# Kmeans - CUDA

Alessandro Marinai, Eleonora Ristori

Aprile 2023

## 1 Introduzione

In questa relazione viene presentato il progetto in cui vengono implementate in CUDA due versioni dell'algoritmo K-Means. Le versioni sono una sequenziale e una parallela. Verrà mostrato come variano le performance di queste due versioni al variare dei parametri di input.

## 2 Implementazione generica dell'algoritmo di K-Means

Entrambe le versioni implementate hanno la stessa logica di funzionamento.
La prima cosa da specificare per la comprensione del codice è che, per ragioni legate alla performance dell'algoritmo, i punti sono espressi in modo linearizzato in un unico array.
In secondo luogo, dato che l'obiettivo di questo lavoro è quello di fare considerazioni sulla parallelizzazione dell'algoritmo e non di stabilire il suo funzionamento, i punti da clusterizzare vengono generati in cluster ben distinti di forma globulare. In particolare, viene generato il primo centroide randomicamente, mentre i successivi vengono selezionati solo se sufficientemente distanti dagli altri. Una volta selezionati i centroidi si procede alla generazione dei punti che avviene utilizzando una distribuzione gaussiana per ciascuna dimensione, centrata in un centroide scelto a caso e con varianza passata come parametro.
Ecco la funzione che si occupa della generazione dei punti:

```
void generateCluster(int num_points,int data_point_dim, int k, float sigma, float*data_points, float*centroids){
    //Assigning centroids
    for(int dim=0; dim<data_point_dim; dim++){
        centroids[dim] = float(std::rand()) / RAND_MAX *sigma*sigma*k;
    }
    int num_assigned_centroids = 1;

    while(num_assigned_centroids != k){

        for(int dim=0; dim<data_point_dim; dim++){
            centroids[num_assigned_centroids*data_point_dim+dim] =  float(std::rand()) / RAND_MAX *sigma*sigma*k;
        }
        int centroid_id = 0;
        float sum_distances;
        do {
            sum_distances = 0;
            for(int dim=0; dim<data_point_dim; dim++) {
                float diff = centroids[centroid_id*data_point_dim+dim] - centroids[num_assigned_centroids*data_point_dim+dim];
                sum_distances += diff*diff;
            }
            centroid_id++;
        }while(centroid_id < num_assigned_centroids && sum_distances > sigma*sigma*100);
        if(centroid_id == num_assigned_centroids){
            num_assigned_centroids++;
        }
    }


    //Generate random points
    std::default_random_engine generator;
    for(int point_id=0; point_id<num_points; point_id++){
        int cluster = int(std::rand()) % k;

        for(int dim=0; dim < data_point_dim; dim++){
            std::normal_distribution<double> distribution( Mean0: centroids[cluster*data_point_dim+dim], Sigma0: sigma);
            data_points[point_id*data_point_dim + dim] = float(distribution( &: generator));
        }
    }
}
```

Figure 1: Funzione per la generazione randomica dei punti

Dopo aver generato i punti, inizia la vera e propria esecuzione di K-means. Il primo step è quello di inizializzare i centroidi dei cluster con dei punti scelti tra l'insieme dei punti da clusterizzare. In particolare il primo viene preso a caso (nel nostro caso, per semplicità, viene scelto il primo punto) mentre i successivi vengono scelti in modo che la loro distanza da tutti gli altri punti scelti sia superiore di una certa threshold passata come parametro al metodo per l'inizializzazione dei centroidi.
Ecco la funzione che sceglie i centroidi iniziali:

```
void initialize_centroids(int data_point_dim, float* centroids, const float *data_points, int k, float threshold){
    for(int dim=0; dim<data_point_dim; dim++){
        centroids[dim] = data_points[dim];
    }
    int num_assigned_centroids = 1;
    int point_id = 1;
    while(num_assigned_centroids != k){

        for(int dim=0; dim<data_point_dim; dim++){
            centroids[num_assigned_centroids*data_point_dim+dim] =  data_points[point_id*data_point_dim+dim];
        }
        int centroid_id = 0;
        float sum_distances;
        do {
            sum_distances = 0;
            for(int dim=0; dim<data_point_dim; dim++) {
                float diff = centroids[centroid_id*data_point_dim+dim] - centroids[num_assigned_centroids*data_point_dim+dim];
                sum_distances += diff*diff;
            }
            centroid_id++;
        }while(centroid_id < num_assigned_centroids && sum_distances > threshold);
        if(centroid_id == num_assigned_centroids){
            num_assigned_centroids++;
            point_id++;
        }
        else {
            point_id++;
        }
    }
}
```

