Reporting flock patterns on the GPU

Marta Fort, J.Antoni Sellarès, and Nacho Valladares*

Departament Informàtica, Matemàtica Aplicada i Estadística. Universitat de Girona.

Abstract

In this paper we study the problem of finding flock patterns in a set of trajectories of moving entities. A flock refers to a large enough subset of entities that move close to each other for a given time interval. We present a parallel approach, to be run on a Graphics Processing Unit, for reporting maximal flocks. We also provide experimental results that show the efficiency and scalability of our approach.

1 Introduction

A trajectory is a sequence of sampled time-stamped point locations describing the path of a moving entity over a period of time. Assuming that the movement of an entity between two consecutive positions is done at constant speed and without changing direction, its trajectory is modelled by the polygonal line, that can self-intersect, whose vertices are the trajectory points.

In this paper we study the flock pattern [1, 2, 3], that identifies a group of entities moving close together during a given time interval (see Figure 1).

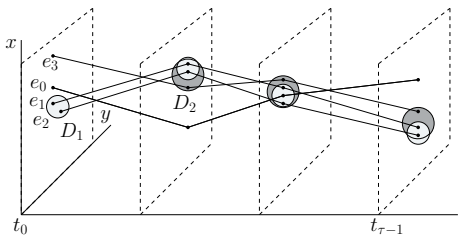

Figure 1: Example of two flocks.

The increasing programmability and high computational rates of Graphics Processing Units (GPUs), together with CUDA (Compute Unified Device Architecture) and some programming languages which use this architecture, make them attractive to solve problems which can be treated in parallel as an alternative to CPUs. GPUs are used in different computational tasks where a big amount of data or operations have to be done, whenever they can be processed or done in parallel. Some recent works show that demanding algorithms in different fields can take advantage of the GPU parallel processing [4, 5, 6].

*Email:(mfort,sellares,ivalladares)@imae.udg.edu.

Authors research supported by TIN2010-20590-C02-02.

In this paper, we present an efficient parallel GPU-based algorithm, designed under CUDA architecture, for reporting maximal flocks. The experimental results obtained with the implementation of our algorithm show the significance of the presented approach according to its performance and scalability.

2 The flock pattern

Let $E = \{e_0, \ldots, e_{n-1}\}$ be a set of n moving entities. The trajectory T_i of the entity e_i is a sequence of τ points in the plane $T_i : e_0^i, \ldots, e_{\tau-1}^i$, where e_j^i denotes the position of e_i at time t_j with $0 \leq j < \tau$. We assume that the positions are sampled synchronously for all the entities, and that entity e_i moves between two consecutive positions e_j^i, e_{j+1}^i with constant speed and without changing direction. Consequently, the trajectory T_i is described by the polygonal line, which may self-intersect, whose vertices are the trajectory points.

Flocks identify groups of entities whose trajectories are close together during a minimum period of time. Formally, according to [2], given a set E of entities, a minimum number of entities $\mu \in \mathbb{N}$, a number of time-steps $\delta \in \mathbb{N}$, a time interval $I_j^\delta = [t_j, t_{j+\delta-1}]$, and a distance $\epsilon \in \mathbb{R}$:

Definition 1 A flock $f(\delta, \mu, \epsilon)$ in a time interval I_j^δ, consists of at least μ entities such that for every discrete time step $t_\ell \in I_j^\delta$, there is a disk of radius ϵ that contains all the μ entities.

The number of reported or determined flocks is a critical issue that can adversely affect the response time and the proper interpretation of the final results. For a time-step $t_\ell \in I_j^\delta$ there may be several circles with radius ϵ that yield a flock with the same subset of entities of E, so we consider that two flocks are different if they involve two different subsets. Since there can be $\Theta(n^2)$ combinatorially distinct ways to place a circle of radius ϵ among n points in the plane, at each time-step there can be $\Theta(n^2)$ flock candidates involving different subsets of entities [9]. An algorithm having as output the position of the entities involved in a flock, would have an output size of $\Theta(n^3)$ per time-step. Moreover, it is easy to observe that a set of entities could be present in many flocks, and even one single entity can be involved in several flocks. For example, a flock of $\mu+1$ entities contains $\mu+1$ flocks of μ entities.

To overcome this problem, as in [3], instead of finding all the existent flocks, we are interested only in maximal flocks, so that the entities of one flock may not be a subset of the entities of another flock. Even in the case of finding maximal flocks, at every time-step there may still asymptotically be $\Theta(n^2)$ flock candidates, although in many real situations the number of candidates decreases considerably.

