Confronto delle implementazioni sequenziale e parallela dell'algoritmo K-Means

Lorenzo Macchiarini
7045205

Andrea Leonardo
7070775

Abstract

In questo progetto abbiamo analizzato come la parallelizzazione dell'algoritmo di clustering K-Means ne migliora le performances rispetto ad un'implementazione sequenziale. Abbiamo svolto i test implementando una versione sequenziale in C++, due versioni parallele: una in OpenMP che fa uso dei cores della CPU ed una in Cuda che fa uso dei cores della GPU.

1. Introduzione a K-Means

K-Means è un algoritmo di clustering e per questo ha come obiettivo quello di classificare in *cluster* un insieme di dati non etichettati. K-Means inoltre si basa sull'idea del *point assignment*: il numero di cluster è fissato a *k* e, una volta inizializzati i k cluster, l'algoritmo assegna ogni dato al cluster più vicino secondo una distanza definita. Ad ogni iterazione viene quindi ricalcolato il punto rappresentante del cluster, che chiameremo centroide, e ad ogni punto viene nuovamente assegnato il cluster più vicino. L'algoritmo termina la sua esecuzione quando raggiunge una condizione di stop. Lo pseudocodice dell'algoritmo è mostrato in Algorithm 1.

Algorithm 1 K-Means Pseudocode

```
1:  k: Number of clusters
2:  procedure K-MEANS(k, points)
3:      clusters ← createClusters(k)
4:      initializeClusters(clusters)
5:      do
6:          for each point in points do
7:              assignPointToCluster(point, clusters)
8:          end for
9:          recalculateCentroids(clusters)
10:     while not stoppingCondition()
11:     return clusters
12: end procedure
```

Nel nostro caso abbiamo che:

- la funzione di distanza utilizzata è la distanza euclidea in quanto abbiamo considerato punti n-dimensionali;
- *initializeClusters* prende un punto a caso dall'insieme di punti e calcola i $k-1$ punti più lontani, quindi i k cluster vengono inizializzati con questi punti come centroidi;
- *assignPointToCluster* calcola la distanza dal punto a tutti i cluster ed aggiunge il punto al cluster più vicino;
- *recalculateCentroids* sposta il centroide di ogni cluster al punto medio dei punti assegnati durante l'ultima iterazione;
- *stoppingCondition* controlla se la distanza fra il vecchi centroidi e i nuovi centroidi è minore di una certa soglia, se tale condizione è verificata l'algoritmo termina.

In Figure 1 è riportata l'esecuzione dell'algoritmo su un set di 50.000 dati appartenenti a 3 clusters. Mentre a sinistra è riportato il dataset creato con la funzione Python *make_blobs* in sklearn, nel centro è riportato il grafico di come l'algoritmo ha associato i punti ai 3 diversi cluster mostrando come la funzione di distanza delimita in modo netto l'insieme dei punti appartenenti ad un dato cluster, sulla destra è riportato il grafico che mostra i punti che sono stati classificati nel modo errato confermando l'ipotesi fatta al punto precedente. Il costo computazionale dell'algoritmo varia in base al numero di iterazioni in cui la stopping condition si verifica. Ciò non è predicibile a priori, quindi la stima che possiamo fare riguarda i costi di ogni funzione e delle iterazioni all'interno dei loop. Consideriamo n il numero di punti in ingresso e k il numero di cluster e notiamo che:

- *initializeClusters* ha un costo $\theta(n)$;
- il ciclo for interno ha un costo $\theta(n*k)$ in quanto svolge n iterazioni in cui vengono calcolate $n*k$ distanze;
- *recalculateCentroids* ha un costo di $\theta(n)$.

