Static and Incremental Overlapping Clustering Algorithms for Large Collections Processing in GPU

Lázaro Janier González-Soler, Airel Pérez-Suárez and Leonardo Chang
Advanced Technologies Application Center (CENATAV)
7ma A # 21406, Playa, CP: 12200, Havana, Cuba
E-mail: jsoler@ceneatav.co.cu and http://www.cenatav.co.cu/index.php/profile/profile/userprofile/jsoler
E-mail: asuarez@cenatav.co.cu and http://www.cenatav.co.cu/index.php/profile/profile/userprofile/asuarez
E-mail: lchang@cenatav.co.cu and http://www.cenatav.co.cu/index.php/profile/profile/userprofile/lchang

Keywords: data mining, clustering, overlapping clustering, GPU computing

Received: November 18, 2016

Pattern Recognition and Data Mining pose several problems in which, by their inherent nature, it is considered that an object can belong to more than one class; that is, clusters can overlap each other. OClustR and DClustR are overlapping clustering algorithms that have shown, in the task of documents clustering, the better tradeoff between quality of the clusters and efficiency, among the existing overlapping clustering algorithms. Despite the good achievements attained by both aforementioned algorithms, they are $O(n^2)$ so they could be less useful in applications dealing with a large number of documents. Moreover, although DClustR can efficiently process changes in an already clustered collection, the amount of memory it uses could make it not suitable for applications dealing with very large document collections. In this paper, two GPU-based parallel algorithms, named CUDA-OClus and CUDA-DClus, are proposed in order to enhance the efficiency of OClustR and DClustR, respectively, in problems dealing with a very large number of documents. The experimental evaluation conducted over several standard document collections showed the correctness of both CUDA-OClus and CUDA-DClus, and also their better performance in terms of efficiency and memory consumption.

Povzetek: OClustR in DClustR sta prekrivna algoritma za gručenje, ki dosegata dobre rezultate, vendar je njuna kompleksnost kvadratnega reda velikosti. V tem prispevku sta predstavljena dva paralelna algoritma, ki temeljita na GPU: CUDA-OClus in CUDA-DClus. V eksperimentih sta pokazala zmožnost dela z velikimi količinami podatkov.

1 Introduction

Clustering is a technique of Machine Learning and Data Mining that has been widely used in several contexts [1]. This technique aims to structure a data set in clusters or classes such that objects belonging to the same class are more similar than objects belonging to different classes [2].

There are several problems that, by their inherent nature, consider that objects could belong to more than one class [3, 4, 5]; that is, clusters can overlap each other. Most of the clustering algorithms developed so far do not consider that clusters could share elements; however, the desire of adequately target those applications dealing with this problem, have recently favored the development of *overlapping clustering algorithms*; *i.e.*, algorithms that allow objects to belong to more than one cluster. An overlapping clustering algorithm that has shown, in the task of documents clustering, the better tradeoff between quality of the clusters and efficiency, among the existing overlapping clustering algorithms, is OClustR [6]. Despite the good achievements attained by OClustR in the task of documents clustering, it has two main limitations:

1. It has a computational complexity of $O(n^2)$, so it could be less useful in applications dealing with a large amount of documents.

2. It assumes that the entire collection is available before clustering. Thus, when this collection changes it needs to rebuild the clusters starting from scratch; that is, OClustR does not use the previously built clustering for updating the clusters after changes.

In order to overcome the second limitation, the DClustR algorithm was proposed by Pérez-Suárez *et al.* in [7]. DClustR introduced a strategy for efficiently updating the clustering after multiple additions and/or deletions from the collection, making it suitable for handling overlapping clustering in applications where the collection changes frequently, specially for those applications handling multiple changes at the same time. Nevertheless, DClustR still suffers from the first limitation; that is, like OClustR, it is $O(n^2)$. This implies that when the collection grows a lot, the time that DClustR uses for processing the changes could make it less useful in real applications. Moreover, when the collection grows, the memory space used by

DClustR for storing the data it needs will also grow, making DClustR a high memory consumer and consequently, making it not suitable for applications dealing with large collections. Motivated by the above mentioned facts, in this work we extend both OClustR and DClustR for efficiently processing very large document collections.

A technique that has been widely used in recent years in order to speed-up computing tasks is parallel computing and specifically, GPU computing. A GPU is a device that was initially designed for processing algorithms belonging to the graphical world, but due to its low cost, its high level of parallelism and its optimized floating-point operations, it has been used in many real applications dealing with a large amount of data.

The main contribution of this paper is the proposal of two GPU-based parallel algorithms, namely *CUDA-OClus* and *CUDA-DClus*, which enhance the efficiency of OClustR and DClustR, respectively, in problems dealing with a very large number of documents, like for instance news analysis, information organization and profiles identification, among others.

Preliminary results of this paper were published in [8]. The main differences of this paper with respect to the conference paper presented in [8] are the following: (1) we introduce a new GPU-based algorithm, named CUDA-DClus, which is a parallel version of the DClustR algorithm, that is able to efficiently process changes in an already clustered collection and to efficiently process large collections of documents, and (2) we introduce a strategy for incrementally building and updating the connected components presented in a graph, allowing CUDA-DClus to minimize the memory needed for processing the whole collection. It is important to highlight that in CUDA-DClus we only analyze the additions of objects to the collection, because this is the case in which it could be difficult to apply DClustR in real applications dealing with large collections, since this is the case that makes the collection grow.

The remainder of this paper is organized as follows: in Section 2, a brief description of both the OClustR and DClustR algorithms are presented. In Section 3, the CUDA-OClus and CUDA-DClus parallel clustering algorithms are proposed. An experimental evaluation, showing the performance of both proposed algorithms on several document collections, is presented in Section 4. Finally, the conclusions as well as some ideas about future directions are presented in Section 5.

2 OClustR and DClustR algorithms

In this section, both the OClustR [6] and DClustR [7] algorithms are described. Since DClustR is the extension of OClustR for efficiently processing collections that can change due to additions, deletions and modifications, the OClustR is first introduced and then, the strategy used by DClustR for updating the clustering after changes is presented. All the definitions and examples presented in this section were taken from [6, 7].

2.1 The OClustR algorithm

In order to build a set of overlapping clusters from a collection of objects, OClustR employs a strategy comprised of three stages. In the first stage, the collection of objects is represented by OClustR as a *weighted thresholded similarity graph*. Afterwards, in the second stage, an initial set of clusters is built through a cover of the graph representing the collection, using a special kind of subgraph. Finally, in the third stage the final set of overlapping clusters is obtained by improving the initial set of clusters. Following, each stage is briefly described.

Let $O = \{o_1, o_2, \ldots, o_n\}$ be a collection of objects, $\beta \in [0, 1]$ a similarity threshold, and S:$O \times O \rightarrow \Re$ a symmetric similarity function. A *weighted thresholded similarity graph*, denoted as $\widetilde{G}_\beta = \langle V, \widetilde{E}_\beta, S\rangle$, is an undirected and weighted graph such that $V = O$ and there is an edge $(v, u) \in \widetilde{E}_\beta$ iff $S(v, u) \geq \beta$; each edge $(v, u) \in \widetilde{E}_\beta, v \neq u$ is labeled with the value of $S(v, u)$. As it can be inferred, in the first stage OClustR must compute the similarity between each pair of objects; thus, the computational complexity of this stage is $O(n^2)$.

Once $\widetilde{G}_\beta$ is built, in the second stage OClustR builds an initial set of clusters through a covering of $\widetilde{G}_\beta$, using *weighted star-shaped sub-graphs*.

Let $\widetilde{G}_\beta = \langle V, \widetilde{E}_\beta, S\rangle$ be a weighted thresholded similarity graph. A *weighted star-shaped sub-graph* (*ws-graph*) in $\widetilde{G}_\beta$, denoted by $G^\star = \langle V^\star, E^\star, S\rangle$, is a sub-graph of $\widetilde{G}_\beta$, having a vertex $c \in V^\star$, called the *center* of $G^\star$, such that there is an edge between c and all the other vertices in $V^\star \setminus \{c\}$; these vertices are called *satellites*. All vertices in $\widetilde{G}_\beta$ having no adjacent vertices (i.e., isolated vertices) are considered *degenerated* ws-graphs.

For building a covering of $\widetilde{G}_\beta$ using ws-graphs, OClustR must build a set $W = \{G_1^\star, G_2^\star, \ldots, G_k^\star\}$ of ws-graphs of $\widetilde{G}_\beta$, such that $V = \bigcup_{i=1}^{k} V_i^\star$, being $V_i^\star, \forall i = 1 \ldots k$, the set of vertices of the ws-graph $G_i^\star$. For solving this problem, OClustR searches for a list $C = \{c_1, c_2, \ldots, c_k\}$, such that $c_i \in C$ is the center of $G_i^\star \in W$, $\forall i = 1..k$. In the following, we will say that a vertex v is *covered* if it belongs to C or if it is adjacent to a vertex that belongs to C. For pruning the search space and for establishing a criterion in order to select the vertices that should be included in C, the concept of *relevance* of a vertex is introduced.

The *relevance* of a vertex v, denoted as $v.relevance$, is defined as the average between the *relative density* and the *relative compactness* of a vertex v, denoted as $v.densityR$ and $v.compactnessR$, respectively, which are defined as follows:

$$v.densityR = \frac{|\{u \in v.Adj / |v.Adj| \geq |u.Adj|\}|}{|v.Adj|},$$

$$v.compactnessR = \frac{|\{u \in v.Adj / AIS(G_v^\star) \geq AIS(G_u^\star)\}|}{|v.Adj|},$$

where $v.Adj$ and $u.Adj$ are the set of adjacent vertices of v and u, respectively; $G_v^\star$ and $G_u^\star$ are the ws-graphs determined by vertices v and u, and $AIS(G_v^\star)$ and $AIS(G_u^\star)$

are the *approximated intra-cluster similarity* of $G^{\star}_{v}$ and $G^{\star}_{u}$, respectively. The *approximated intra-cluster similarity* of a ws-graph $G^{\star}$ is defined as the average weight of the edges existing in $G^{\star}$ between its center and its satellites.

