A GPU-based Interactive Bio-inspired Visual Clustering

Ugo Erra
Dipartimento di Matematica e Informatica
Università della Basilicata
Potenza, Italy
Email: ugo.erra@unibas.it

Bernardino Frola
Dipartimento di Informatica ed Applicazioni
Università di Salerno
Fisciano, Italy
Email: frola@dia.unisa.it

Vittorio Scarano
Dipartimento di Informatica ed Applicazioni
Università di Salerno
Fisciano, Italy
Email: vitsca@dia.unisa.it

***Abstract*—In this work, we present an interactive visual clustering approach for the exploration and analysis of vast volumes of data. Our proposed approach is a bio-inspired collective behavioral model to be used in a 3D graphics environment. Our paper illustrates an extension of the behavioral model for clustering and a parallel implementation, using Compute Unified Device Architecture to exploit the computational power of Graphics Processor Units (GPUs). The advantage of our approach is that, as data enters the environment, the user is directly involved in the data mining process. Our experiments illustrate the effectiveness and efficiency provided by our approach when applied to a number of real and synthetic data sets.**

I. Introduction

Today, data-intensive science consists of the analysis of scientific data captured by instruments or generated by simulations and resulting in vast volumes of data with high dimensionality. For instance, CERNs Large Hadron Collider and astronomy's Pan-STARRS5 array of celestial telescopes are capable of generating several petabytes of data per day, while gene sequencing machines are capable of producing thousands or millions of genome sequences. The amount of data produced by today's technologies is so vast that in order to verify and validate underlying models, data must be visualized and analyzed. Both visualization and data analysis require new approaches that enable us to conduct high-performance processing and intuitive representation when dealing with large amounts of data.

Visual data mining allows us to gain insight into data-intensive science through new approaches to visual data analysis and knowledge discovery. The detection and validation of expected results is facilitated by interactive interfaces that improve the interpretation of data. Scientists can use visual data analysis systems to explore multiple scenarios and examine data using multiple perspectives and assumptions. A number of visualization techniques have developed to support data mining tasks that rely on such types of visualization as clustering, for example.

Clustering is essentially a data mining approach that addresses the problems of growing data and the scarcity of human attention by discovering groups of similar objects. Each group, called a "cluster", consists of objects that are similar to one another and dissimilar to objects of other groups. The meaning and measure of similarity is referred to as the "distance metric," a term used to define the closeness between two objects in a pair. Based on given similarities, data is organized into clusters using an unsupervised learning approach that starts with an unlabeled dataset, from which we want to discover how the objects within that set are organized [1]. Clustering is a common operation that, in addition to discovering groups in data mining applications, has numerous applications. For instance, it may assist in the study of social networks by recognizing communities, in image processing by recognizing objects, and in grouping genes by performing functions similar to the bioinformatics functions genes perform.

There are many clustering techniques . The most widely used are: Hierarchical clustering [2], k-means clustering [3], and self-organized maps [4]. It is impossible to say whether any of these techniques is better than the others because each algorithm has its advantages and disadvantages and performs differently for different problems. For instance, k-means is simple and has good time complexity, but is unstable due to its initial seeds assignment; what is more, the user must decide, a priori, the number of clusters. On the other hand, hierarchical clustering does not require that the number of clusters be determined a priori, but has a complexity that is at least quadratic, in terms of the number of inputs compared to the linear complexity of k-means. Moreover, these algorithms are useful for organizing static data but do not do well with the analysis of extremely large datasets that cannot be held, in their entirety, by a single system's memory. They also do not do well in situations where the time delay before new data is arranged into clusters is essential to refreshing the organization of data. With two- or three-dimensional data, the results of different algorithms can be easily explored using simple visualizations, but data with high dimensionality are much more difficult to visualize and understand. Several techniques try to project the data into two- or three-dimensional space, in order to show the properties of high-dimensional clusters [5].

Thanks to their good price/performance ratio and hardware

Video of this work is available only for this review at http://isis.dia.unisa.it/projects/behavert/cidm2011.wmv

978-1-4244-9927-4/11/$26.00 ©2011 IEEE

programmability, Graphics Processor Units (GPUs) are used today not only for 3D graphics rendering, but for general-purpose computing. In this paper, we present a clustering algorithm inspired by a flocking behavioral model that exploits the parallel architecture of GPUs. High-dimensional data are mapped as agents' features. Each agent is assigned a local behavioral model and moves by coordinating with the motion of other agents in a 3D environment. The effectiveness of our approach relies on the natural organization that arises when a group of agents interacts using a local behavioral model. It enables the organization into clusters of agents with similar features. Using NVIDIA's Compute Unified Device Architecture (CUDA) as our programming environment, we illustrate the model and an efficient implementation that exploits the parallel architecture of GPUs. The proposed clustering does not require the number of clusters as input, and data can be introduced interactively, on the fly. Generally, our approach enables high-performance processing in data analysis and visualization based on an intuitive representation that avoids the projection of high-dimensional data in two- or three-dimensional space. Experimental results show that the quality of clustering that results from using our algorithm is guaranteed, while GPU implementation performs merely well.

The remainder of this paper is organized as follows: In section II, we review previous clustering approaches that are based on GPUs. In section III, we describe the behavioral model that inspired our clustering approach. In section IV, we illustrate our clustering algorithm. In section V, we present its implementation on a GPU. In section VI, we present a brief description of the application. Section VII illustrates the efficiency and performance scalability of some experiments. Finally, section VIII concludes and discusses directions for future work.

