PC-2021/22 KMeans con OpenMP e CUDA

Prasanna Silva
hondamunige.silva1@stud.unifi.it

Abstract

L'algoritmo K-Means è uno degli algoritmi più semplici per effettuare il clustering dei dati. Questo report si concentra sull' implementare e analizzare le prestazioni di tale algoritmo in versione parallelizzata tramite OpenMP e tramite CUDA.

1. Introduction

L'algoritmo K-Means è un algoritmo unsupervised usato per effettuare il clustering dei dati conoscendo a priori il numero dei cluster. Tale algoritmo risulta essere **"embarrassingly parallel"** poiché ogni punto può essere assegnato a un cluster indipendentemente dal resto dei punti. Quindi si presta bene alla parallelizzazione tramite OpenMP ossia usando la parallelizzazione della CPU e soprattutto tramite la GPU. L'algoritmo si basa su tre fasi principali:

1. Calcolo dei centroidi.
2. Assegnamento dei centroidi ai punti del dataset.
3. Aggiornamento dei centroidi

Per semplificare ulteriormente, in questo report è stato scelto di utilizzare soltanto punti di dimensione due e i centroidi iniziali sono stati selezionanti casualmente dai dati. Questo report si pone l'obbiettivo di mostrare le prestazioni delle due versioni parallelizzate rispetto a quella sequenziale.

1.1. Pseudocodice del algoritmo K-Means

Algorithm 1 Pseudocode KMeans

input Dataset $x_1, ...x_p$, centroidi iniziali $c_1, ..., c_k$
  while Condizione di stop **do**
    for $i = 1$ to p **do**
      Assegnazione al punto x_i il centroide più vicino
    end for
    for $i = 1$ to k **do**
      for $j = 1$ to p **do**
        Calcola del punto medio del cluster i esimo
      end for
    end for
    for $j = 1$ to p **do**
      if $distanca(c_i, newcentroid_i) > threshold$ **then**
        $c_i = newcentroid_i$
      end if
    end for
  end while

L'algoritmo si ferma quando tutti i centroidi si spostano di una distanza minore di un threshold scelta a priori, la threshold scelta è e-6.

2. Parallelizzazione di KMeans

La fase del calcolo dei centroidi non è rilevante nel calcolo delle prestazioni, mentre l'assegnamento dei centroidi e il loro aggiornamento sono le fasi importanti in termini di prestazioni.Tra le ultimie due quella più costosa è l'aggiornamento dei centroidi. L'idea su cui si basa la parallelizzazione dell'algoritmo K-Means è la reduction. Nella fase dell'aggiornamento dei centroidi, la fase più rilevante dell'algoritmo viene suddividisa in due ulteriori fasi:

1. Somma dei punti di un cluster e calcolo del punto medio

2. Aggiornamento dei cluster nel caso in cui la distanza tra il nuovo centroide e quello vecchio sia superiore al threshold prescelto.

La reduction subentra nella prima fase dove vogliamo fare la somma dei punti appartenenti a un cluster; infatti, per velocizzare effettuiamo somme parziali dei punti appartenenti a un cluster e infine si ha una somma dei valori parziali ottenuti precedentemente.

3. Implementazione tramite OpenMP

In questa sezione verrà mostrata la versione parallelizzata tramite OpenMP. Come spiegato nella sezione precedente, non è rilevante nelle prestazioni la fase del calcolo dei centroidi,in quanto tale fase itera su un numero che non è rilevante per una parellelizzazione. Quindi partiamo con la parallelizzazione della fase di assegnamento dei centroidi.