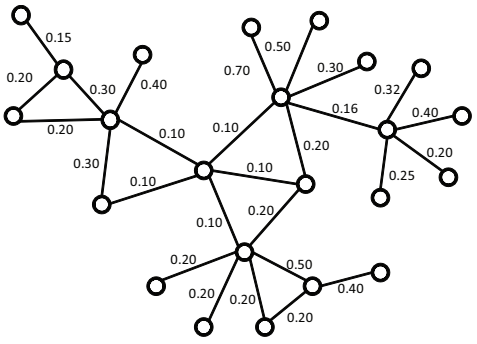
Based on the above definitions, the strategy that OClustR uses in order to build the list C is composed of three steps. First, a *candidate list* L containing the vertices having relevance greater than zero is created; isolated vertices are directly included in C. Then, L is sorted in decreasing order of their relevance and each vertex $v \in L$ is visited. If v is not covered yet or it has at least one adjacent vertex that is not covered yet, then v is added to C. Each selected vertex, together with its adjacent vertices, constitutes a cluster in the initial set of clusters. The second stage of OClustR also has a computational complexity of $O(n^2)$. Figure 1 shows through an example, the steps performed by OClustR in the second stage for building the initial set of clusters.

Finally, in the third stage, the final clusters are obtained though a process which aims to improve the initial clusters. With this aim, OClustR processes C in order to remove the vertices forming a *non-useful* ws-graph. A vertex v forms a non-useful ws-graph if: a) there is at least another vertex $u \in C$ such that the ws-graph u determines includes v as a satellite, and b) the ws-graph determined by v shares more vertices with other existing ws-graphs than those it only contains. For removing non useful vertices, OClustR uses three steps. First, the vertices in C are sorted in descending order according to their number of adjacent vertices. After that, each vertex $v \in C$ is visited in order to remove those non-useful ws-graphs determined by vertices in $(v.Adj \cap C)$. If a ws-graph $G^{\star}_{u}$, with $u \in (v.Adj \cap C)$, is non-useful, u is removed from C and the satellites it only covers are "virtually linked" to v by adding them to a list named $v.Linked$; in this way, those vertices virtually linked to v will also belong to the ws-graph v determines. Once all vertices in $(v.Adj \cap C)$ are analyzed, v together with the vertices in $v.Adj$ and $v.Linked$ constitute a final cluster. This third stage also has a computational complexity of $O(n^2)$. Figure 2 shows through an example, how the final clusters are obtained from the initial clusters showed in Figure 1(d).

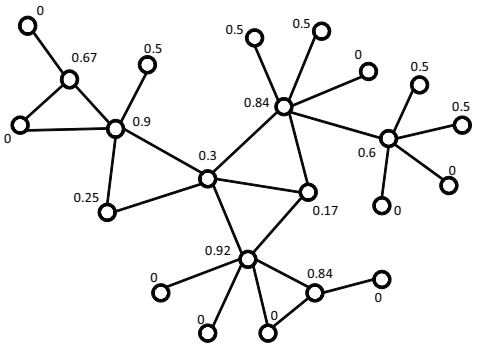
2.2 Updating the clusters after changes: the DClustR algorithm

Let $\widetilde{G}_{\beta} = \langle V, \widetilde{E}_{\beta}, S \rangle$ be the weighted thresholded similarity graph that represents an already clustered collection O. Let $C = \{c_1, c_2, \ldots, c_k\}$ be the set of vertices representing the current covering of $\widetilde{G}_{\beta}$ and consequently, the current clustering. When some vertices are added to and/or removed from O (i.e., from $\widetilde{G}_{\beta}$), there could happen the following two situations:

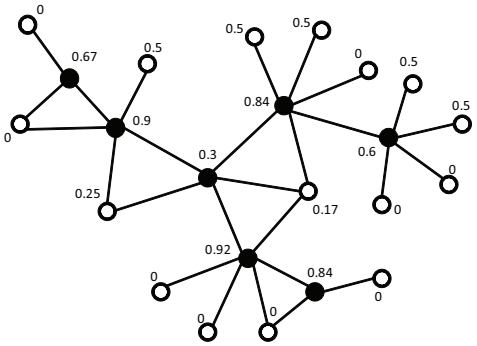
1) Some vertices become uncovered. This situation occurs when at least one of the added vertices is uncovered or when those vertices of C covering a specific vertex were deleted from $\widetilde{G}_{\beta}$.



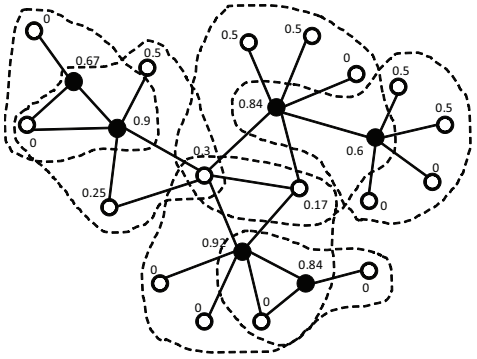
(a) A weighted thresholded similarity graph $\widetilde{G}_{\beta}$



(b) $\widetilde{G}_{\beta}$ using relevance for labeling the vertices

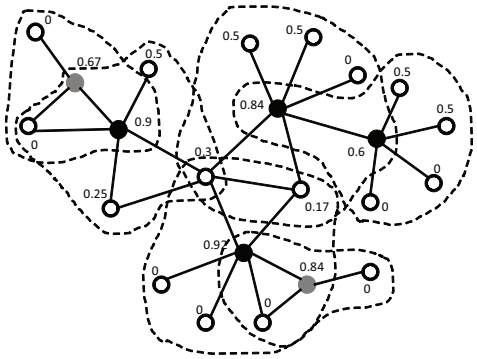


(c) Vertices belonging to set C

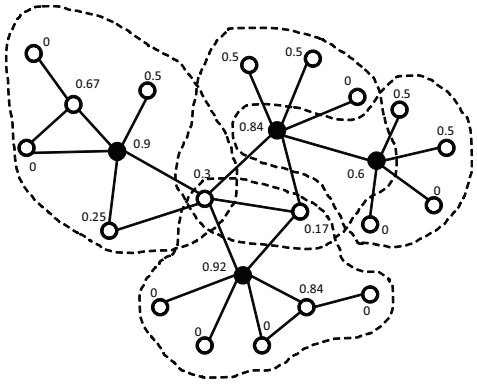


(d) Set of initial clusters

Figure 1: Illustration of how OClustR builds the initial set of clusters.



(a) Vertices determining non-useful ws-graphs (filled with light gray)



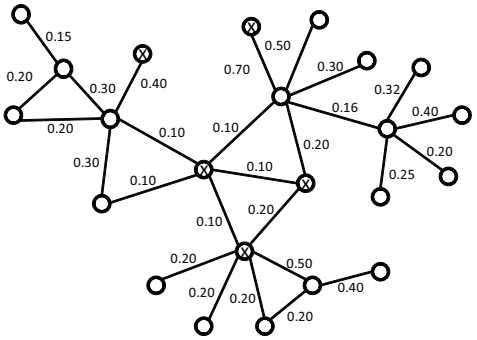
(b) Final set of overlapping clusters

Figure 2: Illustration of how the final clusters are obtained by OClustR in the third stage.

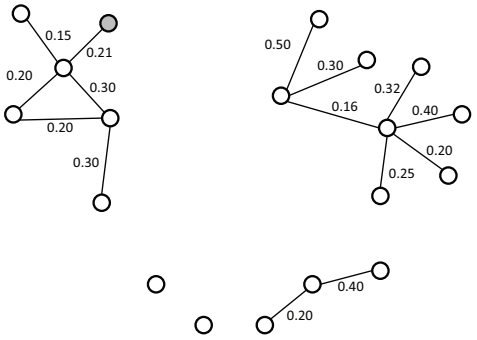
2) The relevance of some vertices changes and, as a consequence, at least one vertex $u \notin C$ appears such that u has relevance greater than at least one vertex in C that covers vertices in $u.Adj \cup \{u\}$. Vertices like u could determine ws-graphs with more satellites and less overlapping with other ws-graphs than other ws-graphs currently belonging to the covering of $\widetilde{G}_\beta$.

Figure 3, illustrates the above commented situations over the graph $\widetilde{G}_\beta$ of Figure 1(a). Figure. 3(a), shows the graph $\widetilde{G}_\beta$ before the changes; the vertices to be removed are marked with an "x". Figure 3(b), shows graph $\widetilde{G}_\beta$ after the changes; vertices filled with light gray represent the added vertices. Figures 3(c) and 3(d), show the updated graph $\widetilde{G}_\beta$ with vertices labeled with letters and with their updated value of relevance, respectively; vertices filled with black correspond with those vertices currently belonging to C. As it can be seen from Figures 3(c) and 3(d), vertices S, F, G, I, H and J became uncovered after the changes, while vertex B, which does not belong to C, has a relevance greater than vertex D, which already belongs to C.

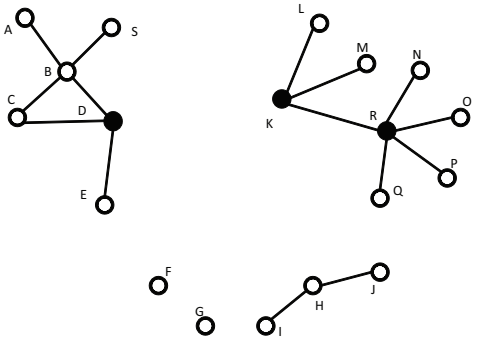
Taking into account the above mentioned situations, in order to update the clustering after changes DClustR first detects which are the connected components of $\widetilde{G}_\beta$ that were affected by changes and then it iteratively updates the covering of these components and consequently, their clus-



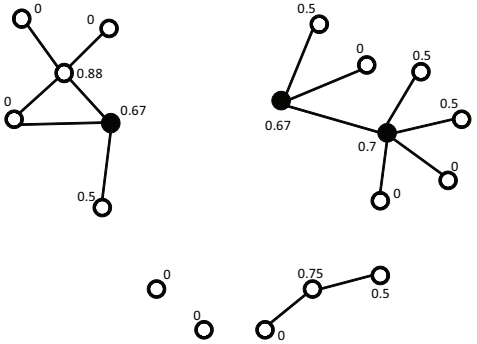
(a) $\widetilde{G}_\beta$ before the changes



(b) $\widetilde{G}_\beta$ after the changes



(c) $\widetilde{G}_\beta$ with vertices labeled with letters



(d) $\widetilde{G}_\beta$ with vertices labeled with their updated value of relevance

Figure 3: Illustration of how some changes in the collection affect the current covering of the graph $\widetilde{G}_\beta$ of Figure 2(b).

tering.

