K-Means in Cuda

Irene Dini
Indirizzo e-mail
irene.dini1@stud.unifi.it

Francesca Natale
Indirizzo e-mail
francesca.natale@stud.unifi.it

Abstract

L'argomento di questo articolo è l'implementazione dell'algoritmo K-Means utilizzando CUDA. Tale algoritmo ha come obiettivo quello di clusterizzare un insieme di N punti in K gruppi, in base alla loro posizione nel piano.

1. K-Means Clustering

1.1. Problema del clustering

Il clustering è il problema di assegnare tutti gli oggetti di un insieme, a dei gruppi detti **clusters** in modo che gli oggetti che appartengono allo stesso cluster siano più simili possibile. In particolare, K-Means si occupa di clusterizzare N punti (x_i, y_i) in K cluster (con N>K), basandosi sulla loro distanza. Per calcolare quest'ultima abbiamo utlizzato la norma-2, ma si potrebbero anche utilizzare criteri diversi. Ogni cluster è identificato dal suo centroide, ovvero un punto (non necessariamente esistente), le cui componenti x ed y corrispondono alla media delle componenti di tutti i punti che appartengono al cluster stesso. Quindi il centroide del generico cluster C_i sarà il punto

$$\left(\frac{\sum_{k=0}^{|C_i|} x_k}{|C_i|}, \frac{\sum_{k=0}^{|C_i|} y_k}{|C_i|} \right), \quad \forall (x_k, y_k) \in C_i$$

.

1.2. Descrizione dell'algoritmo

L'algoritmo opera in modo iterativo, raffinando ad ogni passo la soluzione. Si parte inizializzando i centroidi dei K clusters con K punti scelti casualmente tra gli N da clusterizzare (nel nostro caso i primi K). Inizia quindi il ciclo di iterazioni che si divide in 2 passi:

- Assignment Step: Ciascun punto viene assegnato al cluster più vicino; si calcolano le distanze tra ogni punto ed ogni centroide, e ciascun punto viene collocato nel cluster a distanza minore;
- Update Step: si aggiornano i valori dei centroidi di ogni cluster.

L'algoritmo termina quando il valore di tutti i centroidi varia meno di una quantità pari ad ϵ rispetto all'iterazione precedente.

1.3. Generazione dati

Sia per la versione parallela che per quella sequenziale, l'insieme dei punti da clusterizzare è stato generato attraverso le API CUDA. In particolare abbiamo generato 2N numeri, che corrispondono alle componenti x e y degli N punti. Essi sono numeri pseudocasuali, che variano tra 0 ed 1 secondo una distribuzione uniforme, con un SEED pari a 234ULL. Questa generazione è effettuata dal metodo init qui sotto riportato.

```
int init(float *vectorsDev, int n){
  curandGenerator_t gen;
  CURAND_CALL(curandCreateGenerator(&gen,
      ↪ CURAND_RNG_PSEUDO_DEFAULT));
  CURAND_CALL(curandSetPseudoRandomGeneratorSeed(
      ↪ gen, 234ULL));
  CURAND_CALL(curandGenerateUniform(gen,
      ↪ vectorsDev, 2*n));
  CURAND_CALL(curandDestroyGenerator(gen));
  return 0;
}
```

2. Versione sequenziale

L'algoritmo sequenziale da noi implementato fa un ciclo sulla condizione di arresto, eseguendo ogni volta i due passi del K-Means descritti in precedenza. I parametri del metodo sono i valori di K, N ed ϵ. Qui di seguito descriveremo la struttura di questo algoritmo.

2.1. Strutture dati utilizzate

Le strutture dati utilizzate sono le seguenti:

- Vettore dei dati (hostData): vettore di lunghezza 2N, in cui le componenti $2i$ e $2i+1$ contengono rispettivamente la componente x e la componente y dell'i-esimo punto.
- Vettore dei centroidi (means): vettore di lunghezza 2K, in cui le componenti $2i$ e $2i+1$ contengono rispettivamente la componente x e la componente y del centroide dell'i-esimo cluster.
- Vettore di appartenenza (clusters): vettore di lunghezza N, che nella posizione i contiene l'indice del cluster a cui appartiene l'i-esimo punto.
- oldMeanX, oldMeanY: valori di appoggio utilizzati per calcolare la variazione del valore dei centroidi tra un'iterazione e l'altra, per il criterio di arresto.
- stop: variabile booleana che viene impostata a *true* quando tutti i centroidi hanno subito una variazione minore di ϵ.