Figure 2: Funzione che determina i centroidi iniziali

L'assegnazione dei centroidi iniziali è eseguita da questa funzione sia nella versione sequenziale che in quella parallela.
L'algoritmo poi si articola in 3 fasi che si ripetono in un ciclo do-while:

- assegnazione al cluster: per ogni punto viene calcolata la distanza da ogni centroide e si assegna il punto al cluster con il centroide più vicino;
- calcolo dei nuovi centroidi: per ogni cluster viene ricalcolato il centroide;
- calcolo del criterio di arresto: viene calcolata la somma delle distanze tra la vecchia assegnazione dei centroidi e quella nuova.

Se quest'ultima somma di distanze non supera una certa tolleranza passata come parametro alla funzione significa che la nuova assegnazione è sostanzialmente la stessa della precedente iterazione e quindi l'algoritmo termina. L'algoritmo ha comunque un numero massimo di iterazioni per evitare loop infinito.

# 3 Versione sequenziale

Ecco la porzione di codice relativa all'iterazione principale della versione sequenziale di K-means:

```
// Main loop
int iteration = 0;
 do {

    // Assign each data point to the nearest centroid
    assign_cluster_s(cluster_assignment, data_points, centroids, num_data_points, num_centroids, data_point_dim);


    calculate_centroid_s(data_points, cluster_assignment, centroids, old_centroids, num_data_points, num_centroids, data_point_dim);


    iteration++;

}while (iteration < max_iterations && centroid_distance_s( o_centroids: old_centroids,  n_centroids: centroids,  k: num_centroids, data_point_dim) > tolerance);
```

Figure 3: Iterazione principale K-means sequenziale

La fase di assegnazione al cluster in figura 4 viene implementata scorrendo in un loop tutti i punti e determinando la distanza da ciascun cluster per determinare il più vicino.

```
void assign_cluster_s(int* cluster_assignment, const float* data_points, float* centroids, int num_data_points, int num_centroids, int data_point_dim) {
    for(int data_point_id=0; data_point_id<num_data_points; data_point_id++){
        float min_dist = INFINITY;
        int min_centroid_id = -1;
        for (int centroid_id = 0; centroid_id < num_centroids; centroid_id++) {
            float dist = 0;
            for (int dim = 0; dim < data_point_dim; dim++) {
                float diff = data_points[data_point_id * data_point_dim + dim] - centroids[centroid_id * data_point_dim + dim];
                dist += diff * diff;
            }
            if(dist < min_dist){
                min_dist = dist;
                min_centroid_id = centroid_id;
            }
        }
        cluster_assignment[data_point_id] = min_centroid_id;
    }
}
```

Figure 4: Assegnazione al cluster

Successivamente si ricalcolano i centroidi come media dei punti assegnati al cluster attraverso la funzione calculate_centroid_s() come mostrato in figura 5.

```
void calculate_centroid_s(const float* data_points, int* cluster_assignment, float* centroids, float* old_centroids, int num_data_points, int num_centroids, int data_point_dim)
    for(int centroid_id=0; centroid_id<num_centroids; centroid_id++){
        for (int dim = 0; dim < data_point_dim; dim++) {
            int num_points_assigned = 0;
            float sum = 0;
            for (int data_point_id = 0; data_point_id < num_data_points; data_point_id++) {
                if (cluster_assignment[data_point_id] == centroid_id) {
                    sum += data_points[data_point_id * data_point_dim + dim];
                    num_points_assigned++;
                }
            }
            if (num_points_assigned > 0) {
                old_centroids[centroid_id * data_point_dim + dim] = centroids[centroid_id * data_point_dim + dim];
                centroids[centroid_id * data_point_dim + dim] = sum / num_points_assigned;
            }
        }
    }

}
```