The connected components that are affected by changes are those that contain vertices that were added or vertices that were adjacent to vertices that were deleted from $\widetilde{G}_\beta$. Since DClustR has control over these vertices it can build these components through a depth first search, starting from any of these vertices. Let $G' = \langle V', E', S\rangle$ be a connected component affected by changes, whose covering must be updated. Let $C' \subseteq C$ be the set of vertices of G' which determine ws-graphs (i.e., clusters) covering G'. DClustR follows the same principles of OClustR; that is, it first builds or completes the covering of G' in order to build an initial set of clusters (stage 1) and then, it improves these clusters in order to build the final set of clusters of G' (stage 2). In fact, DClustR uses the same steps that OClustR for the above two mentioned stages, but unlike OClustR, DClustR modifies the way in which the candidate list L, used in stage 1, is built.

In order to build candidate list L, DClustR first recomputes the relevance value of all vertices in G' and it empties the list $c.Linked$, for all vertices $c \in C'$; this last action is supported by the fact that, after changes, there could be ws-graphs that were considered as non useful, which could be no longer so. Let $V_+ \subseteq (V' \setminus C')$ be the set of vertices of G' with relevance greater than zero, which do not belong to C'. For building the candidate list L, both C' and V_+ are processed.

For processing V_+, DClustR visits each vertex $v \in V_+$ and it verifies a) if v is uncovered, or b) if at least one adjacent vertex of v is uncovered, or c) if there is at least one vertex $u \in v.Adj$, such that there is no other vertex in C' covering u whose relevance is greater than or equal to the relevance of v. If any of these three conditions is fulfilled, v is added to L. Additionally, if the last condition is fulfilled, all those vertices like u are marked as "activated" in order to use them when C' is being processed. The computational complexity of the processing of V_+ is $O(n^2)$.

For processing C', DClustR visits the adjacent vertices of each vertex $v \in C'$. Any vertex $u \in v.Adj$ having greater relevance than v is added to L; in these cases, v is additionally marked as "weak". Once all the adjacent vertex of v were visited, if v was marked as "weak" or at least one of its adjacent vertices were previously marked as "active", v is removed from C' since it could be substituted by a more relevant vertex. However, if v has a relevance greater than zero, it is still considered as a candidate and consequently, it is added to L. The computational complexity of the processing of C' is $O(n^2)$.

Figure 4, shows the updated set of overlapping clusters obtained by DClustR when it processes the graph in Figure 3(d); vertices filled with black represent the vertices determining ws-graphs that cover each connected component of $\widetilde{G}_\beta$.

Like OClustR, the computational complexity of DClustR is $O(n^2)$.

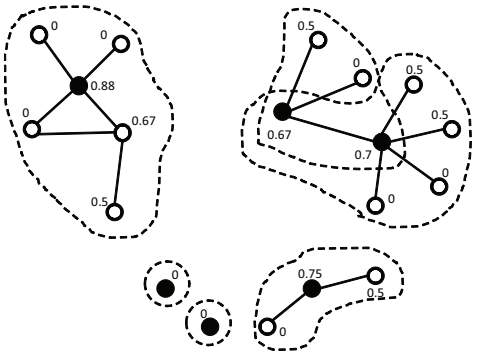


Figure 4: Updated set of overlapping clusters obtained by DClustR.

3 Proposed parallel algorithms

As it was mentioned in Section 1, despite the good achievements attained by OClustR and DClustR in the task of documents clustering, these algorithms are $O(n^2)$ so they could be less useful in applications dealing with a very large number of documents. Motivated by this fact, in this section two massively parallel implementations in CUDA of OClustR and DClustR are proposed in order to enhance the efficiency of OClustR and DClustR in the above mentioned problems. These parallel algorithms, namely *CUDA-OClus* and *CUDA-DClus*, take advantage of the benefits of GPUs, like for instance, the high bandwidth communication between CPU and GPU, and the GPU memory hierarchy.

Although in their original articles both OClustR and DClustR were proposed as general purpose clustering algorithms, the parallel extensions proposed in this work are specifically designed for processing documents. This application context is the same in which both OClustR and DClustR were evaluated and it is also a context in which very large collections are commonly processed. In the context of document processing, both CUDA-OClus and CUDA-DClus use the cosine measure [9] for computing the similarity between two documents; this measure is the function that has been used the most for this purpose [10]. The cosine measure between two documents d_i and d_j is defined as:

$$cos(d_i, d_j) = \frac{\sum_{k=1}^{m} d_i(k) * d_j(k)}{\|d_i\| \cdot \|d_j\|}, \qquad (1)$$

where $d_i(k)$ and $d_j(k)$ are the weights of the k term in the description of the documents d_i and d_j, respectively; $\|d_i\|$ and $\|d_j\|$ are the norms of documents d_i and d_j, respectively.

In experiments conducted over several document collections, it was verified that the first stage of OClustR, the construction of the similarity graph, consumes the 99% of the processing time of the algorithm. The remaining 1% is mainly dominated by the computation of the relevance of the vertices. Based on this fact, the above two mentioned steps are the ones that will be implemented in CUDA

by CUDA-OClus; remaining steps are high memory consuming tasks that are more favored with a CPU implementation. Analogously, in these experiments it was also verified that the most time consuming steps of DClustR are the updating of the graph after changes and the recomputing of the relevance, so these steps will be implemented in CUDA by CUDA-DClus. In this case, it could be noticed also that the detection of the connected components affected by changes is a high memory consuming task performed by DClustR, so it is also important to address this problem in CUDA-DClus.

Finally, it is also important to mention that since we are dealing with the problem of processing very large document collections, CUDA-DClus only tackles additions, which are the changes that could increase the size of the collection. Implementing deletions is irrelevant for overcoming problems related with large document collections.

Following, the CUDA-OClus algorithm is first introduced and then, the CUDA-DClust algorithm is presented.

3.1 CUDA-OClus algorithm

Let $D = \{d_1, d_2 \ldots, d_n\}$ be a collection of documents described by a set of terms. Let $T = \{t_1, t_2, \ldots, t_m\}$ be the list containing all the different terms that describe at least one document in D. CUDA-OClus represents a document $d_i \in D$ by two parallel vectors, denoted by T_{d_i} and W_{d_i}. The first one contains the position that the terms describing d_i have in T, and the second one contains the weights that those terms have in the description of d_i.

For building $\widetilde{G}_\beta = \langle V, \widetilde{E}_\beta, S\rangle$, OClustR demands S to be a symmetric similarity measure, so the similarity between any two documents (i.e., vertices in $\widetilde{G}_\beta$) needs to be computed only once. Based on this fact and considering the inherent order the documents have inside a collection D (i.e., vertices in V), for building the edges relatives to a vertex $v \in V$ it is only necessary to compute the similarity between v and each vertex following v in V. Let Suc_v be the list of vertices that follow a vertex v in V. To speed up the construction of $\widetilde{G}_\beta$, for each vertex $v \in V$, CUDA-OClus will compute in parallel the similarity between v and the vertices in Suc_v.

Considering the definition of the cosine measure, it can be seen from Expression (1) that its numerator is a sum of independent products which could be computed all at once. On the other hand, taking into account that the norm of a document can be computed while the document is being read, the denominator of Expression (1) can be also resolved with no extra time. Based on these facts, CUDA-OClus also parallelizes the computation of the similarity between a pair of vertices, in order to speed up even more the construction of $\widetilde{G}_\beta$.

In order to carry out the previous idea, CUDA-OClus builds a *grid* comprised of k square blocks, each block having a shared memory square matrix (SMM); where $k = \frac{n}{t} + 1$ and t is the dimension of both the blocks and the matrices. A *grid* is a logic representation of a matrix of threads in the GPU. The use of SMM and its low latency will allow CUDA-OClus to not constantly access the CPU memory, speeding up the calculus of the similarity between two vertices. CUDA-OClus assigns to t the maximum value allowed by the architecture of the GPU for the dimension of a SMM.

When CUDA-OClus is going to compute the similarity between a vertex v and the vertices in Suc_v, it first builds a vector P_v of size m. This vector has zero in all its entries excepting in those expressed by the positions stored in T_v; these last entries contain their respective weights stored in W_v. Once P_v has been built, the list Suc_v is partitioned into k sublists. Each one of these sublists is assigned to a block constituting the grid and the SMM associated with that block is emptied; i.e., all its cells are set to zero. When a sublist $Q = \{v_1, v_2, \ldots, v_p\}$ is assigned to a block inside a grid, each vertex in Q is assigned to a column of the block. In this context, to assign a vertex v_i to a column means that each row of the column points to a term describing v_i; in this way, the jth row points to the jth term describing v_i. Figure 5 shows an example of how the list Suc_v is divided by CUDA-OClus into k sublists and how these sublists are assigned to the blocks constituting the grid. The example on Figure 5 shows how the first vertex of the sublist assigned to "block 0" is assigned to the first column of that block; the other assignments could be deduced from this example.