3.1. Assegnamento dei centroidi tramite OpenMP

```
void calculateDistance(double vect_x[], double vect_y[],
                       centroid_point cp, int c_vect[])
{
  double dist,temp;
  int cluster_class;

  #pragma omp parallel for private(dist,temp,cluster_class)

  for (int i = 0; i < DATASET_SIZE; i++)
  {
   dist = distance(vect_x[i],vect_y[i],cp.x[0],cp.y[0]);
   cluster_class = 0;

   for (int j = 1; j < CLUSTER_SIZE; j++)
   {
      temp = distance(vect_x[i],vect_y[i],cp.x[j],cp.y[j]);
      if(dist > temp)
      {
         cluster_class = j;
         dist = temp;
      }
   }
   c_vect[i] = cluster_class;
  }
}
```

Listing 1. Assegnamento dei centroidi OpenMP

La funzione presenta due for annidati, quello più esterno che itera sui dati e quello interno sui cluster. Si effettua una parallelizzazione sul ciclo for più esterno, con schedulazione statica; pertanto a ogni thread viene assegnata una porzione di dati e per ogni dato calcola la distanza con tutti i centroidi, assegnando al dato il cluster più vicino. Non è stato scelto di fare una parallelizzazione anche del for più interno perché il ciclo più interno ha una dimensione piccola e non avrebbe guadagnato in prestazione.

3.2. Calcolo del punto medio di ogni cluster tramite OpenMP

```
void sumMeanCentroids(double vect_x[], double vect_y[],
                      int vect_c[],centroid_point &cp)
{

    int countDataPoints[CLUSTER_SIZE];
    double cx_tmp[CLUSTER_SIZE];
    double cy_tmp[CLUSTER_SIZE];

    for(int i = 0; i < CLUSTER_SIZE; i++)
    {
        cp.x[i] = cp.y[i] = 0;
        cx_tmp[i] = cy_tmp[i] = 0;
        countDataPoints[i] = 0;
    }

    #pragma omp parallel for default(shared) reduction
    (+:countDataPoints[:CLUSTER_SIZE],cx_tmp[:CLUSTER_SIZE]
     ,cy_tmp[:CLUSTER_SIZE] )

    for (int i = 0; i < DATASET_SIZE; i++)
    {
        countDataPoints[ vect_c[i] ] += 1;
        cx_tmp[vect_c[i]] += vect_x[i];
        cy_tmp[vect_c[i]] += vect_y[i];
    }

    for (int i = 0; i < CLUSTER_SIZE; i++)
    {
        cp.x_c[i] = cx_tmp[i]/countDataPoints[i];
        cp.y_c[i] = cy_tmp[i]/countDataPoints[i];
    }

}
```

Listing 2. Calcolo punti medi dei cluster

Questa è la funzione cruciale per le prestazioni, effettuiamo una reduction sulla somma dei punti appartenenti ai cluster; per fare ciò sono necessario delle strutture d'appoggio: utilizziamo quindi tre array di dimensioni del numero dei cluster. L'indice di questi array determinano il cluster

1. Array per mantenere il numero dei punti appartenenti a un cluster
2. Array per sommare la coordinata X dei punti appatenenti a un cluster
3. Array per sommare la coordinata Y dei punti appatenenti a un cluster

Si effettua una parallelizzazione al for che itera sui dati, con schedulazione statica. A ogni

thread è associato un chunk di dati, da ogni dato si estrapola il cluster d'appartenenza e questa informazione verrà usata come indice per gli array creati precedentemente. In base al cluster associato si incrementa il proprio contatore e si somma il punto negli altri due array. Infine, queste informazioni parziali vengono messe insieme, in questo modo si ottiene la reduction. Una volta che si hanno la somma dei punti appartenenti a un cluster e il contatore per ogni cluster si calcola il punto medio.

L'ultima fase ovvero l'aggiornamento dei centroidi non è stata parallelizzata poiché una volta ottenuto il punto medio per ogni cluster si effettua un controllo. Se la distanza tra il vecchio centroide e quello nuovo è minore della threshold non si ha l'aggiormaneto del centroide.

```
bool updateMP(centroid_point &cp, centroid_point cp_new)
{
    double cond;
    int count = 0;
    for (int i = 0; i < CLUSTER_SIZE; i++)
    {
        cond = distance(cp.x[i],cp.y[i],cp_new.x[i],
                                            cp_new.y[i]);

        if( cond <= THRESHOLD ) count++;
        else
        {
            cp.x[i] = cp_new.x[i];
            cp.y[i] = cp_new.y[i];
        }
    }
     if(count >= PERCENTAGE*CLUSTER_SIZE)
        return false;

    return true;
}
```

