Bipartite Graph Matching Computation on GPU

Cristina Nader Vasconcelos[1] and Bodo Rosenhahn[2]

[1] PUC-Rio
crisnv@inf.puc-rio.br
[2] Leibniz Universitaet Hannover
rosenhahn@tnt.uni-hannover.de

Abstract. The *Bipartite Graph Matching Problem* is a well studied topic in Graph Theory. Such matching relates pairs of nodes from two distinct sets by selecting a subset of the graph edges connecting them. Each edge selected has no common node as its end points to any other edge within the subset. When the considered graph has huge sets of nodes and edges the sequential approaches are impractical, specially for applications demanding fast results. In this paper we investigate how to compute such matching on Graphics Processing Units (GPUs) motivated by its increasing processing power made available with decreasing costs. We present a new data-parallel approach for computing bipartite graph matching that is efficiently computed on today's graphics hardware and apply it to solve the correspondence between 3D samples taken over a time interval.

1 Introduction

Graph Matching is one of the fundamental problems in Graph Theory, with a intrinsic combinatorial nature. It can be defined as: given a graph G, its set of edges E and its set of nodes V, a matching M is a set of edges, subset of E, such that no two edges in M are incident to the same node.

Interesting problems in Computer Vision can be formulated as a Graph Matching, specially when an objective function associates weights to the graph edges, semantically related to some benefit or cost of the application. In that case, the weighted graph matching optimization goal is to maximize (or minimize) the sum of the weights of the matched edges.

There exist some parallel algorithms for approximating a graph matching [1,2,3,4]. Usually, the graph is initially distributed over several processors of a parallel computer or a set of computers organized as clusters or distributed systems.

In this paper we are interested in a variant of the general matching proposition, where the considered graph is a bipartite graph. A matching in a bipartite graph is easier to compute than in a general (or non-bipartite) graphs, as the number of possible combinations decreases considerably with the bipartite restriction and that its result can be obtained in a non-approximation way. Such version is usually named *The Assignment Problem* as it can be semantically proposed as: given a set of employees, a set of jobs and some cost (or benefit) function that evaluates the employee-job assignment, it is required to designate the people to accomplish the tasks by assigning exactly one employee to each job in such a way that the total cost of the assignment is minimized. Unbalanced versions of the same problem consider that the number of employees and the number of jobs are not equal. In that case, the one in greater number will have some elements unmatched at the final association.

The need for a parallel formulation to *The Assignment Problem* arises from cases considering huge graphs and from applications demanding fast results. Attending to such demand, we propose a GPU friendly formulation motivated by the modern graphics hardware increasing processing power made available with low costs (specially when compared with other high processing power solutions like distributed systems). This paper presents a parallel algorithm developed using the stream processing paradigm. Our algorithm identifies elements that can be arranged within a stream of data and processed independently. It attends to GPU implementation requirements and scales in a transparent way as the number of processors increases.

As an application to the GPU implementation developed, we propose a new formulation for a Computer Vision classical problem: the correspondence problem, here defined of over independent sets of 3D samples taken over a period of time. The quality of the obtained matchings was tested using microscopy data, taken during different time instants, against ground truth results manually obtained. The time efficiency of our solution was tested comparing it against two sequential implementations on CPU and also observing its answer time with a growing set of artificially generated 3D moving points.

The source code containing an implementation of the bipartite graph matching optimization proposed here, coded using CUDA programming language, is available for download from the author's homepage [5].

This paper is organized as follows. The next section presents a formal definition of the matching problem (section 2) and existing algorithms for computing it are presented in section 3. A brief comparison between CPU and GPU computing is presented in section 4. Our stream processing approach is presented in section 5, while results and conclusions are presented in sections 6 and 7.

2 Bipartite Graph Matching Definitions

According to the formal definition, a graph $G = (V, E)$ is bipartite if there exists a partition of its vertexes (or nodes) into two distinct sets, X and Y, such as that three properties are valid: the original set of vertexes V is formed by the union of the generated sets ($V = X \cup Y$); each vertex of the original set of vertexes V belongs exclusively to one of the created sets and not to both of them ($X \cap Y = \emptyset$); and, all the edges of the graph connect a vertex from one of the vertexes sets to the other ($E \subseteq X \times Y$). A Matching is a subset of the edges set ($M \subseteq E$) such that for every $v \in V$ at most one edge in M is incident to v. A vertex v is said to be matched in M if it is an endpoint of an edge in M, otherwise v is said free.

