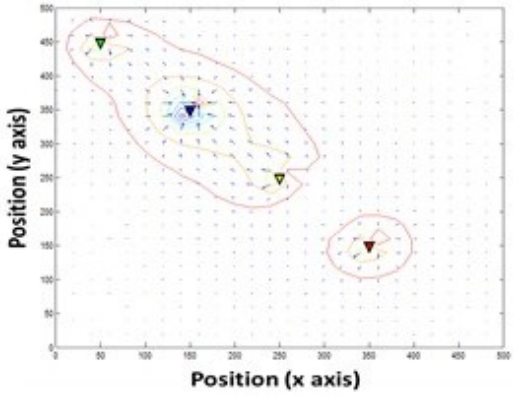
| Simulation No | Leader Position | Formation Scheme |
|---|---|---|
| 1 | (50,450) | Echelon Left |
| 2 | (100,450) | Box |
| 3 | (250,450) | Trail |

The correct place will be the attractive manner point in the potential field for the related UAV and the others positions acts like repulsive sources to avoid collision while platform getting the correct place. For each simulation, four UAV will be used as formation members. Therefore totally four different potential fields will be computed, one for each formation member.

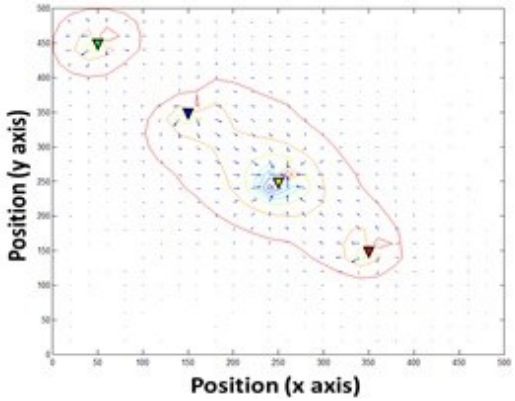
Potential field based global path planning in formation is important to detect and predict collision risk especially fast and limited mobility vehicles like UAVs. Each instant in the formation flight all of the path planning for platforms will be computed globally.



