PC-2020/21 Elaborato Long-Term

Andrea Neri
andrea.neri1@stud.unifi.it

Abstract

Il progetto "Long-Term" prevede l'implementazione dell'algoritmo di clustering K-Means in una versione sequenziale e due versioni parallele. In particolare è stato utilizzato il framework OpenMP per la realizzazione della versione sequenziale e di una versione parallela, per la seconda versione parallela è stato utilizzato CUDA.

1. Introduzione: l'algoritmo K-Means

K-Means è un algoritmo di apprendimento non supervisionato che, dato in input un insieme di dati e un numero K di gruppi, si occupa di dividere i data points a seconda della presenza o meno di una certa somiglianza tra di loro. La somiglianza tra due punti può essere definita, in uno spazio a due dimensioni, dalla distanza euclidea. Ogni cluster avrà un punto particolare detto centroide che rappresenta il centro geometrico dell'insieme.

L'algoritmo può essere rappresentato dalle seguenti fasi:

1) **Inizializzazione:** si definisce il parametro K e il dataset su cui l'algoritmo andrà a lavorare;
2) **Assegnazione dei punti ad un cluster:** ogni punto del dataset viene assegnato al centroide più vicino;
3) **Ricalcolo della posizione del centroide:** si ricalcola la posizione del centroide come la media di tutti i data points appartenenti a quel cluster.

Vediamo dettaglio i tre passi:

Passo 1: Inizializzazione

I K centroidi iniziali vengono disposti casualmente. Scegliendo il numero di centroidi, si scelgono i cluster cui il data set sarà composto e quindi i raggruppamenti che si vogliono effettuare e visualizzare.

Passo 2: Assegnazione del cluster

In questa fase, l'algoritmo analizza ciascuno dei data points e li assegna al centroide più vicino, calcolando la distanza tra ogni data points e ogni centroide. Ogni data points sarà poi assegnato al centroide la cui distanza risulti minima.

In termini matematici:

$$\underset{c_i \in C}{argmin}\ dist(c_i, x)^2$$

dove c_i è un centroide nell'insieme C, x sono i data points e $dist()$ è la distanza.

Passo 3: Aggiornamento della posizione del centroide

Il nuovo valore di un centroide sarà la media di tutti i data points che sono stati assegnati al nuovo cluster.

In termini matematici:

$$c_i = \frac{1}{|s_i|}\sum_{x_i \in s_i} x_i$$

Dove s_i rappresenta la somma dei data points assegnati al cluster i-esimo. Si ottiene la nuova posizione del centroide dalla media di tutti i data points assegnati al cluster nello step precedente.

L'algoritmo continuerà a ripetere i passaggi 2 e 3 finché non si raggiunge una condizione di stop.

Di solito una condizione di stop è rappresentata da una delle seguenti opzioni:

- nessun data points cambia cluster;
- la somma delle distanze è ridotta al minimo;
- viene raggiunto un numero massimo di iterazioni (modalità utilizzata in questo progetto).

Al termine dell'esecuzione dell'algoritmo tutti i punti saranno assegnati ad un solo cluster e non ci saranno punti non assegnati.

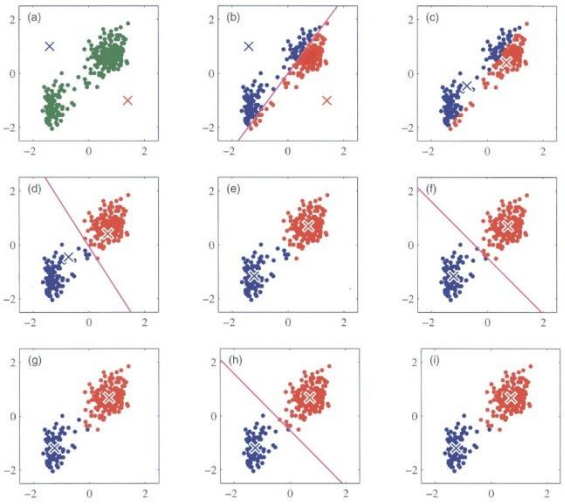

Figura 1: esecuzione dell'algoritmo con K=2

2. Parametri di esecuzione

Per facilitare l'esecuzione del programma e dei test sono state definite delle macro in modo da modificare i parametri di esecuzione in un solo punto del codice.

```
#define RANGE_COORDINATE_MAX 100000
#define CLUSTER_NUMBER 100
#define POINT_NUMBER 5000000
#define CLUSTER_ATTRIBUTES 4
#define POINT_ATTRIBUTES 3
#define THREAD_NUMBER 1
#define ITERATION 30
#define DISTANCE 0
```