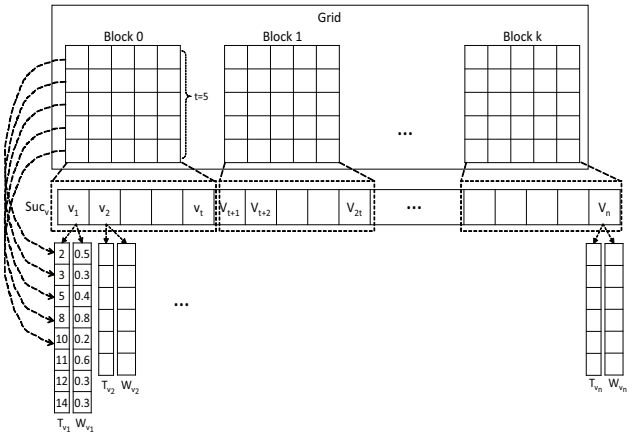


Figure 5: Illustration of how CUDA-OClus divides Suc_v and assigns each resulting sublist to the blocks.

Each row inside a column of a block has a thread that performs a set of operations. In our case, the threads associated with the ith column will compute the similarity between v and its assigned vertex v_i. With this aim, the thread associated with each row inside the ith column will compute the product between the weight that the term pointed by that row has in the description of v_i, and the weight this same term has in the description of vertex v. It is important to note that although the sum in the numerator of Expression (1) runs over all the terms in T, the products that will be different from zero are only those between terms shared

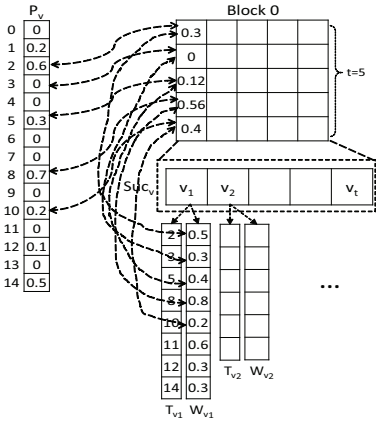
by both documents; this is the reason we only use the terms of v_i and multiply their weights by the weights that these terms have in v; remaining terms in v are useless.

Given that the jth row of the column to which vertex v_i has been assigned, points to the jth term of T_{v_i}, the weight this term has in v_i is stored at the jth position of W_{v_i} and the weight this same term has in v is stored at P_v, in the entry referred by the value stored at the jth position of T_{v_i}. The result of the product between the above mentioned weights is added to the value the jth row already has in the SMM. If the description of a vertex v_i assigned to a column of a block exceeds the length of the column (i.e., t) a *tiling* is applied at this block. Tiling [11] is a technique that consists on dividing a data set into a number of small subsets, such that each subset fits into the block; i.e., the SMM. Thus, when the rows of a column point at the next t terms, the products between the weights these terms have in the description of v_i and v are computed and accumulated into the values these rows have in the SMM. This technique is applied until all the terms describing the vertices assigned to the columns have been processed. Figure 6 shows how the similarity between the vertex v_1 assigned to the first column of "Block 0" and v is computed. In this example, it has been assumed that there are 15 terms describing the documents of the collection, the size of the block is $k = 5$, and $T_v = \{1, 2, 5, 8, 10, 12, 14\}$, $W_v = \{0.2, 0.6, 0.3, 0.7, 0.2, 0.1, 0.5\}$, $T_{v_1} = \{2, 3, 5, 8, 10, 11, 12, 14\}$ and $W_{v_1} = \{0.5, 0.3, 0.4, 0.8, 0.2, 0.6, 0.3, 0.3\}$.

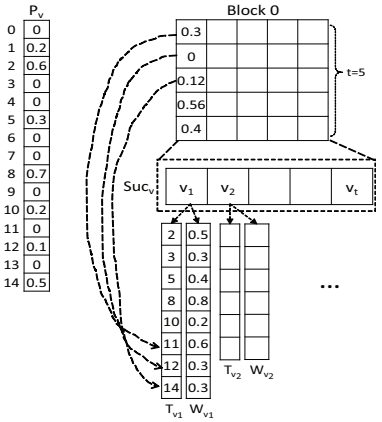
As it can be seen from Figure 6(a), each thread of the t rows of the first column computes the product between the weight of the term it points at, and the weight this same term has in P_v (i.e., the description of v). As it was mentioned before, the computed products are stored in the SMM of that block. Note from Figure 6(a) that the product computed by the second row is zero since vertex v does not contain the term pointed out by this row; i.e., term having index 3rd in T. Figure 6(b) shows how when Tiling is applied, the remaining terms describing v_1 are pointed by the rows of the first column. Figure 6(c) shows how the products between the remaining terms of v_1 and v are performed. Finally, Figure 7 shows which are the values stored in the first column of the SMM of "Block 0", once all the products have been computed.

Once all the terms describing the vertices assigned to a block have been processed, a *reduction* is applied over each column of the block. Reduction [12] is an operation that computes in parallel the sum of all the values of a column of the SMM and then, it stores this sum in the first row of the column. Figure 8 shows the final sum obtained for the first column of "Block 0".

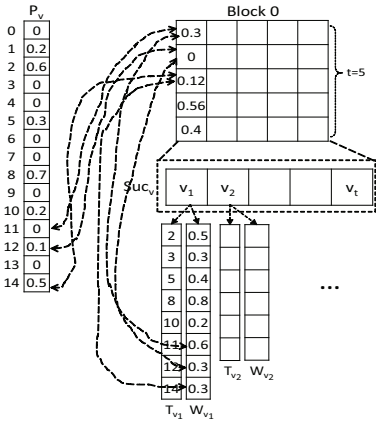
The sum obtained on the column to with vertex v_i has been assigned corresponds with the numerator of the cosine measure between v and v_i. This sum is then divided by the product of the norms of v and v_i, which have been previously computed; the result of this division (i.e., the similarity between v and v_i) is copied to the CPU. Using this result CUDA-OClus decides if it should create or not an edge between v and v_i, during this step CUDA-OClus also updates the value of $AIS(v)$ and $AIS(v_i)$.



(a) Computing the first t products



(b) Applying *Tiling*



(c) Computing remaining products

Figure 6: Illustration of how CUDA-OClus computes the similarity between a vertex v and the vertices in Suc_v.

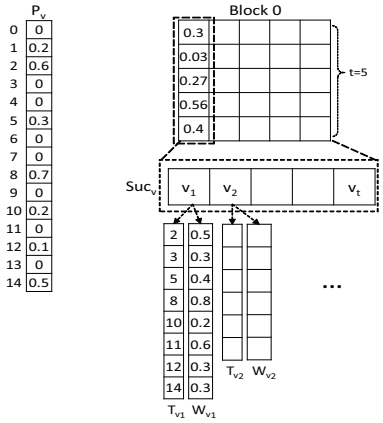


Figure 7: Final results stored in the SMM after processing all terms of v_1.

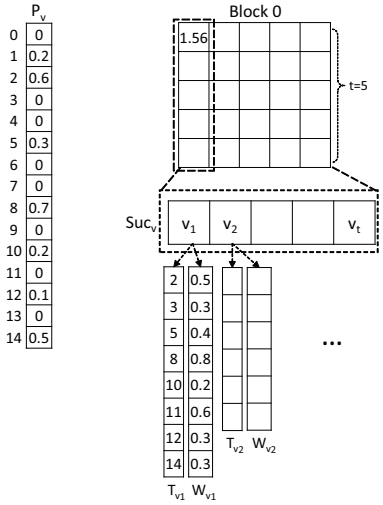


Figure 8: Result of applying Reduction on the first column of "Block 0".

The pseudocode of cosine similarity function is shown in Algorithm 1.

Once the thresholded similarity graph $\widetilde{G}_\beta$ has been built, CUDA-OClus speeds up the computation of the other time-consuming step: the computation of the relevance of the vertices. In order to do that, CUDA-OClus computes in parallel the relevance of all the vertices of $\widetilde{G}_\beta$. Moreover, for each vertex v, CUDA-OClus computes in parallel the contribution each adjacent vertex of v has over the relevance of v, speeding up even more the computation of the relevance of v. In order to accomplish this idea, the list of vertices of $\widetilde{G}_\beta$ is partitioned into k sublists and each sublist is assigned to a block inside a grid. However, in this case, when a vertex v_i of a sublist is assigned to a column of a block, each row in that column will point to an adjacent vertex of v_i; e.g., the jth row points at the jth adjacent vertex of v_i. Different from building graph $\widetilde{G}_\beta$, now the threads associated with a column will compute the relevance of its assigned vertex. With this aim, the thread on each row of that column will compute the contributions the vertex pointed by that row has over the relevance of the vertex assigned to the column.

Algorithm 1: CUDA implementation of the cosine similarity function.

Input: Suc_v the list of vertices that follow a vertex v, P_v weights of vertex v, W_v weights associated to v, T_v position of terms that represent to v, $Norm_v$ is the norm of v
Output: $similarity$: cosine similarity values between v and Suc_v

```
1  __shared__ float SMM[R][C];   // R = C because block are squared
2  int tid = threadIdx.x + blockDim.x * blockIdx.x;
3  if (tid < |Suc_v|) then
4      u = Suc_v[tid];
5      int tid_y = threadIdx.y;
6      float sum = 0;
7      while (tid_y < |W_u|) do
           /* Accumulating the multiplication between
              weights of v and u                          */
8          sum += W_u[tid_y] * P_v[T_u[tid_y]];
9          tid_y += R;        // Applying tiling technique
10     SMM[threadIdx.y][threadIdx.x] = sum;
   /* Waiting that whole threats compute
      multiplications between weights of v and
      u ∈ Suc_v                                           */
11 __syncthreads();
   /* Applying reduction technique to calculate
      Σ_{k=1}^{m} d_i(k) * d_j(k)                         */
12 int i = R/2;
13 while (i != 0) do
14     if (threadIdx.y < i) then
15         SMM[threadIdx.y][threadIdx.x] +=
             SMM[threadIdx.y + i][threadIdx.x];
16     __syncthreads();
17     i = i/2;
18 if (threadIdx.y == 0 && tid < |Suc_v|) then
19     similarity[tid] = 0;
20     if (Norm_v > 0 && Norm_u > 0) then
           /* Dividing between the multiplication of
              norms of v and u                            */
21         similarity[tid] =
             SMM[0][threadIdx.x]/(Norm_v * Norm_u)
```

Let v be a vertex assigned to a column and u one of its adjacent vertices. Vertex u contributes $\frac{1}{|v.Adj|}$ to the relevance of v if $|v.Adj| \geq |u.Adj|$; otherwise, its contribution is zero. This case represents the contribution u has to the relevance of v through the relative density of v. On the other hand, u contributes $\frac{1}{|v.Adj|}$ to the relevance of v if $AIS(v) \geq AIS(u)$; otherwise, its contribution is zero. This other case represents the contribution u has to the relevance of v through the relative compactness of v. The total contribution provided by a vertex is added to the value the row already has in the SMM; similar to the case of building graph $\widetilde{G}_\beta$, the SMM of each block is initially emptied. If v has more than t adjacent vertex, a Tiling is applied. Once all the adjacent vertices of v has been processed, a Reduction is applied in order to compute the relevance of v. Obtained values are then copied to the CPU.

