UNIVERSITÀ
DEGLI STUDI
FIRENZE

K-Means

Alessandro Mini

UNIVERSITÀ DEGLI STUDI FIRENZE

Indice

UNIVERSITÀ DEGLI STUDI FIRENZE

Introduzione

K-Means: Aloritmo di clustering che individua *k clusters* all'interno di un set di oggetti.

- Si deve poter definire una *distanza* tra gli elementi del set.
- *k* deve essere determinato a priori (alcune versioni dell'algoritmo utilizzano procedure di best fit per trovarlo).
- Ogni cluster *C* possiede un punto detto *centroide* che ne rappresenta il centro.
- K-Means è un algoritmo di *hard-clustering* nel senso che:
  1. Ogni punto appartiene ad uno ed un solo cluster.
  2. Non esistono punti non assegnati.

UNIVERSITÀ DEGLI STUDI FIRENZE

Introduzione

Descrizione dell'algoritmo:

1. Si definiscono k, il set in cui si andrà a lavorare ed un numero i di iterazioni.

Per ogni $n = 1 \ldots i$:

1. Ogni punto viene assegnato al cluster più vicino.
2. Aggiorno la posizione del centroide affinchè corrisponda sempre al centro *geometrico* del cluster.

Introduzione

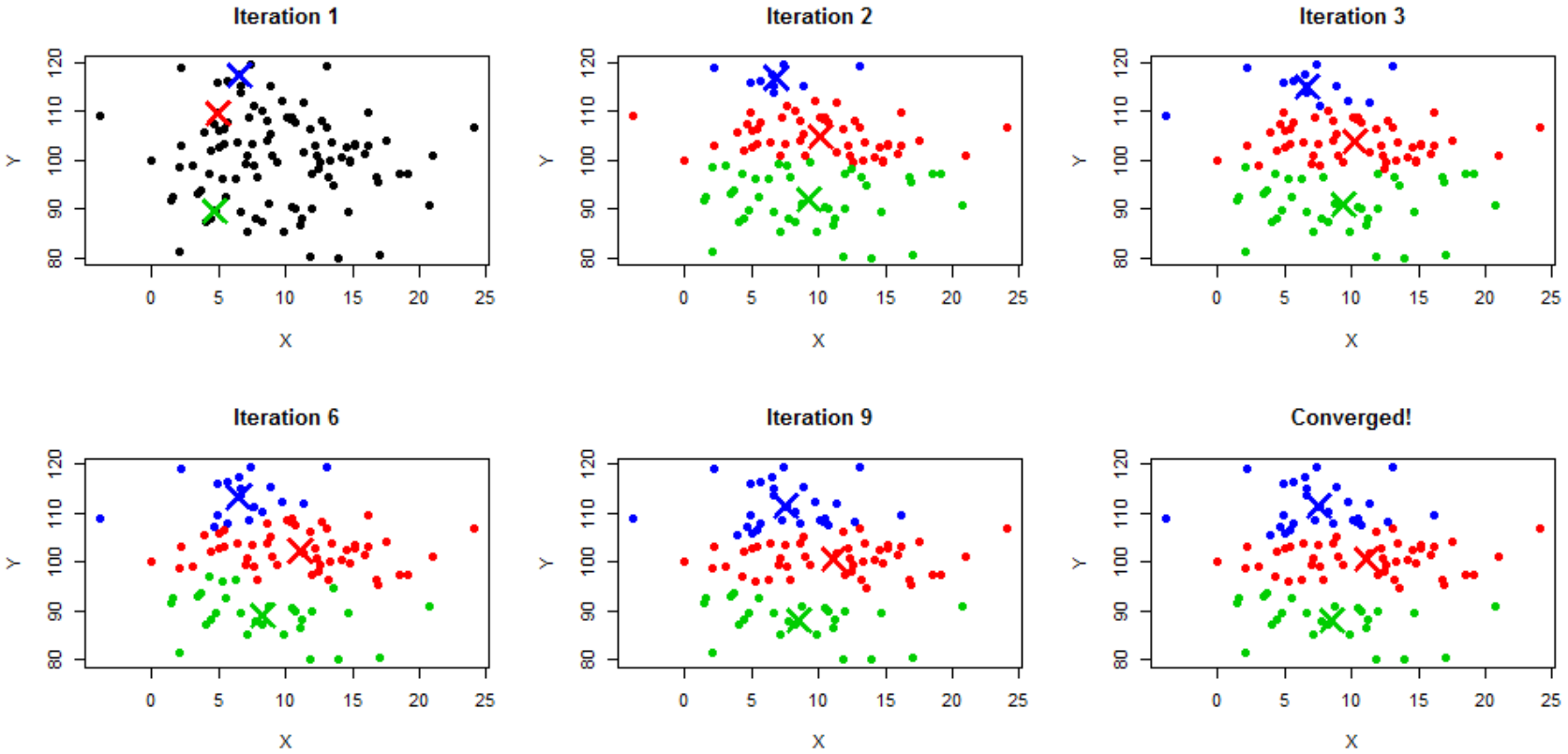

Figure 1: Esecuzione di K-Means.

UNIVERSITÀ DEGLI STUDI FIRENZE

Descrizione del funzionamento

K-Means verrà implementato come segue:

- Una versione OpenMP in cui si può decidere il numero di cores da utilizzare (es. se eseguirlo in modalità sequenziale o parallela).
- Una versione CUDA in cui si può decidere il numero di *threads* da utilizzare ed i TPB (Threads per block).

UNIVERSITÀ DEGLI STUDI FIRENZE

Implementazione OpenMP

I punti e cluster vengono implementati come *matrici* di tipo float, ogni riga della matrice corrisponde ad un elemento.

- Punti: matrice *NUM_POINTx*3 in cui ogni punto è una terna (x, y, c) dove
  - x, y sono le coordinate del punto.
  - c è il cluster di appartenenza.
- Clusters: matrice *NUM_CLUSTERSx*4 in cui ogni cluster è una quadrupla (p, sx, sy, n) in cui:
  - p è il punto centroide, l'indice corrispondente nella matrice dei punti.
  - sx, sy sono valori incrementali che rappresentano la *size*.
  - n è il numero dei punti "dentro" al cluster.

Verranno quindi riportate alcune porzioni di codice rilevanti.

