PC-2019/20 Implementazione in Open-MP e CUDA dell'algoritmo K-Means

Marco Calamai
6314073
marco.calamai@stud.unifi.it

Elia Mercatanti
6425149
elia.mercatanti@stud.unifi.it

Abstract

In questo progetto è stato implementato l'algoritmo K-Means per la clusterizzazione in tre versioni. La prima sequenziale in C++, una versione parallelizzata su CPU usando Open-MP ed un'ultima utilizzando la libreria CUDA su GPU Nvidia. Successivamente è stato fatto un confronto tra le versioni parallele e quella sequenziale per analizzare performance e speedup.

1. Introduction

L'algoritmo K-Means un algoritmo di clustering, che permette di suddividere gruppi di oggetti in K partizioni, chiamati cluster cluster, sulla base dei loro attributi. Si assume che gli attributi degli oggetti possano essere rappresentati come vettori. L'obiettivo dell'algoritmo è di minimizzare la varianza totale intra-cluster. Ogni cluster viene identificato mediante un centroide o punto medio. L'algoritmo segue una procedura iterativa:

1. *Inizializzazione*: seleziona K punti come centroidi iniziali scelti casualmente dall'intero datsatet.
2. *Assegnameto*: assegna ogni punto del dataset al cluster più vicino in base ad una funzione di distanza dai centroidi. Come tipo distanza in questo progetto è stata utilizzata la distanza Euclidea.
3. *Aggiornamento*: aggiorna i nuovi centroidi in base alla media di tutti i punti all'interno cluster
4. Ripeti i passi 2 e 3 fino a quando nessun punto cambia cluster.

L'algoritmo può essere riassunto con lo pseudocodice 1.

Algorithm 1: K-means

Seleziona K punti come centroidi iniziali.
repeat
  Forma K cluster assegnando ogni punto al centroide più vicino.
  Ricalcola i centroidi di ogni cluster.
until *Gli assegnamenti non cambiano.*

2. Implementazione sequenziale del K-Means

Il codice dell'algoritmo K-Means sequenziale è mostrato nel codice 4.

2.1. Rappresentazione del Dataset, centroidi e punti

L'implementazione sequenziale dell'algoritmo K-means è stata sviluppata in C++.
Punti, dataset e centroidi sono stati rappresentati come segue:

- Per i punti una *struct* Point con componenti:
  - *cluster_id* di tipo intero per rappresentare l'etichetta del cluster a cui il punto è stato assegnato.
  - $vector\langle int\rangle$ *dimensions* per memorizzare le dimensioni del punto.
- Per il Dataset un $vector\langle Point\rangle$ *dataset*, ovvero un vettore di *struct* Point.
- Per ogni centroide un $vector\langle Point\rangle$ *centroids*, come per il Dataset.

2.2. Inizializzazione

I k centroidi iniziali sono stati scelti in modo random da tutti i punti del dataset dato in input un valore k del numero di cluster da ricercare. Una volta selezionati sono stati salvati in un vettore di Point in modo tale da essere utilizzati da tutte le varianti dell'algoritmo.
Questa parte è stata implementata nel file di test, comune a

tutte le versioni implementate in modo tale da ri utilizzare sempre gli stessi centroidi iniziali.

2.3. Assegnamento

Una volta presi in input i centroidi iniziali, per ogni punto sono state misurate le distanze con ogni centroide, calcolando nel ciclo for più interno la distanza minima.
Il codice 1 mosta nel dettaglio il metodo *point_assignement* per lo step di assegnamento. Vengono eseguiti due cicli for per iterare su ogni punto ed ogni cluster ed utilizzata una funzione *calculate_distance* per calcolare la distanza euclidea tra punto e centroide su ogni dimensione.

```cpp
void points_assignment(std::vector<Point> &dataset, const int num_clusters,
      const std::vector<Point> &centroids) {
double min_distance, distance;
  int cluster_id;
  for (auto i = 0; i < dataset.size(); i++) {
    min_distance = std::numeric_limits<double>::max();
    for (auto j = 0; j < num_clusters; j++) {
      distance = calculate_distance(dataset[i].dimensions, centroids[j].
       dimensions);
      if (distance < min_distance) {
        min_distance = distance;
        cluster_id = centroids[j].cluster_id;
      }
    }
    dataset[i].cluster_id = cluster_id;
  }
}
```