I parametri, nell'ordine, rappresentano: l'estremo superiore nell'intervallo in cui vengono generati in modo randomico i punti, il numero di cluster e di punti da generare, gli attributi dei punti e dei cluster (spiegati nella prossima sezione), il numero di thread con cui eseguire l'algoritmo, il numero di iterazioni che compirà l'algoritmo, e la macro per la scelta di quale formula utilizzare per il calcolo della distanza (sezione 4).

Nell'implementazione CUDA la macro THREAD_NUMBER è stata sostituita con THREAD_FOR_BLOCK, che rappresenta il numero di thread per ogni blocco.

3. Rappresentazione di punti e cluster

Utilizzando uno spazio a 2 dimensioni un punto può essere rappresentato da 3 elementi:

- c: il cluster a cui verrà assegnato
- x: la coordinata nello spazio relativa all'asse x
- y: la coordinata nello spazio relativa all'asse y

I punti vengono memorizzati all'interno di un array di float di dimensione `POINT_NUMBER * POINT_ATTRIBUTES`.

L'accesso alle singole componenti viene fatto seguendo le seguenti formule:

- `points[0 * POINT_NUMBER + j]`
- `points[1 * POINT_NUMBER + j]`
- `points[2 * POINT_NUMBER + j]`

Un cluster in uno spazio a 2 dimensioni può essere rappresentato utilizzando 4 elementi:

- p: definisce il punto che rappresenta il centroide
- n: il numero di punti assegnato al cluster
- vx: valore che rappresenta la somma delle componenti di ogni singolo punto del cluster lungo l'asse x (valore utilizzato per il ricalcolo del centroide)
- vy: valore che rappresenta la somma delle componenti di ogni singolo punto del cluster lungo l'asse y (valore utilizzato per il ricalcolo del centroide)

I cluster vengono memorizzati all'interno di un array di float di dimensione `CLUSTER_NUMBER * CLUSTER_ATTRIBUTES`.

L'accesso alle singole componenti viene fatto seguendo le seguenti formule:

- `clusters[0 * CLUSTER_NUMBER +j]`
- `clusters[1 * CLUSTER_NUMBER +j]`
- `clusters[2 * CLUSTER_NUMBER +j]`
- `clusters[3 * CLUSTER_NUMBER +j]`

Questa metodologia di memorizzazione è stata utilizzata sia per l'implementazione in OpenMP sia quella in CUDA.

4. Calcolo della distanza

La distanza tra due punti, in particolare tra un punto e un centroide, è possibile calcolarla con due metodi: distanza euclidea e distanza di Manhattan.

4.1. **Distanza euclidea**

Per due punti in uno spazio bidimensionale $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$ la distanza euclidea è calcolata come:

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

E viene implementata con la seguente funzione:

```
float euclideanDistance(float x1, float y1,
float x2, float y2)
{
    return sqrt(pow((x1-x2),2)
                +pow((y1-y2),2));
}
```

4.2. Distanza di Manhattan

Per due punti in uno spazio bidimensionale $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$ la distanza di Manhattan è calcolata come:

$$d(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2|$$

E viene implementata con la seguente funzione:

```
float manhattanDistance(float x1, float y1,
float x2, float y2)
{
      return abs(x1-x2)+abs(y1-y2);
}
```

5. Implementazione con OpenMP

In questa sezione vengono riportati tutti i metodi implementati con il framework OpenMP. Vengono commentati i punti più importanti di ogni metodo e vengono riportati i risultati ottenuti nella versione parallela e in quella sequenziale.

La versione parallela e quella sequenziale condividono lo stesso codice, il parametro che permette la scelta dell'esecuzione di una versione o dell'altra è la macro THREAD_NUMBER. Infatti, grazie a questo parametro, saranno scelti il numero di thread con cui eseguire l'algoritmo.