Configurazioni del programma:

```
#define COORD_MAX 100000
#define CLUSTER_NUM 100
#define POINT_NUM 100000
#define NUM_THREAD 12
#define IT_MAX 20
#define EPSILON 0.001
#define POINT_FEATURES 3
#define CLUSTER_FEATURES 4
```

Main:

```
for(int i = 0; i< IT_MAX;i++){
#pragma omp parallel for default(none) shared(punti, clusters)
↪  num_threads(NUM_THREAD)
        for (int j = 0; j < POINT_NUM; j++) {
            assignPoints(j, punti, clusters);
        }
        clusters_recomputeCenter(punti, clusters);
        remove_points_from_clusters(clusters);
        printf("|", i);
    }
```

Funzione che calcola la distanza tra i punti:

```
float distance(float x1, float x2, float y1, float y2) {
    return sqrt(((x1 - x2) * (x1 - x2)) + ((y1 - y2) * (y1 - y2)));
}
```

Inizializzazione **casuale** di punti e clusters:

```
for(int i = 0;i<POINT_NUM;i++){
    punti[i][0] = random_float();
    punti[i][1] = random_float();
    punti[i][2] = 0;
}
for (int j = 0; j < CLUSTER_NUM; j++) {
    clusters[j][0] = rand() % (POINT_NUM - 0) + 0;
    clusters[j][1] = 0;
    clusters[j][2] = 0;
    clusters[j][3] = 0;
}
```

Assegnamento dei punti ai clusters.

```
void assignPoints(int punto, float punti[POINT_NUM][POINT_FEATURES], float
↪  clusters[CLUSTER_NUM][CLUSTER_FEATURES]) {
    float x_punto, x_cluster, y_punto, y_cluster = 0;
    x_punto = punti[punto][0];
    y_punto = punti[punto][1];
    int best_fit = 0;
    float distMax = FLT_MAX;
    for (int i = 0; i < CLUSTER_NUM; i++) {
        int cluster_index_point = (int)clusters[i][0];
        x_cluster = punti[cluster_index_point][0];
        y_cluster = punti[cluster_index_point][1];
        if (distance(x_punto, x_cluster, y_punto, y_cluster) < distMax) {
            best_fit = i;
            distMax = distance(x_punto, x_cluster, y_punto, y_cluster);
        }
    }
    punti[punto][2] = (float)best_fit;
#pragma omp atomic
    clusters[best_fit][1] = clusters[best_fit][1] + x_punto;
#pragma omp atomic
    clusters[best_fit][2] = clusters[best_fit][2] + y_punto;
#pragma omp atomic
    clusters[best_fit][3] = clusters[best_fit][3] + 1;
};
```