II. Related Works

An example of a system that uses visualization techniques for high-dimensional clustering is OPTICS [6]. The authors of OPTICS created a one-dimensional ordering of databases, representing the density of clustering structures. Cluster points are close in the one-dimensional ordering generated by OPTICS, and their reachability is provided using a distance plot. This visualization system is valuable for understanding and guiding the clustering process. Another approach to high-dimensional clustering is the HD-Eye system [7]. HD-Eye considers clustering a partitioning problem and allows the user to be directly involved in the clustering process - that is, in choosing the dimensions to be considered, in selecting the clustering paradigms, and in partitioning the data sets.

With regards to clustering, GPUs have demonstrated interesting results. The k-means clustering is probably the most studied clustering algorithm on GPUs. The first implementations to show how GPUs can be used to significantly accelerate k-means are [8] [9]. Using an obsolete approach, based on shader languages, the authors of these studies exploited the computational capabilities of GPUs. Today, general purpose languages, like CUDA, offer better support to GPU architectures. The authors of [10] [11] tried to improve the efficiency of k-means using CUDA and optimizations directly targeted at parallel architectures. They obtained a speed that is 14 and 13 times greater, respectively, than that of a CPU's sequential computation.

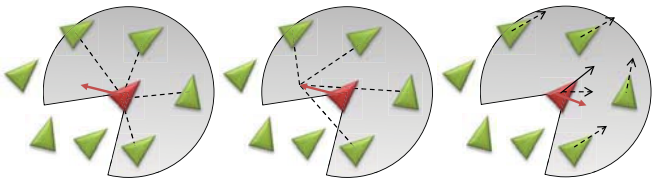

Fig. 1. From left to the right, the steering behaviors: separation, cohesion and alignment. In all cases, the agent's has a local perception of the other agents is limited by its field of view.

The authors of [12] used a shader language to implement hierarchical clustering. Their implementation speed was 2-4 times greater than that of a CPU. [13] explored parallel computation of hierarchical clustering with CUDA and obtains a speed that is 48 times greater.

In [14], authors analyze and group documents based on a flocking clustering model implemented on the GPU. In this implementation, each document is a bird and flies toward other documents that are similar to it. As the authors recognize, the limitation of this approach is its complexity $O(n^2)$. This inefficiency results from the fact that in each step of the simulation, the authors created a thread for each agent pair (n^2 thread in total), in order to compare their locations in 2D virtual space and then compute the distance between them.

Our approach tackles the problem of $O(n^2)$ complexity using a static grid that subdivides 3D space into cubic cells of the same size. This approach enable us to identify, in parallel, all agents within the same cell or within a given region, considering which agents within the cells overlap a region of interest. Experiments showed that this approach can speed up the implementation of a clustering algorithm, allowing for faster exploration of data sets that contain thousands of items.

III. The Behavioral Model

Our clustering approach is inspired by the original behavioral models proposed by Reynolds [15]. In Reynold's model, each agent has a strictly local perception of the space it occupies. None of the creatures that compose the group has full knowledge of the entire group. Hence, agents must base their decisions on what they know of the neighbors that their fields of view allow them to perceive. Based on each agent's visibility, the synchronized aggregated motion of the group is achieved by performing a weighted sum of *steering behaviors*. Reynolds defined three steering behaviors, illustrated in Figure 1. The first, *separation*, tends to keep distance from other neighbors. This is necessary to prevent the collision of agents. A repulsive force $\vec{f_s}$ is calculated as the difference vector between an agent's current position and the position of each of its neighbors, while the steering force

is calculated as the average vectors between all the repulsive forces. The *cohesion* moves the agent toward the center of his local neighborhood. This gives the flock an aggregated aspect. The cohesion force $\vec{f_c}$ is obtained by computing the average position of neighbors. The *alignment* tends to align the agent with other neighbors through group computing. The alignment force $\vec{f_a}$ is calculated as the difference between the average of the neighbors' forward vectors and the forward vector of the agent itself.

The overall steering force $\vec{f_r}$ of the Reynolds model, for the agent i, is achieved by adding the steering forces produced by behaviors

$$\vec{f_r} = w_s\vec{f_s} + w_c\vec{f_c} + w_a\vec{f_a}$$

where, w_s, w_c, and w_a are weights that manage the behavioral impact on the whole steering force.

According to the steering model proposed by Reynolds, the 3D movement of each agent i is defined using the following equations:

$$\vec{a}_i = \vec{a}_i + smoothRate(\vec{f_r} - \vec{a}_i)$$

$$\vec{v}_i = \vec{v}_i + \vec{a}_i$$

$$\vec{d}_i = \left(\vec{d}_i + \vec{a}_i\right) / \left|\left|\vec{d}_i + \vec{a}_i\right|\right|$$

$$\vec{p}_i = \vec{p}_i + \vec{v}_i$$

where, $\vec{a}_i$ is the agent's acceleration vector, $\vec{v}_i$ is the velocity vector, $\vec{p}_i$ is the position vector, and $\vec{d}_i$ is the direction unit vector. The parameter $smoothRate$ indicates how much steering force influences the agent's acceleration. Both the lengths of $\vec{v}_i$ and $\vec{f_r}$ are truncated to a maximum speed.

