PC-2016/17 Progetto mid-term del corso
Implementazione in CUDA dell'algoritmo KMean

Tommaso Ceccarini
6242250
tommaso.ceccarini1@stud.unifi.it

Federico Schipani
6185896
federico.schipani@stud.unifi.it

Abstract

L'algoritmo di KMeans è uno dei più popolari metodi per la clusterizzazione. Nel nostro lavoro forniamo una implementazione in CUDA che fa uso di GPU Nvidia per l'esecuzione dell'algoritmo. Inoltre forniamo un'analisi delle preformance con lo scopo di comparare la nostra implementazione parallela con una implementazione sequenziale scritta in C.

1. Introduzione

L'algoritmo KMeans è uno degli algoritmi di clustering più famosi. Lo scopo del clustering è di dividere dati in gruppi significativi chiamati cluster. Dato un insieme di dati $(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_N})$ dove ogni dato è un vettore reale di dimensione P, lo scopo di KMeans è di partizionare N dati in $K(\leq N)$ insiemi $\boldsymbol{S} = \{S_1, S_2, \ldots, S_K\}$ per minimizzare la distanza dentro ai singoli cluster. In altre parole viene definita una funzione obiettivo del tipo:

$$\arg\min_{\boldsymbol{S}} \sum_{i=1}^{k} \sum_{x \in S_i} ||\boldsymbol{x} - \boldsymbol{\mu}_i||^2 \tag{1}$$

dove μ_i è la media dei punti in S_i.[1]

L'Algoritmo 1 rappresenta lo pseudocodice di un'implementazione iterativa dell'algoritmo KMeans.

I principali passi dell'algoritmo sono:

1. Selezionare casualmente K dati per inizializzare la media dei K cluster. Questo passo è usualmente chiamato *inizializzazione*.
2. Assegnare ogni dato al cluster più vicino, in accordo ad certa funzione di distanza. Questo passo è chiamato *assegnamento*.
3. Calcolare la nuova media dei cluster. Questo passo è chiamato *aggiornamento*.
4. Ripetere il passo due e tre finché nessun dato cambia cluster, oppure se solo pochi lo fanno.

Algoritmo 1 KMeans

```
1:  procedure KMEANS(data[n][p])
2:      mean[k][p], oldMean[k][p]
3:      assignment[n]
4:      (mean, assignment) ← initAss(data)
5:      while !stop do
6:          assignment ← calcMin(mean, data)
7:          oldMean ← mean
8:          mean ← calcMean(assignment, data)
9:          stop ← stopCrit(mean, oldMean)
10:     end while
11:     return mean
12: end procedure
```

1.1. KMeans sequenziale

Prima del ciclo *while* c'è un passo di assegnazione iniziale. In questo step i primi K dati sono assegnati ai primi K cluster. Uno pseudocdice di questo assegnamento iniziale è mostrato in Algoritmo 2

Algoritmo 2 Initial Assignment

```
1:  procedure INITASS(data[n][p])
2:      mean[k][p]
3:      assignment[n]
4:      for i = 0; i < k; i + + do
5:          assignment[i] = i
6:          for j = 0; j < p : j + + do
7:              mean[i][j] = data[i][j]
8:          end for
9:      end for
10:     return (mean, assignment)
11: end procedure
```

Dopo questo step iniziale di assegnamento parte il vero e proprio algoritmo. Nella prima parte del ciclo *while* viene calcolata la distanza euclidea minima tra ognuno dei dati e tutti i cluster. Successivamente si assegna ogni dato al cluster più vicino. Il passo successivo consiste nel ricalcolare

le nuove medie dei cluster, secondo la (2).

$$m_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j \tag{2}$$

L'ultimo step è il criterio d'arresto. In questo semplice algoritmo, mostrato in 3, si iterano l'attuale matrice dei cluster e la matrice dei cluster calcolata al passo $i-1$. Quando $ActualValue - OldValue$ eccede una prefissata tolleranza TOL il metodo ritorna il valore *false*. Questo valore di ritorno porterà ad un'altra iterazione del ciclo *while* esterno.

Algoritmo 3 Stop Criterion

```
1: procedure STOPCRIT(mean[k][p], oldMean[k][p])
2:     for all value, oldValue in mean, oldMean do
3:         if abs(value − oldValue) > TOL then
4:             return false
5:         end if
6:     end for
7:     return true
8: end procedure
```

2. Implementazione

2.1. Come vengono rappresentati i dati ed i cluster

Le due differenti implementazioni dell'algoritmo di KMeans, riportate in sezione 2.2 e 2.3, risolvono il problema del clustering per un valore arbitrario del parametro P. Tuttavia le analisi delle performance, mostrate in Sezione 3, sono state effettuate nel caso in cui $P = 2$, ovvero il caso in cui i dati sono vettori bidimensionali.

Le due maggiori strutture su cui il codice lavora sono la matrice dei dati e la matrice dei centroidi, rispettivamente di dimensione $N * P$ e $K * P$. In entrambe le implementazione viene usato anche un vettore chiamato *assignment* che, per tutti gli N dati, memorizza l'indice del cluster al quale appartengono.