5.1. Assegnamento dei punti ai cluster

Il metodo principale dell'algoritmo è quello che si occupa di assegnare ogni singolo punto al cluster più vicino.

Sfruttando le direttive disponibili all'interno del framework OpenMP il ciclo che itera su tutti i punti del dataset è stato parallelizzato. Vengono creati, in modo statico, un numero di sottoinsiemi di indici indipendenti pari al numero di thread richiesti. Ogni thread condividerà con gli altri l'array dei punti e quello dei cluster.

Il metodo, per ogni punto, recupera le coordinate sull'asse x e sull'asse y e cerca il cluster il cui centroide ha la distanza minore tra il punto in analisi e il centroide stesso.

Una volta trovato il centroide migliore, si crea la relazione fra il punto e il cluster e si aumenta il contatore del numero dei punti all'interno del cluster. Quest'ultima operazione viene protetta dalla direttiva *#pragma omp atomic*, in modo da proteggere l'operazione di aggiornamento da possibili race condition.

Per questo di leggibilità il codice viene riportato in appendice

5.2. Calcolo della somma dei valori sugli assi

In questo metodo per ogni cluster si va a calcolare la somma delle componenti sull'asse x e sull'asse y di ogni punto all'interno del cluster.

```
void calculateValue(float* points, float* clusters){

        #pragma omp parallel for default(none)
        shared(clusters,points)
        num_threads(THREAD_NUMBER) schedule(static)

        for (int i = 0; i < POINT_NUMBER; i++) {
                unsigned int cluster_n=
                points[0 * POINT_NUMBER + i];

                #pragma omp atomic

                clusters[2*CLUSTER_NUMBER+cluster_n=
                clusters[2*CLUSTER_NUMBER+cluster_n]
                +points[1*POINT_NUMBER+i];

                #pragma omp atomic

                clusters[3*CLUSTER_NUMBER+cluster_n=
                clusters[3*CLUSTER_NUMBER+cluster_n]
                +points[2*POINT_NUMBER+i];
        }
}
```

Anche in questo caso sono state utilizzate le direttive per parallelizzare il ciclo e per evitare race condition sull'array dei cluster

5.3. Ricalcolo della posizione del centroide

Questa funzione ha il compito di ricalcolare la posizione del centroide all'interno di ogni cluster secondo le formule:

- $x = \frac{v_x}{n}$
- $y = \frac{v_x}{n}$

E di aggiornare la nuova posizione all'interno dell'array dei punti.

```
void recomputeCentroid(float* points, float* clusters) {

#pragma omp parallel for default(none)
shared(clusters,points)
num_threads(THREAD_NUMBER)
schedule(dynamic)

    for (int i = 0; i < CLUSTER_NUMBER; i++) {
                float centroid_x =
                clusters[2*CLUSTER_NUMBER+i]/
                clusters[1*CLUSTER_NUMBER+i];

                float centroid_y =
                clusters[3*CLUSTER_NUMBER+i]/
                clusters[1*CLUSTER_NUMBER+i];

                unsigned int centroid_index=(int)
                clusters[0*CLUSTER_NUMBER+i];

                points[1*POINT_NUMBER+centroid_index]=
                centroid_x;

                points[2*POINT_NUMBER+centroid_index]=
                centroid_y;

        }
}
```

In questo caso la schedulazione dinamica permette di avere una riduzione dei tempi di esecuzione di circa 100 ms ~ 150 ms.

5.4. Rimozione dei punti dai cluster

Grazie al metodo riportato sotto è possibile eliminare i punti dai cluster, riportando ai valori iniziali n, vx e vy.

```
void removePoint(float* clusters) {
#pragma omp parallel for default(none)
shared(clusters) num_threads(THREAD_NUMBER)
    for (int i = 0; i < CLUSTER_NUMBER; i++) {
        clusters[1*CLUSTER_NUMBER+i] = 0;
        clusters[2*CLUSTER_NUMBER+i] = 0;
        clusters[3*CLUSTER_NUMBER+i] = 0;
    }
}
```

5.5. Funzione Main

All'interno del main vengono creati gli array dei punti e dei cluster e vengono inizializzati in modo random.