L'algoritmo quindi ha un costo $\theta(numIterations*n*k)$. Ci aspettiamo che gli algoritmi paralleli eseguano l'algoritmo in tempi minori perché parallelizzando il ciclo

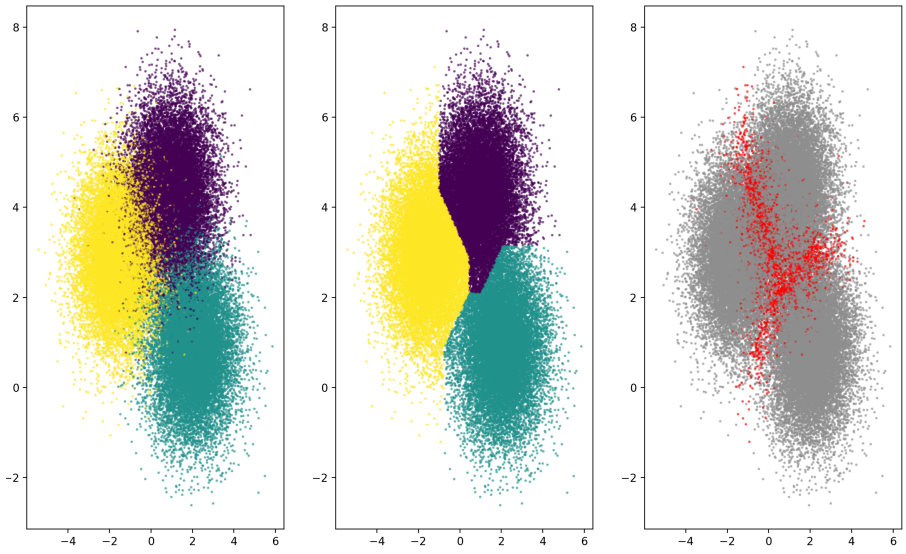

Figure 1. Esempio di esecuzione dell'algoritmo k-means su 50.000 dati in 3 cluster. Sono riportati: a sinistra il dataset, nel centro la classificazione fatta dall'algoritmo, a destra i punti classificati in modo errato.

for interno sui punti, ogni thread dovrà svolgere la computazione su un numero minore di punti facendo diminuire n.

2. Implementazione Sequenziale

Questa implementazione si basa sullo pseudocodice mostrato in precedenza, apportando alcune modifiche concettuali comuni anche alle implementazioni parallele. Il dataset utilizzato in tutte le versioni è stato ottenuto con uno script Python che utilizza la funzione *make_blobs* della libreria *sklearn*.

```
# sampleNumber definito dall'utente
# centerNumber = 3
# dimensions = 2
X, y, centers = make_blobs(
      n_samples=int(sampleNumber),
      centers=int(centerNumber),
      n_features=int(dimensions),
      random_state=0,
      return_centers=True
)
```

I valori ottenuti sono stati salvati in un file CSV in comune con tutte le implementazioni di k-means. Per leggere i files CSV abbiamo utilizzato la libreria *rapidcsv*.
Il codice sequenziale quindi implementa le funzioni riportate nello pseudocodice nel seguente modo:

- I cluster sono rappresentati solamente dai relativi centroidi attuali *double *centroids*, dai centroidi futuri *double *newCentroids* e dal numero di punti contenuti nel cluster *int pointsInCluster[k]*;

- *createClusters* viene implementata da **getDataset** in cui viene fatto un parsing del file CSV descritto in precedenza e viene ritornato un array sequenziale di *double*. Abbiamo scelto questa soluzione per semplicità, avendo testato anche array bidimensionali ed array di oggetti;

- *initializeClusters* viene implementata dalla funzione **getFarCentroids** a cui vengono passati il vettore di punti e le dimensioni di questi. In questa funzione viene preso un punto random nel set di punti, nel nostro caso il primo, e vengono calcolate le distanze fra tutti i punti del dataset e quello scelto; quindi vengono presi i $k-1$ punti con distanze massime ed infine ritornati come centroidi iniziali per i k cluster da inizializzare;