In this paper we are specially interested in weighted bipartite graphs, in which each edge (i, j) is associated to a weight $w(i, j)$. The weight of a matching M is defined as the sum of the weights of edges in M:

$$w(M) = \sum_{e \in M} w(e). \tag{1}$$

There are variants of the weighted bipartite graph matching formulation, including: $w(M)$ maximization or minimization (can be viewed as a maximization problem by just replacing the cost function c with $-c$), perfect matchings and maximum matchings [6].

3 Related Works and Background

In this section we present two approaches for computing a bipartite graph optimal matching in order to evaluate them for a GPU approach. The first one, known as *The Hungarian Algorithm*, is the classical approach from Graphs Theory literature and it is described in Subsection 3.1. In subsection 3.2 we describe a second approach, known as *The Auction Algorithm*, motivated by its distributed formulation.

3.1 The Hungarian Algorithm

The Hungarian Algorithm is a sequential combinatorial optimization algorithm published by Harold Kuhn [7] that solves *The Bipartite Graph Matching Problem* in a polynomial time. It iterates between two phases: the first one is based on the Graph Theory concept of augmenting paths while the second is based on the concept of feasible labellings (dual variable) and equality graphs.

A path in a graph is a sequence of vertexes such that from each of its vertexes there is an edge to the next vertex in the sequence. Both the first and the last vertexes of the sequence are called end or terminal vertexes of the path. Given a matching M and the set of edges E of the graph, a path is alternating if its edges alternate between M and $E - M$ and an alternating path is augmenting if both endpoints are free (see Figure 1).

The property that assures the first phase of *The Hungarian Algorithm* is that an augmenting path has one less edge in M than in $E - M$, thus, it is possible

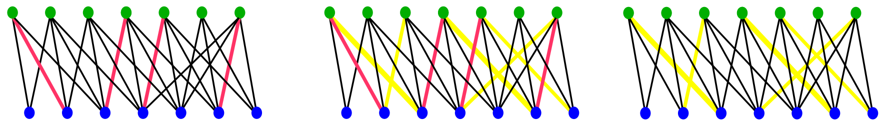


Fig. 1. A matching M represented with red edges (left); an augmenting path formed by the red (edges from M) and yellow edges (edges from the $E - M$ set) (center); the new path with size incremented by one (right)

to increment size of the matching by replacing the M edges by the $E - M$ ones. The second phase of the algorithm is concerned about dealing with the weights. Considering a vertex labeling as a function $l : V \rightarrow \Re$, a feasible labeling is one such that

$$l(x) + l(y) \geq w(x, y), \forall\ x \in X, y \in Y \tag{2}$$

The labeling here works as a dual variable in order to solve the problem. Given equation 2, the Equality Graph (with respect to the labeling l) is defined as $G = (V, E_l)$ where E_l is the set of edges satisfying the equation with equality, that is:

$$E_l = (x, y) : l(x) + l(y) = w(x, y) \tag{3}$$

Figure 2 (left) illustrates an example of weights associated to the vertexes that satisfies equation 2, thus, they compose a feasible labeling. Figure 2 (right) shows an equality graph constructed using the same labeling.

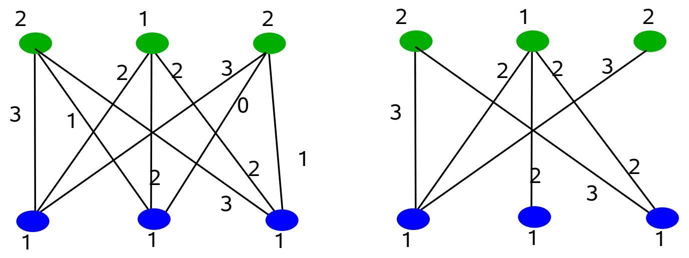


Fig. 2. A feasible labeling (left) and an equality graph (right)

The Kuhn-Munkres Theorem [7] assures that: If l is feasible and M is a perfect matching in E_l then M is a max-weight matching. This theorem transforms the problem from an optimization problem of finding a max-weight matching into a combinatorial one of finding a perfect matching. Its proof assures that for any matching M and any feasible labeling l we have

$$w(M) \leq \sum_{v \in V} L(v) \tag{4}$$

ie., that the sum of the labels in a feasible labeling defines an upper-bound on the cost of any perfect matching.