The pseudocode of cosine similarity function is shown in Algorithm 2.

As it was mentioned before, the remaining steps of OClustR were not implemented in CUDA because they are

Algorithm 2: CUDA implementation of the relevance function.

Input: $\widetilde{G}_\beta$ weights threshold similarity graph,$AIS(\widetilde{G}_\beta)$ is the approximated intra-cluster similarity of $\widetilde{G}_\beta$
Output: $relevance$: relevance values of vertices

```
1  __shared__ float SMM[R][C];   // R = C because block are squared
2  int tid = threadIdx.x + blockDim.x * blockIdx.x;
3  if (tid < |V|) then
4      int tid_y = threadIdx.y;
5      float sum = 0;
6      while (tid_y < |Adj[tid]|) do
           /* Checking if the density and compactness
              conditions are met                          */
7          if (|Adj[tid_y]| ≤ |Adj[tid]|) then
8              sum+ = 1;
9          if (|AIS[tid_y]| ≤ |AIS[tid]|) then
10             sum+ = 1;
11         tid_y += R;        // Applying tiling technique
12     SMM[threadIdx.y][threadIdx.x] = sum;
   /* Waiting that whole threats check density and
      compactness conditions                              */
13 __syncthreads();
   /* Applying reduction technique to calculate
      relevance                                           */
14 int i = R/2;
15 while (i != 0) do
16     if (threadIdx.y < i) then
17         SMM[threadIdx.y][threadIdx.x] +=
             SMM[threadIdx.y + i][threadIdx.x];
18     __syncthreads();
19     i = i/2;
20 if (threadIdx.y == 0 && tid < |V|) then
21     relevance[tid] = 0;
22     if (|Adj[tid] > 0|) then
           /* Dividing between the number of
              adjacents of the current vertex             */
23         relevance[tid] =
             SMM[threadIdx.y][threadIdx.x]/(2 * |Adj[tid]|);
```

more favored with a CPU implementation since they are high memory consumption tasks.

3.2 CUDA-DClus algorithm

In order to update an already clustered collection when changes take effect, in our case additions, DClustR first detects, in the graph $\widetilde{G}_\beta$ representing the collection, which are the connected components that were affected by changes and then, it updates the cover of those components and consequently, the overall clustering of the collection.

As it was stated in Section 2.2, the connected components affected by additions are those containing at least one added vertex. Thus, each time vertices are added to $\widetilde{G}_\beta$, in addition to computing the similarity between these vertices and those already belonging to $\widetilde{G}_\beta$ in order to create the respective edges, DClustR also needs to build from scratch each affected connected component in order to update their covers. In order to reduce the amount of information DClustR needs to store in memory, CUDA-DClus proposes to represent the graph $\widetilde{G}_\beta$ using an array of *partial connected components*, named Arr_{PCC}, and two parallel arrays. The first of these parallel arrays, named V, contains the vertices in the order in which they were added to $\widetilde{G}_\beta$. The second array, named PC_V, contains the index of the partial connected component to which each vertex belongs. This new representation allows CUDA-DClus to not need to rebuild the affected components each time the collection changes, but keeping the affected components updated each time vertices are added to the graph $\widetilde{G}_\beta$, with no extra cost.

Let $\widetilde{G}_\beta = \langle V, \widetilde{E}_\beta, S\rangle$ be the thresholded similarity graph representing the collection of documents. A *partial connected component* (PCC) in $\widetilde{G}_\beta$ is a connected subgraph induced by a subset of vertices of $\widetilde{G}_\beta$. A partial connected component is represented using two arrays: one array containing the indexes the vertices belonging to that component have in $\widetilde{G}_\beta$, and the other array containing the adjacency list of the aforementioned vertices.

The array of partial connected components representing $\widetilde{G}_\beta$ is built once while $\widetilde{G}_\beta$ is being constructed. The strategy used by CUDA-DClus for this purpose is as follows. In the first step, CUDA-DClus adds a vertex in V for each document of the collection and then, PC_V is emptied (i.e., it is filled with -1), meaning that the vertices do not belong to any PCC yet. In the second step, CUDA-DClus processes each vertex $v_i \in V$. If v_i does not belong yet to a PCC, CUDA-DClus creates a new PCC and it puts v_i in this component; when a vertex v is added to a PCC, the index this PCC has in the array Arr_{PCC} is stored in the array PC_V, at the entry referred to by the index v has in array V; this is the way CUDA-DClus uses to indicate that now v belongs to a PCC. Following, CUDA-DClus computes the similarity between v_i and the vertices in Suc_{v_i}, using the strategy proposed by CUDA-OClus. Once these similarities have been computed, CUDA-DClus visits each vertex $u \in Suc_{v_i}$. If $S(v_i, u) \geq \beta$ and u does not belong to any PCC yet, then u is inserted in the PCC to which v_i belongs and the adjacency lists of both vertices u and v_i are modified in order to indicate they are similar to each other; otherwise, if u already belongs to a PCC only the adjacency lists of both vertices are modified. In this last case, if the partial connected components to which both v_i and u belong are not the same, we will say that these partial connected components are *linked*.

As an example, let $\widetilde{G}_\beta$ be initially empty and $D = \{d_1, d_2, \ldots, d_9\}$ be the set of documents that will be added to the collection. For the sake of simplicity, we will assume that CUDA-DClus already added the documents in $\widetilde{G}_\beta$ as vertices and that the similarities existing between each pair of documents are those showed in Table 1. Taking into account the above mentioned information, Figures 9, 10, 11 and 12 exemplify how CUDA-DClus builds the array of partial connected components representing graph $\widetilde{G}_\beta$, for $\beta = 0.3$.

As it can be seen from Figure 9, firstly CUDA-DClus processes vertex v_1 in order to build the first PCC. As the result of the above process vertices v_3 and v_5 are added to the first PCC, which is now constituted by vertices v_1, v_3 and v_5. The second PCC is built when vertex v_2 is processed, see Figure 10; this component is finally constituted by vertices v_2, v_4 and v_7. Afterwards, as it can be seen

Vert./Vert.	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
v_1	-	0	0.4	0	0.5	0	0	0	0
v_2	0	-	0	0.4	0	0	0.5	0	0
v_3	0.4	0	-	0.7	0.6	0.3	0	0	0
v_4	0	0	0.7	-	0	0	0	0	0
v_5	0.5	0	0.6	0	-	0	0	0	0
v_6	0	0	0.3	0	0	-	0	0	0
v_7	0	0	0	0	0	0	-	0	0
v_8	0	0	0	0	0	0	0	-	0.5
v_9	0	0	0	0	0	0	0	0.5	-

Table 1: Similarities existing between each pair of vertices of the example.

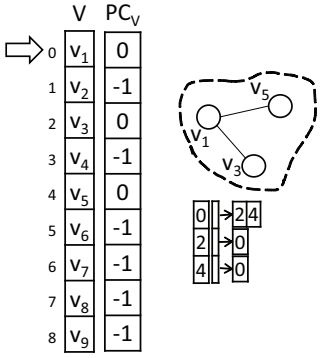


Figure 9: Processing vertex v_1.

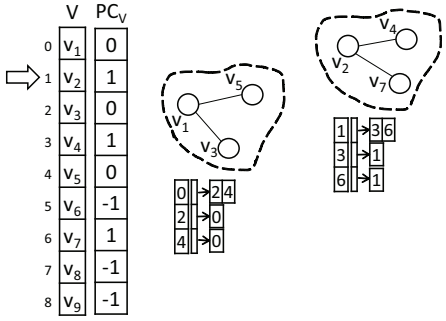


Figure 10: Processing vertex v_2.

from Figure 11, when vertex v_3 is being processed, CUDA-DClus updates the first PCC by adding vertex v_6 and updating the adjacency list of vertices v_3 and v_5; CUDA-DClus also updates the second PCC by modifying the adjacency list of vertex v_4, which is similar to vertex v_3. In this example, these two partial connected components were joined by a dash line in order to illustrate the fact that they are linked since vertices v_3, belonging to the first PCC, and v_4, belonging to the second PCC, are similar. Finally, the third PCC is created when CUDA-DClus processes vertex v_8,

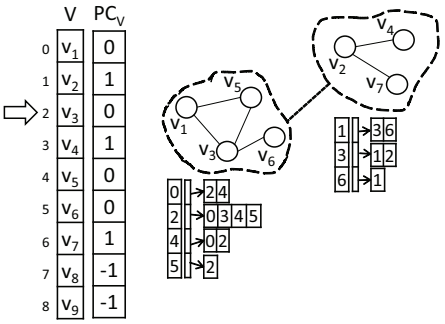


Figure 11: Processing vertex v_3.

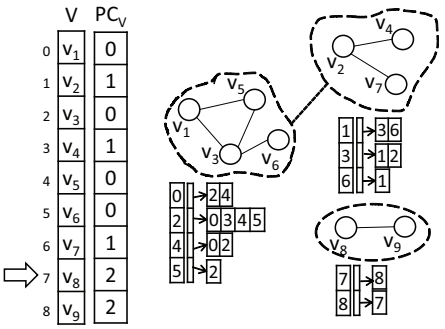


Figure 12: Processing vertex v_8.

as it can be seen in Figure 12. The processing of vertices v_4, v_5, v_6, v_7 and v_9 does not affect the partial connected components built so far, therefore, it was not included in the example.

