Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 369 – 378

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering

Guilherme Andrade[a], Gabriel Ramos[a], Daniel Madeira[a,b], Rafael Sachetto[a], Renato Ferreira[c], Leonardo Rocha[a,*]

[a]*Federal University of São João del-Rei, MG, Brasil*
Computer Science Departament
[b]*Fluminense Federal University, RJ, Brasil*
Computer Science Institute
[c]*Federal University of Minas Gerais, MG, Brasil*
Computer Science Departament

Abstract

With the advent of Web 2.0, we see a new and differentiated scenario: there is more data than that can be effectively analyzed. Organizing this data has become one of the biggest problems in Computer Science. Many algorithms have been proposed for this purpose, highlighting those related to the Data Mining area, specifically the clustering algorithms. However, these algorithms are still a computational challenge because of the volume of data that needs to be processed. We found in the literature some proposals to make these algorithms feasible, and, recently, those related to parallelization on graphics processing units (GPUs) have presented good results. In this work we present the G-DBSCAN, a GPU parallel version of one of the most widely used clustering algorithms, the DBSCAN. Although there are other parallel versions of this algorithm, our technique distinguishes itself by the simplicity with which the data are indexed, using graphs, allowing various parallelization opportunities to be explored. In our evaluation we show that the G-DBSCAN using GPU, can be over 100x faster than its sequential version using CPU.

Keywords: clustering, dbscan, parallel computing, GPU

1. Introduction

With the advent of Web 2.0, we observed a real democratization of the data generation. Several tools are being developed allowing anyone with Internet access to publish and distribute data with a speed never seen before. The large volume of data generated, as well as the high complexity of its relations, has generated in recent years a challenging scenario for several applications: there is more data than that can be effectively analyzed. Thus organizing and finding appropriate information resources to fulfill the needs of users has become one of the most challenging problems in computer science.

New proposals for models and algorithms that are able to handle this data efficiently (response time and appropriate use of computational resources) and effectively (response quality, or the robustness and accuracy to perform a task) are emerging every moment. Among these, we highlight those related to the Data Mining area [1]. Data

*Corresponding author. email: lcrocha@ufsj.edu.br ; Tel.: +55-032-3373-3985 .

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.200

mining applications are highly relevant because of their wide applicability in terms of tasks and target scenarios, improving the quality and variety of the functionalities provided by all sorts of information systems. Nevertheless, they are still a computational challenge because of the data volume to be processed and the irregular nature of most of the existing algorithms, which make both their performance and resource demands quite unpredictable.

A collection of algorithms that illustrates this scenario are those related with clustering [2]. The goal of these algorithms is to organize large sets of objects into different groups (clusters) according to a similarity metric. These techniques can be used in many different scenarios, such as social networks, recommendation systems, bioinformatics etc., making their use even more challenging. In the literature, strategies has been proposed to make these applications feasible, whether through data indexing techniques [3, 4], either through the parallelization of these tasks using different processing units [5, 6]. With respect to parallelization strategies, the use of graphics processing units (GPU's) [7] has been given considerable importance, since these are able of providing a higher level of parallelism than multicore CPU's, associated with a lower energy consumption [8].

Thus, in this work we present a new clustering algorithm, the *G-DBSCAN*, a GPU accelerated algorithm for density-based clustering. Our algorithm is based on the original DBSCAN proposal [9], one of most important clustering techniques, which stands out for its ability to define clusters of arbitrary shape as well as the robustness with which it deals with the presence of data noise. The implementation strategy is quite simple and is divided into two steps. The first step is to construct a graph that will represent the data, where each object is represented as a node in the graph and an edge is created between two objects when the similarity measure between them is less than or equal to a threshold defined as an input parameter (i.e. Euclidean distance less than 3). After the construction of this graph, the second step is to identify the clusters, using a traditional breadth-first search (BFS) to traverse the graph created in the first step. In this work, both steps were implemented using GPUs, resulting in an extremely efficient algorithm regarding to the execution time, achieving a speedup greater than 100×.