Finally, the Hungarian algorithm is described as: start with any feasible labeling l and some matching M in E; while M is not perfect repeat the following:

– find an augmenting path for M in E_l and increase the size of M;
– if M is maximum, then, it has minimum cost among all maximum matchings in G. Stop the search.
– otherwise, if no augmenting path exists, improve l to l' such that $E_l \subset E_{l'}$ (adjust the labels in order to enlarge the Equality Graph). Go back to the first step.

The iteration finishes as in each step of the loop we will either be increasing the size of M or of the set E_l. When it finishes, M is a perfect matching in E_l (if it exists), for some feasible labeling l and, by the Kuhn-Munkres theorem, M is a max-weight matching.

The Hungarian Algorithm Computation. main input is the weights associated with the graph edges that are arranged into a matrix, whose elements (i, j) represent the cost of a matching between the vertexes i and j, thus $w(i, j)$. Its computation is described as:

- Step 1:
  - 1.1 For each row, subtract its smallest element from all its elements.
  - 1.2 For each column, subtract its smallest element from all its elements.
- Step 2:
  - 2.1 Locate a lone zero and assign it.
  - 2.2 Cover the row or column associated with the lone zero depending on where the other zeros are, if any.

  Repeat these operations until there are no lone zeros in the matrix. Upon completion go to Step 3.
- Step 3:
  - 3.1 If all the rows have been assigned, then stop.
  - 3.2 If there are uncovered zeros, then go to Step 4.
  - 3.3 Else (if there are no uncovered zeros), go to Step 5.
- Step 4: Assign any one of the uncovered zeros and cover both its row and its column. Then go to Step 2.
- Step 5: The way this step is carried out depends on whether Step 3.2 had been visited since the last time a new zero was created in Step 2. If this step has not been executed since the last time a new zero was created, then new zero(s) are generated by subtracting the smallest uncovered element of the matrix from all the uncovered elements of the matrix. If this step has already been executed, it is necessary first to cover all the zeros of the matrix with the minimum number of lines (rows and/or columns) and then create a new zero. Then go to Step 2.

Some of the presented steps can be easily computed concurrently, but others intrinsically demand the introduction of a non GPU-friendly message pass scheme or of global synchronization points as they provoke ambiguities if computed concurrently or locally. Thus, the definition of independent processing kernels for coding this algorithm as required for a pure GPU computation, is not straightforward.

3.2 The Auction Algorithm

This section presents *The Auction Algorithm* for *The Assignment Problem.* For our purposes, this algorithm has a huge advantage over the Hungarian as it was originally described as a distributed relaxation method [8], very well suited for parallel computation.

The Auction Algorithm is semantically described as a real auction where persons compete for objects by raising their prices through competitive bidding. Suppose that there are n persons and m objects (where $(n \leq m)$). We want to match them in a way that each person should be assigned to a single object and each object should be assigned to at most a single person. The matching should respect the restriction that each person i can only be assigned to object a j if the pair (i, j) belongs to a given set $A(i)$ of possible matching pairs. Analogously, for each object j it is possible to define $B(j)$ as the set of persons that can be matched with j.

There is a benefit a_{ij} for matching a person i with an object j, such that the goal of the auction is to assign persons to objects so as to maximize the total benefit, defined as

$$\sum_{i=0}^{n-1} a_{ij_i} \tag{5}$$

The auction algorithm introduces an economic equilibrium problem that can be seen as a dual problem. It supposes that an object j has a price p_j and that the person who receives the object must pay the price p_j. As each person associates an benefit a_{ij} with each object, then the object j net value of for person i is related with the difference between the corresponding benefit the object price. Each person i would logically want to be assigned to an object j_i with maximal value, that is, with