The model illustrated requires that within a large environment, neighbors be identified - neighbors being all agents that fall inside the field of view of an agent. This is fundamental because each agent must make decisions only according to its neighbors, and so it must be able to pick out these agents efficiently. A brute force approach requires $O(n^2)$ steps for a proximity screening, i.e., compares each agent to all others and gathers all agents within a given range. This approach is sufficient for a hundred agents, but it is clear that is computationally inefficient for generating interactive results for thousands of agents.

IV. The Clustering Model Approach

In addition to the model proposed by Reynolds, we defined two new behaviors called *Cluster-Cohesion* and *Cluster-Alignment*. These behaviors implement the agent-based clustering algorithm.

The cluster-cohesion force $\vec{f_{cc}}$, for a specific agent i, is computed as

$$\vec{f_{cc}} = \sum_{j \in Neighs(i)} sim_{ij}\vec{s_{ij}} + (1 - sim_{ij})\vec{f_{ij}}$$

where, $Neighs(i)$ are the nearest neighbors of the agent i. The vector $\vec{s_{ij}} = (\vec{p_j} - \vec{p_i}) - \vec{v}_i$ is the seeking force between agents i and j, while $\vec{f_{ij}} = -\vec{s_{ij}}$ is the fleeing force. The function sim_{ij} computes a similarity factor between the features vectors associated with agents i and j and must be between 0 and 1. The cluster-alignment force $\vec{f_{ca}}$, for a specific agent i, is computed as

$$\vec{f_{ca}} = \sum_{j \in Neighs(i)} sim_{ij}\vec{d_j}$$

The steering force $\vec{f}$ used in the flocking clustering algorithm is achieved by adding Reyndol's steering forces and these new forces

$$\vec{f} = \vec{f_r} + w_{cc}\vec{f_{cc}} + w_{ca}\vec{f_{ca}}$$

Also, in this case, we use two weights w_{cc}, and w_{ca} to manage the impact of the clustering algorithm's behaviors.

A. Similarity

In our model, each agent represents an object of the data set, while the features vector associated with each object defines an agent's character. The agents move in the 3D environment within which the most similar agents will be found and grouped. The overall effect is that when an agent finds an agent similar to itself, it stays near this agent but continues to explore the 3D environment, looking for groups of agents that are similar to its group. The purpose of using the 3D environment as search space is twofold. First, the 3D environment enables clustering of high-dimensional data sets without loss of features. Second, the clustering process is visualized in an intuitive and natural fashion, without regards to data set dimensionality.

The similarity between two agents is computed using the values of their associated features. The implementation described in this paper recognizes the angular separation between agent i and j as

$$sim_{ij} = \frac{\vec{c}_i \cdot \vec{c}_j}{\sqrt{\|\vec{c}_i\| \, \|\vec{c}_j\|}}$$

where, $\vec{c}$ is the agent's features vector. The resulting factor is between -1 and 1 and must be between 0 and 1. To achieve this value, the similarity is recalculated as

$$sim_{ij} = (sim_{ij} + 1)/2$$

This similarity factor yields poor results when features vectors are not normalized, which is possible taking into account the mean value and variance of all the features vectors. This kind of normalization is unfeasible when data represent continuous streams. For this reason, we adopted a dynamic adjustment of the similarity value, using the similarities between statistics for agents' neighbors.

At each step of our simulation, each agent collects information about the minimum value (s_{min}), maximum value (s_{max}) and average value (s_{avg}) of its neighbors' similarities. The adaptive similarity $asim$ is then computed as follows:

$$asim_{ij} = \begin{cases} lerp(sim_{ij}, s_{min}, 0.0, s_{avg}, 0.5) & \text{if } sim_{ij} \leq s_{avg} \\ lerp(sim_{ij}, s_{avg}, 0.5, s_{max}, 1.0) & \text{else} \end{cases}$$

where, $lerp(val, x_a, y_a, x_b, y_b)$ gets the linear interpolation of val on the line whose vertexes are (x_a, y_a) and (x_b, y_b). Figure 2 shows the relationship between sim and $asim$.

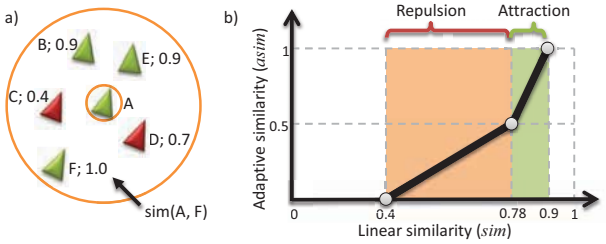

Fig. 2. a) A group of 6 agents (little triangles). Step I: Agent A computes the linear similarity (sim) of each neighbor. It also computes the minimum (0.4), maximum (0.9) and average (0.78) sim among them. Step II: Agent A computes the value of $asim$ for each of its neighbors, taking into account the minimum, maximum and average sim computed during the previous step. b) Adaptive similarity vs. linear similarity. Agent A is repulsed by agents with $asim < 0.5$ (agents C and D) and attracted by those with $asim > 0.5$ (agents B, E and F).

B. Cluster Identification

We implemented a simple local label propagation algorithm for cluster identification. The algorithm consists of two steps:

I. Assign a unique label to each agent.

II. Each agent looks to each of its neighbors, in turn. If its neighbor's label is smaller than its own label, then it replaces its label with that of its neighbor. Repeat this step *LPIterations* times.

The value of *LPIterations* can be set at run-time using one of the slides of the user interface. Figure 3 shows an example of local label propagation.