The remainder of this paper is divided as follows: in Section 2 we describe some related work. In Section 3 we present the original DBSCAN proposal, detail the implementation strategy based on graphs used in the G-DBSCAN as well as the parallelization strategies. Section 4 presents the experiments performed to evaluate our algorithm, and the obtained results. Finally, we present our conclusions in Section 5.

2. Related Work

Data clustering is one of the most common and more used techniques in data mining. Its goal is basically, receiving a dataset as input, organize the data into semantically consistent groups, based on a previously defined similarity metric. In [2] several issues related to the use of clustering techniques are presented, highlighting some of its challenges, such as how to properly set the input parameters, how to specify a good similarity measure metric and how to work with large volumes of data.

Several clustering algorithms are found in the literature [10]. These algorithms range from simpler techniques and widely used in various scenarios, such as k-means [11] to more elaborate and context-driven techniques, such as subspace clustering [12] and partitioning clustering [13]. A set of clustering techniques which is receiving great attention is the one related to density-based clustering [9, 14, 15]. Such techniques are distinguished by their ease of implementation and by the applicability in different contexts. Moreover, these techniques do not need to determine in advance, as an algorithm input, the number of clusters, as is done by the others techniques mentioned above.

Among the density-based clustering techniques aforementioned, the most referenced in the literature is DBSCAN [9]. The DBSCAN technique is even being used as a base for many other techniques [14, 5]. Its operation is based on calculating a proximity radius between each pair of objects, which is defined according to the adopted similarity metric (i.e. Euclidean distance, cosine similarity, etc.). From a minimum proximity radius, defined as an algorithm input, objects are grouped with each other whenever they are within this proximity radius. One of the most used strategies to improve the performance of these algorithms is the data indexing [3, 4]. Among the most commonly used indexing techniques, we can highlight the priority R-Tree [16], that reduces the complexity of the DBSCAN algorithm from $O(n^2)$ to $O(nlogn)$.

Although the use of data indexing techniques improves the performance of density-based clustering algorithms [5], the scalability of these algorithms and making them effectively applicable in a large data volume

scenario remains a major challenge [17]. One of the adopted solutions to address this challenge is the use of parallel computing, either through distributed memory strategies [18, 6] or, more recently and with good results, through strategies that use graphics accelerators [19, 20, 5]. In [19], the authors present an interesting study on how several data mining applications can be implemented using GPU in CUDA architecture [7]. In [20] a GPU version of K-means algorithm is presented, and in [5] is shown a parallel version of the DBSCAN algorithm using CUDA. Moreover, in the last, the authors also present a new data indexing strategy achieving interesting results, where the parallel version is up to $15x$ faster compared to sequential algorithm.

In our work we also propose a parallel version of DBSCAN using GPU, the G-DBSCAN. In our version we use a graph structure for the data indexing. Despite its simplicity, as we shall see in Section 3.3, this structure allows the exploration of several parallelization opportunities using graphics processing units (GPUs). In our evaluation we show that the G-DBSCAN using GPU manages to be more than $100x$ faster than its sequential CPU version.

3. G-DBSCAN

3.1. DBSCAN: An Overview

The DBSCAN algorithm is a density-based clustering technique. Such algorithms assume that clusters are regions of high density patterns, separated by regions of low density in the data space. A cluster (or grouped data set) is defined as a connected dense component and grows in any direction that density leads. The operation of the algorithm is straightforward and is based on calculating a proximity radius between each pair of objects. This calculation is defined according to the adopted similarity metric (i.e. Euclidean distance, cosine similarity etc.). For each object in the dataset, the algorithm evaluates the number of neighbours that an object have by counting the number of other objects that are within a proximity radius (minimum R), defined as an input parameter of the algorithm. Based on this calculation, each object is labelled **core**, **border** or **noise**, according to its number of neighbours. If a given object has more neighbours than a minimum value ($MinPts$), also defined as a parameter of the algorithm, it is classified as **core**, and all objects reachable from it, either directly (direct neighbours) or indirectly (neighbours of neighbours), are classified as **border**. All the others objects, not reachable from any core, are classified as **noise**. Each set of objects associated with a **core** determines a cluster. Figure 1 illustrate this process.

