PC 2020/21 Elaborato Final Term

Alessandro Mini
E-mail address
alessandro.mini1@stud.unifi.it

Abstract

In questo elaborato "final term" si implementa l'algoritmo di clustering "K-Means" in versione sequenziale e parallela utilizzando C++, OpenMP e CUDA. Si utilizzeranno strumenti di profiling come Nvidia VisualProfiler e si analizzeranno tempi e speedups.

La versione sequenziale ed OpenMP condivideranno il codice, si potrà infatti scegliere il numero di cores da utilizzare e quindi se eseguirlo in modalità sequenziale o parallela.

L'implementazione CUDA è separata dalle altre.

1. Introduzione

K-Means è un algoritmo di clustering che individua k *clusters* all'interno di un set di oggetti tra cui è possibile definire una *distanza*, il numero k viene scelto prima dell'esecuzione dell'algoritmo. In alcune varianti k viene identificato eseguendo delle procedure di "best fitting".

Per ogni cluster si definisce un *centroide*, ossia un punto (immaginario o reale) che ne rappresenta il centro.

L'algoritmo k-means è iterativo, il suo funzionamento si può descrivere nelle seguenti fasi:

1. Inizializzazione: si definisce K ed il set su cui si andrà a lavorare.
2. Assegnazione del cluster: ogni data point viene assegnato al cluster (o centroide) più vicino;
3. Aggiornamento della posizione del centroide: ricalcola il punto esatto del centroide basandosi sul concetto di "centro di massa" e in base a questo ne modifica la sua posizione.

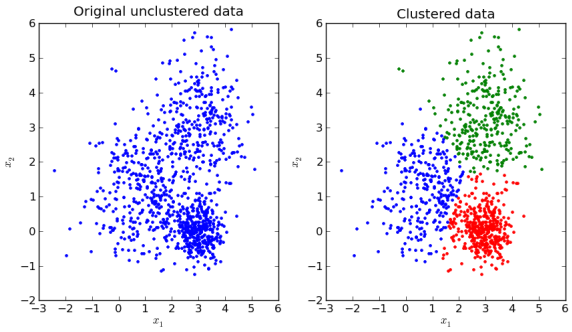

Figure 1. Esempio di esecuzione dell'algoritmo.

K-Means viene definito un algoritmo di "hard clustering" nel senso che un punto p_i può appartenere solo e soltanto ad un certo cluster C_m. Non esistono punti "non associati" a nessun cluster.

Si può dimostrare che l'algoritmo K-Means converge, ovvero in tempo *finito* completerà il processo di clustering su un dataset assegnato.

2. Implementazione Sequenziale/OpenMP

E' stato scelto di implementare il programma cercando di avere un implementazione "uniforme" in termini di strutture dati, tipi, etc.. in modo da avere una misura dello speedup più "pulita" possibile.

La versione sequenziale è implementata insieme ad OpenMP così sarà possibile scegliere il numero dei cores da utilizzare.

L'implementazione è fatta in modo tale che

l'utente possa scegliere il numero dei punti, dei clusters e dei thread attraverso delle defines iniziali.

```
#define COORD_MAX 100000
#define CLUSTER_NUM 100
#define POINT_NUM 10000000
#define POINT_FEATURES 3
#define CLUSTER_FEATURES 4
#define NUM_THREAD 12
#define IT_MAX 20
#define EPSILON 0.001
```

COORD_MAX rappresenta il range di coordinate del piano, CLUSTER_NUM e POINT_NUM rappresentano il numero dei punti e dei clusters.

POINT_FEATURES e CLUSTER_FEATURES rappresentano le caratteristiche dei punti e dei cluster e sono un parametro fisso.

IT_MAX indica il numero massimo di iterazioni dell'algoritmo mentre EPSILON è un parametro che indica la tolleranza per cui due punti sono "sufficientemente vicini".

2.0.1 Punti e clusters

I punti vengono implementati come matrice di numeri di tipo *float*.

```
float clusters[CLUSTER_NUM][CLUSTER_FEATURES];
float punti[POINT_NUM][POINT_FEATURES];
```

Un punto è composto da 3 elementi (x, y, c) dove:

- x, y sono le coordinate nello spazio 2D.
- C è il cluster di appartenenza.