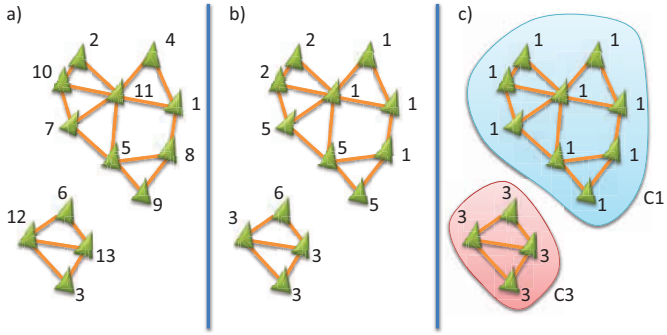

Fig. 3. Example of local label propagation for cluster identification (with *LPIterations* equal to 2). a) Step I: Assign a unique label to each agent. b) Step II, iteration 1: Propagate minimum values using neighborhood connections. c) Step II, iteration 2: Again, propagate minimum values. Agents with the same labels represent clusters.

V. Implementation

Recently, GPUs have evolved to program general-purpose computations using such programming models as CUDA [16]. CUDA is a minimal extension to C language and permits the writing of a serial program called *kernel*. A kernel executes in parallel across a set of parallel threads. Each thread has a private local memory. The programmer organizes these threads into a hierarchy of thread blocks and grids. A thread block is a set of concurrent threads that can cooperate amongst themselves, through barrier synchronization, and that have access to the shared memory, with latency comparable to registers. The grid is a set of thread blocks, each of which may be executed independently. All threads have access to the same global, constant or texture memory. These three memory spaces are optimized for different memory usages and, thus, have different time access. For example, the read-only constant cache and texture cache are shared by all scalar processor cores, and this speeds up reads from the texture memory space and constant memory space.

Grid and block sizes must be defined for every kernel invocation. Each block is mapped to one multiprocessor, and then multiple thread blocks can be mapped onto the same multiprocessor and executed concurrently. Multiprocessor resources (registers and shared memory) are split amongst the mapped thread block. This limits the number of thread blocks that can be mapped onto the same multiprocessor. In order to maximize the number of threads supported by a multiprocessor, the resources required by each kernel must be taken into account.

In order to avoid the $O(n^2)$ complexity of the search for neighbors, we adopt a strategy based on the assumption that the interaction of steering behaviors drops off with distance [17]. Then, we are interested only in efficiently computing a limited number of neighbors. This biologically-based assumption alleviates the computational effort required by the search for neighbors, as well as the difficulty of managing dynamic data structures, which is not trivial when it comes to GPU implementation.

Then, to accomplish this task, a static grid subdivides the 3D environment into cubic cells of the same size. For each agent, we assign a hash value based on its cell. Based on the hash values, a radix sort is performed on the GPU. At the end of this step, groups of agents belonging to the same cell will be located in continuous regions of the GPU's memory. This process quickly results in a list of neighbors. A range query is performed, with each agent looking for neighbors in its own and adjacent cells. A simple linear search that starts from a proper index and is based on cell hash function is then sufficient. Further information, including implementation details, performance and scalability evaluations of this approach are discussed in more detail in our previous works [18] [19].

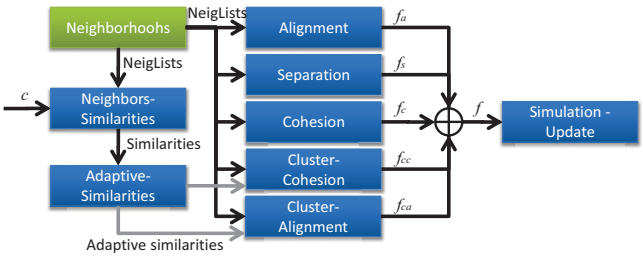

Fig. 4. Implementation schema. Rectangles are kernels, and arrows are data streams. *Neighborhoods* represents a set of kernels.

Figure 4 shows the main kernels and data streams used to implement the proposed clustering method. "Neighborhoods" is composed of several kernels and generates neighbors lists

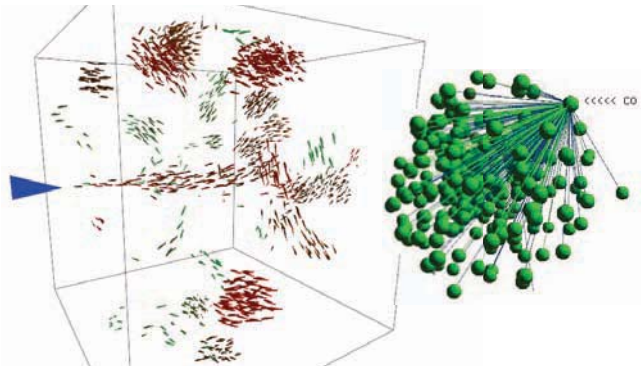

Fig. 5. Left: Agents introduced on the fly in any place in a 3D environment. Agents are rendered with a 3D model of fish. Right: 3D representation of a cluster. Agents are rendered with billboards. Each agent is connected to the agent with the lowest label value amongst the agents that belong to the same cluster.