$$a_{ij_i} - p_{j_i} = max\left\{a_{ij} - p_j\right\}. \tag{6}$$

The Linear Programming Formulation for *The Assignment Problem* associates an assignment A with the set of variables $\{x_{ij}|(i,j) \in A\}$, where $x_{ij} = 1$ if person i is assigned to object j and $x_{ij} = 0$ otherwise. Thus, the value of an assignment is expressed as

$$\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} a_{ij_i} x_{ij} \tag{7}$$

and the restrictions that assure one-to-one mapping are then written as

$$\sum_{j=0}^{m-1} x_{ij} = 1, \text{for every } i \tag{8}$$

and

$$\sum_{i=0}^{n-1} x_{ij} = 1, \text{for every } j \tag{9}$$

As a consequence of the existent duality, it can be assured using Linear Programming Theory that an equilibrium assignment offers the maximum total benefit (and thus solves *The Assignment Problem*), while the corresponding set of prices solves an associated dual problem.

The Auction Algorithm Computation goal is to find an equilibrium assignment and its corresponding price vector. The algorithm iterates between two steps: a bidding phase and an assignment phase.

During the bidding phase, each unassigned person finds an object j which offers maximal value (according to equation 6) and makes a bid for that object offering a bidding increment γ_i calculated as:

$$\gamma_i = v_i - w_i + \epsilon \tag{10}$$

where (v_i) and (w_i) are respectively the maximal and second maximal net values of objects that the person i is interested in. The inclusion of the positive constant ϵ in equation 10 assures that the bidding increment is not zero, which otherwise would happen in cases where a person has more that one object with the maximum net value, ie., cases where (v_i) is equal to (w_i).

After the bidding phase, the algorithm turns into the assignment phase. Then, each object j, if it was selected as a best object by any nonempty set of people $P(j)$, determines the highest bidder by:

$$i_j = arg\ max_{i \in P(j)}\ \gamma_i \tag{11}$$

Using the highest bidding increment the object raises its price and gets assigned to the person i, considered as highest bidder i_j. If the object was previously assigned to other person, that person becomes unassigned.

Iterating between those two phases, the algorithm continues until all persons have an assigned object. The termination with a feasible assignment (if it exists) is assured by noting that once an object is assigned to any person, it will never be turned into an unassigned object again. Besides, if an object receives a bid in k iterations, its price must exceed its initial price by at least $k\epsilon$, thus, at some point of the iteration, an assigned object will become expensive enough to be judged less valuable (according to equation 6) than some other object that has not received a bid so far. It follows an object can receive a bid in a limited number of iterations while some other object still has not yet received any bid. On the other hand, once n objects $(n \leq m)$ receive at least one bid, the auction terminates.

4 Using CPU *versus* GPU for Computer Vision Tasks

Both microprocessors (CPUs) and graphics hardware (GPUs – Graphics Processing Units) are composed in low level by the same components: the transistors. This means that the CPU and GPU are equally benefited by transistors technology advances [9]. Their performance difference can be illustrated comparing the processing power of models on market, like the NVIDIA GeForce 8800 GTX graphic card (330 GFlops and 80 GB/s bandwidth) and the multi-core processor Intel Core Duo (48 GFlops and 10 GB/s bandwidth).

What defines the main difference between CPUs and GPU being responsible for their efficient disparity is their architecture. CPUs are developed to efficiently attend to a variety class of applications, requiring the disposal of many transistors to offer complex control functionality in hardware (such as branch prediction). The GPUs were originally developed aiming 3D graphics processing which allowed their architecture to concentrate the transistors on computation power, rather than on control chips. Nowadays, GPUs are low-cost stream processors specialized on high arithmetic computation over independent elements of a stream.

Modern GPUs can be seen as fine-grained parallel computers. Its programming model requires to formulate the desired algorithms as what is called data algorithms, that means, to be executed with simultaneous operations across large sets of data. Each set of data is organized in what is called a stream, containing similar elements to be processed. Thus, the algorithm to be computed should be decomposed into similar operations to be applied to each element of a stream of data independently. The computations are defined in a operator called kernel describing the algorithm tasks to be applied over a single element of the stream. In such formulation, the parallelism occurs by processing in parallel the same kernel over different elements of the stream. As the number of processors increases, more elements can be processed in parallel, scaling the algorithm in a transparent way to its developers.