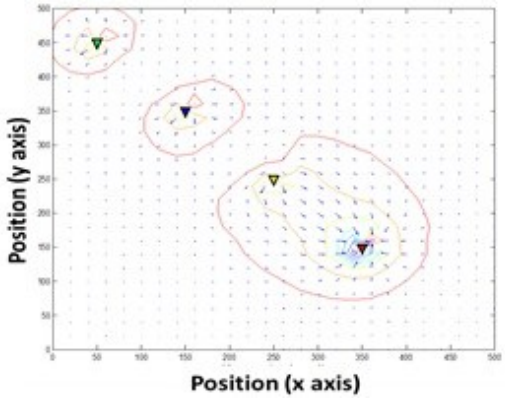

(a) Echelon left formation vector field for leader platform




(b) Echelon left formation vector field for platform no 2



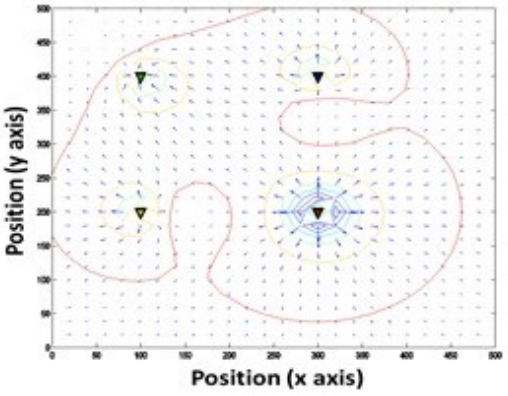

(c) Echelon left formation vector field for platform no3



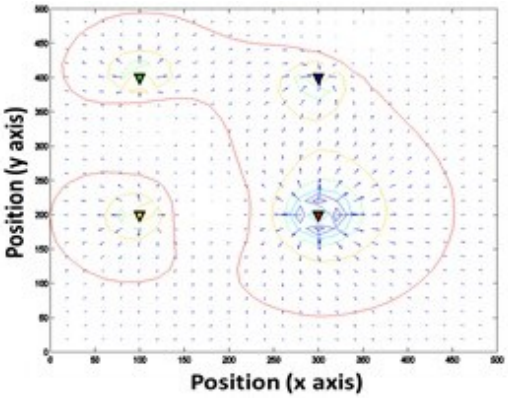

(d) Echelon left formation vector field for platform no 4

Figure 7 GPGPU accelerated potential field based echelon left formation control

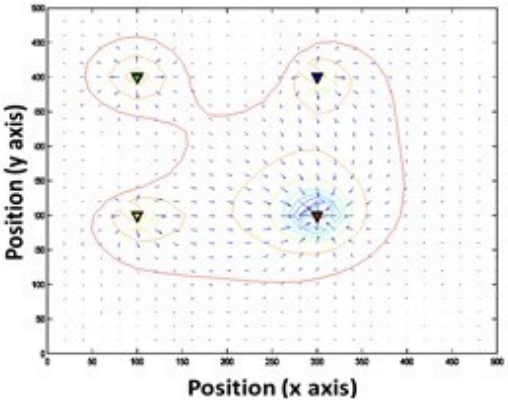
As seen from Fig. 7, potential field based formation control generates vector fields for path planning of each UAV dynamically. GPGPU accelerated parallel computed potential field based formation control for echelon left scheme can be seen from Fig. 7. In Fig. 8 and 9 shows formation control for box and trail schemes respectively.



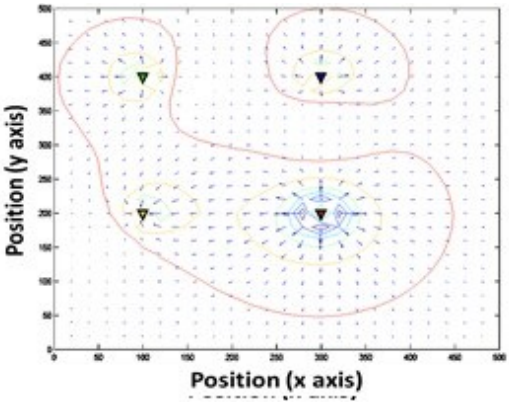

(a) Box formation vector field for leader platform




(b) Box formation vector field for platform no 2

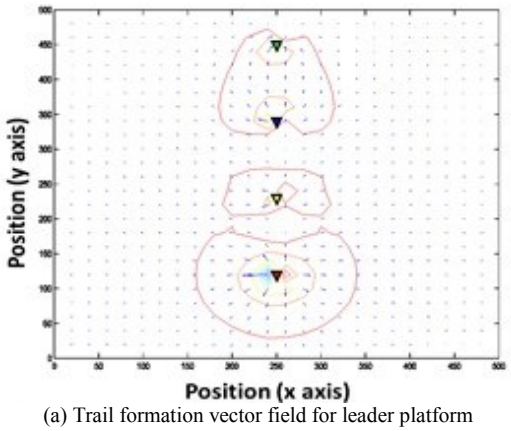



(c) Box formation vector field for platform no3

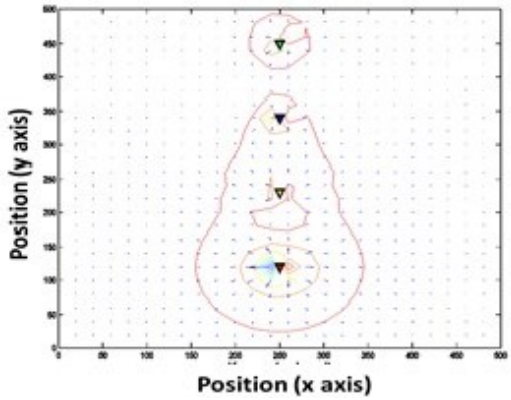



(d) Box formation vector field for platform no 4

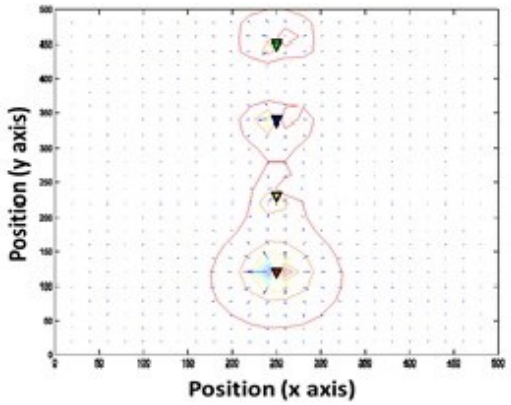
Figure 8. GPGPU accelerated potential field based box formation control