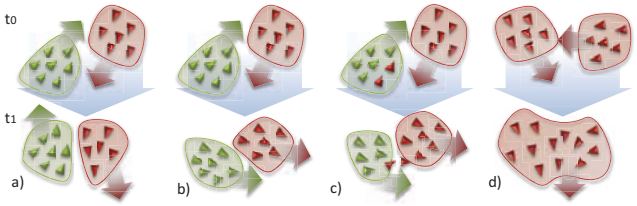

Fig. 6. Some common situations occurred during our experimentation: Impact between flocks representing well-defined clusters (a) and similar clusters (b). Individual exchanges between flocks (c). Flocks mixing (d).

as described above. Given the neighbors list of each agent, the kernel `NeighborhsSimilarities` generates a similarity value for each agent's neighbor. It, therefore, launches one thread for each agent's neighbor. Let *maxNeighs* be the maximum number of neighbors for each agent. One agent is associated with the *maxNeighs* thread, with each agent sharing its features vector and neighbors list. These data are read in parallel and put in the shared memory. Other kernels - `Separation`, `Cohesion`, and `Alignment` - compute the three steering forces of Reynolds' flocking behavior, while kernels `ClusterCohesion` and `ClusterAlignement` turn similarities and neighbors lists into steering forces, respectively, $\vec{f_{cc}}$ and $\vec{f_{ca}}$. These kernels launch a thread for each agent. The steering forces yielded by each behavior are blended in a common accumulator ($\vec{f}$), taking into account the force weights associated with each behavior. The `SimulationUpdate` kernel applies the resulting steering force to the 3D motion of agents, as described by equations in Sect. IV. Each thread computes and stores similarities and similarities statistics (useful for the adaptive similarity method, described in Sect. IV-A)) of only one neighbor agent.

VI. The Application

Efficient implementation of the proposed model enables the introduction of agents into the 3D environment in any place, on the fly, using a sort of agents fountain that eliminates the need to restart the algorithm when new data are available (Fig. 5 left). The objective is to maintain a consistently good clustering of the sequences observed so far, in a small amount of time. After the data stream of agents has entered the environment, it immediately seeks similar clusters, in a natural fashion. This feature is implemented by pre-allocating buffering spaces in the GPU's memory and using these buffers whenever new data is available.

During the simulation, agents belonging to the same cluster may move together and form flocks. These flocks explore the 3D environment, looking for similar groups to join up with. When flocks representing well-defined clusters collide, they bounce off of each other and follow different paths (Fig. 6a). Flocks representing similar clusters move close but do not mix. Some agents act as cluster bridges, moving between two flocks (Fig. 6b. These agents change flock membership, depending on whether the cluster of one flock matches its own features vector better than the cluster of its current flock (Fig. 6c). Flocks representing the same cluster (according to the metric used as the similarity function) merge in a big flock (Fig. 6d). In our experiments, we observed that 2000 iterations were sufficient to reach a stable state, even for thousands of agents.

Several parameters influence the formation of clusters. Aside from the weights of the model illustrated in Sect. IV, we have $worldRadius$, which is the size of the world; $searchRadius$, which is the size of the range query; $separationRadius$, which represents the distance of separation between agents; and $maxNeighbors$, which is the maximum number of neighbors an agent can have.

The visual interface (Fig 7) supports the user in the classification and verification process of output clusters, using the Visual Information Seeking Mantra "Overview first, zoom and filter, then details-on-demand" [20], as described below:

- *Overview first.* During the creation of clusters, the application supports the visualization of the flocking approach. The overview provides the user with a visual summary of clustering results and allows a first evaluation of the number of clusters and relations between clusters. As described in Sect. IV-B, each agent is connected to the agent with the lowest unique index in its cluster. The name of the cluster is the label of the lowest unique index in that cluster. During clustering, the user can modify the simulation parameters during run time, using several sliders and changing point of view in the 3D environment, so as to explore one or more clusters from multiple angles.
- *Zoom and filter.* Because our approach can also handle vast volumes of data, the visual interface allows the user to zoom in, from the initial overview, and filter information, refining the current view. If the user identifies clusters of interest in the overview, these clusters can be selected individually or removed from the clustering process.
- *Details-on-demand.* The user can select one or more agents and show their properties (position, class membership, etc.). Each input data is labeled with actual

class membership, and the application shows detailed information about clusters, using the confusion matrix. Each matrix row represents the instances in a predicted class, while each column represents the instances in an actual class.

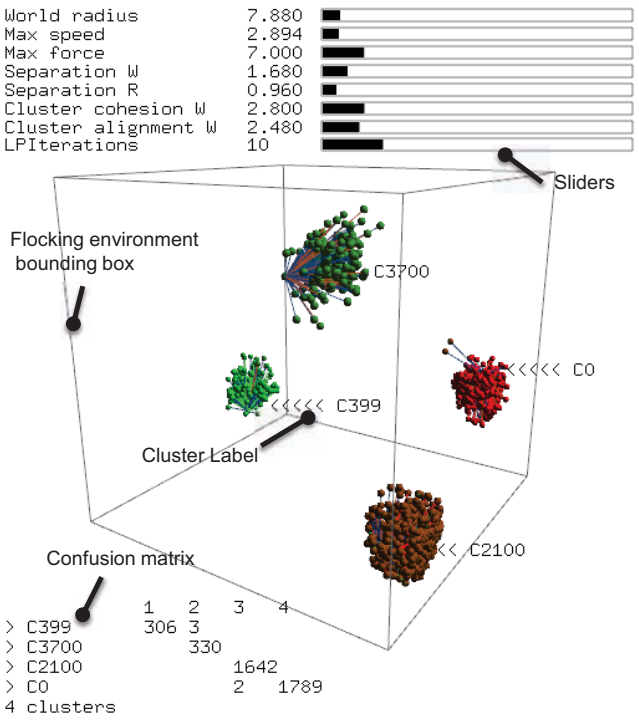

Fig. 7. The software Graphical User Interface (GUI).