2.2. Assignment Step

In questo passo del K-Means l'algoritmo calcola sequenzialmente le distanze tra ogni punto ed ogni cluster. Tale operazione viene fatta tramite due cicli for annidati: il più esterno scorre i vettori da 0 a N-1, mentre il più interno scorre i cluster da 0 a K-1. Per ciascun punto, si fissa inizialmente la distanza minima (minDistance) ad un valore massimo e, ad ogni iterazione, se la distanza tra il punto ed il cluster considerato è minore della distanza minima, si aggiorna quest'ultima a tale valore. Calcolata la distanza da tutti i cluster, si aggiorna il vettore di appartenenza, inserendo il punto considerato nel cluster a distanza minore. Il costo totale di questa operazione è di $\mathcal{O}(N \cdot K)$ operazioni sequenziali.

```
for (int v = 0; v < N; v++) {
  float minDistance = 3.0;
  short minIndex = -1;
  float distance = 0;
  for (int c = 0; c < K; c++) {
    distance = pow((hostData[2*v]
     - mean[2*c]),2)
     + pow((hostData[2*v+1]
     - mean[2*c+1]),2);
    if (distance < minDistance) {
      minIndex = c;
      minDistance = distance;
    }
  }
  clusters[v] = minIndex;
}
```

2.3. Update step

Nell'update step, vengono ricalcolati i valori dei centroidi dei cluster. Tale operazione è eseguita in modo sequenziale, una volta per ogni cluster c; ci serviamo di due strutture dati di appoggio: numComponents cioè una variabile che conta il numero di punti nel cluster considerato, e il vettore arraySum definito come segue:

$$\text{arraySum}[0] = \sum_{i:(x_i,y_i)\in c} x_i$$

$$\text{arraySum}[1] = \sum_{i:(x_i,y_i)\in c} y_i$$

Esso contiene quindi la somma delle componenti x di tutti i punti del cluster nella posizione 0, mentre, in posizione 1, la somma delle componenti y.
Anche in questo caso abbiamo due cicli for annidati: uno esterno che scorre i cluster da 0 a K-1, e uno interno che scorre i vettori da 0 a N-1. Ogni volta che si passa al cluster successivo, le due strutture dati di appoggio vengono inizializzate nuovamente a 0. A questo punto si scorre il vettore di appartenenza ai cluster: ogni volta che si trova una corrispondenza col cluster considerato, si sommano le componenti del vettore ad arraySum, e si incrementa numComponents. Dopo che sono stati considerati tutti i vettori, si aggiorna il centroide. Se il nuovo valore è cambiato di una quantità maggiore di ϵ rispetto a quello precedente, si setta la variabile del criterio di arresto a *false*, così da continuare l'esecuzione dell'algoritmo. Anche la complessità di questo passo è $\mathcal{O}(N \cdot K)$

```
int numComponents = 0;
for (int i = 0; i < K; i++) {
  numComponents = 0;
  float *arraySum = (float*)calloc(2, sizeof(
      ↪ float));
  for (int j = 0; j < N; j++) {
    if (clusters[j] == i) {
      numComponents++;
      arraySum[0] += hostData[2*j];
      arraySum[1] += hostData[2*j+1];

    }
  }
  oldMeanX = mean[2*i];
  mean[2*i] = arraySum[0] / numComponents;
  oldMeanY = mean[2*i+1];
  mean[2*i+1] = arraySum[1] / numComponents;
  if(abs(mean[2*i] - oldMeanX) > EPS || abs(mean
      ↪ [2*i+1] - oldMeanY) > EPS){
    stop = false;
  }
}
```

2.4. Criterio di arresto

Il criterio di arresto si basa sulla valutazione della variabile stop, il cui valore viene settato durante l'update step. In questo passo infatti, ogni volta che si aggiornano i centroidi, si confronta il loro valore con quello che avevano all'iterazione precedente. Se almeno una delle componenti è variata più di ϵ, stop viene posto a *false*. In questo modo, se anche un solo centroide si sposta di una quantità maggiore di ϵ, l'algoritmo K-Means continua la sua esecuzione.