Ricalcolo del centroide dei clusters.

```
void clusters_recomputeCenter(float points[POINT_NUM][POINT_FEATURES],float
↪  clusters[CLUSTER_NUM][CLUSTER_FEATURES]) {
    #pragma omp parallel for default(none) shared(clusters,points) num_threads(NUM_THREAD)
    for (int i = 0; i < CLUSTER_NUM; i++) {
        float newX = clusters[i][1] / clusters[i][3];
        float newY = clusters[i][2] / clusters[i][3];
        int cluster_center_index = (int) clusters[i][0];
        float x = points[cluster_center_index][0];
        float y = points[cluster_center_index][1];
        if(!(abs(x - newX) < EPSILON && abs(y - newY) < EPSILON)) {
            points[cluster_center_index][0] = newX;
            points[cluster_center_index][1] = newY;
        }
    }
}
```

Rimozione dei punti dai clusters:

```
void remove_points_from_clusters(float clusters[CLUSTER_NUM][CLUSTER_FEATURES]) {
#pragma omp parallel for default(none) shared(clusters) num_threads(NUM_THREAD)
    for (int i = 0; i < CLUSTER_NUM; i++) {
        clusters[i][1] = 0;
        clusters[i][2] = 0;
        clusters[i][3] = 0;
    }
}
```

Implementazione CUDA

Si riprende l'implementazione della versione OpenMP ma si *linearizzano* le matrici dei punti e cluster secondo il seguente approccio (Row-Major):

- Se M è una matrice con m righe e k colonne:

$$M[i][j] = m[i \cdot k + j]$$

Metodo Main:

```
for (int i = 0; i < IT_MAX; i++) {
printf("|", i);
//CUDA call to assign points:
assign_clusters << < (POINT_NUM + THREAD_PER_BLOCK - 1) / THREAD_PER_BLOCK, THREAD_PER_BLOCK >>
↪  > (punti_d, cluster_d);
gpuErrchk(cudaPeekAtLastError());
gpuErrchk(cudaDeviceSynchronize());
//CUDA call to recompute centers:
cluster_recomputecenters_cuda << <1, CLUSTER_NUM >> > (punti_d, cluster_d);
gpuErrchk(cudaPeekAtLastError());
gpuErrchk(cudaDeviceSynchronize());
//CUDA call to set each cluster to 0:
cuda_remove_points_cluster << <1, CLUSTER_NUM >> > (cluster_d);
gpuErrchk(cudaPeekAtLastError());
gpuErrchk(cudaDeviceSynchronize());
}
```

A differenza della versione OpenMP ogni punto viene mappato in un *thread* e cercherà il cluster ”migliore” a cui assegnarsi.

```
__global__ void assign_clusters(float* punti, float* clusters) {
        long id_punto = threadIdx.x + blockIdx.x * blockDim.x;
        if (id_punto < POINT_NUM)
        ↪  {
                float x_punto, x_cluster, y_punto, y_cluster = 0;
                x_punto = punti[id_punto * POINT_FEATURES + 0];
                y_punto = punti[id_punto * POINT_FEATURES + 1];
                long best_fit = 0;
                long distMax = LONG_MAX;
                for (int i = 0; i < CLUSTER_NUM; i++) {
                        int cluster_index_point = clusters[i * CLUSTER_FEATURES + 0];
                        x_cluster = punti[cluster_index_point * POINT_FEATURES + 0];
                        y_cluster = punti[cluster_index_point * POINT_FEATURES + 1];
                        if (distance(x_punto, x_cluster, y_punto, y_cluster) < distMax) {
                                best_fit = i;
                                distMax = distance(x_punto, x_cluster, y_punto, y_cluster);
                        }
                }
                punti[id_punto * POINT_FEATURES + 2] = best_fit;
                atomicAdd(&clusters[best_fit * CLUSTER_FEATURES + 1], x_punto);
                atomicAdd(&clusters[best_fit * CLUSTER_FEATURES + 2], y_punto);
                atomicAdd(&clusters[best_fit * CLUSTER_FEATURES + 3],
                ↪  1);
        }
}
```