Successivamente vengono eseguite in successione le funzioni descritte precedentemente e si calcolano le tempistiche di esecuzione.

```
float* points=(float*)malloc(POINT_NUMBER *
POINT_ATTRIBUTES * sizeof(float));

float* clusters = (float*)malloc(CLUSTER_NUMBER
* CLUSTER_ATTRIBUTES * sizeof(float));

generatePointCluster(points, clusters);

auto start_time = high_resolution_clock::now();
for(int i = 0; i < ITERATION; i++){
    assignPointToCluster(points,clusters);
    calculateValue(points, clusters);
    recomputeCentroid(points, clusters);
    removePoint(clusters);
}
auto end_time = high_resolution_clock::now();
```

5.6. Test e risultati

Macchina di test:

- CPU -> AMD Ryzen 2600 (6 core / 12 thread)
- RAM -> 16 GB DDR4 3200 MHz

L'algoritmo viene eseguito per 30 iterazioni. I risultati ottenuti nella versione sequenziale e nella versione parallela sono visibili in appendice.

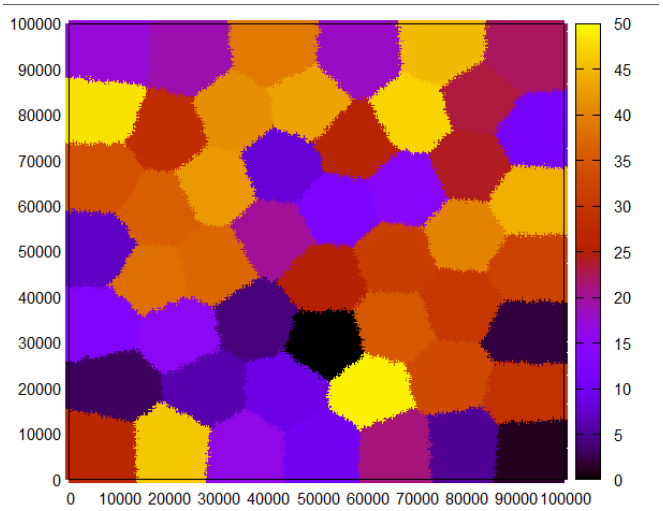

Figura 2: grafico ottenuto con OpenMP, con 5 milioni di punti e 50 cluster

6. Implementazione con CUDA

In questa sezione vengono riportati tutti i metodi implementati in CUDA. Vengono commentati i punti più importanti di ogni metodo e vengono riportati i risultati.

La versione sequenziale in CUDA, ovviamente non è presente e i risultati di questa versione verranno paragonati con la versione sequenziale precedente.

6.1. Assegnamento dei punti ai cluster

L'implementazione di questo metodo è del tutto simile all'implementazione con OpenMP. Una differenza sta nel fatto che non è più presente il ciclo for iniziale, infatti data l'alta presenza di thread in ambiente CUDA, ogni punto verrà assegnato ad un thread che si occuperà di assegnarlo al miglior cluster e di creare l'associazione tra punto e cluster. Anche in questo caso c'è la presenza della funzione atomicAdd per evitare condizioni di Race condition.

Un' ulteriore differenza è la presenza di un if per evitare che i thread non necessari, possano andare ad accedere ad aree di memoria non valide.

Il codice di questa sezione è riportato in appendice.

6.2. Calcolo della somma dei valori sugli assi

Anche in questo caso non si utilizza un ciclo for, ma si fa uso di thread per calcolare la somma delle componenti sull'asse x e sull'asse y di ogni punto all'interno del cluster. Anche in questo caso la presenza di un if è utile per evitare che i thread non necessari, possano andare ad accedere ad aree di memoria non valide.

Per evitare race condition sull'array dei cluster si fa uso della funzione atomicAdd.

```
__global__ void calculateValue(float* points,
float* clusters) {
       unsigned int point_n = threadIdx.x+
       blockIdx.x*blockDim.x;
       if (point_n < POINT_NUMBER) {
         unsigned int cluster_n = points[0*
         POINT_NUMBER+point_n];

         atomicAdd(&clusters[2*CLUSTER_NUMBER+
         cluster_n], points[1*POINT_NUMBER+
         point_n]);

         atomicAdd(&clusters[3*CLUSTER_NUMBER
         +cluster_n], points[2*POINT_NUMBER+
         point_n]);
    }
}
```

