

# KMEANS

## Programmazione di Sistemi Embedded e Multicore

16 marzo 2025

**Nome:** Rideewitage Lachitha Sangeeth

**Cognome:** Perera **Matricola:** 2042904

## 1 Introduzione

Per la parellizzazione del codice sequenziale di KMEANS, sono stati utilizzati due metodi, uno che consiste l'utilizzo di **OpenMP** insieme a **OpenMPI**, e l'altro metodo che utilizza interamente **CUDA**. Prima di analizzare i codici parallelizzati, è necessario capire il funzionamento del codice sequenziale di KMEANS, in quanto sono state apportate delle leggere modifiche per il funzionamento dei test.

La prima modifica effettuata al codice sequenziale è stata l'aggiunta di un parametro in input, che permette di salvare in un file csv il computation time, il quale verrà utilizzato per il calcolo della media dei tempi e confrontato con le altre versioni, per la realizzazione di ciò è stata aggiunta anche una funzione che scrive il computation time nel file specificato in input [2]. I file contenenti i computation times sono salvati in una directory specifica, ovvero:

comp\_time/{versione}/comp\_time{grandezza\_test}.csv.

Oltre a questo, per il corretto funzionamento di tutte le versioni, è stata apportata anche un modifica per quanto riguarda la funzione `euclideanDistance` [1]. Come è possibile notare dal codice, è stata

```
float euclideanDistance(float *point, float *center, int samples)
{
    float dist=0.0;
    for(int i=0; i<samples; i++)
    {
        /* Utilizzo della funzione fmaf per evitare
         * errori di arrotondamento */
        dist = fmaf((point[i]-center[i]), (point[i]-center[i]), dist);
    }
    dist = sqrt(dist);
    return(dist);
}
```

Figura 1: **EuclidianDistance**

```
void writeCompTimeToFile(char *filename, float value)
{
    // Write value in the last line of the file
    FILE *fp = fopen(filename, "a");
    if (fp == NULL) {
        fprintf(stderr, "Error opening file -%s\n", filename);
        return;
    }
    fprintf(fp, "%f\n", value);
    fclose(fp);
}
```

Figura 2: **Write-Comp**

utilizzata la funzione `fmaf` per evitare possibili errori di arrotondamento; questa modifica è stata apportata in tutte le versioni del codice, questo per fare in modo che la versione di CUDA non desse output differenti dal sequenziale, in quanto utilizzando la GPU per i calcoli gli arrotondamenti vengono eseguiti in modo diverso rispetto alla CPU. Con l'utilizzo di `fmaf`, si riduce il numero di arrotondamenti, diminuendo la possibilità di errori.

---

Per quanto riguarda il resto del codice, per parallelizzare il codice sequenziale di KMEANS è stato diviso il ciclo `do - while` in tre parti fondamentali, ovvero:

1. Il reset delle variabili utilizzate ad ogni iterazione [3]
2. Il calcolo dei nuovi centroidi e l'assegnazione dei punti ai cluster [4]
3. Il calcolo della distanza massima tra i centroidi vecchie e quelli nuovi, per il controllo della threshold impostata dai parametri in input [5]

Figura 4: **Second section**

```
for(i=0; i<lines; i++) {
    class=1;
    minDist=FLT_MAX;
    for(j=0; j<K; j++) {
        dist=euclideanDistance(&data[i*samples],
                                &centroids[j*samples],
                                samples);
        if(dist < minDist) {
            minDist=dist;
            class=j+1;
        }
    }
    if(classMap[i]!=class)
        changes++;
    classMap[i]=class;
}
```

Figura 3: **First section**

```
do {
    it++;
    changes = 0;
    ...
    zeroIntArray(pointsPerClass,K);
    zeroFloatMatriz(auxCentroids,K,samples);
    ...
    maxDist = FLT_MIN;
    ...
    while(condizioni);
```

Figura 5: **Third section**

```
for(i=0; i<K; i++){
    distCentroids[i]=euclideanDistance(&centroids[i*samples],
                                        &auxCentroids[i*samples],
                                        samples);

    if(distCentroids[i]>maxDist)
        maxDist=distCentroids[i];
}
```

La parallelizzazione della versione sequenziale, è stata effettuata cercando di parallelizzare le 3 parti appena descritte. Oltre a queste parti, è importante anche capire la gestione dei dati all'interno di ogni iterazione, le variabili che sono state tenute con maggior considerazione durante la parallelizzazione sono: `pointPerClass`, `auxCentroids`, `classMap`, `centroids`, `maxDist`, `classMap`.

---

Per quanto riguarda `pointPerClass` e `auxCentroids` sono utilizzate per il calcolo dei nuovi centroidi, che è una sezione del codice che non è stata tratta precedentemente, in quanto è stata poi accorpata con la seconda e la terza sezione. In quanto la prima parte del calcolo viene effettuata su un ciclo che itera sul numero di linee (come la seconda sezione), mentre la seconda parte del calcolo viene effettuata su un ciclo che itera sul numero di cluster (come la terza sezione).

La variabile `classMap` riporta per ogni dato passato dal file di input il cluster a cui viene assegnato. Il contenuto di questa variabile viene poi trascritto con la funzione `writeResult(filename)` sul file passato in input (specificata negli argomenti utilizzando il seguente path:

`output_files/{versione}/output{grandezza_test}.txt`).