- L'aggiornamento dei cluster all'interno del ciclo è stato leggermente modificato in vista delle versioni parallele. Per ottimizzare la parallelizzazione, il ciclo for sui punti contiene la maggior parte del carico computazionale. In particolare l'aggiunta del punto al cluster non viene fatta aggiungendo il punto a un set di punti appartenenti al cluster, ma sommando le coordinate del punto a quelle del centroide più vicino e incrementando il contatore dei punti contenuti nel cluster. Quest'ultima versione permette di non dover fare un'ulteriore scansione di tutti i punti per calcolare il centroide aggiornato, ma basterà fare una media dei punti dividendo il centroide per il numero di punti

appartenenti al cluster. Di seguito viene riportata l'implementazione dell'aggiornamento dei cluster:

```
// For all data points
for (int i = 0; i < dataLength; i++){
   // Max distance from centroids
   double dist = FLT_MAX;
   // Id of the nearest cluster
   int clustId = -1;
   // Compute distance from k centroids
   for (int j = 0; j < k; j++) {
      // Distance from each Cluster
      double newDist = 0;
      for (int x=0; x<dimensions; x++){
         newDist += fabs(
            points[i*dimensions + x] -
            centroids[j*dimensions + x]
         );
      }
      if(newDist < dist) {
         dist = newDist;
         clustId = j;
      }
   }
   // Add point to the nearest cluster
   for (int x = 0; x < dimensions; x++){
      newCentroids[clustId*dimensions+x]
         += points[i*dimensions + x];
   }
   pointsInCluster[clustId]++;
}
```

Come descritto in precedenza, è possibile notare che ad ogni iterazione del ciclo vengono aggiunte le coordinate del punto al *newCentroid* relativo al cluster più vicino e viene quindi incrementato *pointsInCluster[clustId]*, il contatore dei punti presenti nel cluster;

- *recalculateCentroids* viene implementata da un semplice ciclo su *newCentroids* che per ogni cluster divide le coordinate del centroide con il contatore di punti nel cluster;
- Per valutare *stoppingCondition* viene calcolata la distanza fra i centroidi precedenti e quelli aggiornati nell'iterazione corrente e salvata in *distanceFromOld*. La stopping condition si limita a verificare che questa distanza sia maggiore di una soglia settata a 0.001.

3. Implementazione Parallela

In questa implementazione vengono riutilizzate le funzioni sequenziali che implementano lo pseudocodice di: *createClusters*, *initializeClusters*, *recalculateCentroids* e *stoppingCondition*.
Tale implementazione parallelizza il ciclo su tutti i punti da classificare, poiché è il task che può fornire uno speedup maggiore. L'aggiunta di un punto ad un cluster, nel modo in cui è stata implementata, può essere parallelizzata quasi completamente in quanto il solo calcolo delle distanze non mantiene variabili condivise fra i vari thread. Questi dovranno però essere sincronizzati per andare a sommare il valore del proprio punto al centroide a minima distanza, in quanto la scrittura in una variabile condivisa da più thread rappresenta una sezione critica.
Nell'implementazione proposta i dati sono organizzati in Array of Structures in quanto gli array *newCentroids* e *pointsInCluster* non sono consequenziali. Questa implementazione ha permesso una più veloce implementazione e non ha rappresentato un grande calo di performances in quanto i dati maggiormente utilizzati sono quelli dei punti, organizzati in un singolo array sequenziale.
In questo progetto sono state implementate due diverse versioni parallele: una in *OpenMP* che utilizza la parallelizzazione a livello di CPU ed una in *Cuda* che utilizza quella a livello di GPU.

3.1. OpenMP

L'implementazione con OMP si basa sulla parallelizzazione del for di aggiornamento dei $newCentroids$, poiché si ha un ciclo su un numero elevato di elementi, nel nostro caso i punti da clusterizzare:

```
#pragma omp parallel for num_threads(8)
   private(dist, newDist)
for (int i = 0; i < dataLength; i++) {
   dist = FLT_MAX;
   // Id of the nearest Cluster
   clustId[i] = -1;
   //#pragma omp parallel for shared(points,
   //centroids) reduction(+: newDist)
   for (int j = 0; j < k; j++) {
      //Distance from each Cluster
      newDist = 0;
      for (int x = 0; x < dimensions; x++) {
         newDist += fabs(points[i *
         dimensionsn+ x] - centroids[
         j * dimensions + x]);
      }
      if (newDist < dist) {
         dist = newDist;
         clustId[i] = j;
      }
   }
}
```

Come si può notare da un rapido confronto con la versione sequenziale precedentemente descritta, il ciclo parallelizzato non include l'aggiornamento dei centroidi, ma solamente l'aggiornamento dei $clustId$ a tal proposito.