2.1.1 Come viene generato il dataset

Dopo che sono state dichiarate, e successivamente istanziate, le strutture che sono necessarie per rappresentare i dati e i centroidi viene effettuata una genrazione pseudocasuale dei dati. Questa generazione viene effettuata in maniera parallela attraverso le API CUDA, in particolare è stata usata la libreria cuRAND. Il metodo che è stato sviluppato per questo scopo è `generateRandomDataOnDevice()`. Questo metodo usa un seed prefissato e, con l'aiuto di un altro metodo che fa parte della libreria cuRAND, genera i valori del dataset. Nella Sezione 3 sono mostrate le analisi delle performance nel caso in cui i dati sono generati con una distribuzione uniforme.

2.2. Una implementazione sequenziale in C

Lo step di inizializzazione

Dopo aver generato casualmente il dataset nella memoria globale del device è necessario istanziare la memoria richiesta per memorizzare la matrice dei dati e dei centrodi nella memoria dell'host. Inoltre, per come funziona l'algoritmo, è necessaria un'altra matrice che memorizza il valore della media durante la precedente iterazione. Questa matrice è usata per implementare il criterio d'arresto. L'inizializzazione è effettuata da un semplice ciclo for che assegna `data[i][j]` a `mean[i][j]` per `i` tra 0 e $K-1$. Questa strategia d'assegnamento è motivata dal fatto che i dati sono generati casualmente.

Lo step di assegnamento

Dopo aver effettuato l'assegnazione iniziale l'algoritmo sequenziale implementa il passo di assegnamento attraverso il Codice 1. Nel Codice 1 vengono utilizzati due cicli for: un ciclo esterno che itera su N dati e un ciclo annidato che per ogni dato itera sui K cluster per cercare il più vicino. Al termine del ciclo annidato l'indice del cluster più vicino al dato `i`-esimo verrà memorizzato alla cella `i` del vettore `assignment[]`.

Codice 1: Passo di assegnamento sequenziale

```
1    for (int i = 0; i < N; i++) {
2      float minDistance = 999999.9;
3      short minIndex = -1;
4      float distance = 0;
5      for (int z = 0; z < K; z++) {
6        distance = 0;
7        for (int j = 0; j < COMPONENTS; j++) {
8          distance +=pow(
9              (hostData[i * COMPONENTS + j]
10                 - mean[z * COMPONENTS + j]),2);
11       }
12       if (distance < minDistance) {
13         minIndex = z;
14         minDistance = distance;
15       }
16     }
17     assignment[i] = minIndex;
18   }
```

Lo step di aggiornamento

Successivamente all'assegnazione dei dati al cluster più vicino è necessario ricalcolare nuovamente i valori dei centroidi per i cluster. Questo passo dell'algoritmo è implementato mediante il Codice 2. Come si può notare dal Codice 2 questo passo è implementato mediante due cicli for: un primo ciclo esterno che itera sui K cluster e un ciclo annidato in cui, utilizzando il vettore `assignment[]` viene calcolata la somma dei valori delle singole componenti dei

dati che appartengono al cluster e il numero di dati che appartengono al cluster. Quindi alla fine del ciclo interno, per ogni componente, viene diviso il valore della somma per il numero di dati che appartengono al cluster per calcolare il valore effettivo dei nuovi centroidi.

Codice 2: Passo di aggiornamento della media

```
1    for (int i = 0; i < K; i++) {
2      int numberOfData = 0;
3      float *arraySum = (float*) malloc(COMPONENTS * sizeof(
           ↪ float));
4      for (int x = 0; x<COMPONENTS; x++) {
5        arraySum[x] = 0.0;
6      }
7      for (int j = 0; j < N; j++) {
8        if (assignment[j] == i) {
9          numberOfData++;
10         for (int x = 0; x < COMPONENTS; x++) {
11           arraySum[x] += hostData[j * COMPONENTS + x];
12         }
13       }
14     }
15     for (int j = 0; j < COMPONENTS; j++) {
16       oldMean[i * COMPONENTS + j] = mean[i * COMPONENTS + j
           ↪ ];
17       mean[i * COMPONENTS + j] = arraySum[j] / numberOfData
           ↪ ;
18     }
19   }
```

Il criterio d'arresto

Il criterio d'arresto che viene implementato verifica se il vettore dei centroidi è cambiato nel corso dell'ultima iterazione rispetto a una prefissata tolleranza `TOL`. Per implementare questa strategia, alla riga 16 del Codice 2, prima di aggiornare il valore della media, vengono copiati i valori di `mean[]` nel vettore di appoggio `oldMean[]`. Il Codice 3 verifica quindi se i valori dei centroidi sono cambiati iterando mediante un ciclo for sulle componenti dei vettori `oldMean[]` e `mean[]` verificando se i valori sono uguali a meno di una tolleranza `TOL`.