Definition 2 A maximal flock $mf(\delta, \mu, \epsilon)$ in the time interval I_j^δ, is a maximal flock $f(\delta, \mu, \epsilon)$ in I_j^δ with respect to the subset inclusion.

We will denote $\mathcal{M}_j^\delta$ the family of maximal flocks in the time interval I_j^δ, and $\mathcal{M}^\delta$ the family of all maximal flocks, that is, the flocks of $\mathcal{M}_j^\delta$, for $j = 0, \ldots, \tau - 1$.

2.1 Characterization

Vieira et al. [3] prove that, for each pair of entity locations, there exists two such disks. Thus, $2n^2$ disks per time step must be tested. Since we are interested in those flocks that are maximal, we prove that keeping just one of the two disks is enough. Consequently, only n^2 disks per time step must be tested, thus the number of tests is reduced by half.

The following lemma, similar to one presented in [3], allows us to bound the number of disks to be tested in each time step.

Lemma 3 *Let P be a set of n points and D' be a disk of radius r that covers a subset $P' \subseteq P$, $|P'| = k \leq n$. There exist another disk D'' of radius r such that: a) has at least two points $p_i, p_j \in P'$ on its boundary; b) covers a superset P'' of P', $P' \subseteq P'' \subseteq P$; c) it is located at the right side of points p_i, p_j.*

3 Reporting flock patterns

In this paper study the problem of reporting the family $\mathcal{M}^\delta$ of all maximal flocks. To report $\mathcal{M}^\delta$, we must find the family $\mathcal{M}_j^\delta$ in the time interval I_j^δ, for $j = 0, \ldots, \tau - 1$.

3.1 Computing the family $\mathcal{M}_j^\delta$

Next, we describe the two main steps of the algorithm used to compute the family $\mathcal{M}_j^\delta$ in the time interval I_j^δ.

First step. At each time step t_i in the time interval I_j^δ, we compute the family $\mathcal{F}_i$ of potential flocks containing at least μ entities. Each family $\mathcal{F}_i$ is obtained by using the characterization given in Lemma 3. Next, we find the maximal sets of each family $\mathcal{F}_i$. Abusing of notation, we still denote $\mathcal{F}_i$ the obtained subfamily of potential maximal flocks. At the end of the step, we have the array of families of potential maximal flocks $\mathcal{F}_j^\delta = [\mathcal{F}_j, \ldots, \mathcal{F}_{j+\delta-1}]$ in the time interval I_j^δ.

Second step. We first compute the family $\mathcal{I}_j^\delta$ obtained intersecting the δ families of potential maximal flocks in $\mathcal{F}_j^\delta$. That is, $\mathcal{I}_j^\delta$ contains the sets which are the intersection of δ sets, one of each family $\mathcal{F}_i$ in $\mathcal{F}_j^\delta$. Finally, we find the subfamily of maximal sets of $\mathcal{I}_j^\delta$ that contain at least μ elements. The obtained subfamily is the desired $\mathcal{M}_j^\delta$.

4 GPU implementation

In this Section, we describe how to implement the different steps of the algorithm that computes $\mathcal{M}_j^\delta$, when GPU parallel techniques are used.

4.1 The multi-grid structure $\mathcal{G}$

We denote by $E_i = \{e_i^0, \ldots, e_i^{n-1}\}$ the set of locations of the entities of E at time step t_i, $i \in \{0, \ldots, \tau - 1\}$. In [3], a single grid is used to perform quick disk range searching queries over each set E_i. Instead of using a grid for each time step, we build a multi-grid structure, denoted $\mathcal{G}$, in parallel in the GPU, that allows us to search simultaneously in each one of the δ sets of $E_j^\delta = [E_j, \ldots, E_{j+\delta-1}]$.

The multi-grid structure contains δ regular grids, where the edges of the grid cells have length ϵ. Each grid j contains the set of points corresponding to E_j. In order to maximize the parallel performance, the top-right corner is defined so that the grid is a square, that is, it has the same number of rows and columns (see Figure 2).