Listing 3. Calcolo punti medi dei cluster

4. Prestazioni della versione parallelizzata tramite OpenMP

In questa sezione si analizza le prestazioni ottenute tramite la paralizzazione OpenMP rispetto a quella sequenziale. Quella parallelizzata è effettuata con 8 thread, il massimo disponibile fisicamente dal computer utilizzato per fare i test.

Il primo test effettuato è su dataset di dimensioni variabili e numero di cluster fissato a 16.

Notiamo come la versione parallelizzata sia nettamente superiore alle prestazioni rispetto a quella sequenziale, soprattutto quando il numero di dati è molto elevato.

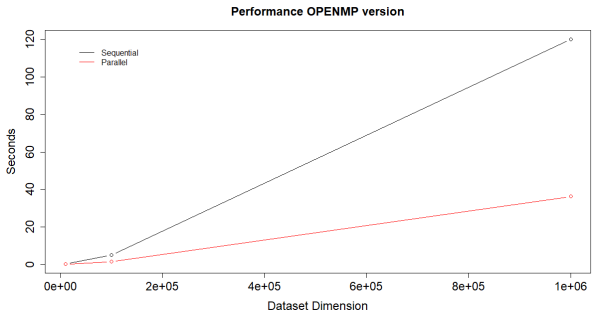

Figure 1. Dimensione dataset variabile e 16 cluster

Il secondo test è stato effettuato fissando il numero di dati a 100.000 e variando il numero di cluster.

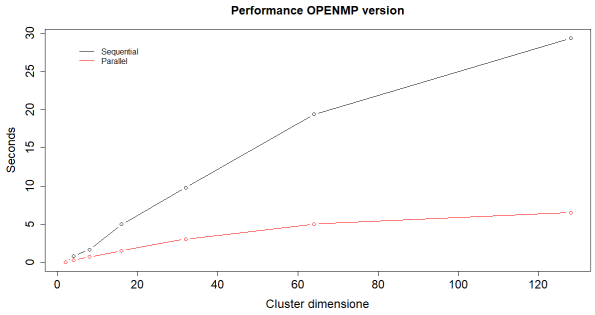

Figure 2. Dimensione dataset variabile e 16 cluster

Il terzo test è uguale a quello precedente, oltre a quello parallelizzato usando 8 thread vogliamo vedere le prestazioni variando il numero di thread utilizzati per parallelizzare.

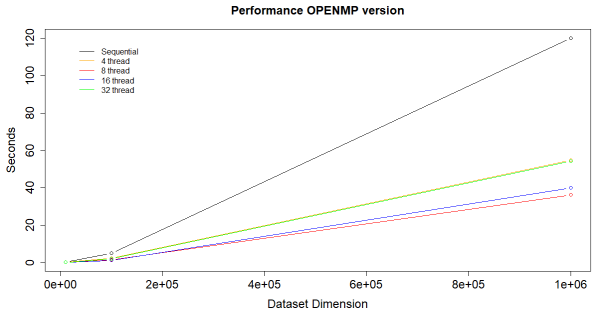

Figure 3. Numero di thread variabile

Si nota che, una volta superata il numero di core disponibile, l'incremento della schedulazione peggiora le prestazioni, sempre mantenendo una prestazione migliore a quella sequenziale.