Questo perché la fase di aggiornamento dei centroidi e dei $pointsInCluster$ include operazioni critiche che, se eseguite in modo atomico, rallenterebbero inutilmente il programma. Di conseguenza esse vengono eseguite sequenzialmente in seguito al precedente ciclo. Poichè dunque la variabile $clustId$ uscirebbe fuori scope alla fine di ogni iterazione del ciclo, i suoi valori vengono salvati nel vettore $clustId[]$, che viene utilizzato dunque nell'aggiornamento seguente. L'istruzione omp *#pragma omp parallel for num_threads(8)private(dist, newDist)* indica al programma le seguenti istruzioni:

- produrre un numero di thread indicati da num_threads (#pragma omp parallel num_threads())
- dividere il for successivo nei threads prodotti (#pragma omp for)
- specifica di un istanza privata per ogni thread delle variabili $dist$ e $newDist$

Dal $pragma$ iniziale si può notare, dunque, che ogni thread necessita della privatizzazione delle variabili $dist$ e $newDist$, poichè devonno essere trattate indipendentemente da ognuno di essi evitando concorrenza. Trattandosi di variabili piccole e non vettoriali esse possono essere memorizzate individualmente da ogni thread in esecuzione. E' possibile notare un commento con un ulteriore $pragma$, che descrive la possibile parallelizzazione del calcolo della distanza tramite $reduction$. La riduzione consiste nel ridurre gli elementi di un array in un unico risultato ottenuto tramite operazioni spesso associative e commutative (come somme e prodotti) calcolate dai threads in esecuzione. Questa però non genererebbe un grande vantaggio viste le poche iterazioni del ciclo in questione. Il concetto che per cicli piccoli sia più inefficiente organizzare un gruppo di threads che eseguire effettivamente le operazioni sequenzialmente si può applicare a tutti i restanti cicli del programma, i quali iterano o sulla dimensione dei punti o sul numero di centri.

3.2. Cuda

Questa implementazione si basa sul fatto che una GPU permette di parallelizzare un ciclo for allocando un nuovo thread per ogni iterazione. Nel nostro caso ogni thread creato sarà responsabile di svolgere la computazione relativa ad un singolo punto. I thread in Cuda vengono suddivisi in blocchi contenuti negli Streaming Multiprocessors. Il numero di thread, e quindi di blocchi necessari, è calcolato dopo aver fatto il parsing dei punti nel file CSV. Definito il numero di thread presenti in un singolo blocco, *threadPerBlock*, il numero di blocchi *numBlocks* viene calcolato come:

```
const int threadPerBlock = 512;
int dataLength = getDataLength();
int numBlocks =
      (dataLength + threadPerBlock - 1)/
      threadPerBlock;
```

Stabiliti questi due parametri, è necessario allocare i dati necessari per la computazione all'interno della memoria della GPU, chiamata device memory. In particolare vengono copiati in **global memory**, la memoria condivisa fra tutti i thread della GPU: i punti del dataset (*points_dev*), i centroidi non aggiornati (*centroids_dev*), i centroidi in aggiornamento (*blocksCentroids*), e il numero parziale di punti in ogni cluster (*blocksPointPerCluster*). Ad ogni iterazione del ciclo while in *centroids_dev* vengono scritti i valori di *newCentroids*, mentre *blocksCentroids* e *blocksPointPerCluster* vengono azzerati.
La prima versione della parallelizzazione prevedeva che ogni thread aggiungesse in modo atomico il valore del proprio punto al centroide più vicino. A causa del grande numero di somme atomiche su ogni centroide, il tempo di computazione risultava molto grande. L'implementazione scelta minimizza l'uso delle funzioni atomiche introducendo una reduction: ogni blocco ha al suo interno una copia locale di *blocksCentroids* e una di *blocksPointPerCluster* (chiamate *newCentroids* e *pointsInCluster*); ogni thread nel blocco esegue una funzione atomica su queste variabili locali e non sulle globali; infine, quando tutti i thread di ogni blocco hanno terminato la loro computazione, viene fatta una **reduction** aggregando i valori parziali di ogni blocco in *blocksCentroids* e *blocksPointPerCluster*. In questo modo il numero di operazioni atomiche fatte in modo concorrente sullo stesso dato è molto minore: mentre inizialmente questo era $\simeq dataLength$, nell'ultima implementazione è :