Figure 5: Ricalcolo dei centroidi

Il calcolo del criterio di arresto che misura la distanza tra i vecchi ed i nuovi centroidi è mostrato in figura 6.

```
float centroid_distance_s(float* o_centroids, float* n_centroids, int k, int data_point_dim){
    float distance = 0;
    float diff;
    for(int id = 0 ; id < k; id++){
        for(int dim=0; dim < data_point_dim; dim++){
            diff = o_centroids[id*data_point_dim+dim] - n_centroids[id*data_point_dim+dim];
            distance += diff * diff;
        }
    }
    return distance;
}
```

Figure 6: Computazione condizione d'arresto

# 4 Versione parallela

Anche in questa versione, dopo aver allocato la memoria per le diverse strutture dati necessarie sulla GPU, l'algoritmo si articola in un ciclo do-while con le stesse condizioni di terminazione del caso sequenziale. In figura 7 si mostra il loop principale dell'algoritmo parallelo.

```
do {
    // Calculate centroid_distance between each data point and centroid
    cudaMemset( devPtr: &d_updated_centroid,  value: 0,  count: num_centroids*data_point_dim*sizeof(float));
    cudaMemset( devPtr: &d_assigned_points,  value: 0,  count: num_centroids* sizeof(int));
    num_blocks = (num_data_points + num_threads_per_block - 1) / num_threads_per_block;

    // Assign each data point to the nearest centroid
    assign_cluster<<<num_blocks, num_threads_per_block>>>(d_cluster_assignment, d_data_points, d_centroids, num_data_points,
                                                          num_centroids, data_point_dim, d_updated_centroid, d_assigned_points);

    // Calculate new centroids
    num_blocks = (num_centroids + num_threads_per_block - 1) / num_threads_per_block;

    calculate_centroid<<<num_blocks, num_threads_per_block>>>(d_updated_centroid, d_assigned_points, d_centroids, old_centroids,
                                                              num_data_points, num_centroids, data_point_dim);

    // Check if centroids have moved more than tolerance
    tot_distance = 0;
    centroid_distance<<<num_blocks, num_threads_per_block>>>(old_centroids, d_centroids, num_centroids, data_point_dim, d_centr_distance);

    cudaMemcpy( dst: centr_distance,  src: d_centr_distance,  count: num_centroids*sizeof(float),  kind: cudaMemcpyDeviceToHost);

    for(int i=0; i<num_centroids; i++){
        tot_distance += centr_distance[i];
    }
    iteration++;
}while (iteration < max_iterations && tot_distance > tolerance);
```

Figure 7: Iterazione di K-Means Parallelo

Le espressioni per il calcolo del numero di blocchi sono volte a calcolare la parte intera superiore della divisione del numero di thread totali richiesti per il numero di thread per blocco.
La fase di assegnazione dei punti ai cluster è rappresentata in figura 8.

```
__global__ void assign_cluster(int *cluster_assignment, const float* data_points, const float* centroids, int num_data_points,
                               int num_centroids, int data_point_dim, float* updated_centroids, int* assigned_points) {
    int data_point_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (data_point_id >= num_data_points) {
        return;
    }
    float min_dist = INFINITY;
    int min_centroid_id = -1;
    for (int centroid_id = 0; centroid_id < num_centroids; centroid_id++) {
        float dist = 0;
        for (int dim = 0; dim < data_point_dim; dim++) {
            float diff = data_points[data_point_id * data_point_dim + dim] - centroids[centroid_id * data_point_dim + dim];
            dist += diff * diff;
        }
        if(dist < min_dist){
            min_dist = dist;
            min_centroid_id = centroid_id;
        }

    }
    cluster_assignment[data_point_id] = min_centroid_id;
    for (int dim = 0; dim < data_point_dim; dim++)
        atomicAdd( address: &updated_centroids[min_centroid_id*data_point_dim+dim],  val: data_points[data_point_id*data_point_dim+dim]);
    atomicAdd( address: &assigned_points[min_centroid_id],  val: 1);

}
```

Figure 8: Funzione parallela per l'assegnazione dei punti al cluster più vicino

Il ricalcolo dei centroidi si basa su un calcolo nel metodo precedente che somma atomicamente le coordinate dei punti via via che li assegna ai cluster mantenendo anche un contatore sul numero totale di punti per cluster. In questo modo si riesce a sfruttare al massimo il parallelismo offerto dalla GPU.

