Parallel Computing
Relazione sull'implementazione dell'algoritmo K-means

Lorenzo Mandelli

2019-2020

1 Abstract

Il presente elaborato si concentra sull'analisi dell'algoritmo di clustering K-means e ne studia le performance al variare delle scelte implementative. In particolare, oltre alla versione sequenziale, sono state realizzate altre versioni che sfruttano il parallelismo dei processori e delle GPU attraverso i linguaggi di programmazione Omp e CUDA.

2 K-means

L'algoritmo K-means é un algoritmo di clustering che consente di suddividere un insieme di N oggetti in K gruppi sulla base dei valori dei loro attributi. Esso prende in ingresso l'insieme degli oggetti e dopo aver inizializzato i centroidi (i centri dei cluster) itera due fasi: l'assegnazione degli oggetti ai cluster e la rilocazione dei centroidi. Nella prima fase ciascun oggetto è assegnato al cluster con centroide piú vicino. Nella seconda la posizione dei centroidi viene aggiornata con la media dei valori degli attributi degli oggetti assegnati ai rispettivi clusters.

Il procedimento continua fino a quando non è soddisfatto un certo criterio di blocco. Nell'implementazione sono stati scelti come criteri:

- Un numero massimo di iterazioni dell'algoritmo.
- Uno spostamento minimo tra i centroidi di due iterazioni successive.

Se il numero di iterazioni supera quello massimo o la distanza tra i centroidi di due iterazioni successive è minore dello spostamento minimo l'algoritmo termina.

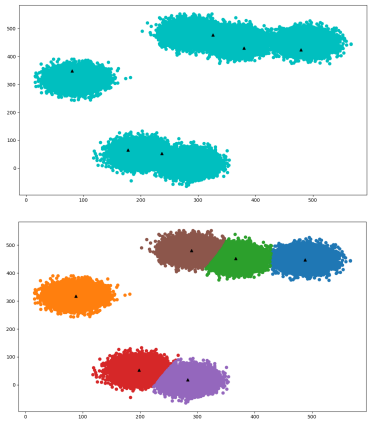

Figura 1: **Esempio di applicazione di kmeans con 6 clusters e $1 * 10^5$ elementi. Le due figure mostrano la disposizione dei cluster nelle iterazioni iniziali e finali.**

2.1 Versione sequenziale

Nella versione sequenziale dell'algoritmo, realizzata nel linguaggio di programmazione c++, i cluster sono stati realizzati attraverso mappe chiave-valore in cui le chiavi sono i centroidi e i valori le liste di oggetti assegnati ai clusters. La struttura del programma è analoga a quella dell'algoritmo: vi è un ciclo *for* esterno per il numero massimo di iterazioni e due cicli *for* interni che corrsipondono alle sue due fasi.

3 Omp

OpenMP è un API multipiattaforma per la programmazione parallela. Esso di basa sull'utilizzo di direttive che consentono la parallelizzazione di sezioni di codice che sarebbero altrimenti eseguite in modo sequenziale.

3.1 Versione parallela

La versione parallela si basa sulla parallelizzazione di entrambe le fasi dell'algorimo attraverso sulla direttiva *"pragma omp parallel for"*.
Di seguito sono riportati i dettagli implementativi.

3.1.1 Implementazione

I clusters sono stati realizzati attraverso mappe chiave-valore in cui le chiavi sono i centroidi e i valori sono dei puntatori a delle liste di tipo lockfree (della libreria boost) di oggetti appartenenti ai clusters.

La fase di assegnazione degli oggetti ai centroidi è parallelizzata attraverso la direttiva *"pragma omp parallel for"*: ogni oggetto è assegnato a un thread che esegue la ricerca del centroide più vicino e assegna il proprio oggetto al rispettivo cluster mediante un push sulla lista corrispondente.
Mediante la stessa direttiva è stata parallelizzata anche la fase di rilocazione dei centroidi: ogni cluster è affidato a un thread che calcola la media dei valori degli oggetti assegnati al proprio cluster e aggiorna con tale valore la posizione del relativo centroide. Durante il calcolo della media, per leggere il valore degli attributi degli oggetti, questi sono rimossi mediante un pop dalle rispettive liste.

4 CUDA

CUDA (Compute Unified Device Architecture) è una piattaforma per la programmazione parallela che rende possibile l'utilizzo del potere di calcolo delle GPU (graphics processing unit).

4.1 Versione parallela

La versione parallela consiste nell'iterazione di due chiamate kernel.
La prima opera l'assegnazione degli oggetti ai cluster ed esegue la somma dei valori degli attributi degli oggetti per ogni cluster. La seconda utilizza questa somma per ricavare la posizione aggiornata dei centroidi.
Di seguito sono riportati i dettagli implementativi delle due chiamate.