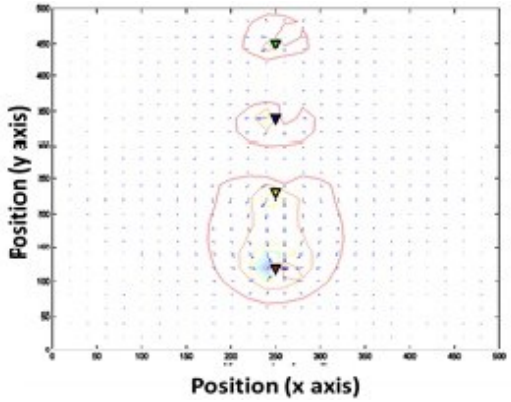

(a) Trail formation vector field for leader platform




(b) Trail formation vector field for platform no 2




(c) Trail formation vector field for platform no3




(d) Trail formation vector field for platform no 4

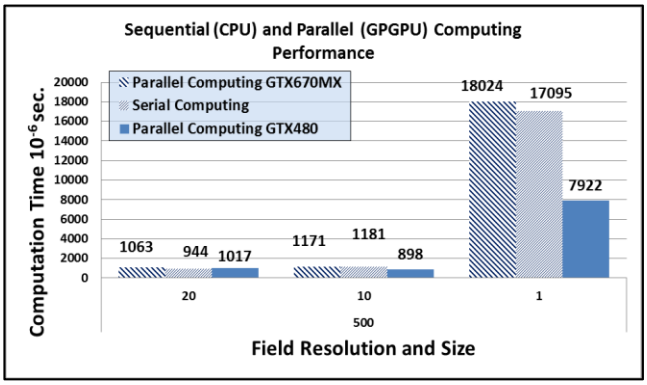
Figure 9. GPGPU accelerated potential field based trail form. control

*D. GPGPU Accelerated Parallel Potential Field Computation Performance*

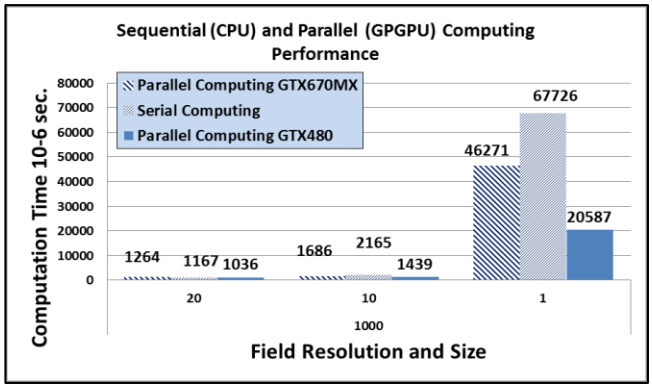
As a result of any movement of the platforms located in formation, all of the potential fields need to be recalculated. Because each UAV in formation is a dynamic obstacle for others and it affects the movement of the platforms. This is a costly process in terms of computational performance. Another factor which is affecting computation performance directly is resolution of the potential field. High resolution provides high-precision maneuvers and it is important to obtain collision avoidance in narrow flight area. But cost will be higher in terms of computation performance. By using the system configuration which is shown in Table 7, different simulations are measured for different field size and resolutions.

TABLE – 7 SIMULATION SYSTEM CONFIGURATION

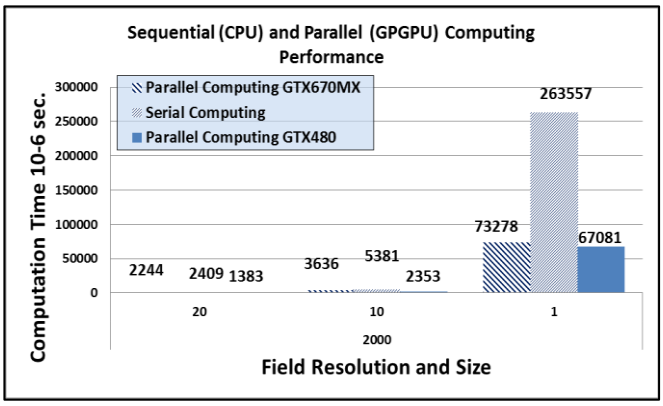
| Properties | Value |
|---|---|
| CPU | Intel® Core™ i7-3630QM @2.40GHz |
| Host Memory | 32 GB DDR3 |
| GPU | NVIDIA GeForce 670MX<br>NVIDIA GeForce GTX480 |
| Operating system | MS Windows 7 ® Ultimate 64 bit |
| CUDA Version | CUDA v5.5 |
| MATLAB Version | Matlab r2013a |
| C++ Version | Microsoft Visual C++ 2010 (v 10.0) |