Il metodo calculate_centroid() semplicemente dovrà di dividere l'accumulazione eseguita nel metodo precedente per il numero di punti assegnati a quel cluster per ottenere la media.

```
__global__ void calculate_centroid(float* updated_centroids, int* assigned_points, float* centroids, float* old_centroids, int num_data_points,
                                   int num_centroids, int data_point_dim) {
    int centroid_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (centroid_id >= num_centroids) {
        return;
    }

    for (int dim = 0; dim < data_point_dim; dim++) {
        if (assigned_points[centroid_id] > 0) {
            old_centroids[centroid_id * data_point_dim + dim] = centroids[centroid_id * data_point_dim + dim];
            centroids[centroid_id * data_point_dim + dim] = updated_centroids[centroid_id*data_point_dim+dim] / assigned_points[centroid_id];
        }
    }
}
```

Figure 9: Calcolo centroidi

Il calcolo della distanza tra vecchi e nuovi centroidi avviene con il seguente metodo:

```
__global__ void centroid_distance(float* o_centroids, float* n_centroids, int k, int data_point_dim, float* distance){
    int centroid_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (centroid_id >= k) {
        return;
    }
    distance[centroid_id] = 0;
    float diff;
    for(int dim=0; dim < data_point_dim; dim++){
            diff = o_centroids[centroid_id*data_point_dim+dim] - n_centroids[centroid_id*data_point_dim+dim];
            distance[centroid_id] += diff * diff;
    }

}
```

Figure 10: Computazione condizione d'arresto

# 5 Analisi dei risultati

Al fine di valutare il beneficio offerto dalla versione parallela in termini di speed-up sono stati eseguiti diversi esperimenti. Lo speed-up è stato calcolato in termini di wall-clock time ovvero determinando il tempo di esecuzione dei due algoritmi.

## 5.1 Valutazione dello speed-up al variare del numero di punti

Il primo esperimento condotto è volto ad analizzare lo speed-up della versione parallela rispetto al caso sequenziale al variare della dimensione del problema. Per questo esperimento sono stati eseguite due misurazioni:

- la prima è stata eseguita considerando la condizione di early termination nel caso in cui i centroidi si siano spostati meno di una certa tolleranza prefissata in input. Si riportano i risultati ottenuti nel grafico in figura 11.

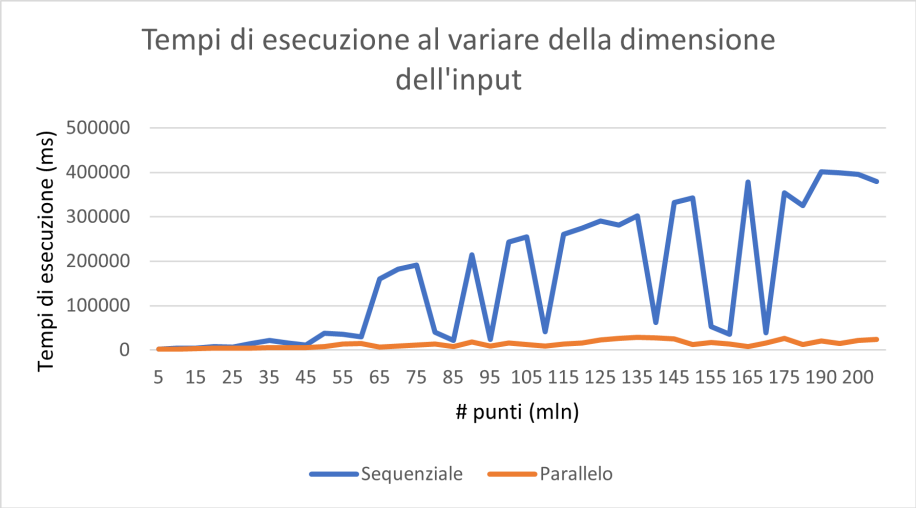


Figure 11: Grafico dei tempi di esecuzione al variare della dimensione dell'input con condizione di early termination