VII. Experimental Results

In this section, we show the results of two types of experiments. The first experiment demonstrates the quality of our approach. The second is related to the efficiency of GPU, versus CPU, implementation. All tests were performed on an AMD Athlon 2800+ CPU, 2GB RAM and a NVIDIA GTX 470 1280Mb RAM (CUDA compute capability 2.0). Software configuration: CUDA SDK v3.1, Windows 7. The rendering of clusters was done with OpenGL [21].

A. Quality

For the quality tests, we selected six of the most popular datasets from UC Irvine's Machine Learning Repository [22]. The selected datasets are Iris, Wine, Yeast, Breast Cancer Wisconsin, Abalone and SPECT Heart. Below is a brief description of these datasets:

- The Iris dataset contains information about Iris flowers. There are three classes of Iris flowers - Iris Setosa, Iris Versicolor and Iris Virginica. The Iris dataset consists of 150 examples of Irises that are classified according to 4 attributes. One class is clearly separable from the others, which overlap in a lot of respects.
- The Wine dataset is the result of a chemical analysis of wines grown in a region of Italy but derived from three different cultivars. There are three classes of wines. The dataset consists of 178 examples of wines, each with 13 continuous attributes. It contains 59 examples of class 1, 71 examples of class 2 and 48 examples of class 3.
- The yeast data set contains 1484 records. The cellular localization sites of proteins are to be determined. There are ten classes.
- The Breast Cancer Wisconsin (B.C.W.) dataset has 699 records of benign and malignant breast cancer tumors. The goal of this dataset is to explain the difference between the two diagnoses.
- The Abalone dataset has a total of 4177 records. Each record represent an abalone instance. The goal of this dataset is to decide the number of rings using various measurements. The number of rings ranges from 1 to 29. A 3-class classification problem is defined for the Abalone dataset, analogously to the Iris dataset, except that here, class 1 has records with 1-8 rings, class 2 has records with 9 or 10 rings, and class 3 has records with 11-29 rings.
- The SPECT Heart dataset has 267 records. In contrast to the other datasets, all of its attributes are binary. The goal of this dataset is to report diagnosis using 0 and 1.

In addition, we used synthetic datasets generated by a Gaussian cluster generator proposed in [23]. We used three synthetic datasets, each containing 4000 records. The first has 10 classes (Synth. 10C), the second has 20 classes (Synth. 20C) and the third 40 classes (Synth. 40C). For each test, we split the given data set 50/50 into training and testing data.

The parameters used for the quality tests are set to $w_a = 0$, $w_c = 0$, $searchRadius = 4$, $separationRadius = 1.5$, $maxNeighbors = 32$. We used training data to empirically find the values of w_s, w_{cc}, and w_{ca} (showed in Table I). The value of $worldRadius$ is computed such that agent density in the 3D environment is always 0.05 world units per agent (in order to ensure a good level of interaction among agents).

For each dataset, we evaluated the correctness of classification results using $precision$ (P) and $recall$ (R). We also used $F - measure$ (F), the harmonic mean of precision and recall. These measures are defined as:

$$P = \frac{tp}{tp + fp} \qquad R = \frac{tp}{tp + fn} \qquad F = 2 \cdot \frac{PR}{P + R}$$

where, tp is the number of true positive patterns, fp the number of false positive patterns, and fn the number of false negative patterns.

Table II shows the averages and standard deviations of the results of the proposed clustering algorithm of 500 iterations, after the simulation reached a stable state (2000 iterations). We compared our results with k-means clustering [3] and hierarchical clustering (single-linkage) [2]. P_{KM}, R_{KM}, and F_{KM} are, respectively, precision, recall, and F-measure of the k-means clustering results. P_{HC}, R_{HC}, and F_{HC} are, respectively, precision, recall, and F-measure of the hierarchical clustering results. We executed the k-means clustering

TABLE I
VALUES OF PARAMETERS

	Iris	Wine	Yeast	B. C. W.	Abalone	SPECT H.	Synth. 10C	Synth. 20C	Synth. 40C
w_s	2.0	3.0	2.0	1.0	2.0	0.5	0.5	0.5	0.5
w_{cc}	3.0	4.0	4.8	2.0	4.0	1.0	3.0	3.0	3.0
w_{ca}	2.0	4.0	3.0	4.0	3.8	6.0	2.5	2.5	2.5