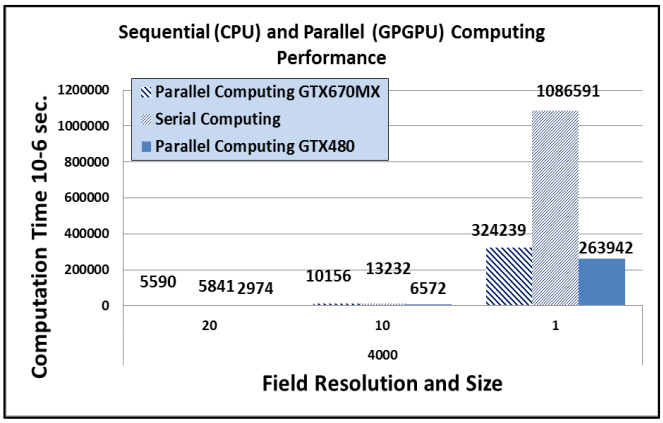

(a) 500 x 500 pixel size computation performance




(b) 1000 x 1000 pixel size computation performance




(c) 2000 x 2000 pixel size computation performance




(d) 4000 x 4000 pixel size computation performance

Figure 10. Sequential (CPU) and Parallel (GPGPU) potential field based formation control algorithm computation performance

For different size and resolutions computation performance of sequential and parallel computing of potential field based formation control algorithm can be seen from Fig. 10. As shown in the Fig. 10, serial and parallel computation performance has been demonstrated in comparison. Analyzing the results of graphics, it can be said that when the resolution and scale of the field grows, parallel computation produces more effective results. For the 4000 x 4000 pixel area and the single resolution value, about 4.1X speed up is achieved.

## IV. Conclusion

In this work; it has been demonstrate that formation control can be achieved by potential field based models which also include collision and obstacle avoidance properties. By increasing field resolution and size, more precise maneuvers can be planned autonomously. Furthermore by modeling global path planning instead of local path planning, exact results can be obtained to provide collision free patterns for each UAV in a dynamic environment. But using global path planning results by modeling large size of environment with high resolution causes computation performance bottlenecks especially for real time dynamic problems like formation control. To overcome performance bottlenecks, parallel algorithms are developed and performed on the stream processors like GPUs. Significant performance improvements are obtained.

As a future work, by tuning the GPGPU based parallel algorithms by considering GPU architectures more performance improvements can be obtained. Furthermore by adding obstacle models, formation control algorithm becomes more complex computation. Also to obtain 3D models multi-layer potential fields can be modeled.

Formation flight control for UAVs are one of the popular research areas for autonomous systems. As a result of this work, it can be said that potential field based formation control is successfully applied as a real-time solution in simulation environment.

## References

[1] G. Elkaim and R. Kelbley, “A lightweight formation control methodology for a swarm of non-holonomic vehicles,” in Aerospace Conference, March 2006.

[2] Meir Pachter, John J. D', Azzo, and Andrew W. Proud. "Tight Formation Flight Control", Journal of Guidance, Control, and Dynamics, Vol. 24, No. 2 (2001), pp. 246-254.

[3] O. Cetin, I. Zagli, G. Yilmaz, “Establishing Obstacle and Collision Free Communication Relay for UAVs with Artificial Potential Fields” Journal of Intelligent & Robotic Systems, January 2013, Volume 69, Issue 1-4, pp. 361-372.

[4] O. Cetin, G. Yilmaz, “GPGPU Accelerated Potential Field Based Autonomous Air Refueling Approach for UAVs”, In proceeding of International Conference on Unmanned Aerial Vehicles ICUAS13.

[5] J. Vascak, “Navigation of Mobile Robots by Computational Intelligence Means”, proceedings of 5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics, January 25-26, 2007.

[6] S. Ryoo, et all, “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA”, proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming pp. 73-82.

[7] D. Luebke, et all, “GPGPU: general-purpose computation on graphics hardware”, Proceedings of the 2006 ACM/IEEE conference on Supercomputing May 1995, Article No. 208.

[8] O. Cetin, G. Yilmaz, “Sigmoid Limiting Functions and Potential Field Based Autonomous Air Refueling Path Planning for UAVs”, Journal of Intelligent & Robotic Systems, January 2014, Volume 73, Issue 1-4, pp. 797-810.

[9] P. Harish. J. Narayanan, “Accelerating Large Graph Algorithms on the GPU Using CUDA”, High Performance Computing – HiPC 2007, Lecture Notes in Computer Science Volume 4873, 2007, pp. 197-208.

[10] Whitepaper, NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110, http://www.nvidia.com/ content/ PDF/kepler/ NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, [last access 14 Feb.2014].