Invece, la variabile `centroids` rappresenta i centroidi attuali, i quali all'inizio vengono generati randomicamente, in seguito, dopo la prima iterazione del ciclo `do - while` viene sovrascritto da `auxCentroids` controllando che la distanza massima di cambiamento, ovvero `maxDist`, sia minore della massima distanza specificata negli argomenti.

## 1.1 Check validity and Testing script

Per quanto riguarda l'esecuzione dei test e il controllo della validità di un'esecuzione sono stati creati due script python (`tester.py` e `run.py`) all'interno della cartella `py_prog/`.

————— `run.py` —————

Il file `run.py` viene utilizzato per l'esecuzione di un singolo test, impostato il file `.sub` per il submit del job sul cluster in base alla versione che si vuole eseguire (MPI + OMP, CUDA o sequenziale). Una volta eseguito il test, per controllare la correttezza dell'output generato controllo il contenuto

del file con il file di output generato dalla versione sequenziale. Per quanto riguarda il setting del file job sono state create 3 funzioni, una per versione, in cui in ognuna modifica il file `.sub` della propria versione all'interno della directory `jobs/` impostando tutti i settaggi necessari per l'esecuzione del test (per esempio il file di input, dove salvare l'output e il computation time, nel caso di MPI + OMP il numero di thread e di processi).

————— `tester.py` —————

Per quanto riguarda l'esecuzione dei test per il calcolo dell'Average Time, è stato creato un'altro script python, ovvero, il file `tester.py` il quale utilizza la libreria `unittest` per l'esecuzione dei test in tutte e tre le versioni. I test vengono eseguiti attraverso la funzione `main` importata da `run.py`; ogni singolo test viene ripetuto 50 volte, per avere dei dati più precisi per quanto riguarda l'AvgTime. Dopo l'esecuzione dei test esegue il calcolo dell'AvgTime del test e viene salvato su un file `{versione}.csv` in cui ogni riga ha il seguente formato:

```
Number of Process, Number of Thread, AvgTime Test 2D2, ..., AvgTime Test 100D2
*Nel caso del sequenziale e di CUDA il numero di processi e thread è 0
```

## 2 CUDA Version

Per la versione CUDA sono state implementate due funzioni kernel, che rappresentano due delle sezioni spiegate nell'introduzione (Figure: 4 e 5), e 3 variabili `__constant__` che sono il numero di cluster, linee e samples. Per ogni chiamata ad una funzione cuda è stata usata `CHECK_CUDA_CALL` per verificare se la funzione passata in input ha generato un errore. Come prima modifica, è sono state aggiunte nella sezione di allocazione della memoria (nella funzione `main`) le funzioni `cudaMalloc` e `cudaMemcpy` per l'allocazione della memoria sulla GPU e il trasferimento dei dati dalla CPU alla GPU per ogni variabile utilizzata all'interno delle funzioni kernel, mentre, per le variabili `__constant__` è stata usata la funzione `cudaMemcpyToSymbol`.

In seguito all'allocazione della memoria e allo spostamento delle variabili, e prima del ciclo `do - while`, vengono impostati le dimensioni di ogni blocco e i thread per blocco in base al numero di linee (determinato dal file di input) e cluster (determinato dal parametro passato in input).

```
dim3 blockSize(1024);
dim3 numBlocks(ceil(static_cast<double>(lines) / blockSize.x));
dim3 numBlocks2(ceil(static_cast<double>(K) / blockSize.x));
```

Come si nota viene utilizzata la funzione `ceil()` per evitare che non vengano scartate le ultime iterazioni.

————— Kernel Functions —————

All'interno del ciclo `do - while` vengono inizialmente reimpostate (attraverso la funzione `cudaMemset`) le seguenti variabili: `d_changes`, `d_maxDist`, `d_pointPerClass`, `d_auxCentroids`. In seguito al reset delle variabili vengono chiamati i due kernel. Il primo definito nel seguente modo:

1. Definizione della funzione kernel:

```
--global-- void assign_centroids(float* d_data, float* d_centroids,
                                int* d_classMap, int* d_changes, int* d_pointsPerClass,
                                float* d_auxCentroids)
```

La funzione viene utilizzata per assegnare ogni punto del file in input al cluster più vicino, e calcolare il numero di punti per ogni cluster e aumenta il numero di cambiamenti per iterazione (ricordiamo che il numero di cambiamenti viene resettato ad ogni iterazione del ciclo) se un punto cambia di cluster. In seguito inizia il calcolo dei nuovi centroidi, che conclude con il secondo kernel.

2. Inizialmente la funzione calcola l'indice del thread e controlla se il thread è minore del numero di linee:

```
int id = (blockIdx.x * blockDim.x) + threadIdx.x;
if (id < d_lines)
```

3. All'interno dell'`if` viene eseguita la seconda sezione [4], lavorando con le variabili che si trovano nella GPU, spostati precedentemente dalla CPU nella sezione dell'allocazione della memoria.

Questa implementazione del kernel è stata fatta per parallelizzare il calcolo dei nuovi centroidi e l'assegnazione dei punti ai cluster più vicini.