We would like to emphasize two facts about the above commented process. The first fact is that, since this is the first time the array Arr_{PCC} representing $\widetilde{G}_\beta$ is built, all these components are already in system memory. The second fact is that if we put a PPC $P_i \in Arr_{PCC}$ into a set Q_{P_i} and then, iteratively we add to Q_{P_i} all the linked PCC of each PCC belonging to Q_{P_i}, the resulting set is a connected component. Proof is straightforward by construction. Hereinafter, we will say that Q_{P_i} is the connected component induced by PCC P_i.

Once the array Arr_{PCC} representing $\widetilde{G}_\beta$ was built, CUDA-DClus processes each of its partial connected components in order to build the clustering. For processing a PCC $P_i \in Arr_{PCC}$ that has not been processed in a previous iteration, CUDA-DClus first builds Q_{P_i} and then, CUDA-DClus recomputes the relevance of the vertices be-

longing to this component using the strategy proposed by CUDA-OClus. Once the relevance of the vertices have been recomputed, CUDA-DClus follows the same steps used by DClustR for updating the covering and consequently, the clustering of Q_{P_i}. Remaining steps of DClustR were not implemented in CUDA because they are more favored with a CPU implementation. Once the clustering has been updated, CUDA-DClus stores the existing partial connected components in the hard drive, releasing in this way the system memory.

Once $\widetilde{G}_\beta$ changes due to the additions of documents to the collection, CUDA-DClus updates the array Arr_{PCC} representing $\widetilde{G}_\beta$ and then, it updates the current clustering. In order to update the array Arr_{PCC}, CUDA-DClus adds for each incoming document, a vertex in $\widetilde{G}_\beta$ and then, CUDA-DClus sets to -1 the entries that these vertices occupy in PC_V, in order to express that they do not belong to any PCC yet. Let $M = \{v_1, v_2, \ldots, v_k\}$ be the set of added vertices. Afterwards, for processing a vertex $v_i \in M$, CUDA-DClus slightly modifies the strategy it uses for creating the partial connected components. Now, rather than computing the similarity of v_i only with the vertices that came after v_i in V (i.e., Suc_{v_i}), CUDA-DClus also computes the similarity of v_i with respect to the vertices that belong to $\widetilde{G}_\beta$ before the changes; that is, the similarities are now computed between v_i and each vertex in $Suc_{v_i} \cup (V \setminus M)$. Remaining steps are the same.

Let $D_1 = \{d_{10}, d_{11}, \ldots, d_{15}\}$ be the set of documents that were added to the collection represented by graph $\widetilde{G}_\beta$, whose array of partial connected components was built in Figure 9, and let $v_{10}, v_{11}, \ldots, v_{15}$ be the vertices that were consequently added in $\widetilde{G}_\beta$ by CUDA-DClus. For the sake of simplicity, in the example it is assumed that none of the vertices belonging to $\widetilde{G}_\beta$ before the changes is similar to the added vertices, with the only exception of v_2 whose similarity with v_{10} is 0.5. Table 2 shows the similarities between each pair of the added vertices. Figures 13, 14, 15 and 16 show, assuming $\beta = 0.3$, how CUDA-DClus updates the array of partial connected components representing $\widetilde{G}_\beta$ after the above mentioned additions. In these figures, vertices filled with light gray are those that were added to the collection.

	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}
v_{10}	-	0.4	0.3	0.6	0	0
v_{11}	0.4	-	0	0.4	0	0
v_{12}	0.3	0	-	0.4	0	0
v_{13}	0.6	0.4	0.4	-	0	0
v_{14}	0	0	0	0	-	0.5
v_{15}	0	0	0	0	0.5	-

Table 2: Similarities existing between each pair of added vertices.

As it can be seen in Figure 13, firstly, CUDA-DClus processes vertex v_{10} and, as a result of this processing, another PCC is created for containing vertices v_{10}, v_{11}, v_{12} and v_{13}. This new PCC was joined with the PCC determined by ver-

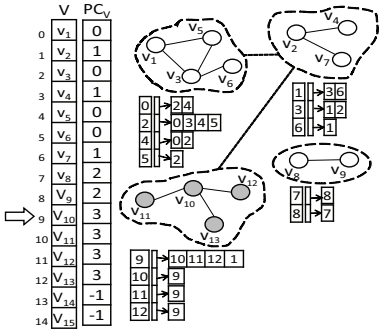


Figure 13: Processing vertex v_{10}.

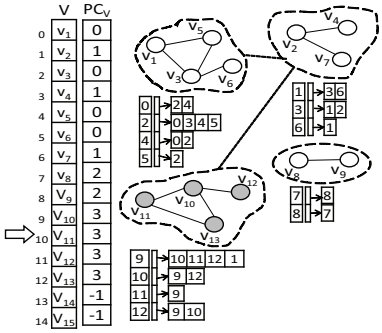


Figure 14: Processing vertex v_{11}.

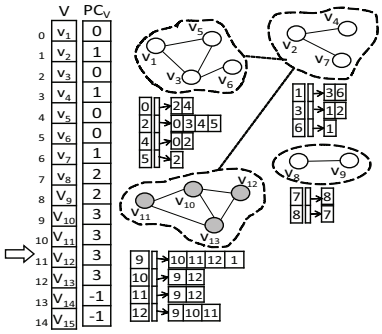


Figure 15: Processing vertex v_{12}.

tex v_2, through a dash line, in order to reflect the fact that they are linked since vertices v_2 and v_{10} are similar. Furthermore, as it can be seen in Figures 14 and 15, this fourth PCC is updated when vertices v_{11} and v_{12} are processed, in order to reflect the fact that they are similar to vertex v_{13}. Finally, a fifth PCC is created when vertex v_{14} is processed,

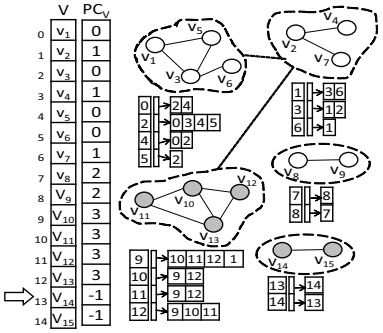


Figure 16: Processing vertex v_{14}.

see Figure 16; this PCC contains vertices v_{14} and v_{15}.

Once the array Arr_{PCC} has been updated, CUDA-DClus processes each new PCC following the same strategy commented above, in order to update the current clustering. It is important to highlight that, different from when Arr_{PCC} was created, this time the partial connected components loaded into the system memory are those belonging to the connected components determined by each new created PCC; the other partial connected components remain in the hard drive. Although in the worst scenario an incoming document can be similar to all existing documents in the collection, generally similarity graphs are very sparse so it is expected that the new representation proposed by CUDA-DClus as well as the strategy it uses for updating the array of partial connected components, help CUDA-DClus to save system memory.

4 Experimental results

In this section, the results of several experiments done in order to show the performance of the CUDA-OClus and CUDA-DClus algorithms are presented. The experiments were conducted over eight document collections and were focused on: (1) assessing the correctness of the proposed parallel algorithms wrt. their original non parallel versions, (2) evaluating the improvement achieved by the proposed algorithms with respect to the original OClustR and DClustR algorithms, and (3) evaluating the memory both CUDA-DClus and DClustR consume when they are processing the same collection. All the algorithms were implemented in C++; the codes of OClustR and DClustR algorithms were obtained from their authors. For implementing CUDA-OClus and CUDA-DClus the CUDA Toolkit 5.5 was used. All the experiments were performed on a PC with Core i7-4770 processor at 3.40 GHz, 8GB RAM, having a PCI express NVIDA GeForce GT 635, with 1 GB DRAM.

The document collections used in our experiments were built from two benchmark text collections commonly used in documents clustering: Reuters-v2 and TDT2. The Reuters-v2 can be obtained from http://kdd.ics.uci.edu, while TDT2 benchmark can be obtained from *http://www.nist.gov/speech/tests/tdt.html*. From these benchmarks, eight document collections were built. The characteristics of these collections are shown in Table 3. As it can be seen from Table 3, these collections are heterogeneous in terms of their size, dimension and the average size of the documents they contain.

Coll.	#Docs.	#Terms	Terms/Docs.
Reu-10K	10000	33370	27
Reu-20K	20000	48493	41
Reu-30K	30000	59413	50
Reu-40K	40000	70348	58
Reu-50K	50000	74720	64
Reu-60K	60000	81632	69
Reu-70K	70000	91490	76
Tdt-65K	65945	114828	210

Table 3: Overview of the collections used in our experiments.

In our experiments, documents were represented using the Vector Space Model (VSM) [13]. The index terms of the documents represent the lemmas of the words occurring at least once in the collection; these lemmas were extracted from the documents using Tree-tagger[1]. Stop words such as: articles, prepositions and adverbs were removed. The index terms of each document were statistically weighted using their term frequency. Finally, the cosine measure was used to compute the similarity between two documents [9].

4.1 Correctness evaluation

As it was mentioned before, the first experiment was focused on assessing the correctness of the proposed algorithms. With this aim, we will compare the clusterings built by CUDA-OClus and CUDA-DClus with respect to those built by the original OClustR and DClustR algorithms, under the same conditions. For evaluating CUDA-OClus we selected the Reu-10K, Reu-20K, Reu-30K, Reu-40K and Reu-50K collections; whilst for evaluating CUDA-DClus we selected Reu-10K, Reu-20K and Reu-30K collections. These collections were selected due to they resemble the collections over which both OClustR and DClustR were evaluated in [6] and [7], respectively.

In order to evaluate CUDA-OClus, we executed OClustR and CUDA-OClus over the Reu-10K, Reu-20K, Reu-30K, Reu-40K and Reu-50K collections, using $\beta = 0.25$ and 0.35. We used these threshold values as these values obtained the best results in several collections as reported in the original OClustR [6] and DClustR [7] articles. Then, we took the clustering results obtained by OClustR as *ground truth* and we evaluateds the clustering results obtained by CUDA-OClus in terms of their accuracy, using the FBcu-

[1] http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger

bed [14] and the Normalized Mutual Information (NMI) [15] external evaluation measures.