Ricalcolo del centroide dei clusters.

```
__global__ void cluster_recomputecenters_cuda(float* points, float* clusters)
↪  {
        long id_cluster = threadIdx.x + blockIdx.x * blockDim.x;
        float sizeX = clusters[id_cluster * CLUSTER_FEATURES + 1];
        float sizeY = clusters[id_cluster * CLUSTER_FEATURES + 2];
        float nPoints = clusters[id_cluster * CLUSTER_FEATURES + 3];
        float newX = sizeX / nPoints;
        float newY = sizeY / nPoints;
        long cluster_center_index = (long)clusters[id_cluster * CLUSTER_FEATURES + 0];
        float x = points[cluster_center_index * POINT_FEATURES + 0];
        float y = points[cluster_center_index * POINT_FEATURES + 1];
        if (!(fabsf(x - newX) < EPSILON && fabsf(y - newY) < EPSILON)) {
                points[cluster_center_index * POINT_FEATURES + 0] = newX;
                points[cluster_center_index * POINT_FEATURES + 1] = newY;
        }
}
```

Eliminazione dei punti dai clusters:

```
__global__ void cuda_remove_points_cluster(float* clusters) {

        long id_cluster = threadIdx.x + blockIdx.x * blockDim.x;
        clusters[id_cluster * CLUSTER_FEATURES + 1] = 0;
        clusters[id_cluster * CLUSTER_FEATURES + 2] = 0;
        clusters[id_cluster * CLUSTER_FEATURES + 3] = 0;
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Dettaglio su trasferimento degli oggetti tra le memorie *host* e *device*.