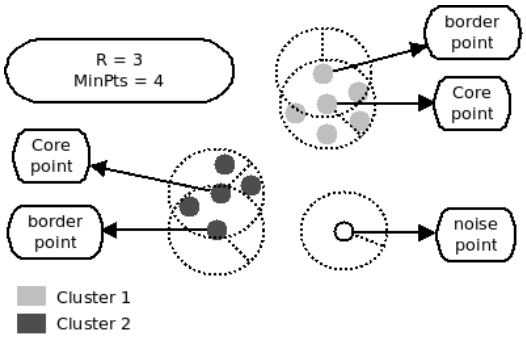


Figura 1. DBSCAN example using $R = 3$ and $MinPts = 4$ as input parameters.

It is worth noting that, unlike other traditional clustering algorithms such as K-means, DBSCAN does not need as input parameter the number of clusters to be found. In density-based clustering that number is derived from the parameters R and $MinPts$, which also define the density of the clusters to be found.

3.2. Serial implementation

In our proposal, we represent the data as a graph $G(V, E)$, where V represents the objects to be clustered and E the edges connecting the objects that are within the minimum proximity radius of each other. As our main goal is the parallelization of this algorithm using GPUs, which have a major memory limitation, we chose to represent the graph using a compact adjacency list. For this we use two vectors, Va that represents the vertices and Ea that stores the adjacency lists. In Va the index represents the vertex and each position of the vector stores two values:

the number of vertices that are adjacent to this vertex (its degree) and the position in vector Ea where its adjacency list begins.

We illustrate this data structure through Figure 2. For instance, to find the adjacency list of the object with label 0, we first have get the values stored in the position 0 of Va. The first value represents the vertex degree (in this case 2) and the second, the index where its adjacency list begins in Ea (index 0). Then we are able to obtain its adjacency list by visiting two positions in Ea, starting from position 0. In this case we obtain the objects 2 and 3. The space complexity of this data structure is $O(V + E)$.

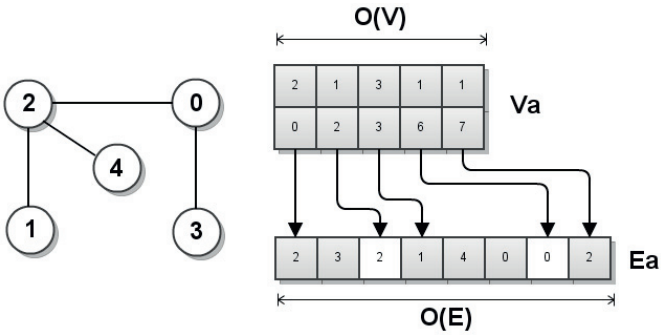


Figura 2. The data structure used to represent the graph.

Algorithm 1 presents a pseudocode that illustrates the construction process of the graph that represents the data. As can be seen, the algorithm receives as input parameters the data (*dataset*), R and $MinPts$ and returns the resulting graph. For each object, we first calculate the distance (proximity radius) to other objects using the function *distance*, which in our case is the Euclidean distance [21]. If the calculated value is lower than the input parameter R, an edge is created between these objects. After this first step, the object is then classified as core, if the total number of neighbors is greater than the parameter $MinPts$, or noise otherwise, by the function $ClassifyObject$. These two steps (calculate all distances from an object and classify it as core or noise) are the first part of our proposed method. The border objects will be identified in the next step, as presented below.

