Progetto Midterm: K-Means

Marco Solarino
marco.solarino@stud.unifi.it

Simone Pezzulla
simone.pezzulla@stud.unifi.it

Abstract

In questo progetto è stato trattato l'algoritmo di clustering k-means. Per elaborare grosse quantità di dati è richiesta una notevole capacità di computazionale. Lo scopo è stato dunque quello di studiare approcci atti a ridurre i tempi di calcolo, tramite la parallelizzazione.

1. Introduzione

Il k-Means è un algoritmo che dati in input n punti nello spazio e un numero k, è in grado di raggruppare i punti in k diversi gruppi denominati $cluster$, minimizzando la distanza tra gli elementi appartenenti allo stesso cluster. Lo scopo del progetto è di ridurre i tempi di calcolo tramite l'impiego di:

- **CUDA**, un toolkit sviluppato da NVIDIA per sfruttare l'accelerazione data dalle GPU [1]
- **Hadoop**, un framework open source per sistemi distribuiti [2]

e di valutare l'eventuale aumento di prestazioni rispetto alla versione sequenziale.

1.1. Algoritmo del K-means

Lo schema di riferimento utilizzato per l'implementazione dell'algoritmo del K-means è il seguente:

Ricevi in input $points, k, L$
$C \leftarrow selectRandom(points, k)$
$i \leftarrow 0$
$changed \leftarrow true$
while $i < L$ & $changed = true$ **do**
  $assignCluster(points, C)$
  $changed \leftarrow updateCentroids(points, C)$
  $i \leftarrow i + 1$
end while

All'inizio sarà dunque necessario specificare anche il numero di iterazioni massime L.
Le funzioni invece hanno il seguente scopo:

- $selectRandom(points, k)$ sceglie k punti casuali tra quelli passati tramite argomento. Tali punti saranno i cosiddetti "centroidi", i centri di ogni cluster.
- $assignCluster(points, C)$ assegna ad ogni punto il centroide più vicino
- $updateCentroids(points, C)$ modifica la posizione di tutti i centroidi assegnando la media delle coordinate dei punti appartenenti al medesimo cluster e restituisce $true$ se la posizione di almeno uno di essi è effettivamente cambiata, $false$ altrimenti.

2. Implementazione

Di seguito verrà fatta una breve descrizione dell'implementazione delle tre versioni dell'algoritmo facendo riferimento allo schema generale della sez. 1.1

2.1. Versione sequenziale

La versione sequenziale è stata realizzata in C++.
Il programma, dopo aver ricevuto in input il numero di cluster da rilevare e il numero di iterazioni da eseguire, legge un file in formato .csv contenente le 3 coordinate di tutti i punti da analizzare. Al termine dell'esecuzione esso genera due file di output: nel primo vi sono le coordinate dei punti seguite dall'id del cluster di appartenenza e nel secondo le coordinate dei vari centroidi che verranno utilizzate nello script per rappresentare graficamente i risultati.
Per fini di valutazione delle performance è stato rimosso il flag $changed$ descritto nella sezione precedente per uniformare la quantità di iterazioni da una esecuzione all'altra poiché altrimenti il numero di iterazioni dipenderebbe dalla scelta dei punti casuali. Quindi L non sarà più il limite superiore ma diventa il numero stesso delle iterazioni.

Il codice e le istruzioni per l'utilizzo degli script sono consultabili al seguente indirizzo:
`https://github.com/MarcoSolarino/Midterm_Parallel_Computing_K-means`

2.2. Versione CUDA

La struttura e le funzionalità della versione in CUDA si basano sulla versione sequenziale in C++, pertanto tutte le implementazioni descritte nella sezione 2.1 rimangono invariate. Di seguito sono descritte le modifiche volte all'uso della GPU.

Il programma, dopo aver scelto i centroidi iniziali, allocato la memoria necessaria nella GPU e aver copiato le coordinate dei punti in global memory, esegue la funzione kernel $kMeansKernel$. Quest'ultima unisce $assignCluster$ e parte di $updateCentroids$ (dello schema della sez. 1.1), mentre la restante parte di quest'ultima è contenuta nella seconda funzione kernel $calculateMeans$. Tale suddivisione è dovuta alla necessità di introdurre un meccanismo di sincronizzazione a livello di tutti i thread per il calcolo delle nuove coordinate dei centroidi. Infatti, tutti i thread devono aver terminato prima di poter avere le somme parziali corrette necessarie al calcolo della media.
La dimensione dei blocchi è 128, ovvero un numero di warp sufficiente per nascondere la latenza di memoria.
Sono stati fatti vari accorgimenti per l'ottimizzazione dell'uso delle memorie:

- **Accesso coalesced**: tutti gli accessi in memoria globale sono effettuati in modo coalesced.
- **Shared memory**: le variabili alle quali viene fatto più di un accesso vengono caricate prima in shared memory.
- **Somme parziali**: la somma parziale delle coordinate dei punti dei diversi cluster, (utilizzata per il calcolo della media) viene effettuata a livello di blocco e quindi salvata in shared memory. Appena tutto il blocco termina le operazioni alcuni thread si occupano di sommare i valori ottenuti dalla shared alla global memory. Viene effettuato in modo analogo il conteggio del numero dei punti appartenenti a ciascun cluster.