Listing 1. Assegnamento sequenziale

2.4. Aggiornamento

La fase di aggiornamento, eseguita nel metodo *update_centroids*, nel codice 2, segue tre step in tre cicli for distinti. Nel primo ciclo for vengono inizializzate a zero tutte le dimensioni di tutti i centroidi. Successivamente nel secondo step mediante un doppio ciclo for su punti e dimensioni, viene calcolata la somma di tutti i punti in ogni cluster, tenendo traccia del numero di punti in ogni partizione con un contatore. Nell'ultimo step vengono calcolate le medie vere e proprie dei punti, ovvero i centroidi aggiornati, iterando con un ciclo for sui centroidi e calcolando ognuno di essi con un semplice rapporto tra le somme precedentemente calcolate ed il contatore dei punti.

```cpp
void update_centroids(const std::vector<Point> &dataset, const int num_clusters
      , std::vector<Point> &centroids) {
  const auto num_dimensions = dataset[0].dimensions.size();
  std::vector<int> num_points_clusters(num_clusters, 0);

  // Reset the centroids.
  for (auto i = 0; i < num_clusters; i++) {
    for (auto j = 0; j < num_dimensions; j++) {
      centroids[i].dimensions[j] = 0;
    }
  }

  // Calculate the sums and total number of points in each cluster.
  for (auto i = 0; i < dataset.size(); i++) {
    for (auto j = 0; j < num_dimensions; j++) {
      centroids[dataset[i].cluster_id].dimensions[j] += dataset[i].dimensions[j
       ];
    }
    num_points_clusters[dataset[i].cluster_id]++;
  }

  // Calculate the new centroids, for all cluster.
  for (auto i = 0; i < num_clusters; i++) {
    for (auto j = 0; j < num_dimensions; j++) {
      centroids[i].dimensions[j] /= num_points_clusters[i];
    }
  }
}
```

Listing 2. Aggiornamento sequenziale

2.5. Criterio d'arresto

È stato predisposto un ciclo while dove viene controllata la convergenza dell'algoritmo, in modo tale che fino a quando gli assegnamenti dei punti ai vari cluster cambiano, vengono ripetute le fasi di assegnamento e aggiornamento. Nello specifico come mostrato nel codice 3, il controllo viene fatto confrontando gli assegnamenti ai cluster di ogni punto tra l'iterazione corrente e quella precedente. Nel caso in cui la procedura di confronto trovi una prima differenza si continua con l'iterazione successiva.

```cpp
bool check_convergence(const std::vector<Point> &dataset, const std::vector<
      Point> &old_dataset) {
  for (auto i = 0; i < dataset.size(); i++) {
    if (dataset[i].cluster_id != old_dataset[i].cluster_id) {
      return false;
    }
  }
  return true;
}
```

Listing 3. Controllo convergenza

```cpp
std::tuple<std::vector<Point>, std::vector<Point>>
sequential_kmeans(std::vector<Point> dataset, const int num_clusters, std::
      vector<Point> centroids) {
  bool convergence = false;
  std::vector<Point> old_dataset = dataset;
  do {
    // Assignment phase, find the nearest centroid for all points, assign the
       point to that cluster.
    points_assignment(dataset, num_clusters, centroids);

    // Update phase, calculate mean of all points assigned to that cluster, for
       all cluster.
    update_centroids(dataset, num_clusters, centroids);

    // Check if the convergence criterion has been reached.
    if (check_convergence(dataset, old_dataset)) {
      convergence = true;
    } else {
      old_dataset = dataset;
    }
  } while (!convergence);

  return {dataset, centroids};
}
```

Listing 4. K-Means sequenziale

3. Versioni parallele

I due step principali dell'algoritmo (assegnamento e aggiornamento) risultano essere parallelizzabili, il primo in quanto le operazioni che vengono fatte per ogni punto sono indipendenti le une dalle altre, ed è quindi possibile prendere ogni punto, calcolare le distanze tra ogni cluster ed assegnargli quello più vicino in modo parallelo.
Anche il secondo step può essere eseguito in modo parallelo, facendo però attenzione durante il calcolo delle somme per l'aggiornamento dei centroidi, che devono essere fatte in modo atomico per evitare problemi di race condition.
Per quanto riguarda il criterio d'arresto, è stato valutato di implementarlo in modo parallelo. Le motivazioni che hanno influito nella decisione di mantenere la versione sequenziale sono due:

1. Le versioni parallele richiedono di iterare sempre e comunque su tutto il vettore per decidere il risultato finale del controllo mentre la versione sequenziale si può fermare non appena trova una prima differenza tra gli assegnamenti.