Codice 3: Criterio di arresto sequenziale

```
1    stopCriterion = true;
2    for (int i = 0; i < K * COMPONENTS; i++) {
3      if (abs(mean[i] - oldMean[i]) > TOL) {
4        stopCriterion = false;
5        break;
6      }
7    }
```

2.3. Implementazione parallela tramite CUDA

2.3.1 Benefici di un'implementazione parallela

In riferimento al codice sequenziale si può notare come gran parte dei calcoli che vengono effettuati siano indipendenti gli uni dagli altri. Questo fatto porta a pensare che un'implementazione parallela avrà dei miglioramenti in termini di prestazioni. In particolare le operazioni parallelizzabili sono:

- Assegnazione iniziale. Infatti è possibile assegnare ad un thread il compito di assegnare il valore di una componente del centroide.
- Calcolo delle distanze. Ciascuna delle $N * K$ distanze può essere calcolata indipendentemente dalle altre.
- Determinazione del cluster più vicino. In questo contesto è possibile assegnare ad un thread il compito di determinare quale sia il cluster più vicino ad un dato. In ogni caso è comunque necessario effettuare una ricerca sequenziale scorrendo K medie.
- Calcolo della media. Utilizzando il vettore degli assegnamenti ogni thread può: determinare a quale cluster il dato corrispondete appartiene, e incrementare il contatore del numero di elementi per cluster che servirà poi per il calcolo effettivo della media.

Seguendo queste idee quindi è stato implementato il metodo `kMeans(float *devData, short* devAssignment, float* devMean)` che riceve in input la matrice dei dati, il vettore degli assegnamenti e la matrice dei centrodi, tutte queste strutture sono allocate nella memoria del device.

Lo step di inizializzazione

Per realizzare il passo di inizializzazione vengono utilizzati i due metodi Kernel riportati in Codice 4 e Codice 5.

Codice 4: Assegnazione iniziale dei dati ai cluster

```
1  __global__ void initializeAssignment(short *assignment, int n
       ↪ ) {
2    short index = (blockIdx.x * blockDim.x) + threadIdx.x;
3    if (index < K) {
4      assignment[index] = index;
5    }
6  }
```

Nel Codice 4 viene effettata l'assegnazione iniziale. Questo viene eseguito con un totale di K thread. Ogni thread che esegue questa funzione Kernel si occupa di inizializzare il vettore degli assegnamenti scrivendo in posizione `index` il proprio indice `index`. A questo scopo la dimensione dei blocchi viene impostata a `dim3 DimBlockInizPart(1024, 1, 1)` e di conseguenza la griglia sarà inizializzata a `dim3 DimGridInizPart((k / 1024) + 1, 1, 1)`.

Codice 5: Inizializzazione della matrice dei centroidi

```
1  __global__ void initializeMean(float *mean, float* dataset,
       ↪ int n,
```

```
    int P) {
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int bdx = blockDim.x;
  int bdy = blockDim.y;
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int row = by * bdy + ty;
  int col = bx * bdx + tx;
  if (row < K && col < P) {
    mean[row * P + col] = dataset[row * P + col];
  }
}
```

Dopo aver assegnato i primi K dati ai relativi cluster il Codice 5 si occupa di assegnare sfruttando questo fatto: infatti è sufficiente assegnare i valori delle componenti del dato `(i, j)` al centroide `(i, j)`. Quindi in questo caso vengono usati $K * P$ thread e la dimensione dei blocchi viene impostata a `dim3 DimBlockInizMean(P, 16, 1)`, e di conseguenza la griglia sarà inizializzata a `dim3 DimGridInizMean(1, k / 16 + 1, 1)`.

Lo step di assegnamento

Per parallelizzare il passo di assegnamento vengono usate due funzioni, riportati in Codice 6 e Codice 7. L'implementazione parallela del passo di assegnamento è realizzata attraverso i due metodi riportati in Codice 6 e Codice 7. Nel Codice 6 ogni thread `(i,j)` è responsabile del calcolo della distanza del dato `i` dal cluster `j`. Quindi la griglia dei blocchi è organizzata in `dim3 DimGridDistances((k / 4) + 1, (n / 256) + 1, 1)` e di conseguenza i blocchi sono stati impostati a `dim3 DimBlockDistances(4, 256, 1)`.

Codice 6: Kernel per il calcolo delle distanze

```
__global__ void computeDistances(float* dataset, float*
    ↪ centroids,
    float* distances, int P, int k) {
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int row = by * blockDim.y + ty;
  int col = bx * blockDim.x + tx;
  float dist = 0.0;
  if (row < N && col < K) {
    for (int i = 0; i < P; i++) {
      dist += (dataset[row * P + i]
          - centroids[col * P + i])
          * (dataset[row * P + i]
              - centroids[col * P + i]);
    }
    distances[row * k + col] = dist;
  }
}
```

L'altro Kernel riportato nel Codice 7 effettua la ricerca del cluster più vicino al dato utilizzando la matrice delle distanze appena calcolata dal Codice 6. In questo contesto vengono utilizzati N thread in cui ciascuno effettua una ricerca sequenziale della distanza minima, memorizzando allo stesso tempo l'indice del cluster più vicino. La griglia è organizzata come `dim3 DimGridMin(1, (n / 256) + 1, 1)`, e i blocchi come `dim3 DimBlockAss(1, 256, 1)`.

Codice 7: Kernel per il calcolo del minimo

```
__global__ void computeMin(float *distances, short*
    ↪ devAssignment, int k,
    int n) {
  unsigned int ty = threadIdx.y;
  int by = blockIdx.y;
  unsigned int row = by * blockDim.y + ty;
  if (row < n) {
    float min = distances[row * k];
    short minInd = 0;
    for (int i = 1; i < k; i++) {
      bool conf = (min - distances[row * k + i] <= 0);
      min = conf * min + (1 - conf) * distances[row * k + i];
      minInd = conf * minInd + (1 - conf) * i;
    }
    devAssignment[row] = minInd;
  }
}
```

Lo step di aggiornamento

L'aggiornamento della media è divisa in tre Kernel, riportati in Codice 8, Codice 9 e Codice 10. Il metodo Kernel, riportato in Codice 8, si occupa di reinizializzare il valore dei centroidi a 0. Questo per realizzare il calcolo delle somme delle componenti dei dati in modo atomico, riportato nel Codice 9.