Come per le altre versioni, il codice è versionato in una repository di GitHub:
`https://github.com/daikon899/Midterm_K-means_CUDA`

2.3. Versione Hadoop

Per ogni iterazione viene lanciato un job e un counter verrà incrementato se la posizione di almeno un centroide è variata. Tale stratagemma permetterà di capire al programma se eseguire altre iterazioni o arrestarsi nel caso in cui il counter non ha subito variazioni al termine dell'iterazione. Prima del lancio di ciascun job verrà scritta nel *context* la posizione aggiornata dei centroidi a cui farà riferimento il nuovo job.

Discutiamo ora dell'implementazione delle classi *Map*, *Combiner* e *Reducer* il cui schema è consultabile in figura 1.

- **Mapper**: ha il compito di esegue le istruzioni della funzione $assignCluster$ dello schema generale. Esso legge le coordinate dei centroidi dal contesto, riceve la stringa contenente le coordinate del punto da elaborare e cerca il centroide più vicino. In uscita dà la coppia chiave-valore $(Centroid, Point)$ dove il primo è una classe WritableComparable che contiene l'id e le coordinate del centroide più vicino al punto in questione e il secondo è una classe Writable dove vi sono le coordinate del punto in questione.
- **Reducer**: riceve in ingresso la coppia $(Centroid, Point)$ e ha la funzione di $updateCentroids$. La classe prende pertanto tutti i punti raggruppati per chiave, quindi appartenenti allo stesso cluster, e aggiorna le coordinate del centroide. In uscita la chiave non verrà utilizzata e quindi sarà un NullWritable, mentre il valore è la classe Centroid che contiene le coordinate aggiornate del centroide. Un'altra funzione del Reducer è di incrementare il contatore se le nuove coordinate del centroide sono diverse da quelle precedenti.
- **Combiner**: ha lo scopo di diminuire i punti da analizzare del Reducer e di ridurre il traffico dati. Esso crea come valore in uscita un oggetto di tipo Point che ha per coordinate le somme parziali delle coordinate dei punti analizzati. Il numero di punti analizzati è salvato dentro la variabile numPoints all'interno della classe Point (di default questo valore è 1). La chiave in uscita invece non subisce variazioni rispetto a quella in ingresso.

Una volta ottenuta la convergenza dell'algoritmo, viene lanciato un job con la sola classe Map. Questo permette di ottenere in uscita il centroide associato ai vari punti.
Per quanto riguarda la scelta dei centroidi iniziali, essendo il framework pensato per elaborare una mole di dati molto più grande di quella che potrebbe essere elaborata da un solo PC, la scelta viene effettuata casualmente sui primi $k * 20$ punti del dataset.

L'implementazione è stata fatta tramite l'ausilio di Maven ed è consultabile al seguente indirizzo:
`https://github.com/daikon899/Midterm_K-means_hadoop/`

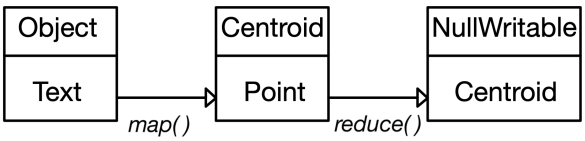

Figura 1: Schema delle varie coppie chiave-valore che sono elaborate in un job nella versione Hadoop

n	sequenziale	CUDA C	speed up
10	0.001 s	0.116 s	x0.008
10^2	0.015 s	0.117 s	x0.1
10^3	0.180 s	0.119 s	x1.5
10^4	1.673 s	0.147 s	x11.4
10^5	8.424 s	0.579 s	x14.5
10^6	83.024 s	5.706 s	x14.5
10^7	804.611 s	54.319 s	x14.8

Tabella 1: Confronto dei tempi di calcolo al variare di n. k è fissato a 10.

t.p.b	tempo
32	147 ms
64	145 ms
128	145 ms
256	146 ms
512	257 ms

Tabella 2: Confronto dei tempi di calcolo al variare del numero di thread per blocco. k e n sono fissati rispettivamente a 10 e 1000

3. Valutazione delle performance

Per la valutazione sono stati presi in considerazione le versioni C++ e CUDA.
Al variare della dimensione del dataset n è possibile apprezzare la bontà delle prestazioni della versione parallela. Come visibile in tabella 1 e nella figura 2 lo speed up raggiunge 14.8. Quest'ultimo ha un andamento linearmente crescente fino a una certa soglia per poi rimanere costante.

Nel secondo test è stato variato il numero di cluster tenendo la dimensione del dataset costante. Lo speed up ottenuto ha un andamento similare al primo test. (vedi figura 3)

Variando invece la dimensione dei blocchi della funzione kernel principale, le prestazioni rimangono invariate per poi subire un calo con blocchi da 512 thread. (tabella 2).