2. Nella versione CUDA rendere il criterio d'arresto parallelo richiederebbe due accessi a memoria per controllo, rendendo di fatto la strategia poco performante.

Analizzando mediante il profiler di CLion l'esecuzione del codice sequenziale si è visto, come riportato in figura 1 che l'algoritmo impiega molto più tempo nel calcolo delle distanze all'interno della fase di assegnamento. Per questo motivo ci si potrebbe concentrare nel parallelizzare soltanto la fase di assegnamento. È stato comunque parallelizzata anche la seconda fase di aggiornamento, ottenendo in Open-MP dei piccoli miglioramenti soprattutto su dataset molto grandi.

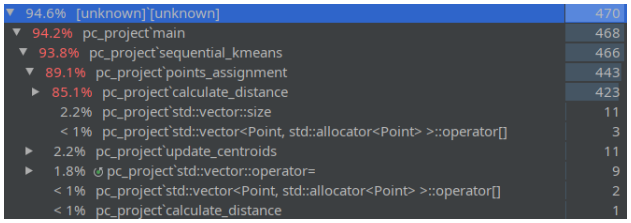

Figura 1. Profiler esecuzione sequenziale

4. Implementazione parallela in Open-MP del K-Means

Per la versione del K-Means in Open-MP, abbiamo creato all'interno del ciclo while una sezione parallela con la clausola *#pragma omp parallel* , che ingloba le due fasi del K-Means come mostrato nel codice 5. Sono state definite *dataset*, *num_clusters*, *centroids*, *num_points_clusters* come variabili shared e *min_distance*, *_distance* e *cluster_id* come private.
Per ogni clausola è stato scelto di mantenere lo scheduler di default in quanto il carico di lavoro di tutte le iterazioni associate ai thread è bilanciato.

```
#pragma omp parallel num_threads(4) default(none) shared(dataset, num_clusters,
        centroids, num_points_clusters)
```

Listing 5. #pragma omp parallel

Per il successivo step di assegnamento è stato richiamato un *#pragma omp for* mostrato nel codice 6 in modo tale da parallelizzare il ciclo for più esterno del metodo e assegnare quindi un certo numero di punti ad ogni thread. La clausula *#pragma omp for* fa in modo tale che i thread non vengano ricreati ogni volta ma vengano utilizzati quelli della clausula *#pragma omp parallel* precedente evitando overhead.

```
void
points_assignment_openmp(std::vector<Point> &dataset, const int num_clusters,
      const std::vector<Point> &centroids) {
  double min_distance, distance;
  int cluster_id;

#pragma omp for
  for (auto i = 0; i < dataset.size(); i++) {
    min_distance = std::numeric_limits<double>::max();
      for (auto j = 0; j < num_clusters; j++) {
        distance = compute_distance_openmp(dataset[i].dimensions, centroids[j].
      dimensions);
        if (distance < min_distance) {
          min_distance = distance;
          cluster_id = centroids[j].cluster_id;
        }
      }
    dataset[i].cluster_id = cluster_id;
  }
}
```

