Algoritmo K-Means in CUDA e confronto prestazioni fra CPU e GPU

Jacopo Sabatino
jacopo.sabatino@edu.unifi.it

Abstract

In questo articolo è stato implementato l'algoritmo K-Means in linguaggio CUDA per poter verificare a livello prestazionale la sua esecuzione su GPU. I risultati ottenuti sono stati confrontati con un'implementazione K-Means eseguita su CPU.

1. Introduzione

Per questo elaborato è stata implementata una versione dell'algoritmo K-Means che, grazie ad un device acceleratore, ha migliori prestazioni sulla parte di calcolo intensivo.

L'algoritmo è stato sviluppato in linguaggio CUDA proprio per poter utilizzare un device acceleratore. L'acceleratore impegnato nei test è una GPU **RTX 3050 Ti**.

2. Implementazione

L'algoritmo K-Means implementato è nella sua versione base. I punti su cui è eseguito l'algoritmo sono tridimensionali. L'esecuzione parte da un numero K di centroidi predefiniti e la condizione di interruzione imposta si basa su un numero finito di esecuzioni del ciclo. Le ultime due condizioni permettono di avere riproducibilità dei test.

L'algoritmo non necessita di essere eseguito nella sua interezza sull'acceleratore, ma è sufficiente solo la parte di calcolo intensiva. In questo caso la parte di calcolo intensivo corrisponde all'assegnazione di ogni punto ad un cluster. Tale parte è eseguita ad ogni iterazione del ciclo.

2.1. Codice Host

Se parte dell'algoritmo è eseguita su device, il restante è eseguito lato host. Guardiamo quindi il codice lato host.

All'avvio del programma il dataset di punti su cui è svolto l'algoritmo è caricato su RAM. La funzione *allocPointDevMemory* si occupa poi di trasferire i punti nella global memory del device. I punti sono passati in memoria attraverso tre vettori, uno per ogni coordinata. I tre vettori avranno quindi dimensione pari al numero di punti. Questa modalità di allocazione permetterà al device di sfruttare i burst di memoria durante le letture eseguite dai thread per ottenere un punto del dataset.

Dopodiché, l'host alloca altri spazi della global memory:

- tre vettori di dimensione K cluster a cui passa i centroidi predefiniti;
- tre vettori di dimensione K cluster che servono per mantenere le coordinate dei nuovi centroidi calcolati;
- un vettore di dimensione K cluster per memorizzare il numero di punti assegnati ad ogni cluster.

Fatto ciò, il codice host entra nel ciclo per eseguire le N iterazioni fissate. Per ogni iterazione è chiamata la funzione globale *assignPointToCluster* che si occupa di assegnare ogni punto del dataset ad un cluster. Essendo una funzione globale, è eseguita sul device. L'host, con un **cudaDeviceSynchronize()**, attende che il device abbia concluso.

Assegnati tutti i punti, l'host recupera dal device i nuovi centroidi e il numero di punti assegnati ad ogni cluster.

Le coordinate dei nuovi centroidi sono in realtà un'accumulazione di tutte le corrispondenti coordinate dei punti appartenenti al cluster. L'host quindi calcola il reale valore delle coordinate dividendole per il numero di punti appartenenti al cluster.

Prima di concludere l'iterazione del ciclo, il codice host aggiorna la global memory del device: le coordinate dei centroidi sono aggiornate, mentre le componenti dei vettori per i nuovi centroidi da calcolare e del numero di punti assegnati ad ogni cluster sono poste a 0.

2.2. Codice Device

Sul device è svolta solo la parte di calcolo intensivo, che in questo caso è stata identificata nell'assegnazione dei punti ai cluster e nel calcolo dei nuovi centroidi. Il kernel *assignPointToCluster* si occupa di svolgere questi compiti. Il numero di thread attivi sul device deve essere almeno pari al numero di punti del dataset, perciò, fissato il numero di thread per blocco, il numero di blocchi varia in modo che ciò sia rispettato. Per ogni blocco è stata allocata una quantità di shared memory pari a 7 volte il numero di cluster:

- tre parti di memoria servono a mantenere le coordinate dei centroidi. Così facendo, il loro numero di letture

da memoria globale si riduce dal numero di thread al numero dei blocchi, aumentando le prestazioni;

- tre parti di memoria servono ad accumulare le varie coordinate dei punti che appartengono allo stesso cluster;
- una parte di memoria serve a mantenere il numero di punti assegnati ad ogni cluster.