The multi-grid structure $\mathcal{G}$ is composed of 7 arrays allocated in the GPU global memory. In Figure 2 we can see an example for $\delta = 3$. The bottom-left and the top-right corners of each grid are stored in the array b_δ of size δ, where $b_\delta[j]$ stores the two corner points of the grid containing E_j. The number of cells per row (or per column) of each grid is stored in $\widetilde{c_\delta}$ and its prefix sum is stored in $\widetilde{p_\delta}$. Both arrays are integer arrays of size δ.

To represent the δ grids, we use the array of integers c_δ, whose size depends on the size of the different grids which directly depend on the distribution of the points. For a given grid cell, we store in the corresponding c_δ position the number of points contained on that cell. The array p_δ of the same size of c_δ is obtained as the prefix sum of c_δ. Note that, for a better understanding, in Figure 2 the arrays c_δ and p_δ are showed as several $2D$ grids, but in the GPU memory they are stored one $1D$ array each, by storing one grid after the other starting from left to right and from top to bottom, thus, the first grid corresponds to the time step t_i and the last one of the time step $t_{i+\delta}$. With this structure we can easily access, in a parallel way, to any element of the δ grids, by using the fact that the set of points of a cell w starts at position $p_\delta[w]$ of e_δ, and is stored in $c_\delta[w]$ consecutive positions of e_δ. In each e_δ position we store the two floats of the

corresponding location. Finally, $\widetilde{e_\delta}$ is an integer array of size $n\delta$ containing at position j the entity id of the $e_\delta[j]$ location.

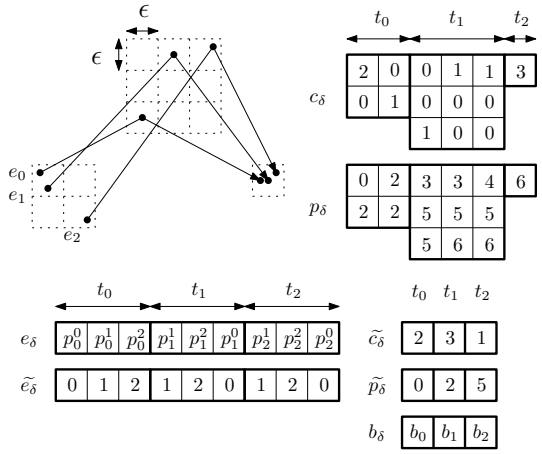

Figure 2: Example of multi-grid $\mathcal{G}$ for $\delta = 3$.

We construct the $\mathcal{G}$ structure in parallel as follows. We allocate CPU arrays $\widetilde{c_\delta}'$, $\widetilde{p_\delta}'$, b'_δ of size δ, and p' and $\widetilde{e_\delta}'$ of size $n\delta$. While we load the input P into p' the arrays $\widetilde{c_\delta}'$, $\widetilde{p_\delta}'$ and b'_δ are accordingly filled. Then $\widetilde{c_\delta}$, $\widetilde{p_\delta}$, b_δ and p are allocated in GPU memory and the $\widetilde{c_\delta}'$, $\widetilde{p_\delta}'$, b'_δ and p' values are transferred to GPU memory to $\widetilde{c_\delta}$, $\widetilde{p_\delta}$, b_δ and p respectively.

Then, c_δ and p_δ are allocated in the GPU memory according to $\widetilde{c_\delta}$ and $\widetilde{p_\delta}$ values. Additionally, the arrays e_δ and $\widetilde{e_\delta}$ are allocated with size $n\delta$ also in GPU memory. Because GPU has no dynamic memory we have to construct $\mathcal{G}$ in two steps.

First, we fill c_δ by counting the number of points contained in each cell. This is done in parallel by launching a kernel with $n\delta$ threads, that is, a thread per point and per time step. Each thread idx reads its $p[idx]$ value and determines its position in the grid by using its position on the plane, the grid corresponding to the time step which the point belongs to, and the $\widetilde{c_\delta}$ and $\widetilde{p_\delta}$ arrays. The corresponding c_δ position is incremented in 1. Because many threads may correspond to the same c_δ position we use atomic operations to ensure no thread interferences.

Once c_δ is computed, p_δ is obtained as the prefix sum of c_δ and we allocate an auxiliary array v with the same size of c_δ, initialized to zeros. Now, we launch a parallel kernel with $n\delta$ threads where each thread idx reads its $p[idx]$ value, determines its position w in the grid and stores $p[idx]$ at $e_\delta[p_\delta[w] + v[w]]$. Every time a thread stores its associated point into e_δ the corresponding v value is incremented in one by using an atomic operation.