Listing 6. Assegnamento Open-MP

Per quanto riguarda lo step di aggiornamento, mostrato nel codice 7, sono stati richiamati quattro *#pragma omp for*.
Il primo è stato inserito prima di un for aggiuntivo rispetto al sequenziale, utilizzato per inizializzare a zero la variabile shared *num_points_clusters* che conta il numero di punti in ogni cluster.
Un secondo *#pragma omp for*, con *collapse(2)* per distribuire meglio le iterazioni ad ogni thread è stato utilizzato per azzerare i centroidi.
Il terzo *#pragma omp for* è stato utilizzato per calcolare le somme dei punti nei cluster e contare il numero di punti in ognuno di essi. Per fare entrambe le operazioni è necessario un *#pragma omp atomic* per evitare problemi di race condition.
Infine, un utlimo *#pragma omp for* con *collapse(2)* per fare le divisioni atomicamente come visto prima e quindi calcolare i nuovi centroidi.

```
void update_centroids_openmp(const std::vector<Point> &dataset, const int
      num_clusters, std::vector<Point> &centroids,
std::vector<int> &num_points_clusters) {
  const auto num_dimensions = dataset[0].dimensions.size();

// Reset shared num_points_clusters.
#pragma omp for
  for (auto i = 0; i < num_clusters; i++) {
    num_points_clusters[i] = 0;
  }

// Reset the centroids.
#pragma omp for collapse(2)
  for (auto i = 0; i < num_clusters; i++) {
    for (auto j = 0; j < num_dimensions; j++) {
      centroids[i].dimensions[j] = 0;
    }
  }

// Calculate the sums and total number of points in each cluster.
#pragma omp for
  for (auto i = 0; i < dataset.size(); i++) {
    for (auto j = 0; j < num_dimensions; j++) {
#pragma omp atomic
      centroids[dataset[i].cluster_id].dimensions[j] += dataset[i].dimensions[j
      ];
    }
#pragma omp atomic
    num_points_clusters[dataset[i].cluster_id]++;
  }

// Calculate the new centroids, for all cluster.
#pragma omp for collapse(2)
  for (auto i = 0; i < num_clusters; i++) {
    for (auto j = 0; j < num_dimensions; j++) {
#pragma omp atomic
      centroids[i].dimensions[j] /= num_points_clusters[i];
    }
  }
}
```

Listing 7. Aggionamento Open-MP

5. Implementazione parallela in CUDA del K-Means

5.1. Rappresentazione del Dataset, centroidi e punti

In CUDA Dataset e centroidi sono stati memorizzati in array bidimensionali semplici dove le colonne rappresentano le dimensioni e le righe sono i punti. Per quanto riguarda

gli assegnamenti, in questo caso è stato predisposto un array unidimensionale, di lunghezza pari al numero di punti del dataset, dove ogni posizione i contiene l'etichetta del cluster associato al punto i.

5.2. Parallelizzazione CUDA

Per l'implementazione dell'algoritmo K-Means in CUDA sono stati definiti 5 kernel di cui due per lo step di assegnamento e tre per la fase di aggiornamento. Tra ognuno di essi è stato prevista una fase di sincronizzazione dei thread, in quanto l'algoritmo ha bisogno dei risultati di ogni fase precedente prima di procedere con lo step successivo.
Per ognuno dei kernel sono state scelte delle apposite griglie e blocchi di thread per fare in modo tale che ogni streaming multiprocessor (SM) abbia assegnato il massimo numero di threads possibile e allo stesso tempo che ci sia il minor numero di thread inutilizzati.

5.3. Step di assegnamento in CUDA

Per la fase di assegnamento sono stati definiti i kernel *compute_distances* e *points_assignment*. Il primo per il calcolo delle distanze di ogni punto da ogni centroide ed il secondo per trovare, per ogni punto, il centroide più vicino ed assegnare tale punto al relativo cluster.
La funzione kernel *compute_distances* è mostrata nel codice 8, dove ogni thread calcola una distanza punto - centroide. Tutte le distanze calcolate sono salvate in un array dove le righe rappresentano i punti e le colonne i centroidi. Questo array di distanze viene passato come parametro alla funzione kernel *points_assignment* mostrata nel codice 9 , dove ogni thread calcola la distanza minima dai centroidi per un punto e assegna tale punto al relativo cluster.

```
__global__
void compute_distances(const double *device_dataset, const double *
      device_centroids, double *device_distances) {
  unsigned int col = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int row = blockIdx.y * blockDim.y + threadIdx.y;
  double distance = 0;

  if (row < const_num_points && col < const_num_clusters) {
    for (auto i = 0; i < const_num_dimensions; i++) {
      distance += pow(device_dataset[row * const_num_dimensions + i] -
       device_centroids[col * const_num_dimensions + i],2);
    }
    device_distances[row * const_num_clusters + col] = sqrt(distance);
  }
}
```

Listing 8. Assegnamento: *compute_distances*

```
__global__
void points_assignment(const double *device_distances, short *
      device_assignments) {
  int unsigned thread_id = blockDim.x * blockIdx.x + threadIdx.x;
  auto min_distance = INFINITY;
  double distance;
  short cluster_id;

  if (thread_id < const_num_points) {
    for (auto i = 0; i < const_num_clusters; i++) {
      distance = device_distances[thread_id * const_num_clusters + i];
      if (distance < min_distance) {
        min_distance = distance;
        cluster_id = i;
      }
    }
  device_assignments[thread_id] = cluster_id;
  }
}
```