Anche i cluster sono implementati come matrice di tipo float.

Un cluster è formato da 4 elementi (p, sx, sy, n) dove:

- p è un numero che corrisponde all'indice di quel punto nella matrice dei punti e definisce il centroide.
- sx, sy rappresentano la *size* su x e su y del cluster.
- n rappresenta invece il numero dei punti associati a quel cluster.

La distanza tra un punto p_i ed un cluster C_m è data dalla distanza euclidea tra p_i ed il centroide di C_m.

La distanza viene implementata con una funzione:

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

2.0.2 Assegnamento punti a cluster

Un punto p_i viene assegnato ad un cluster C_m secondo la seguente procedura:

```
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;
};
```

Si può notare l'utilizzo di "#pragma omp atomic" per "proteggere" il vettore dei cluster da eventuali race conditions.

Potrebbe infatti succedere che aello stesso tempo t due punti p_1 e p_2 competano per essere assegnati allo stesso cluster.

L'inserimento di un punto in un cluster comporta l'aggornamento del campo C (C =

$Cluster$) del punto e dei campi sx, sy, n secondo il seguente schema:

- $sx = sx + p_x$
- $sy = sy + p_y$
- $n = n + 1$

2.0.3 Ricalcolo del centroide

Un cluster deve ricalcolare il centroide secondo la seguente formula:

Se il centro è p con coordinate p_x e p_y le nuove coordinate saranno date da:

- $p_{newX} = \frac{p_x}{sx}$
- $p_{newY} = \frac{p_y}{sy}$

Concretamente questo viene fatto con il metodo:

```
void clusters_recomputeCenter(float
↪  points[POINT_NUM][POINT_FEATURES],float
↪  clusters[CLUSTER_NUM][CLUSTER_FEATURES]) {
↪  //cluster <punto,sizeX,sizeY,n_points>
    #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;
        }
    }
}
```

2.1. Esecuzione dell'algoritmo

Inizialmente vengono costruiti NUM_POINT punti e NUM_CLUSTER cluster inizializzandoli casualmente, si avvia quindi l'algoritmo vero e proprio.

Si utilizza la direttiva openMP per parallelizzare il ciclo for assegnandogli un numero di thread pari a NUM_THREAD. Per ogni iterazione l'algoritmo esegue le seguenti fasi $\forall i = 1..ITMAX$

1. Assgna tutti i punti parallelamente (o sequenzialmente) al proprio cluster.
2. Ricalcola il centroide di tutti i cluster.
3. Libera tutti i cluster (imposta $sx, sy, n = 0$)

Il main viene implementato come segue:

```
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("it: %d\n", i);
}
```

in cui si può notare la direttiva OpenMP che parallelizza il ciclo for.

La porzione di codice che rimuove i punti dai vari clusters (resettandone la size e n) viene implementata come segue:

```
void remove_points_from_clusters(float
↪  clusters[CLUSTER_NUM][CLUSTER_FEATURES]) {
#pragma omp parallel for default(none)
↪  shared(clusters) num_threads(NUM_THREAD)
↪  // <- parallelized
    for (int i = 0; i < CLUSTER_NUM; i++) {
    ↪  // <- <punto,sizeX,sizeY,n_points>
        clusters[i][1] = 0;
        clusters[i][2] = 0;
        clusters[i][3] = 0;
    }
}
```

Il programma presenta quindi i seguenti punti "parallelizzabili":

1. Assegnamento dei punti ai cluster.
2. Ricalcolo del centroide dei cluster.
3. Azzeramento dei cluster.

2.2. Tempi e speedup

In questa sottosezione si analizzeranno i tempi e i vari speedups ottenuti nella versione OpenMP rispetto a quella sequenziale. Si ricorda brevemente l' architettura di test: Una macchina con CPU Ryzen5 3600, 16Gb ram ddr4.