3. Versione parallela

Questa implementazione di K-Means sfrutta la potenza della GPU per parallelizzare l'algoritmo tramite CUDA. Il carico di lavoro di ciascun passo dell'algoritmo viene suddiviso tra più thread, che lavorano in modo parallelo migliorando così le prestazioni. Le strutture dati utilizzate sono le seguenti:

- Vettore dei dati nel device (vectorsDev): vettore di lunghezza 2N, in cui le componenti $2i$ e $2i+1$ contengono rispettivamente la componente x e la componente y dell'i-esimo punto.
- Vettore dei centroidi nel device (meansDev): vettore di lunghezza 2K, in cui le componenti $2i$ e $2i+1$ contengono rispettivamente la componente x e la componente y del centroide dell'i-esimo cluster.
- Matrice delle distanze nel device (distDev): matrice N*K dove l'elemento $\mathrm{distDev_{i,j}}$ contiene la distanza tra il vettore i e il cluster j.
- Vettore di appartenenza ai cluter nel device (clustersDev): vettore di lunghezza N, che nella posizione i contiene l'indice del cluster a cui appartiene l'i-esimo punto.
- Vettore della somma delle componenti nel device (sumComponentsDev): vettore di lunghezza 2*K in cui le componenti $2i$ e $2i+1$ contengono rispettivamente la somma delle componenti x e la somma delle componenti y di tutti i punti appartenenti all'i-esimo cluster.
- Vettore del numero delle componenti nel device (numComponentsDev): vettore di lunghezza K che all'indice i contiene il numero di punti appartenenti all'i-esimo cluster.
- Vettore di 0 nell'host (zeroArray): vettore di lunghezza 2K contentente tutti 0, utilizzato per settare a 0 il vettore sumComponentsDev.
- Variabili per il criterio di arresto nel device e nell'host (changedDev, changedHost): interi che hanno valore maggiore di 0 se la computazione deve terminare, e pari a 0 se deve fermarsi.

3.1. Inizializzazione

Per prima cosa è necessario inizializzare il vettore dei centroidi con i primi K punti. L'inizializzazione del vettore dei centroidi viene effettuata tramite una cudaMemcpy direttamente dal vettore dei dati; il vettore di appartenenza viene inizializzato invece con un ciclo che pone $\mathrm{clustersDev}[i] = i$ per $i = 0 \dots K$. Ad ogni iterazione è inoltre necessario settare nuovamente a 0 i vettori numComponentsDev e sumComponentsDev, e le variabili di arresto.

3.2. Assignment step

Questo passo è diviso in due metodi:

3.2.1 Calcolo delle distanze (calculateDistances)

Questo metodo è eseguito sul device su blocchi di dimensione (1024, 1, 1) su una griglia di dimensione (n/block.x+1, 1, 1). Ciascun thread è associato ad un punto e calcola quindi la distanza tra quel punto e tutti i cluster, scrivendo i valori calcolati nella matrice distDev. Dato che tutti i thread accedono al vettore dei cluster, esso è caricato in memoria shared, per rendere gli accessi più veloci (dal momento che la shared ha dimensione massima pari a 48KB per blocco, questa versione funziona solo per $K \leq 1024$)[1]. Il metodo stesso è quindi diviso in 2 fasi: la fase di caricamento, dove il thread i carica nella memoria shared le componenti x e y del cluster i; e la fase di calcolo dove il thread i calcola la distanza tra il punto i e i K cluster. Le due fasi sono separate da una syncthreads poiché è necessario che tutti i caricamenti nella shared siano stati completati prima di poter procedere. Il costo totale dell'operazione è in questo caso $\mathcal{O}(N+K)$.

```
__global__ void calculateDistances(float*
    ↪ vectorsDev, float* meansDev, float*
    ↪ distDev, int k, int n){
  __shared__ float meansShared[2*1024];
  int tx = blockDim.x*blockIdx.x+threadIdx.x;
  int cluster = threadIdx.x;
  if(cluster<k){
    meansShared[2*cluster] = meansDev[2*cluster];
    meansShared[2*cluster+1] = meansDev[2*cluster
        ↪ +1];
  }
```

[1] Alleghiamo comunque una versione che funziona anche per $K > 1024$, in cui il $\mathrm{thread_{i,j}}$ calcola la distanza tra il vettore i e il cluster j

```
  __syncthreads();
  if(tx<n){
    float v1 = vectorsDev[2*tx];
    float v2 = vectorsDev[2*tx+1];
    for(int c = 0; c<k; c++){
      distDev[k*tx+c] = (v1-meansShared[2*c])*(v1
          ↪ -meansShared[2*c]) + (v2-
          ↪ meansShared[2*c+1])*(v2-meansShared
          ↪ [2*c+1]);
    }
  }
}
```