Algorithm 1 Algorithm for the graph construction

```
function MakeGraph(MinPts, R, dataset, Graph)
    for all c_i ∈ dataSet do
        for all c_j ∈ dataset do
            if distance(c_i, c_j) ≤ R then
                InsertEdge(c_i, c_j, Graph)
            end if
        end for
      dataset = dataset - c_i
      ClassifyObject(c_i,MinPts)
    end for
      return Graph
end function
```

After the creation of data representation and the classification of the objects, the next step is to identify the clusters. In our approach, this identification is done by using a breadth-first search (BFS) in graphs [22]. More specifically, a BFS is started from a node classified as core, and will visit all other vertices reachable from that core node. As these vertices are visited, they are labelled as members of the same cluster, and also have their classification changed to border. Upon the completion of this operation, a new core vertex, which has not been visited by the previously performed BFS, is selected and a new BFS is performed starting from it. Each BFS will find a new cluster and must be repeated as long as there are non-visited core objects. Algorithm 2 illustrates this operation.

Thus, our approach has two distinct stages: the graph construction and the clusters identification. The first has

Algorithm 2 Cluster identification algorithm

```
function IdentifyCluster(Graph) cluster = 0
    for all v ∈ Graph do
        if (v.visited ≠ 1) ∧ (v.type = core) then
            v.visited = 1
            v.cluster = cluster
            BreadthFirstSearch(v,Graph,cluster)
            cluster+ = 1
        end if
    end for
end function
```

a time complexity of $O(V^2)$, since, in the worst case, it will require a comparison between each pair of objects. The second stage has a time complexity of $O(V + E)$ given by the BFS algorithm. So, the total complexity of our approach is $O(V^2)$. As mentioned earlier, although there are more elaborate indexing techniques aimed at reducing this complexity, our choice was based on the parallelization opportunities that can be exploited using GPUs, as we shall see in the next section.

3.3. Parallel implementation

As discussed in the previous section, the G-DBSCAN is divided into two stages: graph construction and identification of clusters through BFS. In our approach, both steps have been parallelized as we shall see in the following sections.

3.3.1. Graph Construction

The computational representation of the graph, as described in Section 3.2, involves two arrays: Va and Ea, with spatial complexity of $O(V)$ and $O(E)$ respectively. In Va each position corresponds to a vertex and stores two values: 1) the number of vertices adjacent to the vertex stored in a position (degree) and 2) the start index for its adjacency list in Ea. Thus, the construction of the graph involves three basic steps: calculating the first value of Va, calculating the second value of Va and finally the construction of Ea. All the steps mentioned before where parallelized using GPUs.

- **First Step (Vertices degree calculation):** For each vertex, we calculate the total number of adjacent vertices. However we can use the multiple cores of the GPU to process multiple vertices in parallel. Our parallel strategy using GPU assigns a thread to each vertex, i.e., each entry of the vector Va. Each GPU thread will count how many adjacent vertex has under its responsibility, filling the first value on the vector Va. As we can see, there are no dependency (or communication) between those parallel tasks (embarrassingly parallel problem). Thus, the computational complexity can be reduced from $O(V^2)$ to $O(V)$.
- **Second Step (Calculation of the adjacency lists indices):** The second value in Va is related to the start index in Ea of the adjacency list of a particular vertex. The calculation of this value depends on the start index of the vertex adjacency list and the degree of the previous vertex. For example, the start index for the vertex 0 is 0, since it is the first vertex. For the vertex 1, the start index is the start index from the previous vertex (i.e. 0), plus its degree, already calculated in the previous step. We realize that we have a data dependency where the next vertex depends on the calculation of the preceding vertices. This is a problem that can be efficiently done in parallel using an **exclusive_scan** operation [23]. For this operation, we used the *thrust* library, distributed as part of the CUDA SDK. This library provides, among others algorithms, an optimized exclusive scan implementation that is suitable for our method.
- **Third Step (Assembly of adjacency lists):** Having the vector Va been completely filled, i.e., for each vertex, we know its degree and the start index of its adjacency list, calculated in the two previous steps, we can now simply mount the compact adjacency list, represented by Ea. Following the logic of the first step, we assign a GPU thread to each vertex. Each of these threads will fill the adjacency list of its associated vertex with all vertices adjacent to it. The adjacency list for each vertex starts at the indices present in the second value of Va, and has an offset related to the degree of the vertex.