2.2.1 Versione sequenziale

Nella versione sequenziale con $IT\_MAX = 20$ si rilevano i seguenti tempi:

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

2.2.2 Versione OpenMP

Sfruttando i 6 core si ottengono i seguenti tempi:

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

3. Implementazione CUDA

Nell'implementazione CUDA si sono linearizzate le matrici dei punti e dei clusters.

```
float* punti = (float*)malloc(POINT_NUM *
↪  POINT_FEATURES * sizeof(float));

float* clusters = (float*)malloc(CLUSTER_NUM *
↪  CLUSTER_FEATURES * sizeof(float));
```

Si utilizza un approccio di tipo row-major, ovvero se M è una matrice con m righe e k colonne:

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

Avere le matrici linearizzate come vettori permetterà di avere una manipolazione più facile con CUDA.

3.0.1 Assegnamento punti a cluster

Per assegnare i punti ai vari cluster si applica la logica:

1. Mappo ogni punto p in un thread CUDA.
2. Il punto p "cerca" il suo *best-fitting* cluster C.
3. Si esegue l'associazione tra p e C

Questo viene implementato come segue:

```
__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);
        }
}
//Output, i assign the results:
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);
}
}
```

3.0.2 Ricalcolo del centroide

Il ricalcolo del centroide dei vari clusters viene eseguito attraverso CUDA, si impiegheranno $CLUSTER\_NUM$ [1] threads.

Ognuno di questi thread ricalcolerà il centroide del proprio cluster attraverso il kernel

```
cuda_remove_points_cluster(float*
↪  clusters)}
```

[1] Il numero dei cluster specificato

La stessa strategia si utilizza per la rimozione dei vari punti dei cluster.

```
__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;
}
```

Si può notare l'azzeramento delle varie variabili.

3.0.3 Esecuzione dell'algoritmo

Si hanno le seguenti defines per impostare i parametri del programma:

```
#define COORD_MAX 100000
#define CLUSTER_NUM 100
#define POINT_NUM 10000000
#define POINT_FEATURES 3
#define CLUSTER_FEATURES 4
#define THREAD_PER_BLOCK 1024
#define IT_MAX 20
#define EPSILON 0.001
```

Il parametro THREAD_NUM della versione OpenMP viene sostituito dal parametro THREAD_PER_BLOCK che, secondo la terminologia CUDA, imposta la dimensione in numero di thread dei vari blocchi.

Il main implementa l seguente logica:

1. Crea i punti ed i cluster con valori casuali nella memoria *host*.
2. Sposta punti e cluster nella memoria *device*.
3. Per $i = 1..2...IT\_MAX$:
   (a) Assegna i Clusters usando CUDA.
   (b) Ricalcola i centroidi usando CUDA.
   (c) Resetta i clusters usando CUDA.
4. Sposta i punti e clusters [2] dalla memoria *device* a quella *host*.
5. Conclude il lavoro.

[2] Dopo l'elaborazione.

3.0.4 Logica CUDA

La porzione del programma che esegue la maggior parte del lavoro è il kernel $assign\_clusters$.

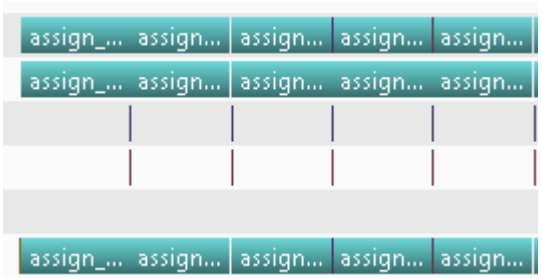

Figure 2. Profiling del programma attraverso Visual Profiler.

Per questo kernel si utilizza la seguente suddivisione in blocchi e thread:

```
assign_clusters << < (POINT_NUM +
↪  THREAD_PER_BLOCK - 1) /
↪  THREAD_PER_BLOCK, THREAD_PER_BLOCK >> >
```

Si creeranno quindi

$$\frac{POINT\_NUM + THREAD\_PER\_BLOCK - 1}{THREAD\_PER\_BLOCK}$$

Blocchi, ognuno dei quali contiene THREAD_PER_BLOCK threads.