3.2.2 Assegnamento al cluster più vicino (assignmentStep)

Le dimensioni dei blocchi e della griglia per questo metodo sono uguali a quelle precedenti. In questo metodo ciascun thread è associato ad un punto; il thread i scorre l'i-esima riga della matrice delle distanze, effettuando una ricerca sequenziale del minimo per trovare il cluster più vicino al punto. Assegna quindi il punto a tale cluster nel vettore dei cluster. La complessità di questa operazione è $\mathcal{O}(K)$.

```
__global__ void assignmentStep(float* distDev,
    ↪ int* clustersDev, int k){
  int index = blockIdx.x*blockDim.x+threadIdx.x;
  float minDist = 3;
  float dist = 3;
  int cluster = 0;

  for(int i = index*k; i<(index+1)*k; i++){
    dist = distDev[i];
    if(dist<minDist){
      minDist = dist;
      cluster = i-(index*k);
    }
  }
  clustersDev[index] = cluster;
}
```

3.3. Update step

Anche questo passo è diviso in 2 metodi:

3.3.1 sumStep

In questo caso le dimensioni di ogni blocco sono (512, 2, 1), mentre quelle della griglia sono ((n/block.x)+1, 1, 1). Ad ogni punto sono associati 2 thread, uno per la componente x (con ty = 0), ed uno per la componente y (con ty = 1). Tali thread hanno il compito di sommare il valore x e il valore y del punto, agli elementi di sumComponentsDev corrispondenti al cluster a cui esso appartiene. Questa somma viene fatta attraverso un'operazione atomica per evitare situazioni di race condition. Tale operazione ha costo costante.

```
__global__ void sumStep(int* clustersDev, float*
    ↪ sumComponentsDev, int *numComponentsDev,
    ↪ float* vectorsDev, int n){
  int tx = blockIdx.x*blockDim.x+threadIdx.x;
  int ty = blockIdx.y*blockDim.y+threadIdx.y;
  if(tx<n){
    int c = clustersDev[tx];
    atomicAdd(&(sumComponentsDev[2*c+ty]),
        ↪ vectorsDev[2*tx+ty]);
    atomicAdd(&(numComponentsDev[c]), 1);
  }
}
```

3.3.2 divideStep

Le dimensioni dei blocchi sono uguali a quelle del metodo precedente, mentre quelle della griglia sono ((k/block.x)+1, 1, 1). In questo metodo ci sono 2 thread per ogni centroide, uno per la componente x ed uno per la componente y. Entrambi calcolano la media per la componente associata, dividendo il contenuto di sumComponentsDev per quello di numComponentsDev diviso 2. Questo perchè nel metodo precedente, dato che ci sono 2 thread per ogni vettore, erano state effettuate il doppio delle somme.
Per il criterio di arresto viene fatto un confronto con i precedenti valori dei centroidi: se si verifica una variazione maggiore di ϵ, viene effettuata una somma atomica su changedDev. Anche questa operazione ha costo costante.

```
__global__ void divideStep(float*
    ↪ sumComponentsDev, int* numComponentsDev,
    ↪ float* meansDev, int* changedDev, int k,
    ↪ float eps){
  int tx = blockIdx.x*blockDim.x+threadIdx.x;
  int ty = blockIdx.y*blockDim.y+threadIdx.y;
  if(tx<k){
    float oldMean = meansDev[2*tx+ty];
    float newMean = sumComponentsDev[2*tx+ty]/(
        ↪ numComponentsDev[tx]/2);
    meansDev[2*tx+ty] = newMean;
    if(abs(oldMean-newMean)>eps){
      atomicAdd(&(changedDev[0]), 1);
    }
  }
}
```

3.3.3 Criterio di arresto

Il criterio di arresto si basa su una coppia di variabili: changedDev e changedHost. Durante l'esecuzione di divideStep, changedDev viene incrementato di uno se almeno una

componente di un centroide varia di un valore maggiore di ϵ. Al termine di ogni iterazione si copia il valore di changedDev in changedHost, in modo che l'host possa terminare nel caso in cui changedHost sia uguale a zero.