Il lavoro svolto dal kernel è diviso in tre parti.

Nella prima, è inizializzata la shared memory di ogni blocco: sono prese da global memory le coordinate dei centroidi e sono poste a 0 tutte le restanti celle. Questa operazione è svolta da un solo thread per blocco, quello con identificativo locale pari a 0. I restanti thread attendono grazie ad un **__syncthreads()**.

Nella seconda parte, ogni thread si occupa di assegnare un punto al cluster con centro più vicino. Individuato il cluster, la shared memory è aggiornata accumulando nelle corrispondenti celle le coordinate e incrementando di 1 il conteggio dei punti del cluster. Questi aggiornamenti sono effettuati con degli **atomicAdd**, cosicché non si generino conflitti. I thread attivi sono in numero maggiore rispetto ai punti presenti in global memory; per questo, la seconda parte del kernel è eseguita dal thread solo se il suo identificativo globale è minore al numero di punti. Un **__syncthreads()** permette l'attesa del completamento delle operazioni da parte di tutti i thread del blocco.

La terza parte del kernel si occupa di accumulare i risultati parziali di tutte le shared memory nella global memory. Ciò che viene accumulato sono le accumulazioni per coordinata dei punti appartenenti allo stesso cluster e il numero di punti a questi assegnati. Anche queste operazioni sono eseguite con degli **atomicAdd**. Il trasferimento dalla shared memory alla global memory deve essere effettuato una volta per blocco, perciò la terza parte è eseguita solo dal thread con identificativo locale pari a 0.

L'attesa che tutti i thread del kernel abbiano finito è eseguita lato host.

3. Test e confronti

Ogni configurazione dei test è stata eseguita dieci volte cosicché il valore utilizzato nel confronto sia più vicino al caso medio.

Il vincolo di arresto dei test è uguale a dieci iterazioni. Tale valore è adeguato per ottenere una corretta divisione in cluster per tutte le configurazioni testate.

Tutti i dataset utilizzati sono composti da quattro blob con distribuzione gaussiana e centri diversi.

I dati relativi alla versione sequenziale e a quella parallela su CPU sono presi da un altro elaborato: **Valutazione effetti della parallelizzazione sull'algoritmo K-Means**.

L'obiettivo primario dei test è valutare se e quando conviene spostare il calcolo intensivo sul device esterno.

3.1. Test al variare del numero di punti

I test sono stati effettuati con le seguenti quantità di punti: 4k, 40k, 400k, 4M, 40M.

Sono state provate diverse configurazioni rispetto al numero di thread per blocco. Fra queste sono presenti anche i due casi limite:

- 32 thread per blocco: caso limite perché 32 è il numero di thread per warp;
- 1024 thread per blocco: caso limite perché 1024 è il numero massimo di thread per blocco.

Per questi test, il numero K di cluster è fissato a 4, pari al numero di blob da cui sono generati i dataset.

Dalla **Figura 1** si può notare che:

- per questa implementazione del K-Means, il numero di thread per blocco non influisce sulle prestazioni (neanche per i casi limite);
- per quantità ridotte di punti, i tempi di esecuzione dell'algoritmo sono simili. Ciò è dovuto al fatto che i tempi di trasferimento dei punti verso il device prevalgono rispetto ai tempi di esecuzione del kernel. Per quantità di dati più elevate prevale invece il tempo di esecuzione del kernel (**Figura 2**).

In **Figura 3** si nota invece che per quantità di punti non elevate lo speedup è minore a 1, quindi la versione sequenziale dell'algoritmo ha prestazioni migliori rispetto alla versione CUDA. Ciò è sempre dovuto all'elevato costo del trasferimento dati su device. Per quantità di dati più elevate invece prevale il guadagno in prestazioni. Tale guadagno supera anche il corrispettivo ottenuto parallelizzando su CPU. La parallelizzazione su CPU considerata utilizza 4 thread, numero ideale per la sua implementazione. Per la versione CUDA è stata scelta invece la configurazione con 256 thread per blocco.

3.2. Test al variare del numero di cluster

Per questi test è stato variato il numero K di cluster e fissato a 4M il numero di punti.