4.2 Finding potential flocks

The computation of the array of potential flocks $\mathcal{F}_j^\delta$ of a given interval I_j^δ is done in two main steps. First, we need to find the center of the disks which are obtained applying the characterization given in Lemma 3, and second, we need to report the entities contained in those disks. Additionally, because GPUs do not have dynamic memory, we split the process in several steps.

First, we allocate an array of integers D_c of size $n\delta$ to count how many disks there exist. This is done in parallel by launching a kernel with $n\delta$ threads, that is, a thread per point within I_j^δ. Using the structure $\mathcal{G}$ each thread idx goes through its neighbors and counts the number of points which are at distance $\leq 2\epsilon$. The results are stored in $D_c[idx]$. Because this is computed in parallel, each disk will be reported twice, once for each of the two points defining the disk. In order to avoid this, we force threads to check those points whose entity id is bigger than itself. Then, D_p of size $n\delta$, is computed as the prefix sum of D_c, and the array D is allocated with size $D_p[n\delta-1]+D_c[n\delta-1]$, where we will store the centers of disks. The same process is repeated, but, instead of counting, each thread reports the center of the disk. Let us recall that only the right disk per paired point has to be reported.

In the second step, we actually report the potential flocks. To do this we use $\mathcal{G}$ to perform the range searching queries, using the values of the array D as the center of the queries with radius ϵ. Note that, because in order that the regular grid structure works for range searching queries, we must be able to locate any point within the grid. It may happen that some disk centers were placed outside the grid for a given time step. Thus, before we perform the range searching queries we have to reconstruct $\mathcal{G}$ so that the center of the disks were contained in $\mathcal{G}$. We could modify $\mathcal{G}$, but it is faster to rebuild it than to modify it.

To count the number of points contained on each disk we first allocate P_c, the array of integers of size equal to the total number of disks reported which is $D_p[n\delta-1]+D_c[n\delta-1]$, initialize it to zeros and launch a kernel with one thread per disk. Each thread idx reads the center of its disk $D[idx]$, locates $D[idx]$ on the grid and checks the distance between the center of the disk and the points of its cell neighbors. Each thread counts how many points are at distance $\leq \epsilon$ to its disk center, and if there are at least μ the number is stored in $P_c[idx]$. Finally, we compute P_p as the prefix sum of P_c.

If there is at least one disk per time step with at least μ entities, we continue to report the potential flocks. The last position of P_p plus the last position of P_c gives us ω, the size of the array used to store the potential flocks. Thus, the array P is allocated as an integer array with size ω and is filled in parallel using a thread per disk. The process is the same as before but this time, instead of counting, each thread stores in P the entities contained inside each disk.

Note that, P_c, P_p and P denote a structure contain-

ing δ families of sets. This is the array of potential flocks $\mathcal{F}_j^\delta$, that is, δ families of sets containing the potential flocks of δ consecutive time steps starting at time step t_j. We denote $\mathcal{F}_j$ the family of potential flocks of the time step t_j.

The last step is to reduce each $\mathcal{F}_i \in \mathcal{F}_j^\delta$ so that it only contains the sets which are maximal, with respect to the partial order induced by the subset relation in each family. To this end, we use the parallel algorithm of Fort et al. [7]. When the process finishes, we have in $\mathcal{F}_j^\delta$ the maximal potential flocks with μ or more entities at each time step of the interval I_j^δ.

4.3 Reporting $\mathcal{M}^\delta$ flocks

We want to report all the flocks M_j^δ for all $I_j^\delta, j = [0, \ldots, \tau - \delta]$. Thus, we could perform the same algorithm over each I_j^δ, but many information computed for a given time step within a time interval can be reused for the following time intervals. The idea is to compute the potential flocks for a given interval and compute their intersections in a way that we can reuse the information for the next time interval. To this aim we proceed as follows.

We start at the interval I_0^δ and we compute $\mathcal{F}_0^\delta$. Then, in order to obtain $\mathcal{M}_0^\delta = \mathcal{F}_{\delta-1} \cap \mathcal{F}_{\delta-2} \cap \cdots \cap \mathcal{F}_0$ we start at $\mathcal{F}_{\delta-1}$ and we intersect $F_{\delta-2} \cap \mathcal{F}_{\delta-1}$. Next, we intersect $\mathcal{F}_{\delta-3} \cap \mathcal{F}_{\delta-2} \cap \mathcal{F}_{\delta-1}$ and so on. After each intersection, we compute the maximal sets for each resulting intersection discarding those sets which are non maximal or containing less than μ entities. When we are done, we add, to $\mathcal{M}^\delta$, the family $\mathcal{M}_0^\delta$, which contains the flocks of I_0^δ.