$$\simeq max(numBlock, threadPerBlock) \ll dataLength \tag{1}$$

. La funzione *assignClusterReduction* è chiamata su *numBlocks* blocchi contenenti *threadPerBlock* thread, ha i seguenti parametri in ingresso:

```
__global__ void assignClusterReduction
      <<<numBlocks, threadPerBlock>>>(
      dataLength,
      dimensions,
      numBlocks,
      points_dev,
      centroids_dev,
      blocksCentroids,
      blocksPointPerCluster
);
```

Coerentemente con quanto riportato sopra, la funzione

assignClusterReduction:

- inizialmente alloca per ogni blocco due vettori condivisi fra threads in **shared memory**:

```
__shared__ double newCentroids[
                              dimensions*k];
__shared__ int pointsInCluster[k];
if(threadIdx.x < dimensions*k) {
   newCentroids[threadIdx.x] = 0;
}
if(threadIdx.x < k) {
   pointsInCluster[threadIdx.x] = 0;
}
__syncthreads();
```

- se l'indice del thread è $\leq$ della lunghezza totale dei dati calcola il cluster di appartenenza in modo analogo all'implementazione sequenziale e quindi aggiunge in modo atomico il proprio punto al centroide condiviso in shared memory:

```
for (int x = 0; x < dimensions; x++) {
   atomicAdd(
   &(newCentroids[clustId*dimensions+x]),
   points[idx*dimensions + x]
   );
}
atomicAdd(&(pointsInCluster[clustId]),1);
```

- con una barriera vengono aspettati tutti i thread, quindi somma in modo atomico i risultati parziali di ogni blocco alle variabili in global memory:

```
__syncthreads();

if(threadIdx.x < dimensions*k) {
   atomicAdd(
   &(blocksCentroids[threadIdx.x]),
   newCentroids[threadIdx.x]
   );
}
if(threadIdx.x < k) {
   atomicAdd(
   &(blocksPointPerCluster[threadIdx.x]),
   pointsInCluster[threadIdx.x]
   );
}
```

Terminata la funzione, vengono copiati i valori di *blocksCentroids* e *blocksPointPerCluster* in *newCentroids* e *pointsInCluster*, quindi dal device all'host. In modo sequenziale vengono calcolati i valori dei centroidi aggiornati.

4. Test

I test svolti sulle varie versioni di k-means mettono in relazione i tempi di computazione ed il numero di dati in ingresso. Il numero di cluster k e il numero di dimensioni sono fissati rispettivamente a 3 e 2 in tutti i test. Per eseguire i test in modo automatico abbiamo implementato uno script *bash* che genera dataset aventi come dimensioni le potenze di 10 fino a 10 milioni e che esegue 20 volte le 3 versioni di k-means su ogni dataset.
Nei test svolti abbiamo notato come i tempi di computazione aumentino in modo lineare con il numero di punti. In particolare nella versione sequenziale è molto evidente questa relazione. Le versioni parallele riportate seguono questa linearità ma, a seconda dell'algoritmo, variano questa legge.

4.1. Versione Parallela OpenMP

I test per la parallelizzazione tramite OMP sono stati svolti su un numero incrementante di punti (esponenziale 10^x) e con un numero costante di centri (3). innanzitutto è stato fatto un paragone tra il tempo di esecuzione dell'algoritmo sequenziale, con la sua versione parallela a 4 threads. Il comportamento della versione parallela tende a migliorare all'incremento dei punti, portando ad un notevole miglioramento al superamento del milione di punti. Si ha invece un comportamento migliore della versione sequenziale per quantità relativamente piccole di punti, ovvero da 10 a $\sim$ 5000, come possibile notare dalla Figura2:

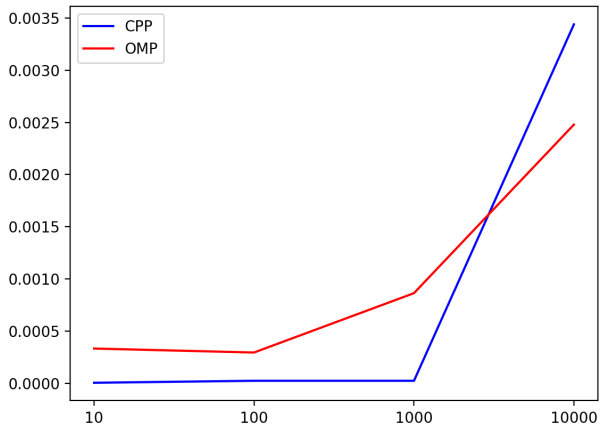

Figure 2. Grafico che confronta i tempi di esecuzione del codice C++ e OMP(4 threads) da 10 a 10000 punti.

Il comportamento generale del codice parallelo risulta però migliore con numeri significativi di punti. A questo punto sono stati svolti dei test per verificare il

comportamento per le versioni parallele al variare dei threads in esecuzione.

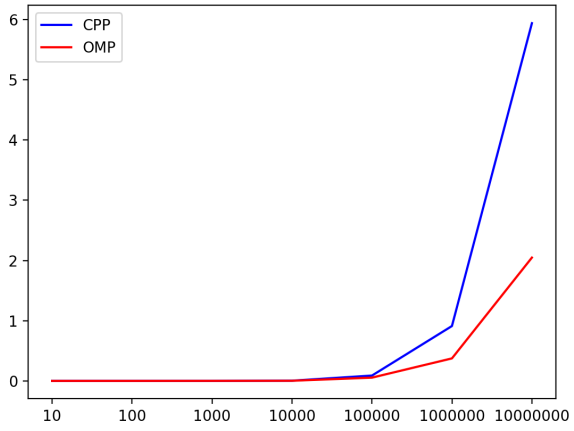

Figure 3. Grafico che confronta i tempi di esecuzione del codice C++ e OMP(4 threads) da 10 a $10mln$ punti.

I test sono stati effettuati per 4, 8, 16 e 32 threads. Il comportamento risulta sempre migliore, anche se di poco, all'aumentare del numero di threads, nonostante un lieve deficit per piccole quantità di dati dove l'organizzazione dei threads risulta svantaggiosa proporzionatamente al numero degli stessi.

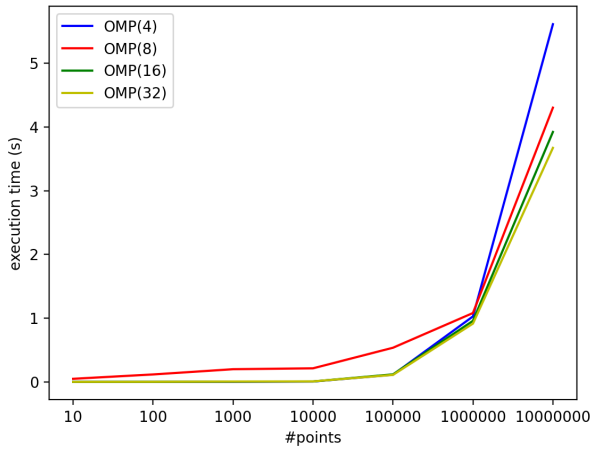

Figure 4. Grafico che confronta i tempi di esecuzione del codice OMP al variare del numero di threads.