Computer vision tasks are well suited for Graphics Processing Unit (GPUs) hardware as many of their tasks can be seen as similar arithmetically-intensive operations over huge sets of data, exactly the nature of problem to which such hardware is developed for.

Next section presents our formulation to solve the bipartite graph matching problem using graphics hardware by decomposing *The Auction Algorithm* into a data-parallel formulation proper to moderns GPU architectures.

5 The Bipartite Graph Matching on GPU

Our proposal reformulates the auction algorithm for a GPU computation using one kernel for the bidding phase, one kernel for the assignment phase (both processed on the device - GPU) and a loop that iterates between the phases triggering the GPU threads until convergence (controlled by the host - CPU). The iteration cycle is illustrated in Figure 3.

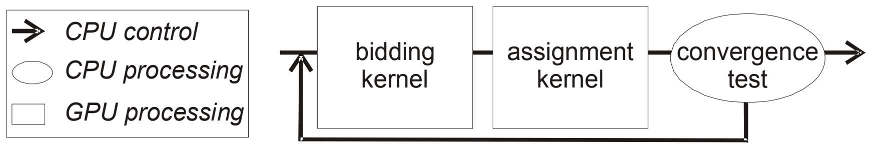


Fig. 3. Our GPU formulation iteration cycle

Before the matching computation begins, data streams have to be created representing (or reserving storage space for) the input, the output and temporary data used by the algorithm.

The initialization task includes the bipartite graph creation, that is, its disjoint sets of nodes and its edges set representation into data streams. The two sets of nodes (X and Y) represent the elements to match. For clarity we will call them here as the set of persons and the set of objects respectively. *The Assignment Problem* looks for a set of one-to-one associations represented as edges in this graph. This means that one should include edges from a vertex $x_i \in X$, to every vertexes $y_j \in Y$, that represents a possible matching in the final association. Consequently, the number of edges created for each vertex is related to the number of possible matchings for each node of the graph.

Any structure used by our algorithm is represented as a 1D or a 2D linear data stream. In our model, the graph nodes, edges and weights are represented within a single two-dimensional stream of constant elements (their values are set during initialization and kept constant during the algorithm), containing the benefit for matching a person i with an object j set as a matrix.

Both requirements that each person i can only be assigned to an object j, or that each object j can only be assigned to a person i, if those pairs belongs to a given set of possible matching pairs (existent edges) are imposed to the algorithm computation in our proposal by setting negative infinity values for the corresponding association of unwanted pairs within the benefit matrix during initialization phase. Thus, the rows and columns of such matrix indirectly represent the existent nodes, while the existent edges are represented with positive values in their corresponding matrix positions.

Other structures used by our algorithm are represented as unidimensional streams. They are dynamic value streams created to represent: the objects prices (initially set as zero); the objects index associated with each person (or a sentinel value if not associated); the person index associated with each object (or a sentinel value if not associated); the bids value suggested by each person in last iteration; the bids target from each person in last iteration (an object index, if any).

Observe that we split a natural two-dimensional data representation for the bids value of each person i to each object j, into two unidimensional streams, by observing that each person can only bid for a single object in each iteration. We create this organization in order to save graphics card memory and to induce faster memory transfers while keeping independent access to stream elements during the kernel computation. With such improvement each person can still write its bidding concurrently to all the others in our unidimensional streams, with no communication between them, but requiring much less storage space.

Once the iteration cycle starts, our algorithm turns what is considered as the input stream that will drive the parallel computation, alternating between a stream composed over the nodes of X and a stream composed over the nodes of Y. During the bidding phase a kernel is coded driven to process a single element of X, while during the assignment phase, a kernel is coded driven to process a single element of Y (see figure 4).

The bidding kernel (bk) is executed by every person concurrently. During its execution, the person decides if he is going to suggest a bid and to which object to bid for, or if he is currently associated to an object (does not ask for another).