6.3. Ricalcolo della posizione del centroide

Con l'utilizzo di CLUSTER_NUMBER thread si ricalcola la posizione del centroide con le stesse formule viste in precedenza.

```
__global__ void recomputeCentroid(float* points,
float* clusters) {
    unsigned int cluster_n=threadIdx.x+blockIdx.x*
    blockDim.x;

    float centroid_x=clusters[2*CLUSTER_NUMBER+
    cluster_n]/clusters[1*CLUSTER_NUMBER+cluster_n];

    float centroid_y=clusters[3*CLUSTER_NUMBER+
    cluster_n]/clusters[1*CLUSTER_NUMBER+cluster_n];

    unsigned int cluster_index=(unsigned
    int)clusters[0 *CLUSTER_NUMBER+cluster_n];

    points[1*POINT_NUMBER+cluster_index]=centroid_x;
    points[2*POINT_NUMBER+cluster_index]=centroid_y;

}
```

6.4. Rimozione dei punti dai cluster

Con l'utilizzo di CLUSTER_NUMBER thread si eliminano i punti dai cluster, riportando ai valori iniziali n, vx e vy.

```
__global__ void removePoint(float* clusters) {
     unsigned int cluster_n=threadIdx.x +
     blockIdx.x*blockDim.x;

     clusters[1*CLUSTER_NUMBER+ cluster_n]=0;

     clusters[2*CLUSTER_NUMBER+ cluster_n]=0;

     clusters[3*CLUSTER_NUMBER+ cluster_n]=0;
}
```

6.5. Funzione Main

La funzione main in CUDA inizializza 4 array: due array di punti e due array di cluster. Una coppia verrà utilizzata a livello di host e l'altra coppia verrà utilizzata a livello di device. Vengono generati in modo randomico punti e cluster e vengono copiati nella memoria del device.

Successivamente vengono eseguite in successione le funzioni descritte precedentemente e si calcolano le tempistiche di esecuzione. I dati elaborati sul device vengono riportati nella memoria dell'host per generare il grafico. Infine, viene liberata la memoria del device.

Le funzioni assignPointToCluster e calculateValue vengono lanciate con un numero di blocchi pari a `(POINT_NUMBER + THREAD_FOR_BLOCK - 1) / THREAD_FOR_BLOCK)` ognuno dei quali lavorerà con THREAD_FOR_BLOCK thread.

Le altre due funzioni vengono lanciate con 1 blocco composto da CLUSTER_NUMBER thread.

Il codice di questa sezione è riportato in appendice.

6.6. Test e risultati

Macchina di test:

- CPU -> AMD Ryzen 1400 (4 core / 8 thread)
- RAM -> 8 GB DDR4 3200 MHz
- Scheda video -> Nvidia GT 1030

L'algoritmo viene eseguito per 30 iterazioni e un numero di thread per blocco pari a 256.
I risultati sono visibili in appendice.

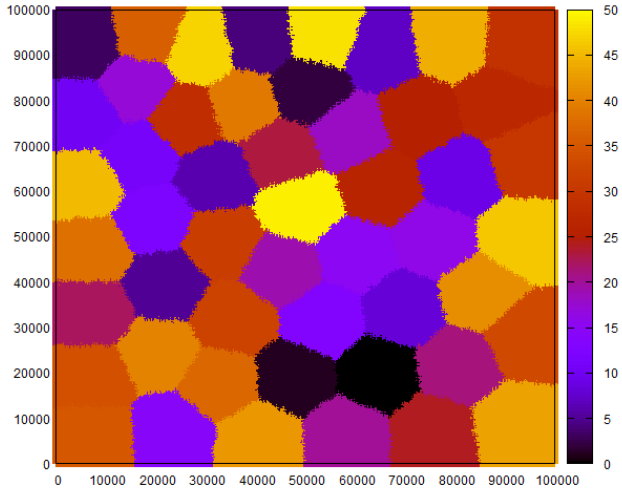

Figura 3:grafico ottenuto con CUDA, con 5 milioni di punti e 50 cluster

7. Considerazioni e conclusioni

Osservando i risultati è possibile notare come lo speedup aumenta all'aumentare della dimensione dell'insieme dei punti e del numero di cluster. In particolare, con l'utilizzo di OpenMP con un numero di cluster minore di 20 non c'è presenza di speedup. Sfruttando CUDA, c'è presenza di speedup anche con pochi punti e 10 cluster.