5. Implementazione tramite CUDA

In questa sezione verrà mostrata la versione parallelizzata tramite CUDA. Come discusso precedentemente, l'algoritmo K-means è **"embarrassingly parallel"** e abbiamo visto tramite la parallelizzazione effettuata con OpenMP che le prestazioni migliorano significativamente all'aumentare dei thread fino al numero di core disponibili; quindi, è naturale pensare che la parallelizzazione effettuata con la scheda video possa dare delle prestazione ancora più performanti. Dato il numero elevato dei thread, che è possibile eseguire parallelamente rispetto al processore, e dato che i thread sono più leggeri rispetto a quelli del processore, il cambio di contesto è più leggero. Abbiamo notato che quando subentra la schedulazione nel processore le prestazioni iniziano a calare. Utilizziamo, il paradigma SIMT(Single Istruciton Multiple Thread) di CUDA; quindi, vogliamo associare a ogni thread la responsabilità di computare un dato e questo permetterà di incrementare notevolmente le prestazioni. Oltre ad adottare il paradigma SIMT, vogliamo effettuare una reduction nella fase dell'aggiornamento dei cluster come è stato fatto durante la parallelizzazione tramite OpenMP.

5.1. Assegnamento dei centroidi tramite CUDA

```
__global__ void calculateDistanceCuda(double vect_x[],
double vect_y[],double cp_x[],double cp_y[],int c_vect[])
{
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx >= DATASET_SIZE) return;

    double dist, temp;
    int cluster_class;

    __shared__ double s_vect_cx[CLUSTER_SIZE];
    __shared__ double s_vect_cy[CLUSTER_SIZE];

    if (threadIdx.x == 0)
    {
        for (int i = 0; i < CLUSTER_SIZE; i++)
        {
            s_vect_cx[i] = cp_x[i];
            s_vect_cy[i] = cp_y[i];
        }
    }
    __syncthreads();

    dist = distance(vect_x[idx], vect_y[idx],
                          s_vect_cx[0],s_vect_cy[0]);
    cluster_class = 0;

    for (int j = 0; j < CLUSTER_SIZE; j++)
    {
        temp = distance(vect_x[idx], vect_y[idx],
                              s_vect_cx[j],s_vect_cy[j]);
        if (dist > temp)
        {
            cluster_class = j;
            dist = temp;
        }
    }

    // updating to the beloging cluster
    c_vect[idx] = cluster_class;

}
```

Listing 4. Assegnazione dei cluster ai punti, cuda

Quindi, adottando il paradigma SIMT, vogliamo che ogni thread sia responsabile nell'assegnare un cluster a un punto; per velocizzare ulteriormente la proccedura è stato utilizzata la shared memory che è una memoria molto veloce, in questa fase gli unici dati che vengono riletti da tutti i thread di un blocco sono i centroidi, pertanto, è stato scelto di salvare all'interno della shared memory i centroidi. Sarà il primo thread di ogni blocco a essere incaricato di memorizzare i centroidi nella shared memory. Un'altra possibilità è quella di incaricare i thread stessi di farlo, ma è possibile soltanto nel caso in cui i centroidi siano in numero minore rispetto ai thread di un blocco. I thread non possono calcolare il centroide più vicino fino a che il primo thread non abbia finito di copiare i dati, *_syncthreads()*, una volta finito di copiarli, ogni thread associa il centroide più vicino al punto assegnato.

5.2. Calcolo del punto medio di ogni cluster tramite CUDA

```
__global__ void calculateCentroidMeans(int vect_c[],
double vect_x[],double vect_y[],double sum_c_x[],
                          double sum_c_y[],int num_c[])
{
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx >= DATASET_SIZE) return;

    __shared__ double s_vect_x[BLOCK];
    __shared__ double s_vect_y[BLOCK];
    __shared__ int s_vect_c[BLOCK];

    __shared__ double partial_sum_x[CLUSTER_SIZE];
    __shared__ double partial_sum_y[CLUSTER_SIZE];
    __shared__ int    partial_num[CLUSTER_SIZE];

    s_vect_x[threadIdx.x] = vect_x[idx];
    s_vect_c[threadIdx.x] = vect_c[idx];
    s_vect_y[threadIdx.x] = vect_y[idx];
```

```
    __syncthreads();

    if (threadIdx.x == 0)
    {
        int j;
        for (int i = 0; i < CLUSTER_SIZE; i++)
        {
            partial_sum_x[i] = 0;
            partial_sum_y[i] = 0;
            partial_num[i] = 0;
        }
        int q = 0;
        if (DATASET_SIZE - (blockIdx.x * BLOCK) < BLOCK)
            q = DATASET_SIZE - (blockIdx.x * BLOCK);
        else
            q = BLOCK;

        for (int i = 0; i < q; i++)
        {
            j = s_vect_c[i];
            partial_sum_x[j] += s_vect_x[i];
            partial_sum_y[j] += s_vect_y[i];
            partial_num[j] += 1;
        }
        for (int i = 0; i < CLUSTER_SIZE; i++)
        {
            atomicAdd(&sum_c_x[i], partial_sum_x[i]);
            atomicAdd(&sum_c_y[i], partial_sum_y[i]);
            atomicAdd(&num_c[i], partial_num[i]);
        }

    }
}
```