4.1.1 Implementazione

L'insieme degli oggetti, dei centroidi e dei vari vettori di supporto sono sta-

ti realizzati secondo una struttura *SoA* (*Structure of Arrays*).
La prima chiamata riceve in ingresso:

- il numero degli oggetti N.
- il numero dei centroidi K.
- un vettore R_i di dimensione N per ogni attributo degli oggetti. Questi vettori contengono l'informazione complessiva dei valori del dataset.
- un vettore C_i di dimensione K per ogni attributo dei centroidi. Questi vettori contengono l'informazione dei valori dei centroidi.
- un vettore S_i di dimensione K per ogni attributo dei dati. Questi vettori hanno lo scopo di contenere per ogni cluster la somma dei valori dei rispettivi attributi degli oggetti a loro assegnati.
- un vettore di interi D di dimensione K da utilizzare per mantenere traccia del numero di oggetti assegnati a ogni cluster.
- un vettore di interi I di dimensione N che servirà a indicare a quale cluster ciascun oggetto è stato assegnato.

La chiamata kernel identifica un thread per ogni oggetto appartenente al dataset attraverso gli indici di blocco e di thread (*"blockIdx.x"*, *"blockDim.x"*, *"threadIdx."*) ed opera nel seguente modo:

- Istanzia nella memoria condivisa per ciascun blocco i valori degli attributi dei centroidi permettendo di ridurre il numero di letture nella memoria globale.
- Identifica mediante i propri indici di thread e di blocco l'oggetto del dataset R che dovà gestire.
- Confronta la posizione del oggetto corrente con quella dei centroidi per trovare quello a distanza minima per poi assegnare l'oggetto al cluster del centroide scelto mediante il vettore di interi I
- Attraverso un *AtomicAdd* somma i valori degli attributi dell'oggetto corrente agli elementi dei vettori S_i individuati dal cluster scelto.
- Attraverso un *AtomicAdd* somma il valore uno all'elemento del vettore D individuato dal cluster scelto.

Alla fine dell'esecuzione quindi i vettori S_i di somma dei valori degli attributi dei clusters, i vettori D_i di dimensione dei clusters e il vettore I di assegnazione degli oggetti ai clusters risultano aggiornati.

La seconda chiamata prende in ingresso:

- I vettori C_i di dimensione K contenenti l'informazione dei valori dei centroidi.
- I vettore S_i di dimensione K contenenti la somma dei valori degli oggetti assegnati ai clusters.
- Il vettore di interi D di dimensione K contenente il numero di oggetti assegnato a ogni clusters.

La chiamata identifica un thread per ogni cluster attraverso l'indice di thread ("*threadIdx.x*") ed opera eseguendo la divisione dei valori della somma-

toria dei cluster per la relativa dimensione e aggiornando con tale valore la posizione dei centroidi.

5 Risultati

Nei test sono stati presi in considerazione oggetti con dimensialitá due, ovvero dotati di due attributi. Ogni test svolto opera eseguendo la media di 10 esperimenti. A ogni esperimento l'insieme degli oggetti, in ingresso alle varie versioni dell'algoritmo k-measn è generato randomicamente.
I parametri del numero massimo di iterazioni e di spostamento minimo dei centroidi per i criteri di arresto trattati nella sezione 2 sono stati impostati rispettivamente a 100 e 0.01.
I test sono stati eseguiti su un computer con processore Intel(R) Core(TM) i5-8250U con 4 core fisici e 8 core logici, 8.00 GB di RAM e scheda grafica NVIDIA GeForce MX130.

I risultati dello speedup ottenuto attraverso Omp sono mostrati nelle seguenti figure. La figura 2 mostra la variazione dello speedup al variare del numero di oggetti N del dataset con numero di cluster K pari a 10, come si può vedere esso tende a stabilizzarsi intorno al valore 5.

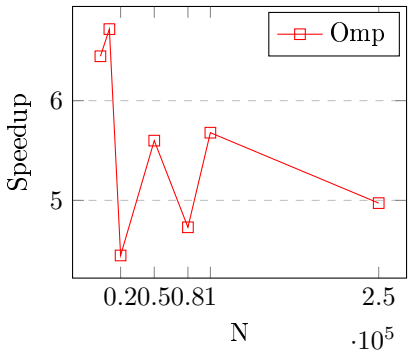

Figura 2: **Speedup di Omp al variare di N con K=10 fissato.**

La figura 3 mostra la variazione dello speedup al variare del numero di clusters K con numero di oggetti N pari a $5 * 10^5$. Si può notare come l'aumento del numero di clusters corrisponda a un incremento dello speedup. Ciò è dovuto alla maggiore parallelizzazione dell'algoritmo che si ottiene nella sua fase di rilocazione dei centroidi, in essa infatti il numero dei threads lanciati dipende dal numero K come spiegato nella sezione 3.1.1.

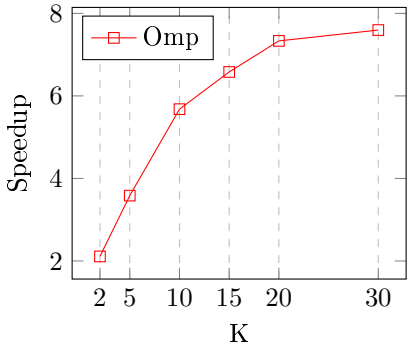

Figura 3: **Speedup di Omp al variare di K con N= $5 * 10^5$.**

Nel linguaggio CUDA sono state realizzate due versioni dell'algoritmo che differiscono tra loro dall'utilizzo o meno della memoria condivisa dei blocchi. Nei successivi grafici è indicata con CUDA 1 la versione priva di memoria condivisa, con CUDA 2 la versione che invece ne fa uso e con TPB la dimensione dei blocchi. Le figure 4 e 5 mostrano la differenza di speedup delle due versioni di CUDA rispettivamente al variare del numero di oggetti N e al variare del numero di clusters K.