Confrontando i valori di speedup massimi ottenuti nelle due versioni parallele è possibile notare come in CUDA si ottiene un valore quasi 3 volte superiore rispetto all'implementazione con OpenMP.

8. Appendice

8.1. Risultati

Versione sequenziale con OpenMP

cluster number	10	15	20	30	50	100
point number						
5000	15,63	31,25	31,25	31,25	46,88	93,75
50000	125,01	156,26	188,57	266,70	437,54	860,48
500000	1315,88	1488,84	1870,74	2929,62	4308,55	8373,58
5000000	13440,10	15801,70	20016,80	28244,10	45006,20	85179,70
10000000	25895,60	31957,70	39570,60	56504,00	88307,00	172595,00

Versione parallela con OpenMP

cluster number	10	speedup	15	speedup	20	speedup	30	speedup	50	speedup	100	speedup
point number												
5000	31,25	0,50	31,25	1,00	31,25	1,00	46,88	0,67	31,25	1,50	31,25	3,00
50000	218,77	0,57	172,98	0,90	140,64	1,34	157,34	1,70	156,26	2,80	187,51	4,59
500000	1640,66	0,80	1693,30	0,88	1568,25	1,19	1410,59	2,08	1332,77	3,23	1579,29	5,30
5000000	16652,10	0,81	17634,10	0,90	15567,60	1,29	14782,40	1,91	14487,80	3,11	16968,70	5,02
10000000	33039,40	0,78	34338,70	0,93	29679,30	1,33	26522,30	2,13	25520,40	3,46	30037,50	5,75

Versione parallela con CUDA

cluster number	10	speedup	15	speedup	20	speedup	30	speedup	50	speedup	100	speedup
point number												
5000	8,79	1,78	12,02	2,60	12,29	2,54	10,73	2,91	15,94	2,94	17,68	5,30
50000	18,44	6,78	27,91	5,60	33,11	5,69	35,11	7,60	51,25	8,54	89,55	9,61
500000	114,69	11,47	157,94	9,43	175,45	10,66	237,81	12,32	307,78	14,00	586,91	14,27
5000000	981,43	13,69	1198,61	13,18	1290,92	15,51	1840,52	15,35	2874,60	15,66	5288,10	16,11
10000000	1666,71	15,54	2173,28	14,70	2412,61	16,40	3545,59	15,94	5553,89	15,90	10232,50	16,87

8.2. Codici

```
void assignPointToCluster(float* points, float* clusters) {
#pragma omp parallel for default(none) shared(points, clusters)
num_threads(THREAD_NUMBER) schedule(static)
    for (int index=0; index < POINT_NUMBER; index++) {

        float x_cluster, y_cluster = 0;
        //si recuperano le coordinate del punto da assegnare
        float x_point = points[1 * POINT_NUMBER + index];
        float y_point = points[2 * POINT_NUMBER + index];
        unsigned int best_fitting = 0;
        float old_distance = FLT_MAX;
        float actual_distance = 0;

        for (int i = 0; i < CLUSTER_NUMBER; i++) {
            //si recuperano le coordinate del centroide
            unsigned int centroid_index = (int) clusters[0 * CLUSTER_NUMBER + i];
            x_cluster = points[1 * POINT_NUMBER + centroid_index];
            y_cluster = points[2 * POINT_NUMBER + centroid_index];

            if (DISTANCE == 0) {
                actual_distance = euclideanDistance(x_point, y_point, x_cluster,
y_cluster);
            } else {
                actual_distance = manhattanDistance(x_point, y_point, x_cluster,
y_cluster);
            }
            //se la distanza tra il centroide in esame è minore della vecchia distanza
            //allora si aggiorna il migliore centroide per il punto
            if (actual_distance < old_distance) {
                best_fitting = i;
                old_distance = actual_distance;
            }
        }
        //si  crea la relazione fra il punto e il suo miglior cluster
        points[0 * POINT_NUMBER + index] = (float) best_fitting;

       #pragma omp atomic
       clusters[1*CLUSTER_NUMBER+best_fitting]=clusters[1*CLUSTER_NUMBER+best_fitting]+1;
    }
}
```