3.4. Vantaggi della parallelizzazione

Utilizzando la parallelizzazione si sono riscontrati dei vantaggi in tutti e 4 i metodi. Per quanto riguarda il calcolo delle distanze, nella versione sequenziale, tutte le N*K distanze venivano calcolate sequenzialmente; in quella parallela, ciascun thread calcola in modo sequenziale solamente K distanze.
Per quanto riguarda la ricerca della distanza minima tra ciascun punto ed ogni cluster, nella versione parallela, ogni thread calcola il minimo di un vettore di dimensione K, mentre in quella sequenziale l'unico thread deve calcolare il minimo tra N vettori (sempre di lunghezza K).
Nel passo di aggiornamento invece, nella versione parallela prima 2N thread eseguono una somma ciascuno, e poi 2K thread eseguono una divisione. In quella sequenziale invece dobbiamo calcolare sequenzialmente 2N somme e 2K divisioni.
Quindi la complessità totale dell'algoritmo diminuisce di molto, riducendo notevolmente il tempo di esecuzione (per input grandi). In particolare si passa da $\mathcal{O}(N \cdot K)$ a $\mathcal{O}(N)$, in quanto $N \geq K$.

4. Tempi di esecuzione

Tabella 1: Tabella dei tempi di esecuzione

K	N	ϵ	sequenziale	parallela	iter
3	10	10^{-3}	0.00005	0.00028	2
10	100	10^{-3}	0.000447	0.000976	7
25	1000	10^{-3}	0.059922	0.015227	52
250	1000	10^{-3}	0.078118	0.012872	7
250	10000	10^{-4}	2.19656	0.056175	34
1000	50000	10^{-6}	60.1764	1.42126	52
1000	80000	10^{-6}	134.85	2.47823	72
1000	100000	10^{-6}	191.808	2.4717	84
1000	150000	10^{-6}	371.731	3.30942	108

Come si può osservare nella tabella, per valori piccoli di N la versione sequenziale risulta più efficiente. Infatti i tempi di esecuzione di quella parallela sono maggiori, dal momento che in essa devono essere gestite più strutture dati. Per valori più grandi, invece, la versione parallela risulta migliore. Ad esempio per N=150000 e K=1000, la versione sequenziale impiega 371.731 secondi, mentre quella parallela 3.30942 secondi: facendo un confronto, a questa serve quindi un tempo quasi 100 volte inferiore per terminare. Se consideriamo K=1000, nell'algoritmo parallelo per N=80000 e N=100000 abbiamo dei tempi di esecuzione molto simili tra loro, ossia 2.47823 e 2.4717 secondi rispettivamente. Nella versione sequenziale riscontriamo invece dei tempi pari a 134.85 e 191.808: in questo caso la differenza è evidente.

4.1. Speed UP

Tabella 2: Tabella dello speed UP

K	N	speed UP
250	1000	6.069
250	10000	39.102
1000	50000	42.340
1000	80000	54.414
1000	100000	77.602
1000	150000	112.325

Come si può notare dalla tabella, all'aumentare di N e di K, le prestazioni aumentano notevolmente. Per N = 150000 e K = 1000, lo speed UP è addirittura maggiore del 100%.

4.2. Esempio di clusterizzazione

Ecco qui riportato un esempio di clusterizzazione con N=1000 e K=100.

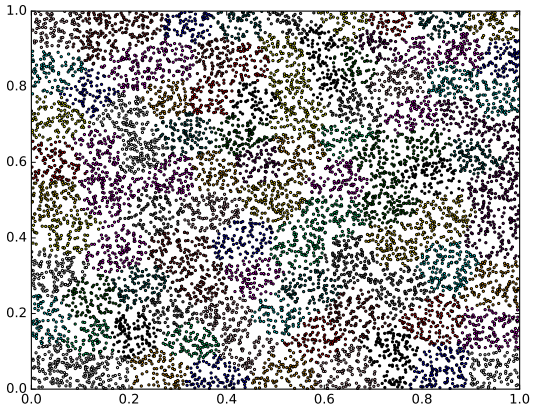

Figura 1: Clusterizzazione con N=1000 e k=100

4.3. Conclusioni

In conclusione, la versione parallela di K-Means, risulta molto più efficiente di quella sequenziale, come abbiamo mostrato in precedenza. Questo perchè i thread possono eseguire molte operazioni in parallelo, in quanto esse sono

indipendenti tra loro. I difetti di questa implementazione sono dovuti al fatto che alcune operazioni richiedono un accesso in mutua esclusione a locazioni della memoria, causando rallentamenti. Inoltre, quando molti thread scrivono in un unica variabile, si può avere perdita di precisione, rendendo il numero di iterazioni leggermente variabile.