- la seconda invece è stata eseguita con un numero di iterazioni fisso escludendo la condizione precedenti. Il grafico in figura 12 mostra i risultati ottenuti in questo secondo caso.

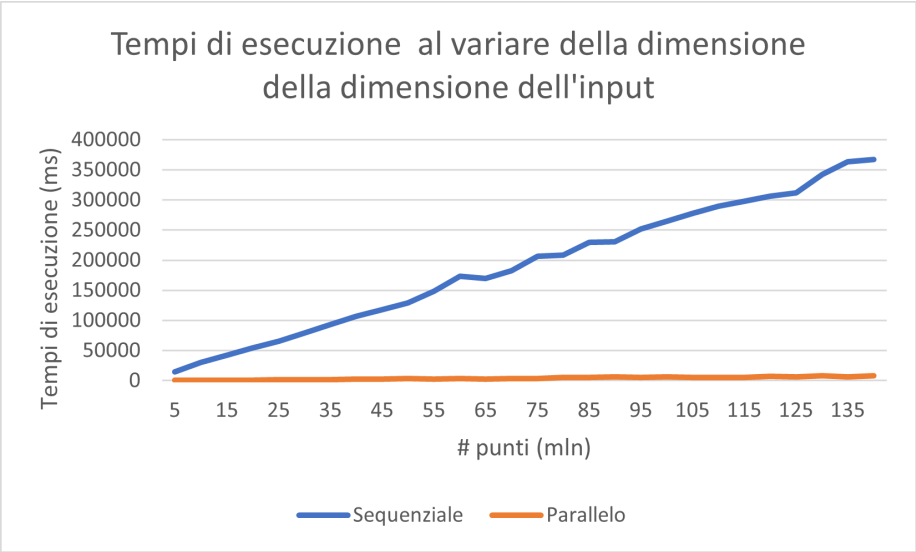


Figure 12: Grafico dei tempi di esecuzione al variare della dimensione dell'input con un numero di iterazioni fisso

In entrambi i casi mostrati sopra si osserva che l'andamento dei tempi di esecuzione per input che variano da 5 a 140 milioni di punti con un netto speed-up della versione parallela che si assesta intorno al valore 51.

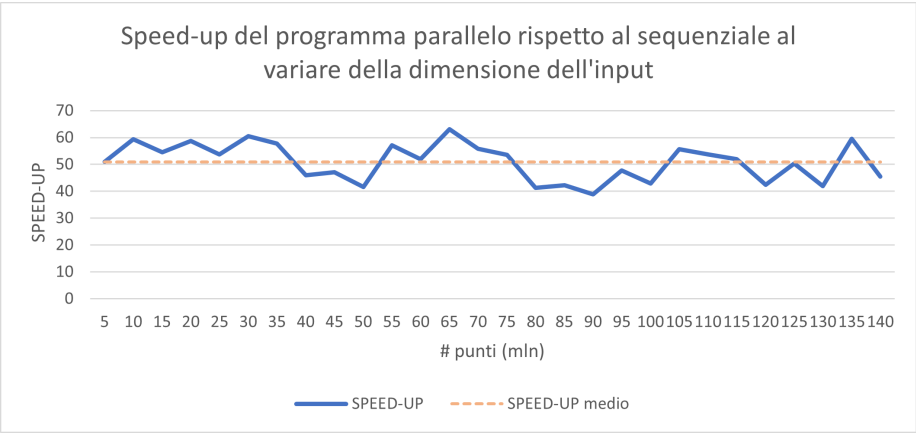


Figure 13: Grafico dello speed-up della versione parallela rispetto a quella sequenziale