A titolo di esempio se si devono "clusterizzare" $1 \cdot 10^6$ punti con un TPB [3] di 1024 si hanno 977 blocchi da 1024 thread per un totale di 1020928 threads.

Delle opportune barriere impediranno ad i thread in eccesso di eseguire lavoro.

[3] Thread per block.

3.1. Tempi e speedups.

Per la versione sequenziale si rimanda a Versione sequenziale sezione 2.2.1.

La versione CUDA con una GPU Nvidia GTX 950 produce i seguenti tempi:

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

Si può notare, come era prevedibile, che l'implementazione CUDA produce speedups molto maggiori della versione OpenMP.

Il tempo di lavoro massimo che nella versione sequenziale si aggirava intorno ai $298ms$ viene ridotto a $13ms$ per via dell'estrema parallelizzazione data dall'architettura CUDA.

4. Confronto OpenMP e CUDA

In questa sezione si confronteranno tempi e si eseguirà il plotting dei risultati per OpenMP e CUDA.

4.1. Tempi

Si esegue una comparazione dei 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

4.2. Plotting

Per eseguire il plotting si utilizza gnuplot, si formatta un file .txt con punti e cluster nella forma $x : y : cluster$. Si eseguono 2 plot nel caso $1 \cdot 10^7$ punti e 100 clusters.

4.2.1 Plot OpenMP

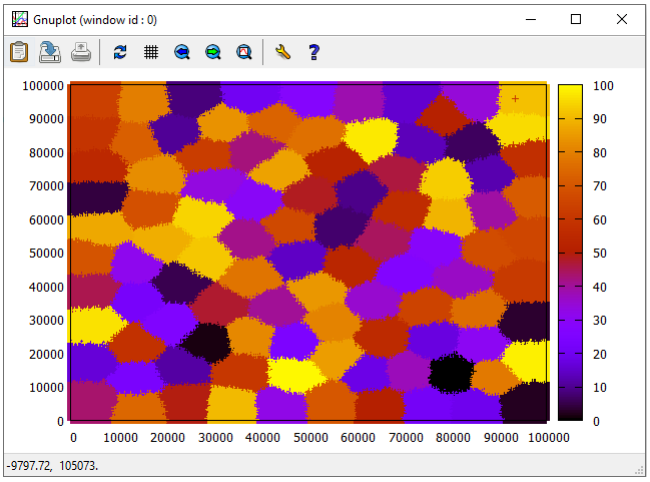

Figure 3. Clustering con OpenMP.

4.2.2 Plot CUDA

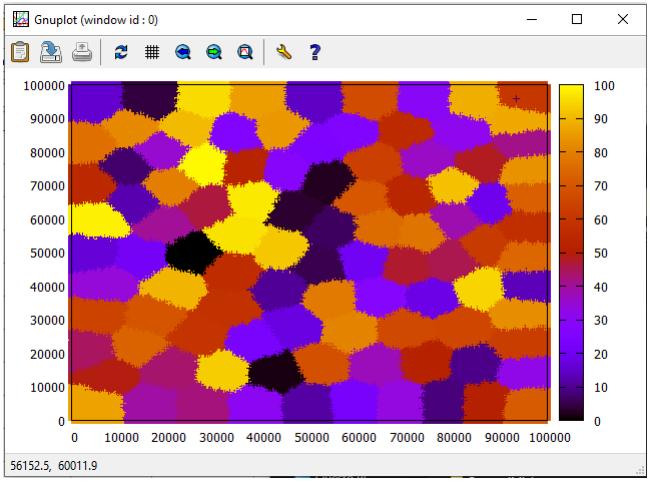

Figure 4. Clustering con CUDA.

5. Conclusioni

Si è implementato il clustering K-Means con OpenMP e CUDA. La versione OpenMP produce speedup più bassi ma con una maggior efficenza per via del numero più basso di cores utilizzati.

La versione CUDA attraverso l'enorme parallelizzazione che fornisce quest'architettura permette di ottenere speedup molto maggiori, circa 21 volte rispetto alla versione sequenziale e 3.30 volte rispetto alla versione OpenMP.