5. Codice completo

5.1. Versione sequenziale

Listing 1: Versione sequenziale

```
#include <iostream>
#include <numeric>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda.h>
#include <curand.h>
#include <math.h>
#include <fstream>
#include <sstream>
#include <cstdio>
#include <ctime>

static void CheckCudaErrorAux (const char *,
    ↪ unsigned, const char *, cudaError_t);
void sequentialKMeans(int N, int K, int EPS,
    ↪ float*devData);

#define N 80000
#define K 200
#define EPS 0.000001

#define CUDA_CHECK_RETURN(value)
    ↪ CheckCudaErrorAux(__FILE__,__LINE__, #
    ↪ value, value)
#define CUDA_CALL(x) do { if((x)!=cudaSuccess) {
    ↪ printf("Error at %s:%d\n",__FILE__,
    ↪ __LINE__);return EXIT_FAILURE;}} while(0)
#define CURAND_CALL(x) do { if((x)!=
    ↪ CURAND_STATUS_SUCCESS) {printf("Error at
    ↪ %s:%d\n",__FILE__,__LINE__); return
    ↪ EXIT_FAILURE;}} while(0)
using namespace std;


int init(float *vectorsDev){
  curandGenerator_t gen;
  CURAND_CALL(curandCreateGenerator(&gen,
      ↪ CURAND_RNG_PSEUDO_DEFAULT));
  CURAND_CALL(curandSetPseudoRandomGeneratorSeed(
      ↪ gen, 234ULL));
  CURAND_CALL(curandGenerateUniform(gen,
      ↪ vectorsDev, 2*N));
  CURAND_CALL(curandDestroyGenerator(gen));
  return 0;
}

int main(int argc, char* argv[]){
  std::clock_t start;
  double duration;

  int iter = 0;
  float *devData, *hostData, *mean;
  bool stopCriterion = false;

  short *clusters;
  float oldMeanX, oldMeanY;

  hostData = (float*) malloc(2*N*sizeof(float));
  mean = (float*) malloc(2*K*sizeof(float));
  clusters = (short*) malloc(N*sizeof(short));


  CUDA_CALL(cudaMalloc((float **)&devData, 2*N*
      ↪ sizeof(float)));
  init(devData);
  CUDA_CHECK_RETURN(cudaMemcpy(hostData, devData,
      ↪  2*N*sizeof(float),
      ↪ cudaMemcpyDeviceToHost));
  cudaDeviceSynchronize();

  memcpy(mean, hostData, 2*K*sizeof(float));

  start = std::clock();
  while (!stopCriterion) {
    stopCriterion = true;
    /*
     * distanze e minimo
     */
    for (int v = 0; v < N; v++) {
      float minDistance = 3.0;
      short minIndex = -1;
      float distance = 0;
      for (int c = 0; c < K; c++) {
        distance = pow((hostData[2*v] - mean[2*c
            ↪ ]),2)+pow((hostData[2*v+1] - mean
            ↪ [2*c+1]),2);

        if (distance < minDistance) {
          minIndex = c;
          minDistance = distance;
        }
```

```
      }
      clusters[v] = minIndex;
    }
    /*
     * nuova media
     */
    for (int i = 0; i < K; i++) {
      int numComponents = 0;
      float *arraySum = (float*)calloc(2, sizeof(
          ↪ float));
      for (int j = 0; j < N; j++) {
        if (clusters[j] == i) {
          numComponents++;
          arraySum[0] += hostData[2*j];
          arraySum[1] += hostData[2*j+1];

        }
      }
      oldMeanX = mean[2*i];
      mean[2*i] = arraySum[0] / numComponents;
      oldMeanY = mean[2*i+1];
      mean[2*i+1] = arraySum[1] / numComponents;
      if(abs(mean[2*i] - oldMeanX) > EPS || abs(
          ↪ mean[2*i+1] - oldMeanY) > EPS){
        stopCriterion = false;
      }
    }

    iter++;
  }
  printf("Numero di iterazioni: %d\r\n", iter);
  duration = ( std::clock() - start ) / (double)
      ↪ CLOCKS_PER_SEC;
  std::cout<<"durata: "<< duration <<'\n';

  free(hostData);
  free(mean);
  free(clusters);

  CUDA_CHECK_RETURN(cudaFree(devData));

  return EXIT_SUCCESS;
}

/**
 * Check the return value of the CUDA runtime API
     ↪  call and exit
 * the application if the call has failed.
 */
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);
}
```