3.3.2. Clusters identification

For this step, we decided to parallelize the BFS. Our parallelization approach in CUDA is based on the work presented in [22], which performs a level synchronization, i.e. the BFS traverses the graph in levels. Once a level is visited, it is not visited again. The concept of border in the BFS corresponds to all nodes being processed at the current level. In our implementation we assign one thread to each vertex. Two Boolean vectors, $Borders$ and $Visiteds$, namely Fa and Xa, respectively, of size V are created to store the vertices that are on the border of BFS (vertices of the current level) and the vertices already visited. In each iteration, each thread (vertex) looks for its entry in the vector Fa. If its position is marked, the vertex removes its own entry on Fa and marks its position in the vector Xa (it is removed from the border, and it has been visited, so we can go to the next level). It also adds its neighbours to the vector Fa if they have not already been visited, thus beginning the search in a new level. This process is repeated until the boundary becomes empty. We illustrate the functioning of our BFS parallel implementation in Algorithm 3 and 4.

Algorithm 3 CPU BFS

```
function BreadthFirstSearch(v, Graph, cluster)
    Declare Xa[1..V]
    Declare Fa[1..V]
    Initialize Fa and Xa with 0
    Fa[v] = 1
  while Fa ≠ ∅ do
    for all v ∈ V in parallel do
      Invoke BreadthFirstSearchKernel(Graph, Fa, Xa) on the grid
    end for
  end while
  Return Xa to host
  for all v ∈ V do
    if (Xa[v] = 1) then
      v.cluster = cluster
      v.visited = 1
      if NOT v.type = core then
        v.type = border
      end if
    end if
  end for
end function
```

Algorithm 3 runs on the CPU while Algorithm 4 runs on GPU. The main loop of Algorithm 3 ends when all levels of the graph are visited and vector border is empty. Next, all nodes visited beginning from a core node are assigned to the same cluster, and the classification is changed to border. As discussed in Algorithm 2, while there are unvisited core nodes, new searches are performed and in each of them a new cluster is set. Therefore, we can note that this implementation has, in the worst case, a complexity directly proportional to the diameter of the graph. In the next section we present the detailed experimental evaluation regarding the performance of the G-DBSCAN.

4. G-DBSCAN Evaluation

4.1. Experimental evaluation

To evaluate the proposed G-DBSCAN algorithm, we developed a set of tests ranging the size of the input data set between $5,000$ and $700,000$ objects in a two dimensional space (two attributes). For each data size we extracted the execution times for the construction of the graph and the BFS, as well as the total time spent by the application, both for the implementation that uses only CPU as well as the version that uses the GPU. From these data we evaluate and compare the *speedup* achieved by the GPU implementation in each part of the algorithm, as well as in the application as a whole. In all our tests we used data sets with 20 randomly generated Gaussian

Algorithm 4 GPU BFS

```
function BreadthFirstSearchKernel(Graph, Fa, Xa)
    tid = getThreadID
    if Fa[tid] then
        Fa[tid] = 0, Xa[tid] = 1
        for all neighbors vid of tid do
            if NOT Xa[nid] then
                Fa[nid] = 1
            end if
        end for
    end if
end function
```

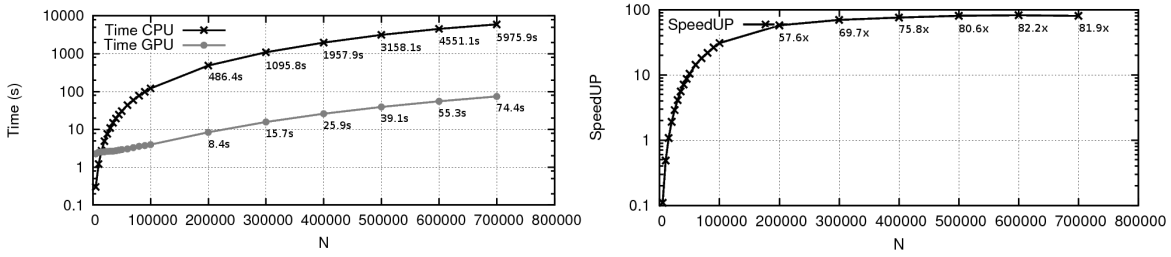
clusters. Furthermore, the input parameters are fixed for all tests being $MinPts = 4$ and $radius = 0.05$. The choice of this configuration was based on that presented in [5], in order to compare the algorithm proposed in that paper with our proposal.