FBcubed is one of the external evaluation measures most used for evaluating overlapping clustering algorithms and unlike of other external evaluation metrics, it meets with four fundamental constrains proposed in [14] (cluster homogeneity, cluster completeness, rag bag and cluster size vs quantity). On the other hand, NMI is a measure of similarity borrowed from information theory, which has proved to be reliable [15]. Both metrics take values in $[0, 1]$, where 1 means identical results and 0 completely different results. In order to take into account the inherent data order dependency of CUDA-OClus, we executed CUDA-OClus twenty more times over the above mentioned collections, for each parameter value, varying the order of their documents. Table 4 shows the average FBcubed and NMI values attained by CUDA-OClus for each selected collection, using $\beta = 0.25$ and 0.35.

FBCubed					
Threshold	Reu-10K	Reu-20K	Reu-30K	Reu-40K	Reu-50K
β**=0.25**	0.999	0.999	1.000	0.998	1.000
β**=0.35**	0.999	1.000	1.000	1.000	0.999
NMI					
Threshold	Reu-10K	Reu-20K	Reu-30K	Reu-40K	Reu-50K
β**=0.25**	0.997	0.999	1.000	0.999	1.000
β**=0.35**	0.998	1.000	1.000	1.000	0.999

Table 4: Average FBcubed and NMI values attained by CUDA-OClus for each selected collection.

As it can be seen from Table 4, the average FBcubed and NMI values attained by CUDA-OClus are very close to 1, meaning that the clusters CUDA-OClus builds are almost identical to those built by OClustR. The differences between the clusterings are caused by the inherent data order dependency of the algorithms and also because of the different floating point arithmetic used by CUDA.

In order to asses the validity of CUDA-DClus, in the second part of the first experiment, we will compare the clustering results built by CUDA-DClus with respect to those obtained by DClustR. With this aim, we obtain a ground truth by executing DClustR over the Reu-30K collection, also using $\beta = 0.25$ and $\beta = 0.35$, and then, we process Reu-20K and Reu-10K collections, in this order, as if they were additions of documents to the collection. That is, we are going to add the documents contained in Reu-20K to the current collection (i.e., Reu-30K) and update the clustering using DClustR and after that, we are goind to add Reu-10K to the collection resulting from previous additions (i.e., Reu-30K union Reu-20K) and update the clustering again. We repeated the above mentioned execution under the same parameter configuration but using CUDA-DClus instead of DClustR and afterwards. Then, we take the results obtained by DClustR as ground truth and we evaluate each of the three clustering results obtained by CUDA-DClus in terms of their accuracy, using the FBcubed and NMI external evaluation measures. Like in the first part of this experiment, we executed CUDA-DClus twenty times under the above mentioned experimental configuration, each time varying the order of the documents inside the collections. Table 5 shows the average FBcubed and NMI values attained by CUDA-DClus for each selected collection, using $\beta = 0.25$ and 0.35.

FBCubed			
Threshold	Reu-30K	Reu-30K+Reu-20K	Reu-30K+Reu-20K+Reu-10K
$\beta = 0.25$	0.999	0.995	0.998
$\beta = 0.35$	0.995	0.996	0.991
NMI			
Threshold	Reu-30K	Reu-30K+Reu-20K	Reu-30K+Reu-20K+Reu-10K
$\beta = 0.25$	0.998	0.994	0.999
$\beta = 0.35$	0.997	0.998	0.995

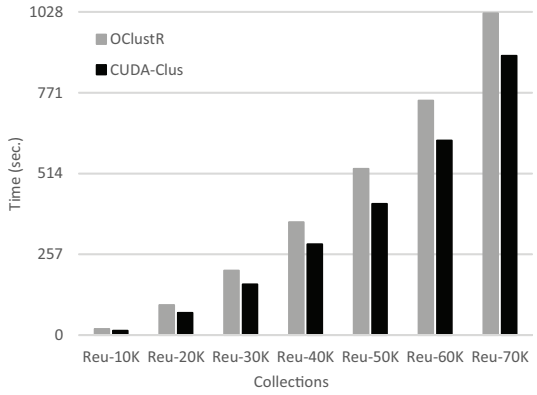
Table 5: Average FBcubed and NMI values attained by CUDA-DClus for each selected collection.

As it can be seen from Table 5, the average FBcubed and NMI values attained by CUDA-DClus are very close to 1, meaning that the clusters it builds are almost identical to those built by DClustR. From this first experiment, we can conclude that the speed-up attained by CUDA-OClus and CUDA-DClus does not degrade their accuracy wrt. the original non parallel versions.

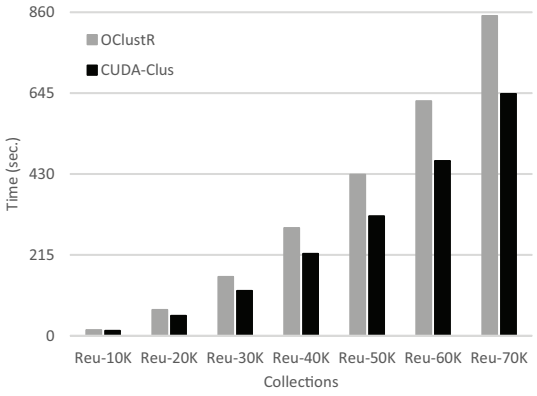
4.2 Execution time evaluation

In the second experiment, we evaluate the time improvement achieved by CUDA-OClus and CUDA-DClus with respect to OClusR and DClustR, respectively. With this aim, we execute both OClustR and CUDA-OClus over Reu-10K, Reu-20K, Reu-30K, Reu-40K, Reu-50K, Reu-60K and Reu-70K, using $\beta = 0.25$ and 0.35 and we measured the time they spent. Like in the previous experiment, in order to take into account the data order dependency of both algorithms, we repeated the above mentioned executions twenty times, for each collection and each parameter configuration, but varying the order of the documents of the collections. Figure 17 shows the average time both OClustR and CUDA-OClus spent for clustering each selected collection, for each parameter configuration.

As it can be seen from Figure 17, CUDA-OClus is faster than OClustR over each selected dataset and for both values of β; for $\beta = 0.25$ and $\beta = 0.35$, CUDA-OClus is respectively 1.26x and 1.29x faster than OClustR. It is important to note from Figure 17 that as the size of the processed collection grows, the difference in the time spent for each algorithm also grows; this behavior shows how well CUDA-OClus scale when the size of the collection grows. We would like to highlight the fact that the specifications of the computer used in the experiments provided advantage to CPU-based algorithms over GPU-based algorithms, since a Core i7-4770 processor at 3.40 GHz with 8GB RAM is superior to a PCI express NVIDA GeForce GT 635, with 1 GB DRAM, which only has two streaming processors and a limited memory. Hence, taking into account the execution model of a GPU, in which the grid blocks are numerated and they distributed among all streaming multiprocessors, which execute simultaneously one



(a) $\beta = 0.25$



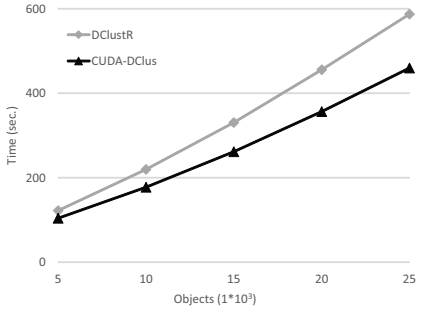
(b) $\beta = 0.35$

Figure 17: Time spent by OClustR and CUDA-OClus for clustering the selected experimental datasets, using $\beta = 0.25$ and 0.35.

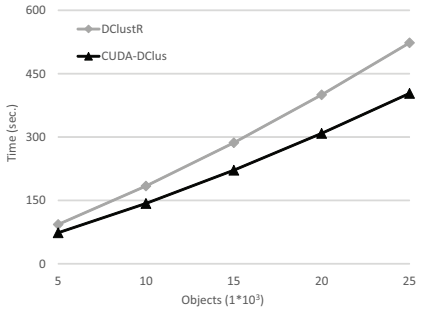
task over a specific block, then we expect that if we use a powerful GPU with more streaming multiprocessor, the difference between the processing time achieved by parallel version and sequential version will be higher than the one showed in this experiments.

In order to compare both DClustR and CUDA-DClus, in the second part of the second experiment, we clustered the Reu-50K collection using both algorithms and then, we measured the time each algorithm spent for updating the current clustering each time N documents of Tdt-65K collection are incrementally added to the existing collection. In this experiment we also used $\beta = 0.25$ and 0.35, and we set $N = 5000$ and $N = 1000$ which are much greater values than those used to evaluate DClustR [7]. In order to take into account the data order dependency of both algorithms, the above mentioned executions were also repeated twenty times, for each collection and each parameter configuration, but varying the order of the documents of the collections. Figure 18 shows the average time both DClustR and CUDA-DClus spent for updating the current clustering, for each parameter configuration.