Listing 5. Calcolo dei punti medi per ogni cluster, cuda

Come spiegato precedentemente, anche nel caso della parallelizzazione cuda andiamo ad adottare una strategia di reduction e per far ciò è necessario suddividere l'ultima fase, ossia l'aggiornamento dei centroidi, in due: la prima dove calcola i punti medi per ogni centroide e la seconda dove fa l'aggiornamento dei centroidi.

Anche in questa fase utilizziamo la shared memory per velocizzare gli accessi fatti dai thread dei blocchi. L'idea è di copiare i dati utilizzati da un blocco nella shared memory e usare 3 array memorizzati anch'essi nella shared memory. Due array servono per sommare i punti dei centroidi e l'ultimo è impiegato per contare i punti appartenenti a un centroide.Gli indici di questi tre array rappresentano i cluster.

In questo caso, i thread di un blocco non sono incaricati di calcolare il punto medio di un cluster, ma bensì per copiare i valori del punto associato a essi nella shared memory.Una volta che i thread del blocco hanno copiato i punti nella memoria shared, il primo thread del blocco eseguirà le somme parziali e il conteggio dei punti per ogni cluster.

Infine, questi dati devono ricongiungersi in un'unica struttura dati, questi sono memorizzati nella memoria globale. Per fare ciò è necessario adottare *atomicAdd()*, altrimenti se piu thread di blocchi differenti scrivono nelle stesse strutture dati nella memoria globale ci possono essere delle race condition. Questo punto rappresenta il vero e unico punto debole di questa implementazione. Un'altra alternativa sarebbe stata far aggiungere a ogni thread il proprio dato nella cella ad esso associata, ma questo avrebbe comportato un peggiorato le prestazione in quanto tutti avrebbero dovuto eseguire l'operazione in maniera atomica. Infine, la seconda parte , ossia la l'aggiornamento dei centroidi.

```
__global__ void updateC(double sum_c_x[], double sum_c_y[],
int num_c[], double cp_x[], double cp_y[], double* count)
{

    const int idx = blockIdx.x * blockDim.x + threadIdx.x;

    __shared__ int c[BLOCK_C];

    if (idx >= CLUSTER_SIZE){return;}

    for (int i = 0; i < BLOCK_C; i++){c[i] = 0;}

    // Calculating the means of the centroids
    if (num_c[idx] == 0){num_c[idx] = 1;}

    sum_c_x[idx] = sum_c_x[idx] / num_c[idx];
    sum_c_y[idx] = sum_c_y[idx] / num_c[idx];

    double dist = distance(cp_x[idx], cp_y[idx],
                           sum_c_x[idx], sum_c_y[idx]);

    if (dist <= THRESHOLD){c[threadIdx.x] = 1;}
    else
    {
        c[threadIdx.x] = 0;
        cp_x[idx] = sum_c_x[idx];
        cp_y[idx] = sum_c_y[idx];
    }
    __syncthreads();
    // caluculating unchange centroids
    if (threadIdx.x == 0)
    {
        double sum = 0;
        for (int i = 0; i < BLOCK_C; i++)
        { sum += c[i]; }
        atomicAdd(count,sum);
    }
    // setting the partial sum vectors to 0
    sum_c_x[idx] = 0;sum_c_y[idx] = 0; num_c[idx] = 0;
}
```

Listing 6. Aggiornamento dei centroidi, cuda

Quest'ultimo non necessitava di essere parallelizzato per i dataset di test usati, ma nel caso in cui il numero di cluster sia elevato, parallelizzare l'aggiornamento dei centroidi può risultare significativo per le prestazioni.