Listing 9. Assegnamento: *points_assignment*

5.4. Step di aggiornamento in CUDA

Per la fase di aggiornamento sono stati definiti i kernel *initialize_centroids*, *compute_sums* e *update_centroids*. Il primo si occupa di inizializzare a zero il valore dei centroidi dove ogni thread pone a zero una coppia centroide - dimensione, come mostrato nel codice 10.
La funzione kernel *compute_sums*, mostrata nel codice 11, si occupa, per ogni cluster, di sommare tutti i punti in esso e calcolare il loro numero. Nello specifico ogni thread gestisce una coppia punto - dimensione eseguendo le operazioni in modo atomico per evitare race condition. In questo modo l'incremento del contatore avviene per ogni dimensione del punto, quindi al momento dell'aggiornamento dei centroidi andrà diviso per il numero di dimensioni.
La funzione kernel *update_centroids*, mostrata nel codice 12, si occupa di calcolare i centroidi aggiornati dove ogni thread gestisce una coppia cluster - dimensione.

```
__global__
void initialize_centroids(double *device_centroids) {
  int unsigned col = blockDim.x * blockIdx.x + threadIdx.x;
  int unsigned row = blockDim.y * blockIdx.y + threadIdx.y;

  if (row < const_num_clusters && col < const_num_dimensions) {
    device_centroids[row * const_num_dimensions + col] = 0;
  }
}
```

Listing 10. Aggiornamento: *initialize_centroids*

```
__global__
void compute_sums(double *device_centroids, const double *device_dataset, const
      short *device_assignments,
  int *device_num_points_clusters) {
  int unsigned col = blockDim.x * blockIdx.x + threadIdx.x;
  int unsigned row = blockDim.y * blockIdx.y + threadIdx.y;

  if (row < const_num_points && col < const_num_dimensions) {
    short cluster_id = device_assignments[row];
    doubleAtomicAdd(&(device_centroids[cluster_id * const_num_dimensions + col
      ]),
    device_dataset[row * const_num_dimensions + col]);
    atomicAdd(&(device_num_points_clusters[cluster_id]), 1);
  }
}
```

Listing 11. Aggiornamento: *compute_sums*

```
__global__
void update_centroids(double *device_centroids, const int *
      device_num_points_clusters) {
  int unsigned col = blockDim.x * blockIdx.x + threadIdx.x;
  int unsigned row = blockDim.y * blockIdx.y + threadIdx.y;

  if (row < const_num_clusters && col < const_num_dimensions) {
    device_centroids[row * const_num_dimensions + col] =
    device_centroids[row * const_num_dimensions + col] / (double(
      device_num_points_clusters[row]) / const_num_dimensions);
  }
}
```

Listing 12. Aggiornamento: *update_centroids*

6. Test e risultati

I test sono stati eseguiti su CPU quad core i7 3770K e GPU Nvidia GTX 1080.
I centroidi iniziali sono stati selezionati in modo random da tutto il dataset e riutilizzati per l'esecuzione di ogni versione del K-Means.
Per ogni versione è stato calcolato il tempo di esecuzione e

lo speed-up ottenuto rispetto alla versione sequenziale.
I test sono stati eseguiti sia su datasets random che su datasets reali.
I dataset random sono stati generati variando numero di punti e numero di dimensioni e l'algoritmo K-Means è stato eseguito al variare del numero di cluster da ricercare tra 10, 100 e 1000.
Nella tabella in figura 3 sono mostrati i risultati ottenuti con le tre versioni del K-Means implementate su datasets random.
Dai risultati ottenuti si può vedere come al crescere della dimensione del problema, la versione Open-MP raggiunge uno speed-up quasi lineare, pari circa a 4, sulla CPU di test.
Dalla tabella si nota che lo speed-up di entrambe le versioni parallele incrementa all'aumentare del numero di clusters e dimensioni in quanto si aumenta il carico di lavoro, soprattutto per la parte relativa al calcolo delle distanze.
Entrambe le versioni parallele risultano decisamente più efficienti della versione sequenziale, soprattutto quella in CUDA dove si ottengono tempi di esecuzione notevolmente più bassi.
I datasets reali considerati sono i seguenti:

- Iris, contenente istanze di piante iris.
- S1-Set, Birch1-set e worms che rappresentano datasets sintetici.
- Letter riguardante le 26 lettere dell'alfabeto inglese.