Codice 8: Kernel per reinizializzare un vettore

```
__global__ void initializeVectorToValue(float *vector, float
    ↪ value, int bound) {
  int tx = threadIdx.x;
  int bdx = blockDim.x;
  int bx = blockIdx.x;
  int index = bx * bdx + tx;
  if (index < bound) {
    vector[index] = value;
  }
}
```

Il Codice 9 utilizza $N * P$ thread in cui il thread `(i,j)` si occupa di verificare, attraverso il vettore degli assegnamenti a quale cluster appartiene la componente `j` del dato `i` e di incrementare il conteggio degli elementi appartenenti al cluster del dato.

Codice 9: Calcolo della somma e del numero di elementi di un cluster

```
__global__ void computeSum37(float* dataset, short*
    ↪ devAssignment,
    float* centroids, int* counter) {
  int tx = threadIdx.x;
```

```
  int ty = threadIdx.y;
  int bdx = blockDim.x;
  int bdy = blockDim.y;
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int row = by * bdy + ty;
  int col = bx * bdx + tx;
  if (row < N) {
    int clusterIndex = devAssignment[row];
    atomicAdd(&(centroids[clusterIndex * P + col]),
        dataset[row * P + col]);
    atomicAdd(&(counter[clusterIndex]), 1);
  }

}
```

Infine nel Codice 10, $K * P$ thread si occupano di fare il calcolo effettivo della media dividendo il risultato delle somme calcolate nel Codice 9 per il numero di elementi che appartengono al relativo cluster. I $K * P$ thread sono divisi in blocchi `dim3 DimBlockMean(P, 256, 1)` e la griglia è formata da `dim3 DimGridMean(1, (K / 256) + 1, 1)` blocchi.

Codice 10: Kernel per il calcolo della media

```
__global__ void computeMean37(float* centroids, int* counter)
    ↪ {
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int bdx = blockDim.x;
  int bdy = blockDim.y;
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int row = by * bdy + ty;
  int col = bx * bdx + tx;

  if (row < K) {
    centroids[row * P + col] = centroids[row * P + col]
        / (counter[row] / 2);
  }
}
```

Il criterio d'arresto

Per implementare il criterio d'arresto sono state prese in considerazione due possibili strade:

1. Utilizzare una variabile globale che conta il numero di dati che cambiano il cluster di appartenenza nel corso di una singola iterazione.

2. Riutilizzare il criterio di arresto implementato per la versione sequenziale, mostrato in Codice 3

L'implementazione della possibilità 1 è stata realizzata incrementando atomicamente la variable globale in questione tutte le volte in cui il Codice 7 determina un cambiamento di cluster. Questa soluzione porta ad un incremento della divergenza a causa del controllo necessario per verificare se il cluster del dato cambia. Quindi è stata valutata la possibilità di riutilizzare il codice che effettuava il criterio d'arresto nell'implementazione dell'algoritmo sequenziale. Sono state condotte una serie di verifiche per determinare se ci fossero delle differenze in termini di prestazioni tra le due diverse possibilità. A seguito di queste verifiche è stato riscontrato che la versione sequenziale del criterio d'arresto è generalmente, anche se di poco, migliore della controparte parallela.

Per realizzare una versione più efficiente della possibilità 2 è stata considerata la possibilità di effettuare una verifica parallela sul vettore delle medie utilizzando la tecnica di riduzione. Ad ogni modo verifiche condotte in merito hanno mostrato che tale approccio non portava alcun tipo di beneficio in termini di prestazioni. Questo tipo di tecnica ha il merito di sfruttare al meglio le potenzialità della memoria shared utilizzandole in maniera efficiente, e senza generare conflitti, per ridurre un vettore ad un singolo elemento. D'alta parte questo approccio richiede di iterare sempre e comunque su tutto il vettore per decidere il risultato finale del controllo, mentre la versione sequenziale appena verifica che una media nel corso dell'ultima iterazione è cambiata a meno di una tolleranza `TOL` si ferma restituendo il fallimento del controllo. Ad ogni modo dovendo operare su strutture allocate nella memoria del host sono necessarie due `cudaMemCpy()` del tipo `DeviceToHost` per il vettore delle medie aggiornato e quello vecchio.

3. Analisi delle performance e risultati sperimentali

Una volta che le due diverse versioni dell'algoritmo di KMeans sono state sviluppate sono stati condotti una serie di esperimenti per verificare quale fosse il vantaggio effettivo ottenuto dall'implementazione parallela. Le Figure 2 3 4 riportano il risultato della clusterizzazione ottenuta attraverso l'implementazione in CUDA con diversi valori di N e K.