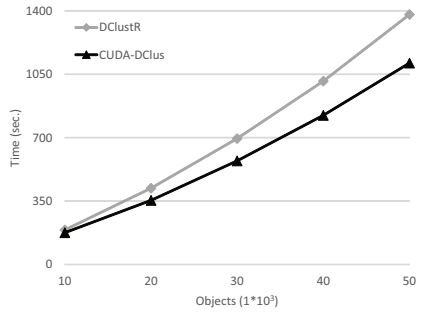
As it can be seen from Figure 18, CUDA-DClus has a better behavior than DClustR, for each parameter configu-



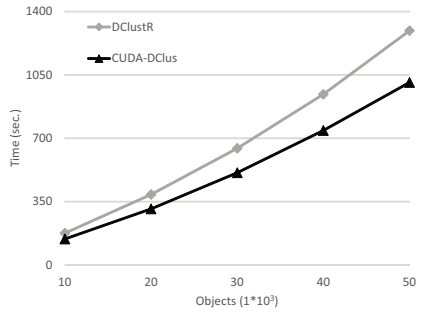
(a) $\beta = 0.25$, $N = 5000$



(b) $\beta = 0.35$, $N = 5000$



(c) $\beta = 0.25$, $N = 10000$



(d) $\beta = 0.35$, $N = 10000$

Figure 18: Time spent by DClustR and CUDA-DClus for updating the current clustering, using $\beta = 0.25$ and 0.35, for $N = 5000$ and 10000.

ration, when multiple additions are processed over the selected dataset, showing an average speed up of 1.25x and 1.29x for $\beta = 0.25, N = 5000$ and $\beta = 0.35, N = 5000$ respectively. Moreover, it also showed an average speed up of 1.19x and 1.26x for $\beta = 0.25, N = 10000$ and

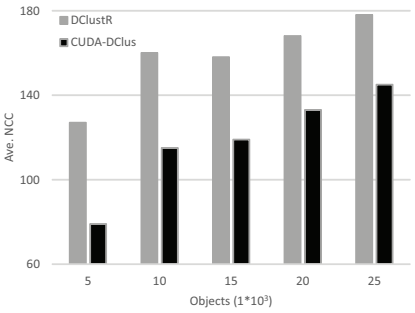
$\beta = 0.35, N = 10000$ respectively. As in the first part of this experiment, it can be seen also from Figure 18, that the behavior of CUDA-DClus, with respect to that of DClustR, becomes better as the size of the collection grows; in this way, we can say that CUDA-DClus also scales well as the size of the collection grows.

4.3 Memory use evaluation

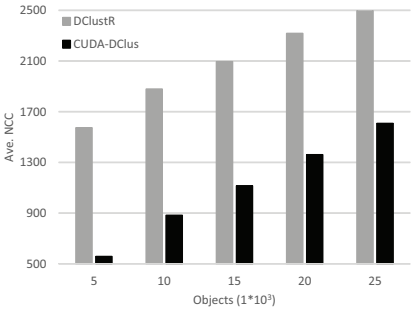
Although the spatial complexity of both algorithm is $O(|V| + \left|\widetilde{E}_\beta\right|)$, the strategy CUDA-DClus proposes for representing $\widetilde{G}_\beta$ should allow to reduce the amount of memory needed to update the clustering each time the collection changes. Thus, in the third experiment, we compare the amount of memory used by CUDA-DClus against that used by DClustR, when processing the changes performed in the second experiment. The amount of connected component loaded by both algorithms when they are updating the current clustering after changes, is directly proportional to the memory used. Based on this, Figure 19 shows the average number of connected components (i.e., Ave. NCC) each algorithm load into system memory, when processing the changes presented in Figure 18, for each parameter configuration.

As it can be seen from Figure 19, CUDA-DClus consumes less memory than DClustR, for each parameter configuration, thereby hence resulting the memory usage of CUDA-DClus is respectively 22.43% for $\beta = 0.25$ and 42.46% for $\beta = 0.35$ less than the one of DClustR. The above mentioned characteristic, plus the fact that CUDA-DClus is also faster than DClustR, makes CUDA-DClus suitable for applications processing large document collections.

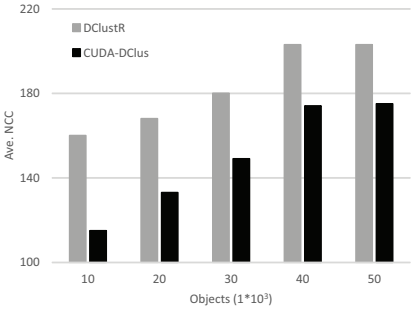
We would like to highlight that in the worst scenario, if the clustering of all the connected components needs to be updated, all the partial connected components will be loaded to system memory and thus, our proposed CUDA-DClus and DClustR will have a similar behavior. Additionally, taking into account the results of experiments in sections 4.1 and 4.2, we can conclude that the strategy proposed for reducing the memory used by CUDA-DClus does not include any considerable cost in the overall processing time of CUDA-DClus or in its accuracy.



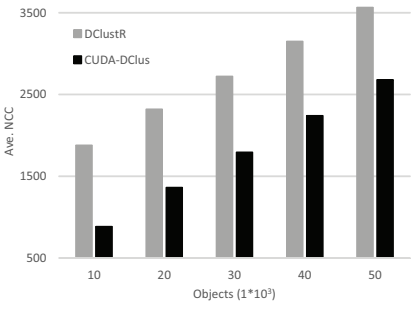
(a) $\beta = 0.25$, $N = 5000$



(b) $\beta = 0.35$, $N = 5000$



(c) $\beta = 0.25$, $N = 10000$



(d) $\beta = 0.35$, $N = 10000$

Figure 19: Average number of connected components DClustR and CUDA-DClus load into system memory when they are updating the current clustering, using $\beta = 0.25$ and 0.35, for $N = 5000$ and 10000.

5 Conclusions

In this paper, we introduced two GPU-based parallel versions of the OClustR and DClustR clustering algorithms, namely CUDA-OClus and CUDA-DClus, specifically tailored for document clustering. CUDA-OClus proposes a strategy in order to speed up the most time consuming steps of OClustR. This strategy is reused by CUDA-DClus in order to speed up the most time consuming steps of DClustR. Moreover, CUDA-DClus proposes a new strategy for representing the graph $\widetilde{G}_\beta$ that DClustR uses for representing the collection of documents. This new representation allows CUDA-DClus to reduce the amount of memory it needs to use and also it helps CUDA-DClus to avoid rebuilding the affected components each time the collection changes but still keep them updated after each changes,

with no extra cost.

The proposed parallel algorithms were compared against their original versions, over several standard document collections. The experiments were focused on: (a) assess the correctness of the proposed parallel algorithms, (b) evaluate the speed-up achieved by CUDA-OClus and CUDA-DClus with respect to OClustR and DClustR, respectively, and (c) evaluate the memory both CUDA-DClus and DClusR consumes when they are processing changes. From the experiments, it can be seen that both CUDA-OClus and CUDA-DClus are faster than OClustR and DClustR, respectively, and that the speed up these parallel versions attain do not degrade their accuracy. The experiments also showed that CUDA-DClus consumes less memory than DClustR, when both algorithms are processing changes over the same collection.

Based on the obtained results, we can conclude that both CUDA-OClus and CUDA-DClus enhance the efficiency of OClustR and DClustR, respectively, in problems dealing with a very large number of documents. These parallel algorithms could be useful in applications, like for instance news analysis, information organization and profiles identification, among others. We would like to mention, that even when the proposed parallel algorithms were specifically tailored for processing documents with the purpose of using the cosine measure, the strategy they propose can be easily extended to work with other similarity or distance measures like, for instance, euclidean and manhattan distances.

As future work, we are going to explore the use in CUDA-OClus and CUDA-DClus of other types of memories in GPU such as texture memory, which is a faster memory than the one both CUDA-OClus and CUDA-DClus are using now. Besides, we are going to evaluate both algorithms over more faster GPU cards, in order to have a better insight of the performance of both algorithms when the number of CUDA cores are increased.

References

[1] E. Bae, J. Bailey, G. Dong, A clustering comparison measure using density profiles and its application to the discovery of alternate clusterings, Data Mining and Knowledge Discovery 21 (3) (2010) 427–471.

[2] A. K. Jain, M. N. Murty, P. J. Flynn, Data clustering: a review, ACM computing surveys (CSUR) 31 (3) (1999) 264–323.

[3] S. Gregory, A fast algorithm to find overlapping communities in networks, in: Machine learning and knowledge discovery in databases, Springer, 2008, pp. 408–423.

[4] J. Aslam, K. Pelekhov, D. Rus, Static and dynamic information organization with star clusters, in: Proceedings of the seventh international conference on Information and knowledge management, ACM, 1998, pp. 208–217.

[5] A. Pons-Porrata, R. Berlanga-Llavori, J. Ruiz-Shulcloper, J. M. Pérez-Martínez, Jerartop: A new topic detection system, in: Progress in Pattern Recognition, Image Analysis and Applications, Springer, 2004, pp. 446–453.

[6] A. Pérez-Suárez, J. F. Martínez-Trinidad, J. A. Carrasco-Ochoa, J. E. Medina-Pagola, Oclustr: A new graph-based algorithm for overlapping clustering, Neurocomputing 121 (2013) 234–247.

[7] A. Pérez-Suárez, J. F. Martínez-Trinidad, J. A. Carrasco-Ochoa, J. E. Medina-Pagola, An algorithm based on density and compactness for dynamic overlapping clustering, Pattern Recognition 46 (11) (2013) 3040–3055.

[8] L. J. G. Soler, A. P. Suárez, L. Chang, Efficient overlapping document clustering using gpus and multicore systems, in: Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications - 19th Iberoamerican Congress, CIARP 2014, Puerto Vallarta, Mexico, November 2-5, 2014. Proceedings, 2014, pp. 264–271.

[9] M. W. Berry, M. Castellanos, Survey of text mining, Computing Reviews 45 (9) (2004) 548.

[10] R. Gil-García, A. Pons-Porrata, Dynamic hierarchical algorithms for document clustering, Pattern Recognition Letters 31 (6) (2010) 469–477.

[11] J. Sanders, E. Kandrot, CUDA by example: an introduction to general-purpose GPU programming, Addison-Wesley Professional, 2010.

[12] D. B. Kirk, W. H. Wen-mei, Programming massively parallel processors: a hands-on approach, Newnes, 2012.

[13] G. Salton, A. Wong, C.-S. Yang, A vector space model for automatic indexing, Communications of the ACM 18 (11) (1975) 613–620.

[14] E. Amigó, J. Gonzalo, J. Artiles, F. Verdejo, A comparison of extrinsic clustering evaluation metrics based on formal constraints, Information retrieval 12 (4) (2009) 461–486.

[15] A. Lancichinetti, S. Fortunato, J. Kertész, Detecting the overlapping and hierarchical community structure in complex networks, New Journal of Physics 11 (3) (2009) 033015.