The implementation was written in C and C for CUDA (nVidia) [7], and all experiments were performed on a Intel® Xeon® 2.40GHz equipped with a Tesla M2050. The Tesla M2050 card has 448 CUDA cores and 3 GB of global memory. In the following subsections we present the results obtained in the construction of the graph, the BFS and finally in the application as a whole, presenting the speedup of each step and showing the achieved gains.

4.2. Graph construction evaluation

As described in previous sections, the graph construction step has the greater time complexity ($O(V^2)$), thus requiring most of the processing time of the entire application. The graphs of Figure 3 present the results obtained with the GPU parallelization of this step of the algorithm.

As can be seen, there is a significant reduction in the execution time of this step. The speedup increases significantly until $N = 200,000$. For values of N greater than $200,000$ the growth is less pronounced stabilizing around $N = 600,000$ with a speedup of 82×. This stabilization is explained by the fact that when we increase the input size we also increase the number of GPU threads/blocks. Then the overhead generated to schedule the amount of threads becomes a limiting factor, stagnating the speedup at a certain point. Anyway, a 82× speedup can be considered an excellent result, since we use only one processing node (GPU).



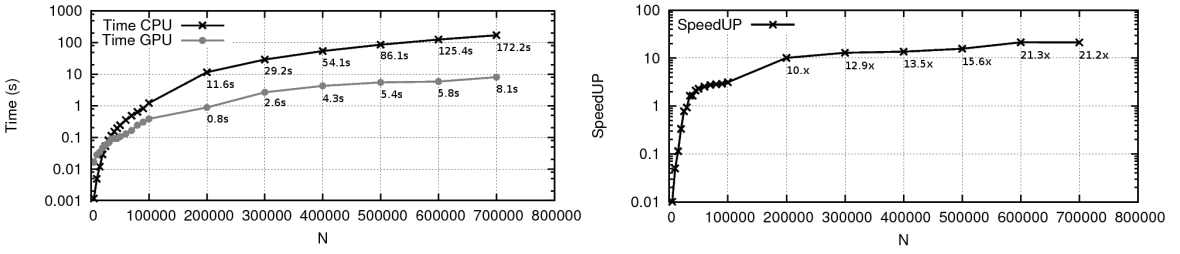
(a) Runtime (b) Speedup

Figura 3. Runtime and speedup of graph construction

4.3. BFS evaluation

The BFS accounts for the second part of the application runtime and its results are shown in the graphs of Figure 4. As we can see by these graphs, although the BFS demands less time than the graph construction, our parallel implementation achieved interesting results. The speedup stabilizes around 21× with inputs greater than $600,000$ objects. The explanation for this stabilization is similar to that presented in the graph construction.

Still, we have an impressive reduction of the execution time for the BFS, dropping the time from 172.2s to 8.1s comparing our CPU and GPU implementations, in the largest input instance.