6. Prestazioni della versione parallelizzata tramite CUDA

In questa sezione si analizza le prestazioni ottenute tramite la parallelizzazione CUDA rispetto a quella sequenziale. Verranno effettuate tre prove.

1. Dimensione di dataset variabile, fissato il numero di cluster e grandezza del grid
2. Dimensione del dataset e grandezza del grid fissato, numero di cluster variabile
3. Dimensione del dataset e numero di cluster fissato, grandezza del grid variabile.

Grafico della prima prova con dataset di grandezza: 10.000, 100.000, 1000.000 con 16 cluster.

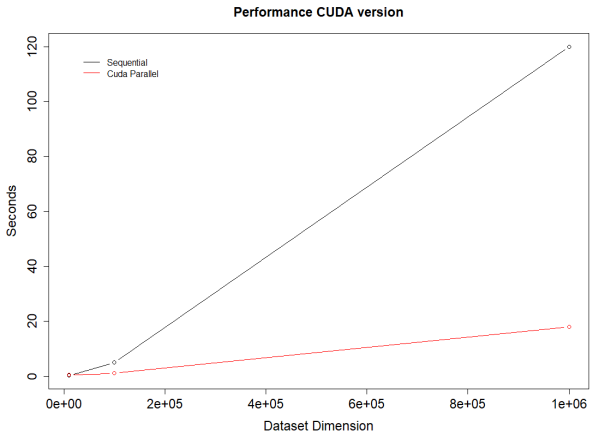

Figure 4. Dimensione del dataset variabile

Notiamo come la versione parallelizzata sia nettamente superiore alle prestazioni rispetto a quello sequenziale, soprattutto quando il numero di dati è molto elevato.

Grafico della seconda prova con dataset di dimensione 100.000 e cluster da: 2,4,8,32,64,128.

Per l'ultima prova le dimensioni del blocco utilizzato sono 32,64,128,256,1024. La dimensione del dataset è di 100000 con 16 cluster. Osserviamo come la grandezza del blocco ottimale sia 256. Si evince che la grandezza del blocco ottimale per tale implementazione è 256 thread a blocco.

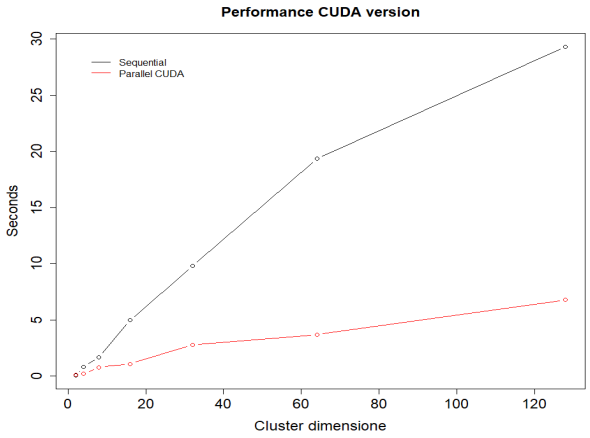

Figure 5. Numero di cluster variabile

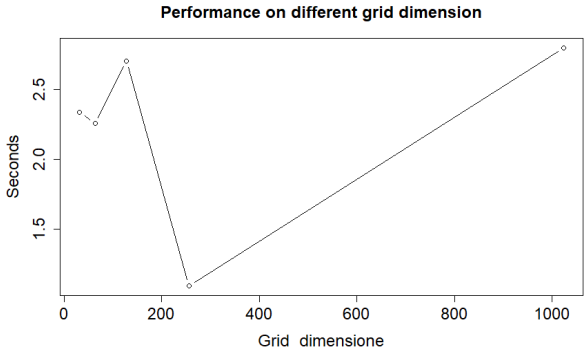

Figure 6. Grandezza blocco variabile

7. Confronto di prestazione tra OpenMP e CUDA

In questa sezione andremo a confrontare le prestazione tra OpenMP e CUDA. Le prove che andremo a testare sono quelle fatte precedentemente quindi:

1. Dimensione di dataset variabile, fissato il numero di cluster
2. Dimensione del dataset, numero di cluster variabile

Il dataset di dimensione 100.000 con 16 cluster. Useremo la versione di OpenMP con 8 thread e la versione di Cuda con grandezza del blocco di 256.

Notiamo nella fig. 7 che cuda ha delle prestazione migliori rispetto a quelle ottenute con OpenMP, quasi il doppio.

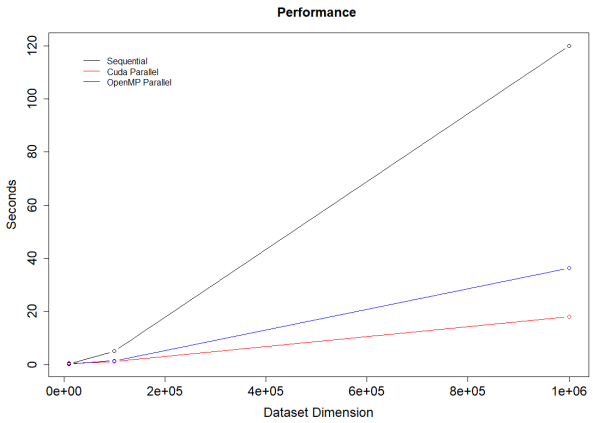

Figure 7. Grandezza blocco variabile

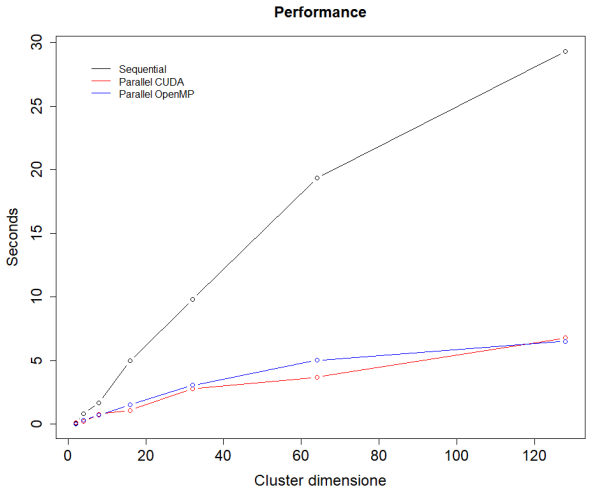

Figure 8. Grandezza cluster variabile

Prestazioni			
Dataset	Sequenziale	OpenMP	CUDA
10000	272 ms	90 ms	547 ms
100000	4976 ms	1418 ms	1092 ms
1000000	120065 ms	36243 ms	18072 ms

Table 1. Tempo di risoluzione

Per quanto riguarda l'ultima prova, su il numero di cluster variabile, le due versioni sono simili.

Entramabe le versioni sono migliori della versione sequenziale.I risultati trovati dalla versione CUDA rispetto a quello di OpenMP, non si differiscono di molto come ci saremmo aspettati di vedere, questo è dovuto molto probabilmente ai tre atomic necessari per effettuare la reduction.

8. Esperimenti

Ogni esperimento è stato svolto dieci volte. E viene poi fatta la media per avere un valore significativo, poichè il risultato dipende dai centroidi iniziali. Il computer utilizzato per questi esperipenti ha:

1. Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz 2.30 GHz
2. Ram: 16,0 GB
3. Scheda Video: Nvidia GeForce GTX 1050

La scheda video GTX 1050 con:

1. 640 cuda core
2. 5 Streaming Multiprocessors