Codice 1: metodo per assegnamento dei punti ai cluster con OpenMP

```
__global__ void assignPointToCluster(float* points, float* clusters) {
    unsigned int point_n = threadIdx.x + blockIdx.x * blockDim.x;
    if (point_n < POINT_NUMBER) {
        float x_cluster, y_cluster = 0;
        float x_point = points[1 * POINT_NUMBER + point_n];
        float y_point = points[2 * POINT_NUMBER + point_n];
        unsigned int best_fitting = 0;
        float old_distance = FLT_MAX;
        float actual_distance = 0;

        for (int i = 0; i < CLUSTER_NUMBER; i++) {
            unsigned int centroid_index = clusters[0 * CLUSTER_NUMBER + i];
            x_cluster = points[1 * POINT_NUMBER + centroid_index];
            y_cluster = points[2 * POINT_NUMBER + centroid_index];

            if (DISTANCE == 0) {
                      actual_distance = euclideanDistance(x_point, y_point, x_cluster, y_cluster);
            }
            else
            {
                      actual_distance= manhttanDistance(x_point, y_point, x_cluster, y_cluster);
            }
            if (actual_distance < old_distance) {
                best_fitting = i;
                old_distance = actual_distance;
            }
        }

        points[0 * POINT_NUMBER + point_n] = best_fitting;
        atomicAdd(&clusters[1 * CLUSTER_NUMBER + best_fitting], 1);
    }
}
```

Codice 2: metodo per assegnamento dei punti ai cluser con CUDA

```
float* points_host = (float*)malloc(POINT_NUMBER * POINT_ATTRIBUTES * sizeof(float));
    float* clusters_host = (float*)malloc(CLUSTER_NUMBER * CLUSTER_ATTRIBUTES * sizeof(float));
    float* points_device = 0;
    float* clusters_device = 0;
    //generazione e copia dati su device
    generatePointCluster(points_host, clusters_host);
    cudaMalloc(&points_device, POINT_NUMBER * POINT_ATTRIBUTES * sizeof(float));
    cudaMalloc(&clusters_device, CLUSTER_NUMBER * CLUSTER_ATTRIBUTES * sizeof(float));
    cudaMemcpy(points_device, points_host, POINT_NUMBER * POINT_ATTRIBUTES * sizeof(float),
    cudaMemcpyHostToDevice);
    cudaMemcpy(clusters_device, clusters_host, CLUSTER_NUMBER * CLUSTER_ATTRIBUTES * sizeof(float),
    cudaMemcpyHostToDevice);

    auto start_time = high_resolution_clock::now();
    for (int i = 0; i < ITERATION; i++) {
        //assegnamento punti al cluster
        assignPointToCluster << < (POINT_NUMBER + THREAD_FOR_BLOCK - 1) / THREAD_FOR_BLOCK,
        THREAD_FOR_BLOCK >> > (points_device, clusters_device);
        gpuErrchk(cudaDeviceSynchronize());

       //calcolo del valore su asse x e y utilizzato per il ricalcolo del centroide
        calculateValue << < (POINT_NUMBER + THREAD_FOR_BLOCK - 1) / THREAD_FOR_BLOCK,
        THREAD_FOR_BLOCK >> > (points_device, clusters_device);
        gpuErrchk(cudaDeviceSynchronize());

        //ricalcolo del centroide
        recomputeCentroid << <1, CLUSTER_NUMBER >> > (points_device, clusters_device);
        gpuErrchk(cudaDeviceSynchronize());

        //rimozione dei punti da ogni cluster
        removePoint << <1, CLUSTER_NUMBER >> > (clusters_device);
        gpuErrchk(cudaDeviceSynchronize());
    }
    gpuErrchk(cudaDeviceSynchronize());

    auto end_time = high_resolution_clock::now();
    duration<double, std::milli> total_time = end_time - start_time;


    cudaMemcpy(points_host, points_device, POINT_NUMBER * POINT_ATTRIBUTES * sizeof(float),
    cudaMemcpyDeviceToHost);
    cudaMemcpy(clusters_host, clusters_device, CLUSTER_NUMBER * CLUSTER_ATTRIBUTES * sizeof(float),
    cudaMemcpyDeviceToHost);
    cudaFree(points_device);
    cudaFree(clusters_device);
```

Codice 3: Main in CUDA