3.1. Tempi di esecuzione

In Tabella 1 sono riportati i tempi di esecuzione per i due algoritmi implementati. Come si può osservare consultando la tabella la versione parallela è migliore di quella sequenziale. I risultati riportati sono stati ottenuti impostando un valore della tolleranza di $5*10^{-6}$. In Tabella 2 è riportato lo speedup che la versione parallela raggiunge rispetto a quella sequenziale per gli esperimenti riportati nella Tabella 1. In Figura 1 sono riportati le percentuali di tempo dei metodi kernel eseguiti dall'implementazione parallela. Tali dati sono stati ottenuti con $N = 80000$ e $K = 200$, eseguendo il programma con lo strumento di profiling di Nvidia.

Tabella 1: Tempi di esecuzione

Punti	Cluster	Sequenziale	Parallelo	Iterazioni
2000	100	0.142902 s	0.009067 s	20
10000	200	2.45001 s	0.067496 s	42
25000	350	12.564984 s	0.533056 s	52
50000	500	56.518483 s	2.519471 s	86
100000	750	182.013450 s	8.965721 s	93
200000	1000	758.144052 s	32.357209 s	143

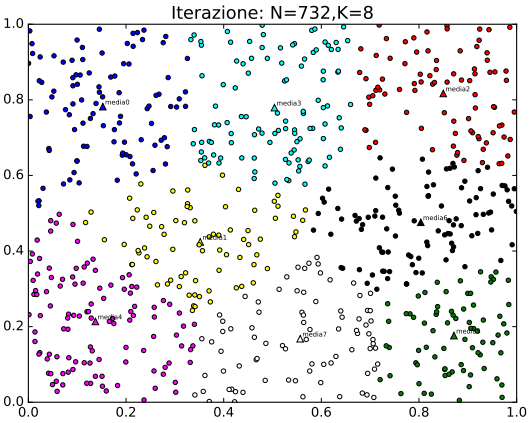

Figura 2: Plot per $N = 732$ e $K = 8$

Tabella 2: Speedup

Punti	Cluster	Speedup
2000	100	15.760
10000	200	36.300
25000	350	23.571
50000	500	22.433
100000	750	20.301
200000	1000	23.430

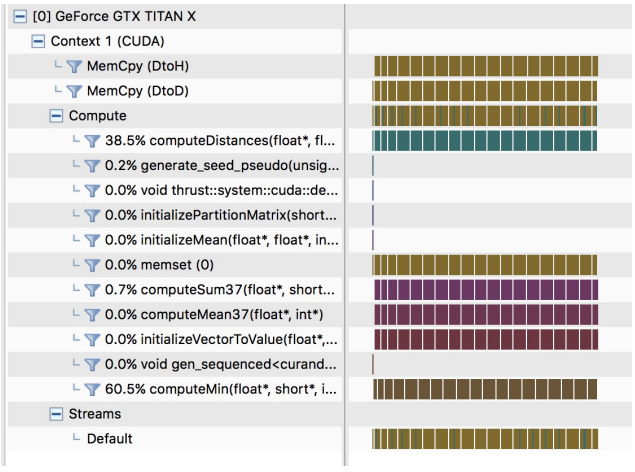

Figura 1: Percentuale del tempo di esecuzione dei diversi kernel

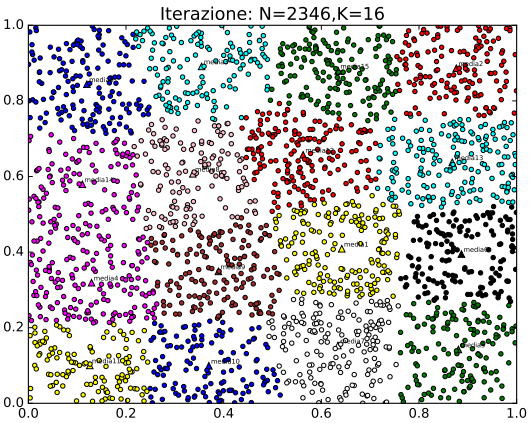

Figura 3: Plot per $N = 2346$ e $K = 16$

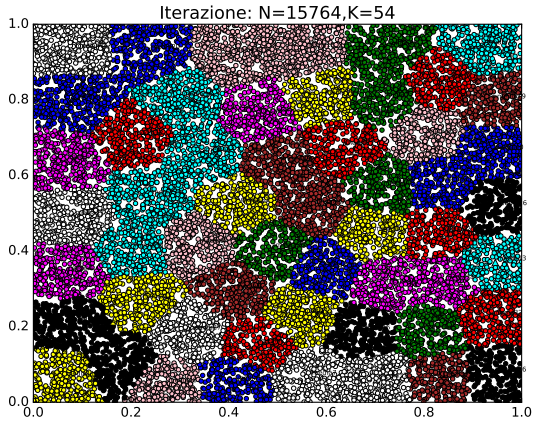

Figura 4: Plot per $N = 15764$ e $K = 54$

4. Conclusioni

È quindi possibile affermare che l'algoritmo di KMeans trae notevoli benefici da un'implementazione parallela che fa uso di una GPU. Infatti per gran parte dei calcoli è stato possibile realizzare una versione parallela in quanto indipendenti gli uni dagli altri. Alla luce degli esperimenti riportati non è possibile affermare che la percentuale di miglioramento della versione parallela rispetto alla sequenziale (speedup) sia legata alle due dimensioni principali del problema che sono state prese in considerazione, ovvero N e K. Ad ogni modo la versione implementata in CUDA si è rivelata essere notevolmente più efficiente rispetto alla versione sequenziale sviluppata in C.