For the following time steps, instead of recomputing all the potential flocks and all the intersections, we reuse the previous computations. If for a given time step t_j, a family of potential flocks $\mathcal{F}_k, j \leq k < j+\delta$ is not inside the array $\mathcal{F}_j^\delta$, we compute $\mathcal{F}_{j+\delta-1}^\delta$. That is, instead of computing the potential flocks of one single time step we compute them in δ intervals. Then, we perform the corresponding intersections. This technique does not decrease the number of intersection we have to do, but, in practice, the δ families we intersect in each step are smaller than the ones containing the potential flocks, leading to faster results. We repeat the process until $j = \tau - \delta$.

The intersection process is also performed in parallel using the GPU algorithm presented by Fort et al in. [8].

5 Experimental results

The experimental results are obtained using an Intel Core2 CPU 6400 with a Nvidia GTX 480. We split the running time into 2 main steps, the potential flocks reporting process and the intersection process. The accumulated value, corresponding to the columns height, gives the total running time of the algorithm.

The algorithm has been tested with a data set extracted from [10], that consists of 145 trajectories of 2 school buses around Athens metropolitan area in Greece with a total of 66,096 time steps.

The results (Figure 3) shows that when we vary ϵ while we maintain $\mu = 5$ and $\delta = 10$, the number of flocks reported increases as long as we increase ϵ. The most affected part is the intersection process. This is because the more flocks we report the more intersections we have to compute. When the parameter μ is varied, while $\epsilon = 1200$ and $\delta = 10$, the condition to form flock is more restrictive. Thus, the number of flocks decreases and, consequently, the running times too.

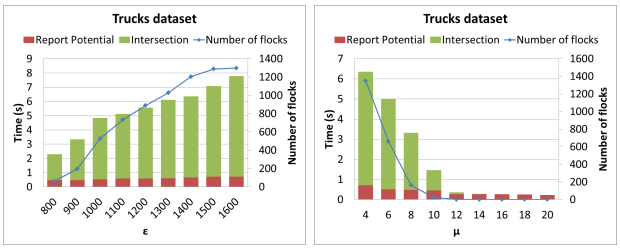

Figure 3: Running times varying ϵ and μ.

References

[1] J. Gudmundsson, M. J. van Kreveld, B. Speckmann, Efficient Detection of Patterns in 2D Trajectories of Moving Points, GeoInformatica 11 (2) (2007) 195–215.

[2] M. Benkert, J. Gudmundsson, F. Hübner, T. Wolle, Reporting flock patterns, Computational Geometry 41 (3) (2008) 111–125.

[3] M. R. Vieira, P. Bakalov, V. J. Tsotras, On-line discovery of flock patterns in spatio-temporal data, in: D. Agrawal, W. G. Aref, C.-T. Lu, M. F. Mokbel, P. Scheuermann, C. Shahabi, O. Wolfson (Eds.), GIS, ACM, 2009, pp. 286–295.

[4] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T. J. Purcell, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (1) (2007) 80–113.

[5] N. Coll, M. Fort, N. Madern, J. A. Sellarès, Multi-visibility maps of triangulated terrains, International Journal of Geographical Information Science 21 (10) (2007) 1115–1134.

[6] M. Fort, J. A. Sellarès, N. Valladres, Computing popular places using graphics processors, in: Proc. SSTDM'10 in cooperation with IEEE ICDM'10, IEEE Computer Society, 2010, pp. 233–241.

[7] M. Fort, J.A. Sellaès, N. Valladares, Finding extremal sets on the GPU (Submitted).

[8] M. Fort, J.A. Sellaès, N. Valladares, Intersecting two families of sets on the GPU (Submitted).

[9] J. Gudmundsson, M. van Kreveld, B. Speckmann, Efficient Detection of Motion Patterns in Spatio-Temporal Data Sets, in: D. Pfoser, I. F. Cruz, M. Ronthaler (Eds.), GIS, ACM, 2004, pp. 250–257.

[10] Y. Theodoridis, R-Tree portal, `http://www.rtreeportal.org` (2011).