Si noti come per n molto piccoli l'overhead della versione CUDA renda i tempi più lunghi rispetto alla sua controparte, ma in generale le migliorie delle prestazioni ottenute tramite l'impiego della GPU sono evidenti. Difatti l'aumento di speed up è indice del fatto che più aumentano i punti (o i cluster) da analizzare e maggiori saranno i benefici. Tale beneficio è visibile a colpo d'occhio nelle figure 2 e 3 dove l'andamento dei tempi di calcolo di CUDA sale molto lentamente.
Per quanto riguarda il numero di thread per blocco, l'alto parallelismo dell'algoritmo non rende evidenti le differenze fino a 256 thread per blocco, mentre 512 risulta essere una dimensione troppo grande per sfruttare ottimamente le risorse degli Streaming Multiprocessor. Non è stato infatti possibile eseguire i test per dimensioni maggiori dei blocchi (ad esempio 1024 thread per blocco) perché non si avevano a disposizione abbastanza registri.

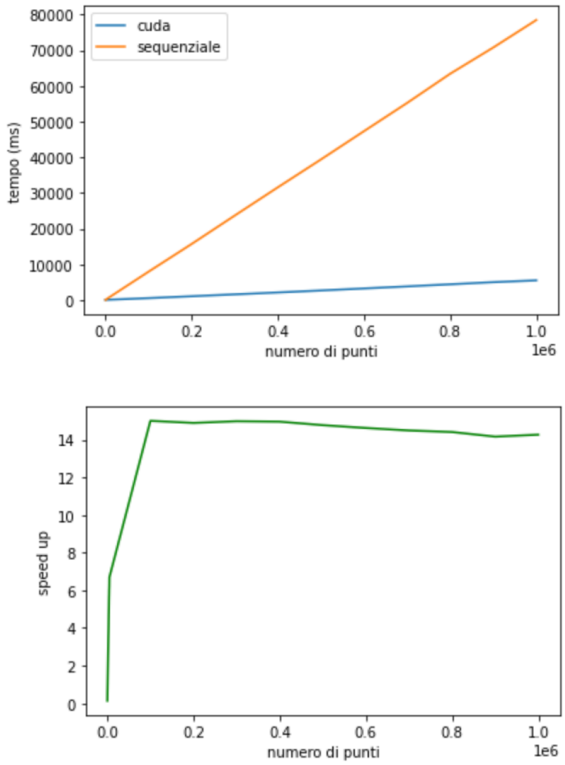

Figura 2: Prestazioni e relativo speedup in riferimento alla tabella 1.

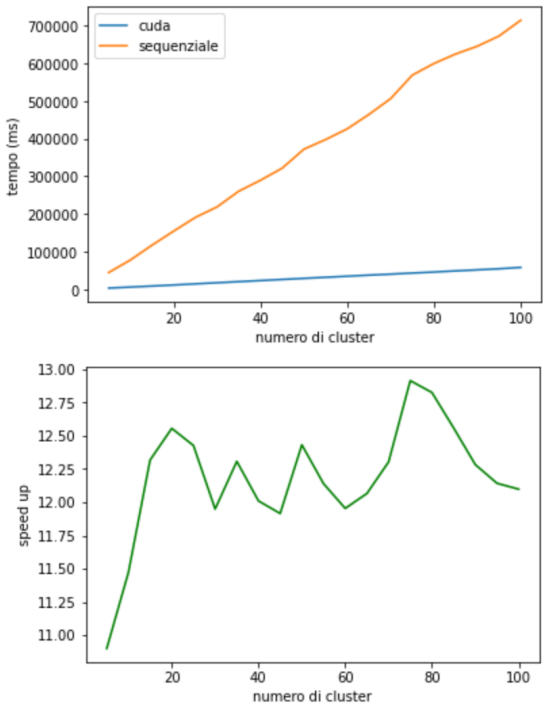

Figura 3: Prestazioni e speedup al variare di k (con $n = 1000000$)

3.1. Precisazioni

I risultati sono valutati su una media di 10 esecuzioni e con il numero di iterazioni prefissato a 50.
È stata inoltre introdotta la funzione $warmUpGpu$ nel codice CUDA per evitare l'overhead al primo uso della GPU.
Le specifiche della macchina sulla quale sono stati eseguiti i test sono le seguenti:

- processore Intel® Core™ i7-860
- scheda grafica NVIDIA GeForce GTX 980 Ti
- sistema operativo ubuntu Xfce

4. Conclusioni

Tramite l'impiego delle schede grafiche è stato possibile migliorare le performance in modo soddisfacente. Purtroppo non è stato possibile un confronto con la versione Hadoop, che è pensato per elaborare un numero di dati più alto di quello che potrebbe essere salvato su un singolo disco locale.

Riferimenti bibliografici

[1] *https://developer.nvidia.com/cuda-toolkit*.

[2] *https://hadoop.apache.org/*.