A. Codice

Codice 11: Codice completo

```

#include <iostream>
#include <numeric>
#include <stdlib.h>
#include <stdio.h>
#include <curand.h>
#include <curand_kernel.h>
#include <math.h>
#include <cuda.h>
#include <stdbool.h>
#include <thrust/device_ptr.h>
#include <thrust/fill.h>
#include <cfloat>
#include <fstream>
#include <sstream>
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <cmath>
#include <time.h>

static void CheckCudaErrorAux(const char *, unsigned, const char *,
    cudaError_t);
#define CUDA_CHECK_RETURN(value) CheckCudaErrorAux(__FILE__, __LINE__, #value, value)
#define P 2
#define K 233
#define N 35067
#define TOL 0.000005
#define SEED 1234ULL
#define TD 1024

#define CURAND_CALL(x) do { \
  if ((x)!=CURAND_STATUS_SUCCESS) { \
    printf("ERROR AT %s:%d\r\n", __FILE__, __LINE__);\
    return EXIT_FAILURE;\
  }\
} while(0)

using namespace std;

__global__ void initializeVectorToValue(float *vector, float value, int bound) {
  int tx = threadIdx.x;
  int bdx = blockDim.x;
  int bx = blockIdx.x;
  int index = bx * bdx + tx;
  if (index < bound) {
    vector[index] = value;
  }
}

__global__ void initializeAssignment(short *assignment, int n ) {
  short index = (blockIdx.x * blockDim.x) + threadIdx.x;
  if (index < K) {
    assignment[index] = index;
  }
}
__global__ void initializeMean(float *mean, float* dataset, int n,
    int P) {
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int bdx = blockDim.x;
  int bdy = blockDim.y;
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int row = by * bdy + ty;
  int col = bx * bdx + tx;
  if (row < K && col < P) {
    mean[row * P + col] = dataset[row * P + col];
  }
}

__global__ void computeSum37(float* dataset, short* devAssignment,
```

```
    float* centroids, int* counter) {
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int bdx = blockDim.x;
  int bdy = blockDim.y;
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int row = by * bdy + ty;
  int col = bx * bdx + tx;
  if (row < N) {
    int clusterIndex = devAssignment[row];
    atomicAdd(&(centroids[clusterIndex * P + col]),
        dataset[row * P + col]);
    atomicAdd(&(counter[clusterIndex]), 1);
  }

}

__global__ void computeMean37(float* centroids, int* counter)
    ↪  {
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int bdx = blockDim.x;
  int bdy = blockDim.y;
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int row = by * bdy + ty;
  int col = bx * bdx + tx;

  if (row < K) {
    centroids[row * P + col] = centroids[row * P + col]
        / (counter[row] / 2);
  }
}

__global__ void computeMean(float *centroids, float* dataset,
    short* devAssignment, int n, int P) {
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int bdx = blockDim.x;
  int bdy = blockDim.y;
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int row = by * bdy + ty;
  int col = bx * bdx + tx;
  float mean = 0.0;
  int counter = 0;
  if (row < K) {
    for (int i = 0; i < n; i++) {
      mean += (devAssignment[i] == row) * dataset[i * P + col
          ↪ ];
      counter += (devAssignment[i] == row);
    }

    centroids[row * P + col] = mean / counter;
  }
}

__global__ void computeDistances(float* dataset, float*
    ↪ centroids,
    float* distances, int P, int k) {
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int row = by * blockDim.y + ty;
  int col = bx * blockDim.x + tx;
  float dist = 0.0;
  if (row < N && col < K) {
    for (int i = 0; i < P; i++) {
      dist += (dataset[row * P + i]
          - centroids[col * P + i])
          * (dataset[row * P + i]
              - centroids[col * P + i]);
    }
    distances[row * k + col] = dist;
  }
}

__global__ void computeMin(float *distances, short*
    ↪ devAssignment, int k,
    int n) {
  unsigned int ty = threadIdx.y;
  int by = blockIdx.y;
  unsigned int row = by * blockDim.y + ty;
  if (row < n) {
    float min = distances[row * k];
    short minInd = 0;
    for (int i = 1; i < k; i++) {
      bool conf = (min - distances[row * k + i] <= 0);
      min = conf * min + (1 - conf) * distances[row * k + i];
      minInd = conf * minInd + (1 - conf) * i;
    }
    devAssignment[row] = minInd;
  }
}

__global__ void assignPartition(short* S, int* minIndex, int
    ↪ n) {
  int by = blockIdx.y;
  int ty = threadIdx.y;
  int row = by * blockDim.y + ty;

  if (row < n) {
    int pos = minIndex[row];
    S[pos * n + row] = true;
  }

}

int generateRandomDataOnDevice(float *devData, int n, int P)
    ↪ {
  curandGenerator_t generator;
  CURAND_CALL(curandCreateGenerator(&generator,
      ↪ CURAND_RNG_PSEUDO_DEFAULT));
  CURAND_CALL(curandSetPseudoRandomGeneratorSeed(generator,
      ↪ SEED));
  CURAND_CALL(curandGenerateUniform(generator, devData, n * P
      ↪ ));
  CURAND_CALL(curandDestroyGenerator(generator));
  return 0;
}

void printFile(std::string path, std::string filename, int n,
    ↪  int k,
    int P, float *devDataset, short *devAssignment,
    float *devMean) {

  float *mean, *dataset;
  short *assignment;
  mean = (float*) malloc(k * P * sizeof(float));
  dataset = (float*) malloc(n * P * sizeof(float));
  assignment = (short*) malloc(n * sizeof(short));