I risultati ottenuti su tali datasets sono mostrati in figura 4 e descritti in 2. Anche per quest'ultimi le considerazioni fatte precedentemente sono confermate.

DATASETS DESCRIPTIONS

DATESET	# PUNTI	# DIMENSIONI	#CLUSTERS
IRIS	150	4	3
S1-SET	5000	2	15
BIRCH1-SET	100000	2	100
LETTER	20000	16	26
WORMS_2D	105600	2	35
WORMS_64D	105600	64	35

Figura 2. Datasets descripions

7. GitHub

https://github.com/elia-mercatanti/parallel-k-means

Per eseguire le tre versioni del K-Means (sequenziale, Open-MP e CUDA) su datasets locali è necessario passare al programma il percorso del dataset e il numero di cluster da ricercare. Il file test.cpp si occuperà di testare il dataset selezionato con tutte le versioni dell'algoritmo.
Nel caso in cui non venga passato nessun parametro al programma allora il file test.cpp si occuperà di generare tutti i datsets random visti precedentemente (nel caso in cui non siano già stati creati) e terminerà l'esecuzione.

DATASET

# CLUSTERS		Random 1000x10	Random 1000x100	Random 1000x1000	Random 10000x10	Random 10000x100	Random 10000x1000	Random 100000x10	Random 100000x100	Random 100000x1000	Random 1000000X10
10	**Sequenziale**	0.0995483	0.638251	2.03217	10.2006	52.0753	134.221	162.108	2622.35	14810.8	3873.36
	Open-MP	0.0426349	0.173283	0.547636	3.46449	14.405	38.6007	55.2159	710.503	4081.3	1320.14
	Speedup Open-MP	2.33	3.68	3.71	2.94	3.62	3.48	2.94	3.69	3.63	2.93
	CUDA	0.00684048	0.0262636	0.0918535	0.823457	0.988186	2.76661	24.9337	45.3077	289.722	584.845
	Speedup CUDA	14.55	24.30	22.12	12.39	52.70	48.51	6.50	57.88	51.12	6.62
	# ITERAZIONI	22	16	5	228	130	33	363	639	366	861
100	**Sequenziale**	0.390759	2.33382	7.84521	29.5799	150.701	705.307	940.892	-	-	-
	Open-MP	0.102776	0.595576	1.99878	7.78527	38.7765	180.048	248.872	-	-	-
	Speedup Open-MP	3.80	3.92	3.92	3.80	3.89	3.92	3.78			
	CUDA	0.00773597	0.0408883	0.13143	0.489796	2.22139	10.0715	14.3992	-	-	-
	Speedup CUDA	50.51	57.08	59.69	60.39	67.84	70.03	65.34			
	# ITERAZIONI	10	6	2	76	39	18	242	-	-	-
1000	**Sequenziale**	0.786591	7.8893	78.2989	65.8951	733.294	1555.92	3840.99	-	-	-
	Open-MP	0.198559	1.98681	19.8394	16.776	189.338	401.735	995.712	-	-	-
	Speedup Open-MP	3.96	3.97	3.95	3.93	3.87	3.87	3.86			
	CUDA	0.0147833	0.134762	1.15328	1.00158	10.5292	21.8604	57.8113	-	-	-
	Speedup CUDA	53.21	58.54	67.89	65.79	69.64	71.18	66.44			
	# ITERAZIONI	2	2	2	17	19	4	99	-	-	-

Figura 3. Results on random datasets

DATASET

With K clusters

	IRIS K=3	S1-Set K=15	Birch1-set K=100	Letter K=26	worms_2d K=35	worms_64d K=35
Sequenziale	0.000429641	0.113001	82.724	36.3694	35.5275	579.928
Open-MP	0.000396073	0.0519949	24.2833	10.2238	12.6263	152.624
Speedup Open-MP	1.08	2.17	3.41	3.56	2.81	3.80
CUDA	0.000340643	0.0223294	2.29935	0.774311	3.70815	37.303
Speedup CUDA	1.26	5.06	35.98	46.97	9.58	15.55
# ITERAZIONI	3	14	99	109	106	63

Figura 4. Results on real datasets