```
//Trasferimento host → device.
cudaMalloc(&punti_d, POINT_NUM * POINT_FEATURES * sizeof(float));
cudaMalloc(&cluster_d, CLUSTER_NUM * CLUSTER_FEATURES * sizeof(float));
cudaMemcpy(punti_d, punti, POINT_NUM * POINT_FEATURES * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(cluster_d, clusters, CLUSTER_NUM * CLUSTER_FEATURES * sizeof(float),
↪  cudaMemcpyHostToDevice);
.... //lavoro
//Trasferimento device → host.
cudaMemcpy(punti, punti_d, POINT_NUM * POINT_FEATURES * sizeof(float), cudaMemcpyDeviceToHost);
cudaMemcpy(clusters, cluster_d, CLUSTER_NUM * CLUSTER_FEATURES * sizeof(float),
↪  cudaMemcpyDeviceToHost);
//Free della memoria occupata.
cudaFree(punti_d);
cudaFree(cluster_d);
```

Premesse

Si ricorda che:

- Punti e clusters vengono generati *randomicamente*.
- Sarà utilizzato uno spazio bidimenzionale di dimensione $1 \cdot 10^5$.
- Il risultato del clustering viene visualizzato stampando i punti su file nella forma $x : y : c$ e si invia come comando a gnuplot.
- architettura di test:
  - CPU: Ryzen 5 3600.
  - GPU: GTX 950 2GB.
  - RAM: 16GB DDR4.
- I tempi saranno indicati in millisecondi.

Tempi versione sequenziale:

Num. Punti	Num. Clusters	Tempo sequenziale (ms)
100000	20	0,707
100000	30	1,021
1000000	20	7,223
1000000	30	10,288
10000000	20	72,748
10000000	30	98,251