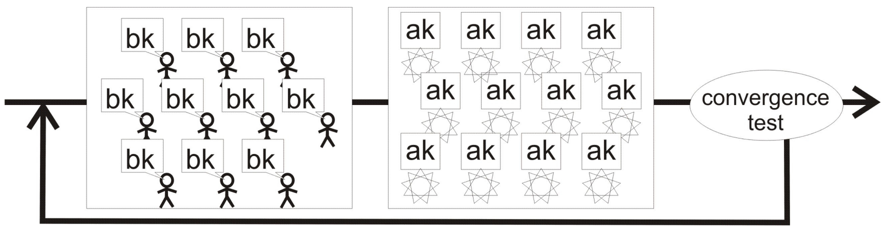


Fig. 4. Independent Parallel Processing

During the next phase of the cycle, every object executes the assignment kernel (ak), each one individually has the task of testing if it has received any bid recently. In such case, the object is responsible for updating its own price, for changing the current bidder (if it has one) and the bid value for the most recently ones. The object is also responsible for setting previous bidder free to let him to start bidding again.

5.1 The Convergence Test

Once that GPU can not trigger its own processes and threads, the convergence decision has to involve the CPU at least to decide if the algorithm cycle stops or if the CPU has to trigger the bidding kernel (bk) and the assignment kernel (ak) once more.

The task of computing the convergence test itself in CPU would require to retrieve data from the GPU streams to CPU memory space. Data transfers are one of the most expensive operations in CPU-GPU programming and such cost is directly proportional to the amount of data retrieved. Based on these facts, our goal is to reduce the amount of data consulted on CPU for the convergence decision.

The convergence criteria evaluates if all persons have already been assigned to an object. A first solution for its computation would involve the transfer of the unidimensional stream containing the objects' index associated with each person and checking in CPU if there is any person associated with the sentinel value. If not, the algorithm has converged.

In a better solution, we observe the algorithm cases when the total number of assigned persons changes and track those cases from the algorithm processing kernels. Supposing that the algorithm starts with no assignments, such number increases only when a free object receives the first bid. In cases when and assigned object changes its corresponding bidder, the total number of assigned persons remains unchanged. This observation is assured by the property of the algorithm that once an object is assigned for any person it never turns to unassigned again, so our counter can never decrease.

With the presented assumption, our algorithm uses a transfer of a single value between the GPU and CPU, representing a counter of the assigned persons total. Using such data, the convergence decision can be taken on CPU as it can be compared with the number of people, which is known as an initial input for the algorithm.

The task of updating the total number of assigned persons on GPU concurrently over several objects can be implemented using a single variable on global space memory, but accessed using atomic operations. In architectures where atomic operators are not available, the kernels can be implemented to write in an unidimensional stream containing a boolean value indicating for each object its status (assigned/unassigned). The total number of assigned objects can be retrieved reducting such boolean-valued stream to a single integer value (the reduction operator can be consulted in [10]).

6 Application and Results

A human being can normally solve visual correspondences quickly and easily, even when the sets of samples observed contain significant amount of noise. Different classes of Computer Vision tasks involve computing correspondences between sets of samples taken from distinct cameras or in distinct time instants.

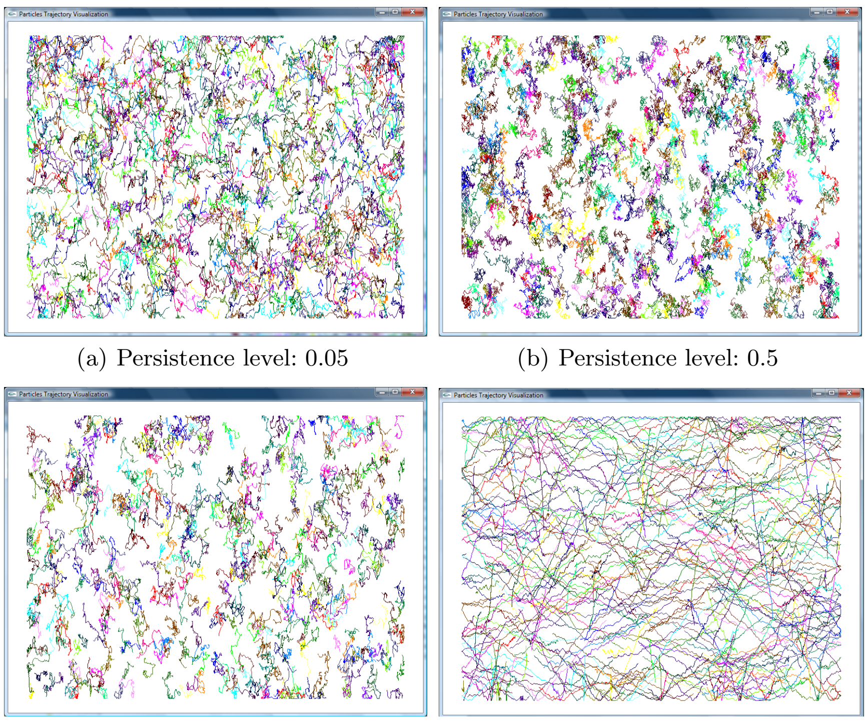
In this section we use the bipartite graph matching to model and compute the search for the best one-to-one association (according to the metric adopted) that correlates an input data composed by distinct sets of samples. The application developed models a variant of the correspondence problem as a discrete optimization over a bipartite graph. More specifically, we model a correspondence problem between $X = \{x_0, x_1, ..., x_n\}$ and $Y = \{y_0, y_1, ..., y_m\}$, that represent two independent sets of 3D samples to be matched. To attend the algorithm description and convergence criteria presented in section 5, in cases the samples sets have different size we associate X with the smaller one.