```

```
  CUDA_CHECK_RETURN(
      cudaMemcpy(mean, devMean, P * k * sizeof(float),
          cudaMemcpyDeviceToHost));
  CUDA_CHECK_RETURN(
      cudaMemcpy(dataset, devDataset, n * P * sizeof(float),
          cudaMemcpyDeviceToHost));
  CUDA_CHECK_RETURN(
      cudaMemcpy(assignment, devAssignment, n * sizeof(short)
          ↪ ,
          cudaMemcpyDeviceToHost));
  cudaDeviceSynchronize();
  ofstream outputfile;
  outputfile.open(path + filename, ios::out);
  for (int i = 0; i < K; i++) {
    std::string toFile = "media" + to_string(i);
    for (int j = 0; j < P; j++) {
      toFile = toFile + " " + to_string(mean[i * P + j]);
    }
    toFile = toFile + "|";
    outputfile << toFile;
  }
  for (int i = 0; i < N; i++) {
    std::string toFile = "";
    toFile = to_string(assignment[i]);
    for (int j = 0; j < P; j++) {
      toFile = toFile + " " + to_string(dataset[i * P + j]);
    }
    toFile = toFile + "|";
    outputfile << toFile;
  }
  outputfile.close();
  free(mean);
  free(dataset);
  free(assignment);
}

bool stopCriterionSequential(short *assignment, short *
    ↪ oldAssignment, int len) {
  for (int i = 0; i < len; i++) {
    if (assignment[i] != oldAssignment[i]) {
      return false;
    }
  }
  return true;
}

bool stopCriterionSequential(float *vector1, float *vector2,
    ↪ int len) {
  for (int i = 0; i < len; i++) {
    if (abs(vector1[i] - vector2[i]) > TOL) {
      return false;
    }
  }
  return true;
}

int kMeans(float *devData, short *devAssignment, float *
    ↪ devMean, int n, int k,
    int P) {
  printf("KMEANS nuovo \r\n");
  dim3 DimBlockInizToZero(256, 1, 1);
  dim3 DimGridInizToZero(((2 * K) / 256) + 1, 1, 1);
  dim3 DimBlock2(n, k, 1);
  dim3 DimBlockAssignment(1, n, 1);
  dim3 DimBlockUpdate(P, k, 1);
  dim3 DimGrid(1, 1, 1);
  dim3 DimBlockSum(P, 256, 1);
  dim3 DimGridSum(1, (N / 256) + 1, 1);
  dim3 DimBlockMean(P, 256, 1);
  dim3 DimGridMean(1, (K / 256) + 1, 1);
  dim3 DimBlockDistances(4, 256, 1);
  dim3 DimGridDistances((k / 4) + 1, (n / 256) + 1, 1);
  dim3 DimBlockMin(1, 256, 1);
  dim3 DimGridMin(1, (n / 256) + 1, 1);
  dim3 DimBlockAss(1, 256, 1);
  dim3 DimGridAss(1, (n / 256) + 1, 1);
  dim3 DimGridInizPart((k / 1024) + 1, 1, 1);
  dim3 DimBlockInizPart(1024, 1, 1);
  dim3 DimGridInizMean(1, k / 16 + 1, 1);
  dim3 DimBlockInizMean(P, 16, 1);
  cudaMemset(devAssignment, 0, n * sizeof(short));
  initializeAssignment<<<DimGridInizPart, DimBlockInizPart
      ↪ >>>(
      devAssignment, n);
  cudaDeviceSynchronize();
  initializeMean<<<DimGridInizMean, DimBlockInizMean>>>(
      ↪ devMean, devData, n,
      P);
  cudaDeviceSynchronize();
  short *devOldAssignment;
  CUDA_CHECK_RETURN(
      cudaMalloc((void** )&devOldAssignment, n * sizeof(short
          ↪ )));
  float *hostData, *hostMean, *hostOldMean;
  hostData = (float*) malloc(n * P * sizeof(float));
  short *hostAssignment, *hostOldAssignment;
  hostAssignment = (short*) malloc(n * sizeof(short));
  hostOldAssignment = (short*) malloc(n * sizeof(short));
  int *deviceOutputVector;
  hostMean = (float*) malloc(k * P * sizeof(float));
  hostOldMean = (float*) malloc(k * P * sizeof(float));
  CUDA_CHECK_RETURN(
      cudaMalloc((void** )&deviceOutputVector, k * sizeof(int
          ↪ )));
  bool stopCriterion = false;
  float *devDistances;
  CUDA_CHECK_RETURN(
      cudaMalloc((void** )&devDistances, n * k * sizeof(float
          ↪ )));
  int* devCounter, *hostCounter;
  CUDA_CHECK_RETURN(cudaMalloc((void** )&devCounter, k *
      ↪ sizeof(int)));
  hostCounter = (int*) malloc(k * sizeof(int));
  while (!stopCriterion) {
    CUDA_CHECK_RETURN(cudaMemset(devOldAssignment, 0, n *
        ↪ sizeof(short)));
    CUDA_CHECK_RETURN(cudaMemset(devCounter, 0, k * sizeof(
        ↪ int)));
    cudaDeviceSynchronize();
    CUDA_CHECK_RETURN(
        cudaMemcpy(devOldAssignment, devAssignment, n *
            ↪ sizeof(short),
            cudaMemcpyDeviceToDevice));