(a) Runtime (b) Speedup

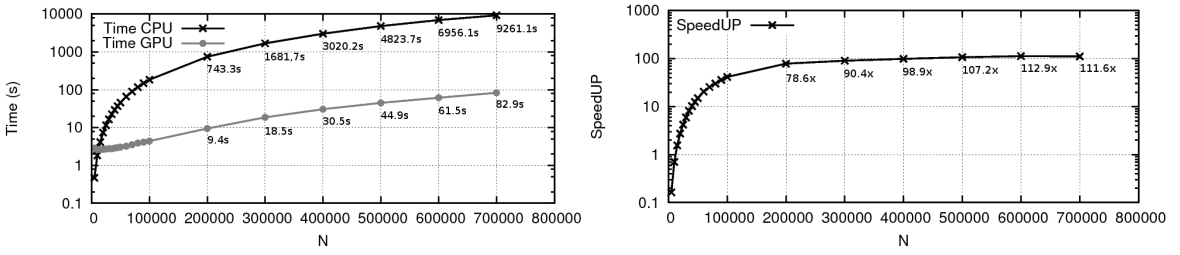
Figura 4. Runtime and speedup of BFS

4.4. G-DBSCAN evaluation

In Figure 5 we present the overall evaluations. As previously discussed, the presented implementation of the G-DBSCAN has a total complexity of $O(V^2)$, while the strategy presented in [5] has a complexity of $O(VlogV)$. However, the time of our CPU implementation and the one presented in [5] are relatively close, making the comparison of our speedups with [5] fair.One explanation for this may lie in the constant that multiply the complexity presented in [5].

Evaluating the runtime and speedup achieved by our algorithm, we can see that the division of the clustering process between graph construction and BFS allowed us to explore several parallelization opportunities. Consequently, this fact adds a much higher level of parallelism that the strategy presented in [5]. This division ensured that the execution times of our GPU application was lower than the times shown in [5], increasing the overall speedup. The maximum speedup achieved was 112x, decreasing the execution time from $9,261.1s$ on CPU to $82.9s$ on GPU, with $700,000$ objects. The parallel version presented in [5], which is based on a traditional implementation of DBSCAN, managed a maximum speed-up of 15x.

Finally, it is important to mention that, considering the total execution time, the speedup stabilization occurred with $600,000$ objects. This fact is justified by the stabilization in the graph construction and BFS.



(a) Runtime (b) Speedup

Figura 5. Runtime and Speedup of G-DBSCAN

5. Conclusions and Future Work

In this paper, we propose G-DBSCAN, a new version of an important algorithm for density-based clustering, the DBSCAN. In our proposal we use a very simple approach for data indexing using a graph $G(V, E)$, where V represents the objects to be clustered and E edges connecting objects that are within a minimum proximity R of each other. This R is defined as one of the algorithm parameters, along with the parameter $MinPts$, which is used to classify a vertex as a core vertex (vertices that have more than $MinPts$ neighbors), border (vertices reachable

from a core vertex) and noise (vertices that are not reachable from any core vertex). Using this graph, a BFS is applied at each unvisited core vertex, where all vertices reachable from a given core are assigned to the same cluster. The total time complexity of this approach is $O(V^2)$. We implemented this algorithm using GPUs, where each of these steps were parallelized using an efficient approach.

In order to evaluate our proposal, we developed a test set in which the number of objects to be clustered ranged between $5,000$ and $700,000$ objects in a two-dimensional space. In order to compare our proposal with the CUDA-DClust presented in [5], one of the most referenced versions of the GPU parallel DBSCAN, the parameters R and $MinPts$ were configured with the same values of that paper, 0.05 and 4, respectively. The CUDA-DClust uses traditional implementation strategies of DBSCAN, getting a total time complexity of $O(NlogN)$. In our results, we demonstrate that, for our proposal, both the parallelization of the graph construction and the cluster identification achieved excellent speed-ups(82× for graph construction and 21× for the clusters identification). Moreover, the total execution time of our sequential algorithm were very close to the times achieved in the sequential version of CUDA-DClust. However, in terms of speedup, while the CUDA-DClust achieved 15×, our algorithm achieved 112×, which is an extremely interesting result.

As future work, we aim to evaluate the G-DBSCAN on real data scenarios, where the volume is very large and the data is represented by several dimensions. Considering the great importance of data mining techniques in current scenarios as Web 2.0, where the data volume grows absurdly, our goal is to propose, implement and evaluate these techniques using multiples GPUs.

Acknowledgments

This work was partially funded by CNPq, CAPES, FINEP, Fapemig, and INWEB.