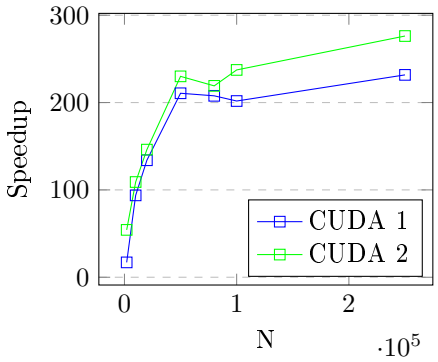

Figura 4: **Confronto degli speedup delle 2 versioni di CUDA dell'algoritmo al variare di N con K=10, TPB=512 fissati. La versione 2 utilizza la memoria condivisa al contrario della versione 1.**

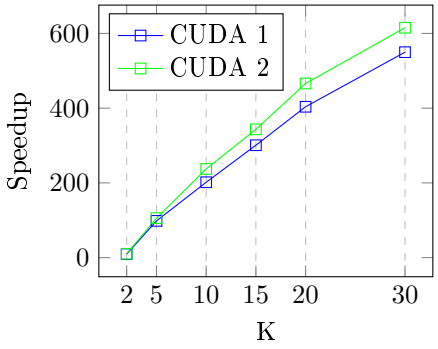

Figura 5: **Confronto degli speedup delle 2 versioni di CUDA dell'algoritmo al variare di K con N= $5 * 10^5$, TPB=512 fissati.**

Come si può vedere la memoria condivisa permette di ottenere un reale aumento delle prestazioni a parità di condizioni. Inoltre l'aumento del numero di centroidi implica per la versione 1 un sempre maggior numero di letture delle loro posizioni in memoria globale al contrario della versione 2, che salva queste informazioni nella memoria condivisa di più veloce accesso.
Sono stati eseguiti inoltre dei test sulla dimensione dei blocchi in cui il valore 512 è risultato il più efficiente come mostrato in figura 6.

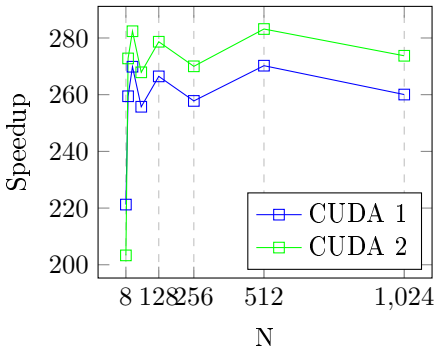

Figura 6: **Confronto degli speedup delle 2 versioni, con e senza l'utilizzo di memoria condivisa, di CUDA dell'algoritmo K-means al variare di TPB con K=10, N= $5 * 10^5$ fissati.**

Come mostrato nelle precedenti sperimentazioni l'utilizzo di CUDA e delle GPU ha permesso di ottenere ingenti speedup, sull'ordine di 250, molto maggiori di quelli ricavati mediante l'utilizzo di OpenMP e dei processori, che restano inferiori a 10. Di seguito nelle figure 7 e 8 sono riportati i confronti dei tempi di esecuzione dell'algoritmo nelle versioni sequenziali e parallele, di OpenMP e di CUDA al variare di N e di K.

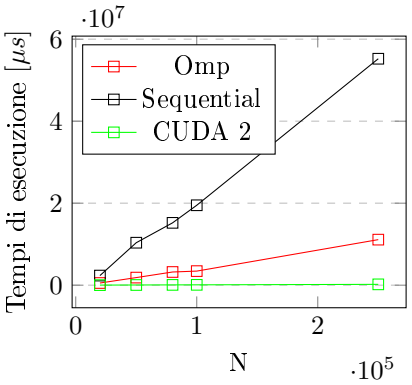

Figura 7: **Confronto dei tempi di esecuzione ottenuti attraverso la versione sequenziali, di Omp e di CUDA al variare di N con K=10, TPB=512 fissati.**

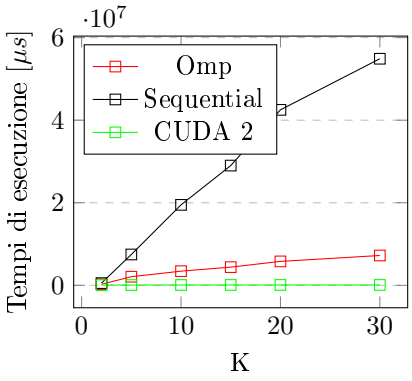

Figura 8: **Confronto dei tempi di esecuzione ottenuti attraverso la versione sequenziali, di Omp e di CUDA al variare di K con N= $5 * 10^5$, TPB=512 fissati.**