4.2. Versione Parallela Cuda

Come esposto nella Section 3.2, tale versione parallela è stata implementata in due varianti: una solamente parallelizzando il for ed una che ottimizza l'uso delle operazioni atomiche con una reduction. Nella Figure 5 possiamo vedere quanto questa ottimizzazione abbia portato a dei miglioramenti nei tempi di esecuzione. Possiamo vedere che quando il numero di punti è basso, per esempio fino a 1000, i tempi di esecuzione sono quasi uguali. Da 10.000 punti invece la versione iniziale esegue $\simeq dataLength = 10.000$ operazioni atomiche, mentre la versione ottimizzata, con 1024 thread in ogni blocco, ne esegue , come riportato nell'Equazione 1, *max(10, 1024) = 1024*, con *numBlock = (dataLength + threadPerBlock-1)/ threadPerBlock*. Aumentando ancora di più il numero di dati, questo comportamento degenera; per esempio con 100.000 punti la versione non ottimizzata impiega circa 8 secondi mentre quella ottimizzata solamente 0.16 secondi.

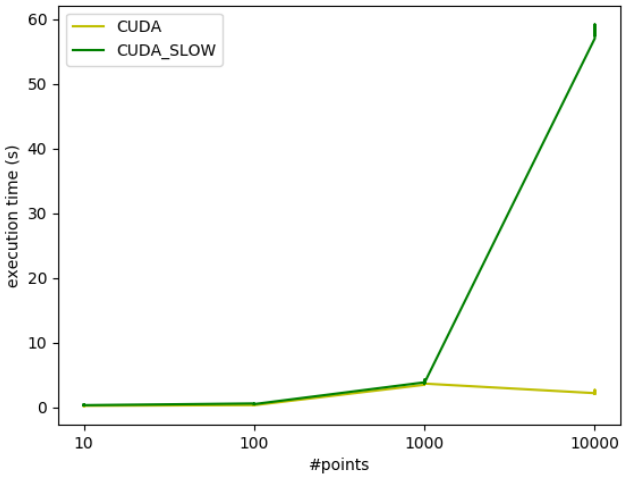

Figure 5. Grafico che confronta la versione ottimizzata di Cuda in giallo e la versione non ottimizzata in verde. La differenza di prestazioni fra le due versioni è data dal numero di operazioni atomiche eseguite.

Abbiamo testato anche come la versione ottimizzata varia il tempo di computazione al variare del numero di threads in ogni blocco; in particolare l'abbiamo testato con 128, 256, 512 e 1024 threads per block. Abbiamo notato come all'aumentare del numero di dati, il numero di thread per block migliore cambi. In particolare in Figure 6 notiamo che fino a 10000 punti la versione più prestante è quella con 128 threads. Questo è dato dal fatto che il numero di operazioni atomiche, come riportato nell'Equazione 1, è $\simeq threadPerBlock$ fino a quando $numBlock \leq threadPerBlock$, quindi fino a quando non abbiamo $dataLength \simeq threadPerBlock^2$. Quindi con pochi punti minore è il numero di thread per blocco, più veloce sarà l'esecuzione.

All'aumentare del numero di punti si vede che la versione con 1024 thread per blocco è la più prestante in quanto il numero di blocchi diminuisce all'aumentare il numero di

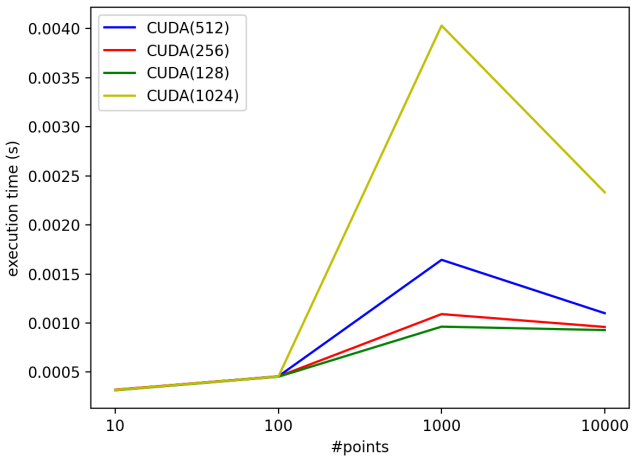

Figure 6. Grafico che confronta i tempi di esecuzione del codice Cuda al variare del numero di thread in ogni blocco. I colori associati ad ogni numero sono riportati sulla sinistra. Con pochi dati la versione con 128 thread è la più veloce.

thread nel blocco. Tale risultato a priori sarebbe contro intuitivo in quanto usando blocchi da 1024 thread non è possibile sfruttare tutti i thread della GPU, che ne ha 3840, lasciandone inutilizzati 768. Questa perdita di thread è però bilanciata con l'incremento delle performances dato dal numero minore di operazioni atomiche.