Referências

[1] I. H. Witten, E. Frank, Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations, Morgan Kaufmann Publishers Inc., 2000.
[2] A. K. Jain, M. N. Murty, P. J. Flynn, Data clustering: a review, ACM Comput. Surv. 31 (3) (1999) 264–323.
[3] P. Christen, A survey of indexing techniques for scalable record linkage and deduplication, IEEE Transactions on Knowledge and Data Engineering 24 (2012) 1537–1555.
[4] G.-H. Cha, Y.-I. Yoon, Clustered indexing technique for multidimensional index structures, in: Proceedings of the 13th International Conference on Database and Expert Systems Applications, DEXA '02, Springer-Verlag, London, UK, UK, 2002, pp. 935–944.
[5] C. Böhm, R. Noll, C. Plant, B. Wackersreuther, Density-based clustering using graphics processors, in: Proceedings of the 18th ACM conference on Information and knowledge management, CIKM '09, ACM, New York, NY, USA, 2009, pp. 661–670.
[6] M. Chen, X. Gao, H. Li, Parallel dbscan with priority r-tree, 2010, pp. 508–511.
[7] M. Fatica, D. Luebke, High performance computing with CUDA, Supercomputing 2007 tutorial. In Supercomputing 2007 tutorial notes (November 2007).
[8] Reducing the Energy Consumption of Embedded Systems by Integrating General Purpose GPUs, Technical reports in computer science, TU, Department of Computer Science, 2010.
URL `http://books.google.com.br/books?id=uBfUYgEACAAJ`
[9] M. Ester, H. peter Kriegel, J. S, X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, AAAI Press, 1996, pp. 226–231.
[10] P. Berkhin, A survey of clustering data mining techniques, Grouping Multidimensional Data (2006) 25–71.
[11] V. Faber, Clustering and the continuous k-means algorithm, Los Alamos Science 22 (1994) 138–144.
[12] R. Agrawal, J. Gehrke, D. Gunopulos, P. Raghavan, Automatic subspace clustering of high dimensional data for data mining applications, in: Proceedings of the 1998 ACM SIGMOD international conference on Management of data, SIGMOD '98, ACM, New York, NY, USA, 1998, pp. 94–105.
[13] S. A. Elavarasi, D. J. Akilandeswari, D. B. Sathiyabhama, A survey on partition clustering algorithms (2011).
[14] M. Ankerst, M. M. Breunig, H.-P. Kriegel, J. Sander, Optics: ordering points to identify the clustering structure, in: Proceedings of the 1999 ACM SIGMOD international conference on Management of data, SIGMOD '99, ACM, New York, NY, USA, 1999, pp. 49–60.
[15] A. Hinneburg, H.-H. Gabriel, Denclue 2.0: fast clustering based on kernel density estimation, in: Proceedings of the 7th international conference on Intelligent data analysis, IDA'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 70–80.
[16] D. M. Gavrila, R-tree index optimization, in: In Sixth International Symposium on Spatial Data Handling, 1994, pp. 771–791.
[17] P. Bradley, U. Fayyad, C. Reina, Scaling clustering algorithms to large databases, AAAI Press, 1998, pp. 9–15.
[18] A. Freitas, A survey of parallel data mining, in: H. Arner, N. Mackin (Eds.), Proc 2nd Int Conf on the Practical Applications of Knowledge Discovery and Data Mining, The Practical Application Company, London, 1998, pp. 287–300.
[19] D. Kumarihamy, L. Arundhati, Implementing data mining algorithms using NVIDIA CUDA (2009).
URL `http://hdl.handle.net/1900.100/3013`

[20] R. Farivar, D. Rebolledo, E. Chan, R. H. Campbell, A parallel implementation of K-means clustering on GPUs, in: H. R. Arabnia, Y. Mun (Eds.), Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'08), CSREA Press, Las Vegas, Nevada, USA, 2008, pp. 340–345.
[21] M. M. Deza, E. Deza, Encyclopedia of Distances, Springer Berlin Heidelberg, 2009.
[22] P. Harish, P. J. Narayanan, Accelerating large graph algorithms on the gpu using cuda, in: Proceedings of the 14th international conference on High performance computing, HiPC'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 197–208.
URL http://dl.acm.org/citation.cfm?id=1782174.1782200
[23] G. E. Blelloch, Prefix sums and their applications., Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University. (1990).