5.2. Versione parallela

Listing 2: Versione parallela

```
#include <iostream>
#include <numeric>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda.h>
#include <curand.h>
#include <math.h>
#include <fstream>
#include <sstream>
#include <cstdio>
#include <ctime>

static void CheckCudaErrorAux (const char *,
    ↪ unsigned, const char *, cudaError_t);


#define N 80000
#define K 200
#define EPS 0.000001

#define CUDA_CHECK_RETURN(value)
    ↪ CheckCudaErrorAux(__FILE__,__LINE__, #
    ↪ value, value)
#define CUDA_CALL(x) do { if((x)!=cudaSuccess) {
    ↪ printf("Error at %s:%d\n",__FILE__,
    ↪ __LINE__);return EXIT_FAILURE;}} while(0)
#define CURAND_CALL(x) do { if((x)!=
    ↪ CURAND_STATUS_SUCCESS) {printf("Error at
    ↪ %s:%d\n",__FILE__,__LINE__); return
    ↪ EXIT_FAILURE;}} while(0)

using namespace std;


int init(float *vectorsDev){
  curandGenerator_t gen;
```

```
  CURAND_CALL(curandCreateGenerator(&gen,
      ↪ CURAND_RNG_PSEUDO_DEFAULT));
  CURAND_CALL(curandSetPseudoRandomGeneratorSeed(
      ↪ gen, 234ULL));
  CURAND_CALL(curandGenerateUniform(gen,
      ↪ vectorsDev, 2*N));
  CURAND_CALL(curandDestroyGenerator(gen));
  return 0;
}


__global__ void initClustersArray(int*
    ↪ clustersDev){
  int i = blockIdx.x*blockDim.x+threadIdx.x;
  if(i<K){
    clustersDev[i]=i;
  }
}

__global__ void calculateDistances(float*
    ↪ vectorsDev, float* meansDev, float*
    ↪ distDev){

  __shared__ float meansShared[2*1024];
  int tx = blockDim.x*blockIdx.x+threadIdx.x;
  int cluster = threadIdx.x;
  if(cluster<K){
    meansShared[2*cluster] = meansDev[2*cluster];
    meansShared[2*cluster+1] = meansDev[2*cluster
        ↪ +1];
  }
  __syncthreads();
  if(tx<N){
    float v1 = vectorsDev[2*tx];
    float v2 = vectorsDev[2*tx+1];
    for(int c = 0; c<K; c++){
      distDev[K*tx+c] = (v1-meansShared[2*c])*(v1
          ↪ -meansShared[2*c]) + (v2-
          ↪ meansShared[2*c+1])*(v2-meansShared
          ↪ [2*c+1]);
    }
  }
}


__global__ void assignmentStep(float* distDev,
    ↪ int* clustersDev){

  int index = blockIdx.x*blockDim.x+threadIdx.x;
  float minDist = 3;
  float dist = 3;
  int cluster = 0;

  if(index<N){
    for(int i = index*K; i<(index+1)*K; i++){
      dist = distDev[i];
      if(dist<minDist){
        minDist = dist;
        cluster = i-(index*K);
      }
    }
    clustersDev[index] = cluster;
  }
}

__global__ void sumStep(int* clustersDev, float*
    ↪ sumComponentsDev, int *numComponentsDev,
    ↪ float* vectorsDev){
  int tx = blockIdx.x*blockDim.x+threadIdx.x;
  int ty = blockIdx.y*blockDim.y+threadIdx.y;
  if(tx<N){
    int c = clustersDev[tx];
    atomicAdd(&(sumComponentsDev[2*c+ty]),
        ↪ vectorsDev[2*tx+ty]);
    atomicAdd(&(numComponentsDev[c]), 1);
  }

}

__global__ void divideStep(float*
    ↪ sumComponentsDev, int* numComponentsDev,
    ↪ float* meansDev, int* changedDev){
  int tx = blockIdx.x*blockDim.x+threadIdx.x;
  int ty = blockIdx.y*blockDim.y+threadIdx.y;
  if(tx<K){
    float oldMean = meansDev[2*tx+ty];
    float newMean = sumComponentsDev[2*tx+ty]/(
        ↪ numComponentsDev[tx]/2);
    meansDev[2*tx+ty] = newMean;
    if(abs(oldMean-newMean)>EPS){
      atomicAdd(&(changedDev[0]), 1);
    }
  }
}


int main(int argc, char* argv[]){
  std::clock_t start;
  double duration;

  int iter = 0;
  float *vectorsDev, *meansDev, *distDev, *
      ↪ sumComponentsDev, *zeroArray;