As a discrete optimization, we are interested in finding the minimum cost matching between the samples in X and Y. The weights of our graph are defined by a energy function indicating the cost of associating each 3D sample in X to each sample in Y. As we are assuming that the input data does not contain any feature that identifies the individual particles, but only their position in 3D space, the cost function is defined as the Euclidean distance between the time-sampled points.

As our first test set, we explore a data set taken with a microscopy collecting positioning data samples from bacterias moving around 3D space over a certain period of time. The problem is to find the correspondence between each bacteria sampled in a instant of time to the same bacteria in the next set of samples taken in *a posteriori* instant. The single feature taken as input is the sets of 3D positions obtained from the microscopy data. For each pair of samples sets taken during consecutive time instants, we initially consider that any bacteria in the first set of samples can be matched to any bacteria in the second set.

The evaluation of the matching accuracy for those microscopy data reveled that our model has found the right correspondences between all bacterias presented in both samples sets. Wrong matchings appear caused by the fact that a moving bacteria could left the microscopy vision field, while others can enter. That is, wrong cases happened when a particle leaves the vision field from one time instance sampled to the other and at the same time a new particle enters in the field. In those cases, our model do not identify that they are actually two different moving particles and it matches them like if they were the same. The disadvantage of such data is that it does not offer a data set with continuous growing size (neither many different data samples of each size) to evaluate correctly the matching efficiency.

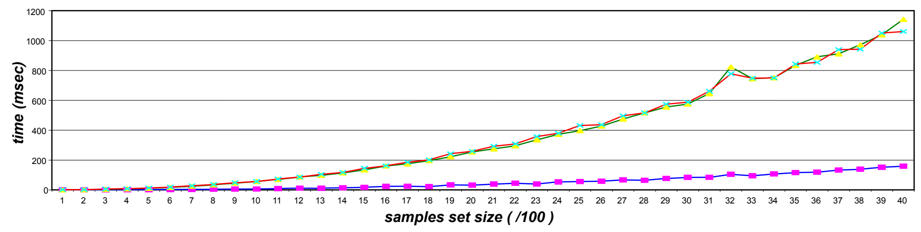
For measuring the timing results, we are interested in creating bipartite graphs with increasing number of nodes and edges. Aiming to produce data sets containing several, increasing size, test samples (with ground truth) we simulated sets of 3D moving points, moving inside a 3D bounding box in a random biased and correlated movement and sampled them in different iteration times. The velocity vector of our particles are composed by two vectors: the correlation vector, as a persistence tendency to keep the particle moving in the same direction, resulting in a correlation between successive steps of the simulation; and the bias vector, a randomly generated vector to disturb regular movement at each time sample. The persistence level in our model is a value between $[0.0, 1.0]$ that indicates a


Particles Trajectory Visualization
Particles Trajectory Visualization
Particles Trajectory Visualization
Particles Trajectory Visualization