10000000	100	289,484

Tempi versione OpenMP:

Punti	Clusters	Tempo OpenMP (ms)	Speedup
100000	20	0,231	3,06
100000	30	0,242	4,22
1000000	20	1,827	3,95
1000000	30	1,97	5,22
10000000	20	17,6	4,13
10000000	30	19,45	5,05
10000000	100	43,917	6,59

Tempi versione CUDA:

Num. Punti	Clusters	Tempo CUDA (ms)	Speedup
100000	20	0,064	11,05
100000	30	0,066	15,47
1000000	20	0,4	18,06
1000000	30	0,483	21,30
10000000	20	3,32	21,91
10000000	30	4,77	20,60
10000000	100	13,7	21,13

Comparazione di tutti i tempi:

N.	Clusters	Seq. (ms)	OMP (ms)	CUDA (ms)
$1 \cdot 10^5$	20	0,707	0,231	0,064
$1 \cdot 10^5$	30	1,021	0,242	0,066
$1 \cdot 10^6$	20	7,223	1,827	0,4
$1 \cdot 10^6$	30	10,288	1,97	0,483
$1 \cdot 10^7$	20	72,748	17,6	3,32
$1 \cdot 10^7$	30	98,251	19,45	4,77
$1 \cdot 10^7$	100	289,484	43,917	13,7

Si eseguiranno quindi 2 plots nel caso $1 \cdot 10^7$ punti con 100 clusters.

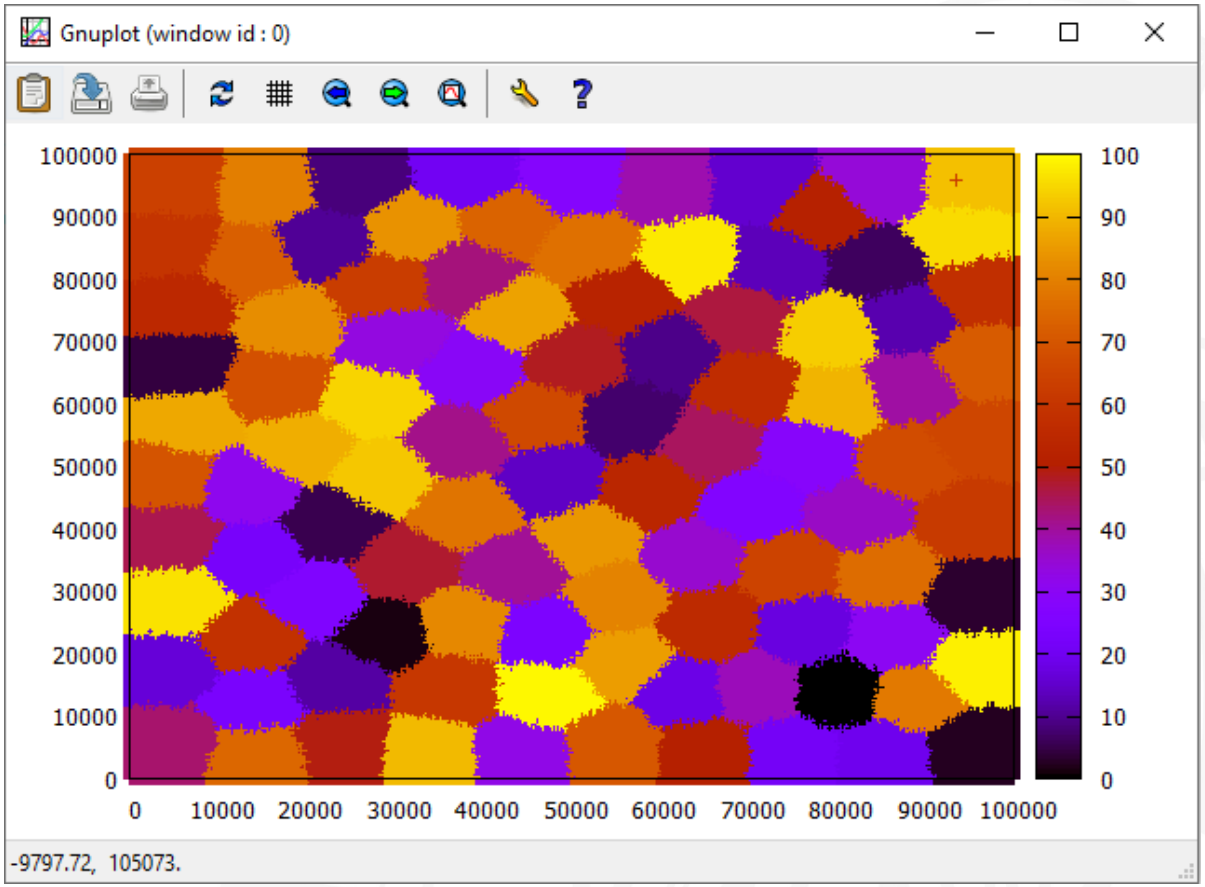

Figure 2: Plot 100 clusters OpenMP.

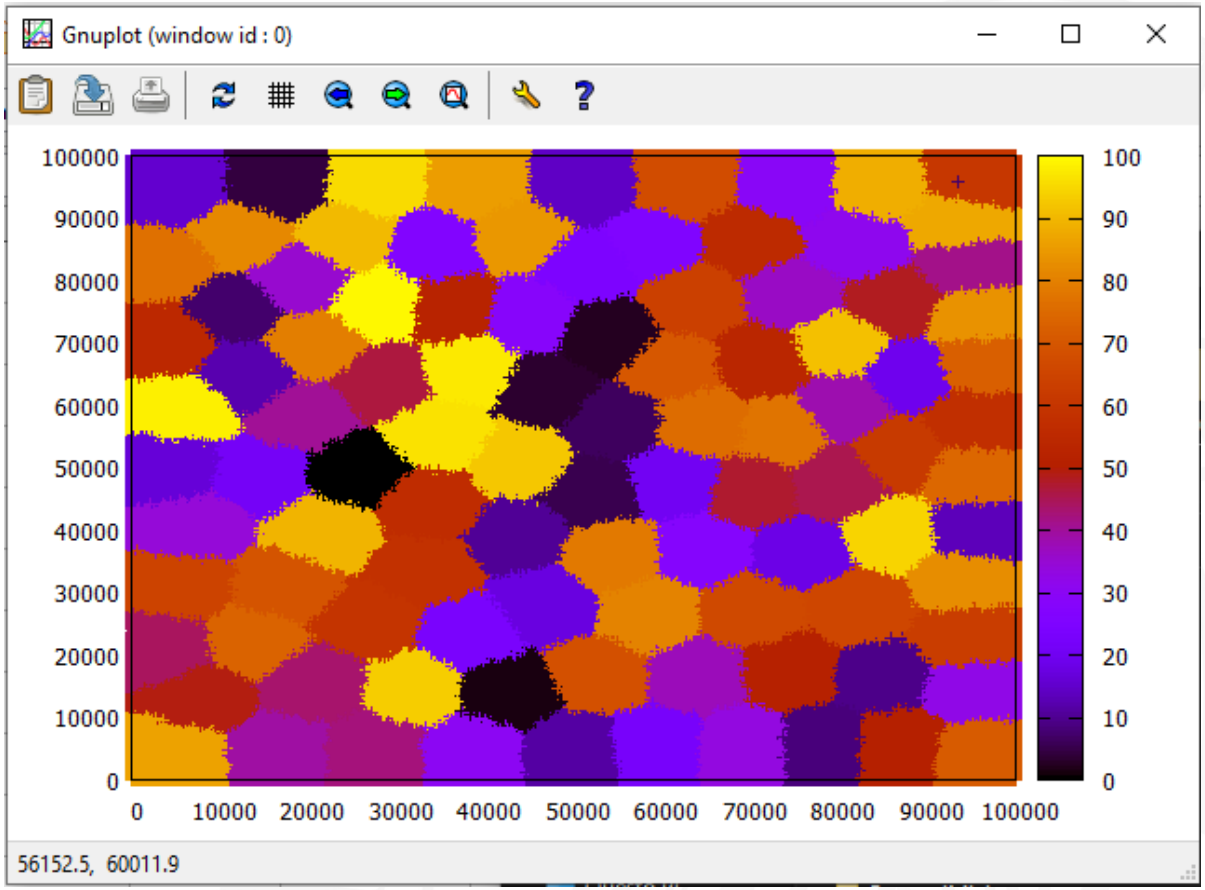

Figure 3: Plot 100 clusters CUDA.

UNIVERSITÀ DEGLI STUDI FIRENZE

Conclusioni

Si è implementato k-Means in OpenMP e CUDA, si può notare che CUDA per via dell'enorme parallelizzazione produce speedup molto maggiori, di circa 21 volte rispetto alla versione sequenziale e 3.30 volte rispetto ad OpenMP.