  int *clustersDev, *changedDev, *
      ↪ numComponentsDev;
  int changedHost = 1;
```

```
zeroArray = (float *)calloc(2*K, sizeof(float))
    ↪ ;


CUDA_CALL(cudaMalloc((float **)&vectorsDev, 2*N
    ↪ *sizeof(float)));
CUDA_CALL(cudaMalloc((int **)&clustersDev, N*
    ↪ sizeof(int)));
CUDA_CALL(cudaMalloc((float **)&meansDev, 2*K*
    ↪ sizeof(float)));
CUDA_CALL(cudaMalloc((float **)&distDev, N*K*
    ↪ sizeof(float)));
CUDA_CALL(cudaMalloc((int **)&changedDev,
    ↪ sizeof(int)));
CUDA_CALL(cudaMalloc((float **)&
    ↪ sumComponentsDev, 2*K*sizeof(float)));
CUDA_CALL(cudaMalloc((int **)&numComponentsDev,
    ↪  K*sizeof(int)));


init(vectorsDev);

dim3 block1(K,1,1);
dim3 grid1(ceil(K/block1.x),1,1);

dim3 block3(1024,1,1);
dim3 grid3(ceil(N/block3.x)+1,1,1);

dim3 block4(1024,1,1);
dim3 grid4(ceil(N/block4.x)+1,1,1);

dim3 block5(512,2,1);
dim3 grid5(ceil(N/block5.x)+1,1,1);

dim3 block6(512,2,1);
dim3 grid6(ceil(K/block6.x)+1,1,1);

initClustersArray<<<grid1, block1>>>(
    ↪ clustersDev);

CUDA_CHECK_RETURN(cudaMemcpy(meansDev,
    ↪ vectorsDev, 2*K*sizeof(float),
    ↪ cudaMemcpyDeviceToDevice));

start = std::clock();

while(changedHost>0){

  iter++;

  //reset values
    CUDA_CHECK_RETURN(cudaMemset(changedDev, 0,
        ↪ sizeof(int)));
    CUDA_CHECK_RETURN(cudaMemcpy(sumComponentsDev
        ↪ , zeroArray, 2*K*sizeof(float),
        ↪ cudaMemcpyHostToDevice));
    CUDA_CHECK_RETURN(cudaMemset(numComponentsDev
        ↪ , 0, K*sizeof(int)));

    //compute
    calculateDistances<<<grid4,block4>>>(
        ↪ vectorsDev, meansDev, distDev);
    cudaDeviceSynchronize();
    assignmentStep<<<grid3,block3>>>(distDev,
        ↪ clustersDev);
    cudaDeviceSynchronize();
    sumStep<<<grid5,block5>>>(clustersDev,
        ↪ sumComponentsDev, numComponentsDev,
        ↪ vectorsDev);
    cudaDeviceSynchronize();
    divideStep<<<grid6, block6>>>(
        ↪ sumComponentsDev, numComponentsDev,
        ↪ meansDev, changedDev);
    cudaDeviceSynchronize();

    CUDA_CHECK_RETURN(cudaMemcpy(&changedHost,
        ↪ changedDev, sizeof(int),
        ↪ cudaMemcpyDeviceToHost));
  }

    duration = ( std::clock() - start ) / (double
        ↪ ) CLOCKS_PER_SEC;

    std::cout<<"durata: "<< duration <<'\n';

  printf("numero iterazioni = %d\n", iter);

  free(vectorsHost);
  free(clustersHost);
  free(zeroArray);

  CUDA_CHECK_RETURN(cudaFree(vectorsDev));
  CUDA_CHECK_RETURN(cudaFree(clustersDev));
  CUDA_CHECK_RETURN(cudaFree(meansDev));
  CUDA_CHECK_RETURN(cudaFree(distDev));
  CUDA_CHECK_RETURN(cudaFree(changedDev));

  return EXIT_SUCCESS;
}

/**
 * Check the return value of the CUDA runtime API
     ↪  call and exit
 * the application if the call has failed.
```

```
 */
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);
}
```

5.3. Versione alternativa di calculateDistances

```
__global__ void calculateDistances(float*
    ↪ vectorsDev, float* meansDev, float*
    ↪ distDev){

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int c = (blockDim.x*blockIdx.x)+tx;   //cluster
      ↪  index
  int v = (blockDim.y*blockIdx.y)+ty;   //vector
      ↪ index

  if(c<K && v<N){
    float dist = (vectorsDev[2*tx]-meansShared[2*
        ↪ c])*(vectorsDev[2*tx]-meansShared[2*c
        ↪ ]) + (vectorsDev[2*tx+1]-meansShared
        ↪ [2*c+1])*(vectorsDev[2*tx+1]-
        ↪ meansShared[2*c+1]);
    distDev[K*v+c] = dist;
  }

}
```