    cudaDeviceSynchronize();
    computeDistances<<<DimGridDistances, DimBlockDistances
        ↪ >>>(devData,
        devMean, devDistances, P, k);
    cudaDeviceSynchronize();
    computeMin<<<DimGridMin, DimBlockMin>>>(devDistances,
        ↪ devAssignment, k,
        n);
    cudaDeviceSynchronize();
    CUDA_CHECK_RETURN(
```

```
        cudaMemcpy(hostOldMean, devMean, P * k * sizeof(float ),
            cudaMemcpyDeviceToHost));
    initializeVectorToValue<<<DimGridInizToZero, DimBlockInizToZero>>>(
        devMean, 0.0, P * K);
    cudaDeviceSynchronize();
    computeSum37<<<DimGridSum, DimBlockSum>>>(devData, devAssignment,
        devMean, devCounter);
    cudaDeviceSynchronize();
    computeMean37<<<DimGridMean, DimBlockMean>>>(devMean, devCounter);
    cudaDeviceSynchronize();
    CUDA_CHECK_RETURN(
        cudaMemcpy(hostMean, devMean, P * k * sizeof(float),
            cudaMemcpyDeviceToHost));
    stopCriterion = stopCriterionSequential(hostOldMean, hostMean,
    K * P);
    /*
     printFile("/home/cecca/Desktop/RemoteCompilation/TXTDATA /",
     to_string(iter) + ".txt", n, k, P, devData,
     devAssignment, devMean);
    */
  }
  return 0;
}


void printFileForSeq(std::string path, std::string filename, float *dataset,
    short *assignment, float *mean) {
  ofstream outputfile;
  outputfile.open(path + filename, ios::out);
  for (int i = 0; i < K; i++) {
    std::string toFile = "media" + to_string(i);
    for (int j = 0; j < P; j++) {
      toFile = toFile + " " + to_string(mean[i * P + j]);

    }
    toFile = toFile + "|";
    outputfile << toFile;
  }
  for (int i = 0; i < N; i++) {
    std::string toFile = "";
    toFile = to_string(assignment[i]);
    for (int j = 0; j < P; j++) {
      toFile = toFile + " " + to_string(dataset[i * P + j]);
    }
    toFile = toFile + "|";
    outputfile << toFile;
  }
  outputfile.close();
}


int main(int argc, char *argv[]) {
  system("rm -rf /home/cecca/Desktop/RemoteCompilation/ TXTDATA");
  system("mkdir /home/cecca/Desktop/RemoteCompilation/TXTDATA ");
  size_t n;
  size_t i;
  int k;
  int P = 2;
  if (argc > 1) {
    n = atoi(argv[1]);
  } else {
    n = N;
  }
  if (argc > 2) {
    k = atoi(argv[2]);
  } else {
    k = K;
  }

  float *devCentroids, *hostCentroids;
  int sizeOfData = P * n * sizeof(float);
  short *hostAssignment, *devAssignment;
  float *devData, *hostData;
  float *devMean, *hostMean;
  hostData = (float*) malloc(sizeOfData);
  hostCentroids = (float*) malloc((sizeOfData * k) / n);
  hostMean = (float*) malloc((sizeOfData * k) / n);
  hostAssignment = (short*) malloc(n * sizeof(short));
  CUDA_CHECK_RETURN(cudaMalloc((void** )&devData, sizeOfData) );
  CUDA_CHECK_RETURN(cudaMalloc((void** )&devMean, (sizeOfData  * k) / n));
  CUDA_CHECK_RETURN(cudaMalloc((void** )&devAssignment, n * sizeof(short)));
  CUDA_CHECK_RETURN(cudaMalloc((void** )&devCentroids, ( sizeOfData * k) / n));
  generateRandomDataOnDevice(devData, n, P);
//################################
  clock_t start, end;
  double cpu_time_used;
  start = clock();
  kMeans(devData, devAssignment, devCentroids, n, k, P);
  end = clock();
  cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
  printf("TEMPO: %f SECONDI \r\n", cpu_time_used);
//################################
  CUDA_CHECK_RETURN(cudaFree(devAssignment));
  CUDA_CHECK_RETURN(cudaFree(devMean));
  CUDA_CHECK_RETURN(cudaFree(devData));
  free(hostData);
  free(hostMean);
  free(hostAssignment);

  return 0;

}

static void CheckCudaErrorAux(const char *file, unsigned line ,
    const char *statement, cudaError_t err) {
  if (err == cudaSuccess)
    return;
  std::cerr << statement << " returned " << cudaGetErrorString(err) << "("
      << err << ") at " << file << ":" << line << std::endl;
  exit(1);
}
```

Riferimenti bibliografici

[1] Wikipedia. K-means clustering — wikipedia, the free encyclopedia, 2017. [Online; accessed 23-January-2017]. 1