TABLE II
RESULTS OF QUALITY TESTS

	Iris	Wine	Yeast	B. C. W.	Abalone	SPECT H.	Synth. 10C	Synth. 20C	Synth. 40C
P	0.98 ± 0.00	0.77 ± 0.00	0.33 ± 0.01	0.86 ± 0.00	0.54 ± 0.00	0.67 ± 0.00	0.91 ± 0.01	0.92 ± 0.01	0.79 ± 0.03
P_{KM}	0.83 ± 0.15	0.75 ± 0.01	0.33 ± 0.04	0.96 ± 0.00	0.51 ± 0.00	0.56 ± 0.02	0.85 ± 0.06	0.79 ± 0.06	0.77 ± 0.05
P_{HC}	0.84	0.47	0.27	0.83	0.28	0.46	0.74	0.74	0.73
R	0.97 ± 0.00	0.72 ± 0.00	0.33 ± 0.01	0.87 ± 0.00	0.50 ± 0.00	0.70 ± 0.00	0.90 ± 0.00	0.91 ± 0.01	0.77 ± 0.05
R_{KM}	0.85 ± 0.12	0.72 ± 0.00	0.34 ± 0.04	0.95 ± 0.00	0.52 ± 0.00	0.69 ± 0.03	0.87 ± 0.07	0.78 ± 0.06	0.75 ± 0.05
R_{HC}	0.68	0.36	0.22	0.50	0.41	0.48	0.71	0.70	0.72
F	0.97 ± 0.00	0.71 ± 0.00	0.29 ± 0.01	0.86 ± 0.00	0.51 ± 0.00	0.68 ± 0.00	0.90 ± 0.00	0.92 ± 0.01	0.76 ± 0.03
F_{KM}	0.84 ± 0.13	0.70 ± 0.01	0.30 ± 0.03	0.95 ± 0.00	0.51 ± 0.01	0.43 ± 0.02	0.85 ± 0.06	0.77 ± 0.07	0.75 ± 0.05
F_{HC}	0.58	0.25	0.20	0.40	0.33	0.48	0.68	0.66	0.73

algorithm 500 times for each dataset. With all datasets, we achieved results that were superior to the results we achieved with hierarchical clustering. With Iris, Wine, and SPECT Heart, we achieved better results than with k-means, while with Yeast and Abalone, we achieved similar results and, with Breast Cancer Wisconsin, slightly worse results. Tests with synthetic data show that datasets with high numbers of classes are properly classified.

B. Performances

For performance tests, we used datasets with different number of instances, features and classes, generated by Gaussian-based synthetic datasets [23]. The parameters are set to $w_s = 0.8$, $w_a = 0$, $w_c = 0$, $w_{cc} = 3.0$, $w_{ca} = 2.5$, $searchRadius = 4$, $separationRadius = 1.5$, $maxNeighbors = 32$, and $worldRadius = 0.05$.

For performance evaluation, we developed a serial version executed on a CPU Opteron 252 2.6Ghz equipped with 2GB RAM and based on the OpenSteer steering library [24]. Tests were executed by comparing how many milliseconds it takes for GPUs and CPUs to implement each algorithm iteration. We also compared the results of our GPU implementation with those of Matlab's k-means serial implementation, in order to give an idea of the results of a classical clustering approach. We executed the k-means clustering algorithm 500 times with each configuration and took the average elapsed time of a single execution.

For the GPU, the CPU, and Matlab's k-means, Figure 8 compares the results obtained with different numbers of instances [NOTE: Please specify instances of what]. With 1000 instances, CPU implementation is more efficient than GPU implementation (due to the data-reordering overhead, as described in [18])), though the latter scales better than the former. We achieved a speed of up to 30 with a dataset of 65000 agent instances (or size of dataset). Figure 8 also shows that CPU implementation can run up to 2000 instances at interactive frame rates, while GPU implementation can run up to 32000 instances at interactive frame rates.

Figure 9 (left) compares the results of k-means with the

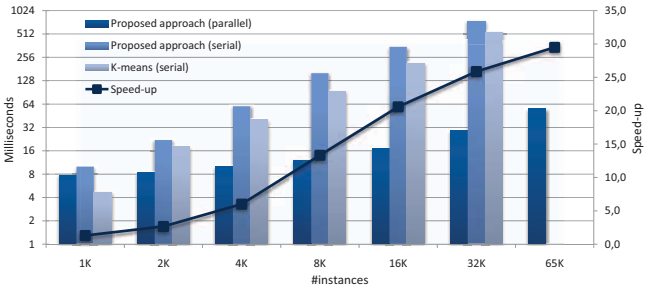

Fig. 8. The speed-up is between our CPU implementation (proposed approach - serial) and our GPU implementation (proposed approach - parallel). GPU implementation scales better than CPU implementation. GPU implementation is affected by an overhead that dominates in overall performance, with a low number of instances (up to 1000).

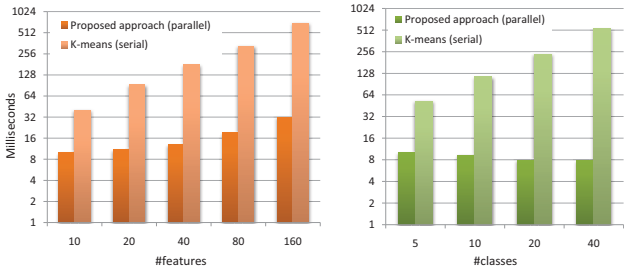

Fig. 9. Left: The GPU implementation (proposed approach - parallel) scales worse than a classical clustering algorithm. This is due to the parallelization scheme we chose. Scalability on #instances is advantaged, compared to scalability on #features. Right: The performance of our GPU implementation does not decrease with a high number of classes.

results of the proposed GPU implementation. The performance of the proposed approach scales slightly worse than k-means. This is due to the `NeighborhsSimilarities` kernel that computes similarities amongst agents. This kernel launches a thread for each agent's neighbor, ensuring good performance, with a high number of instances and a small number of features (up to 40). In future works, a new version of this kernel will address the problem of poor performance. A good solution to this problem is to launch a thread for each feature of each

agent.

Figure 9 (right) illustrates an interesting point. The computation time of the classical k-means increases in proportion to the number of classes. The computation time of GPU implementation decreases (indeed, it slightly decreases) because a high number of classes leads to high fragmentation of agents in the 3D environment (one flock for each class) leading to a decrease in the average size of agents' neighbors lists. Thus, when the number of classes is high, the neighbors searching phase is slightly more efficient.