(a) Persistence level: 0.05 (b) Persistence level: 0.5

(c) Persistence level: 0.7 (d) Persistence level: 0.95

Fig. 5. Tracking 500 samples moving in 3D with varying persistence levels



Samples set size	400	800	1200	1600	2000	2400	2800	3200	3600	4000
■ Our proposal on GPU (ms)	1.05	3.84	11.41	23.31	31.84	53.46	64.29	104.51	118.24	158.95
▲ CPU Hungarian (msec)	6.62	33.43	86.16	160.17	253.95	372.34	515.65	823.37	890.93	1141.91
✕ CPU Auction (msec)	7.75	36.12	85.82	161.47	257.74	379.45	515.95	777.59	853.24	1060.25

Fig. 6. Time comparison between CPU Hungarian, CPU Auction and GPU Auction

linear combination factor between the correlation vector and the bias vector in the composition for the particles movement (Figure 5).

Figure 6 presents the timing results for sequential implementations of the Hungarian and Auction algorithms (presented in sections 3.1 and 3.2) and for our parallel proposal computed on GPU (presented in section 5). The timings represent the mean answer time (in msec) for a hundred different data sets randomly generated given the total number of particles. The tests were performed using a Intel Core 2 Duo processor E6550 2.33Ghz with 2GB of RAM memory processor and a nVidia GeForce 9600 GT (512MB) graphics card. The GPU code was implemented using CUDA [11].

7 Conclusion

This paper presented a GPU formulation for computing a bipartite graph matching. In that sense, we described a data-parallel formulation proper to modern graphics cards architectures. The processed features (nodes, edges, their weights and intermediary data generated by the algorithm) used as the algorithm input and output were reviewed as streams of data, while algorithms applied to them are reformulated as "processing kernels" to be applied over several elements of each stream independently and in parallel.

The results presented show that our approach considerably accelerates the bipartite graph matching computation, opening the possibility of considering such graph and technique as a model in applications using huge sets of data and demanding fast results. In the future we are interested to compare our approach with other algorithms, like the *invisible hand algorithm* presented in [12].

Aiming to let the scientific community to be able to analyze their own experiments on bipartite graph matching applications, the source code is available for download from the author's homepage [5].

References

1. Fischer, T., Goldberg, A.V., Haglin, D.J., Plotkin, S.: Approximating matchings in parallel. Inf. Process. Lett. 46(3), 115–118 (1993)
2. Hougardy, S., Vinkemeier, D.E.: Approximating weighted matchings in parallel. Inf. Process. Lett. 99(3), 119–123 (2006)
3. Lotker, Z., Patt-Shamir, B., Rosen, A.: Distributed approximate matching. In: PODC 2007: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing, pp. 167–174. ACM, New York (2007)
4. Lotker, Z., Patt-Shamir, B., Pettie, S.: Improved distributed approximate matching. In: SPAA 2008: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures, pp. 129–136. ACM, New York (2008)
5. Vasconcelos, C.N., Rosenhahn, B.: Bipartite graph matching computation on gpu public code, `http://crisnv.googlepages.com/bgm`
6. Alexander, H., Saip, B., Lucchesi, C.: Matching algorithms for bipartite graphs. Technical report, DCC-UNICAMP (March 1993)
7. Kuhn, H.W.: The hungarian method for the assignment problem. Naval Research Logistics Quarterly 2, 83–97 (1955)
8. Bertsekas, D.P.: Auction algorithms for network flow problems: A tutorial introduction. Computational Optimization and Applications 1, 7–66 (1992)
9. Fernando, R.: GPU Gems 2 - Programming techniques for High-Performance Graphics and General Purpose Computation. Addison-Wesley Professional, USA (2005)
10. Vasconcelos, C.N., Sá, A., Teixeira, L., Carvalho, P.C., Gattass, M.: Real-time video processing for multi-object chromatic tracking. In: Proceedings of the 12th (BMVC 2008), pp. 113–122 (2008)
11. Cuda programming guide 1.1 (2007), `http://developer.download.nvidia.com/`
12. Kosowsky, J.J., Yuille, A.L.: The invisible hand algorithm: solving the assignment problem with statistical physics. Neural Netw. 7(3), 477–490 (1994)