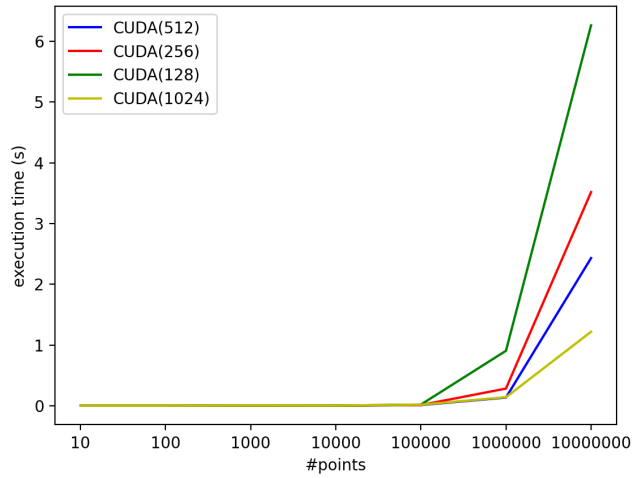

Figure 7. Grafico che confronta i tempi di esecuzione del codice Cuda al variare del numero di thread in ogni blocco. Con molti dati la versione con 1024 thread è la più veloce.

4.3. Confronto

Abbiamo confrontato le versioni più veloci di OpenMP (8 threads) e Cuda (1024 thread per blocco) con la versione sequenziale in C++ nella Figure 8. Come atteso i risultati delle versioni parallele sono molto migliori rispetto alla versione sequenziale. In particolare per grandi dimensioni del dataset la versione in Cuda impiega 0.5 secondi, la versione in OpenMP 2.3 secondi e in C++ 4.5 secondi. Lo speedup non può essere maggiore in quanto il problema non è imbarazzantemente parallelo a causa delle operazioni atomiche necessarie nell'aggiornamento dei centroidi.

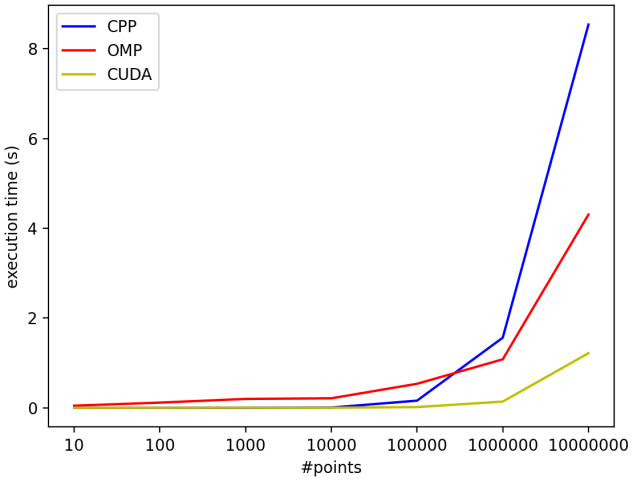

Figure 8. Grafico che confronta i tempi di esecuzione dei codici in C++, OpenMP e Cuda al variare del numero di punti in ingresso.

4.4. Hardware utilizzato

L'hardware utilizzato per la computazione è:

- **Intel Core i7-6700K**: 4 cores, 8 threads, 4,00 GHz;
- **Nvidia Quadro P6000**: 3840 cuda cores, GPU Memory: 24 GB GDDR5X, Memory bandwidth: 432 GB/s;
- Ram: 16 gb.

5. Conclusioni

Come atteso, abbiamo visto come l'utilizzo di una GPU con 3840 cuda cores possa eseguire l'algoritmo più velocemente rispetto ad una CPU con 8 cores e ad una con 1 core solo. Questo è dovuto sia alla formulazione del problema che lo rende quasi imbarazzantemente parallelo sia al modo in cui sono state minimizzate le operazioni atomiche negli algoritmi paralleli.