VIII. Conclusions and Future Works

We proposed a bio-inspired clustering model and presented an efficient implementation that exploits the parallel architecture of GPUs. Each features vector is represented by an agent that follows simple local rules while moving in a 3D environment. By following such simple rules, agents exhibit complex global behavior, while agents similar to each other gradually merge together to form a cluster. Also, we proposed an efficient implementation for GPUs, based on a static grid that tackles the problem of identifying neighbors. This implementation is a key aspect to obtain an interactive visualization-based result on GPUs, enabling incoming data to cluster without taking into account all of the data processed. Our approach is able to diagnose changes in evolving input data. It can also distinguish data that is introduced into a 3D environment and must join old clusters or form new clusters. One advantage of our approach is that it does not require a priori knowledge of the number of clusters. Nor does it require a priori knowledge of the amount of data that will cluster. As the input data stream evolves during computation, the number of natural clusters will change. This enables the user to interactively introduce data streams into a user-defined 3D space, in a way that is similar to the 'in vitro' procedure used in biology. In addition, we implemented a local label propagation approach to automatically identify clusters. The detection and validation of our results was facilitated by the use of a visualization technique that relies on interactive interfaces to improve data interpretation. Experimental results show that our approach can improve the quality and performance of clustering.

In the future, we would like to explore several improvements to our approach. We would like to: (*i*) Perform further experiments to validate the effectiveness of clustering. (*ii*) Automatically identify the best parameters setting. (*iii*) Evaluate other similarity metrics that perform better as the number of features increases. (*iv*) Add a new feature, called "assisted clustering", that enables users to select, at runtime, two or more clusters and try to merge them.

References

[1] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Comput. Surv.*, vol. 31, no. 3, pp. 264–323, 1999.

[2] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, ser. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[3] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, L. M. L. Cam and J. Neyman, Eds., vol. 1. University of California Press, 1967, pp. 281–297.

[4] T. Kohonen, M. R. Schroeder, and T. S. Huang, Eds., *Self-Organizing Maps*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001.

[5] L. Yang, "Interactive exploration of very large relational datasets through 3D dynamic projections," in *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. New York, NY, USA: ACM, 2000, pp. 236–243.

[6] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "OPTICS - Ordering points to identify the clustering structure," *SIGMOD Rec.*, vol. 28, no. 2, pp. 49–60, 1999.

[7] A. Hinneburg, D. A. Keim, and M. Wawryniuk, "HD-Eye - Visual clustering of high dimensional data: A demonstration," *Data Engineering, International Conference on*, vol. 0, p. 753, 2003.

[8] J. D. Hall and J. C. Hart, "GPU acceleration of iterative clustering," in *ACM Workshop on General Purpose Computing on Graphics Processors*, August 2004.

[9] S. A. Shalom, M. Dash, and M. Tue, "Efficient k-means clustering using accelerated graphics processors," in *DaWaK '08: Proceedings of the 10th international conference on Data Warehousing and Knowledge Discovery*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 166–175.

[10] M. Zechner and M. Granitzer, "Accelerating k-means on the graphics processor via CUDA," *Intensive Applications and Services, International Conference on*, vol. 0, pp. 7–15, 2009.

[11] R. Farivar, D. Rebolledo, E. Chan, and R. H. Campbell, "A parallel implementation of k-means clustering on GPUs," in *PDPTA*, 2008, pp. 340–345.

[12] Q. Zhang and Y. Zhang, "Hierarchical clustering of gene expression profiles with graphics hardware acceleration," *Pattern Recogn. Lett.*, vol. 27, no. 6, pp. 676–681, 2006.

[13] D.-J. Chang, M. M. Kantardzic, and M. Ouyang, "Hierarchical clustering with CUDA/GPU." in *ISCA PDCCS*, J. H. Graham and A. Skjellum, Eds. ISCA, 2009, pp. 7–12. [Online]. Available: http://dblp.uni-trier.de/db/conf/ISCApdcs/pdccs2009.html#ChangKO09

[14] J. S. Charles, T. E. Potok, R. M. Patton, and X. Cui, "Flocking-based document clustering on the graphics processing unit," in *NICSO*, 2007, pp. 27–37.

[15] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1987, pp. 25–34.

[16] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[17] I. D. Couzin and J. Krause, "Self-organization and collective behavior in vertebrates," *Advances in the Study of Behavior*, vol. Volume 32, pp. 1–75, 2003.

[18] U. Erra, B. Frola, V. Scarano, and I. Couzin, "An efficient GPU implementation for large scale individual-based simulation of collective behavior," *High Performance Computational Systems Biology, International Workshop on*, vol. 0, pp. 51–58, 2009.

[19] U. Erra, B. Frola, and V. Scarano, "BehaveRT: A GPU-based library for autonomous characters," in *Motion in Games*, ser. Lecture Notes in Computer Science, R. Boulic, Y. Chrysanthou, and T. Komura, Eds. Springer Berlin Heidelberg, vol. 6459, pp. 194–205.

[20] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in *Proceedings of the 1996 IEEE Symposium on Visual Languages*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 336–. [Online]. Available: http://portal.acm.org/citation.cfm?id=832277.834354

[21] OpenGL ARB, D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.

[22] http://archive.ics.uci.edu/ml/datasets.html.

[23] http://dbkgroup.org/handl/generators/.

[24] C. Reynolds, "OpenSteer - steering behaviors for autonomous characters," 2004, http://opensteer.sourceforge.net/.