Dal grafico in **Figura 4** si nota che, all'aumentare del numero di cluster, il tempo di esecuzione aumenta linearmente. Ciò è dovuto ad un incremento dei confronti che ogni thread deve effettuare per assegnare un punto ad un cluster. Le configurazioni utilizzate per i test hanno 256 thread per blocco.

Osservando invece **Figura 5** si può concludere che per la versione CUDA lo speedup cresce all'aumentare del numero di cluster. Aver scelto una quantità consistente come numero di punti (4M), fa sì che già per 2 cluster lo speedup ottenuto con la GPU sia maggiore rispetto a quello ottenuto

parallelizzando con la CPU. Detto ciò, al crescere del numero di cluster aumenta anche la scarto fra gli speedup. Ciò porta a preferire l'utilizzo di un device per parallelizzare questo algoritmo.

4. Conclusioni

Considerando i test effettuati su questa implementazione dell'algoritmo K-Means, si può concludere che l'utilizzo di un device esterno alla CPU comporti un alto guadagno in prestazioni. Per ottenere effettivamente questo guadagno, la quantità di dati su cui lavora l'algoritmo deve essere elevata, altrimenti le prestazioni saranno peggiori rispetto al corrispettivo sequenziale. Rispettata questa condizione, parallelizzare con un device esterno fa registrare migliori prestazioni rispetto alla controparte sequenziale e a parallelizzare con la CPU.

Aumentare il numero di cluster aumenta ulteriormente il guadagno in prestazioni.

Per l'implementazione di questo algoritmo si nota inoltre che i due casi limite di thread per blocco non comportano perdite in prestazioni. Nel caso dei 32 thread per blocco, cioè nel caso in cui ogni blocco abbia un solo warp, ciò potrebbe essere dovuto alla totale esecuzione del warp senza interruzioni. Quindi avere un solo warp per blocco non incide sulle prestazioni. Nel caso dei 1024 thread per blocco invece significa che anche un possibile utilizzo non adeguato delle risorse di uno streaming multiprocessor non influisce sulle prestazioni.

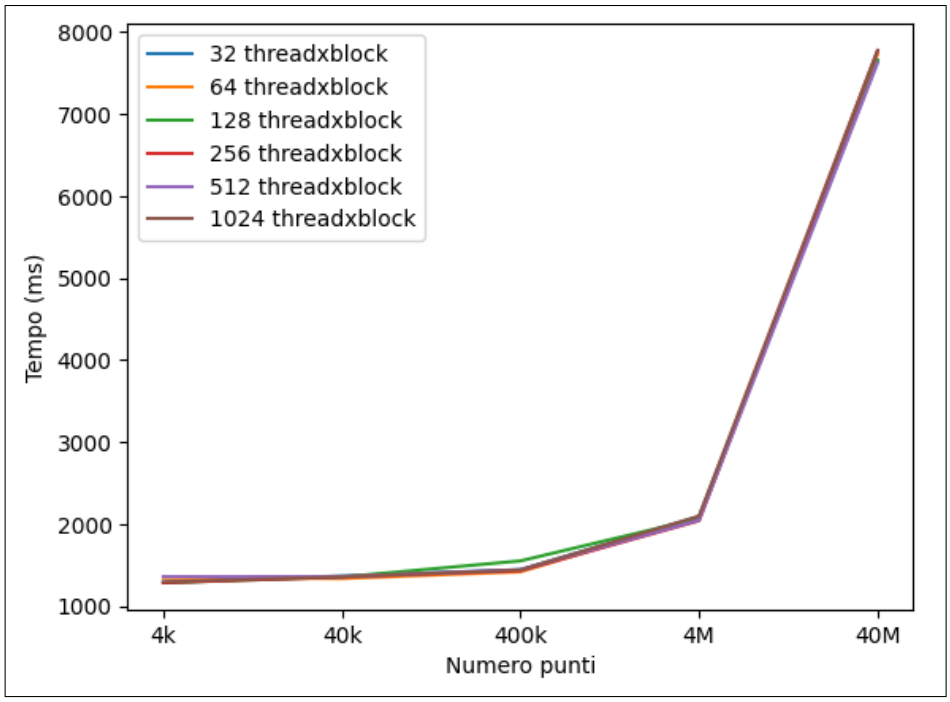

Figura 1: Tempo per eseguire l'algoritmo versione CUDA al variare del numero di punti. Il numero di cluster è fissato a 4.

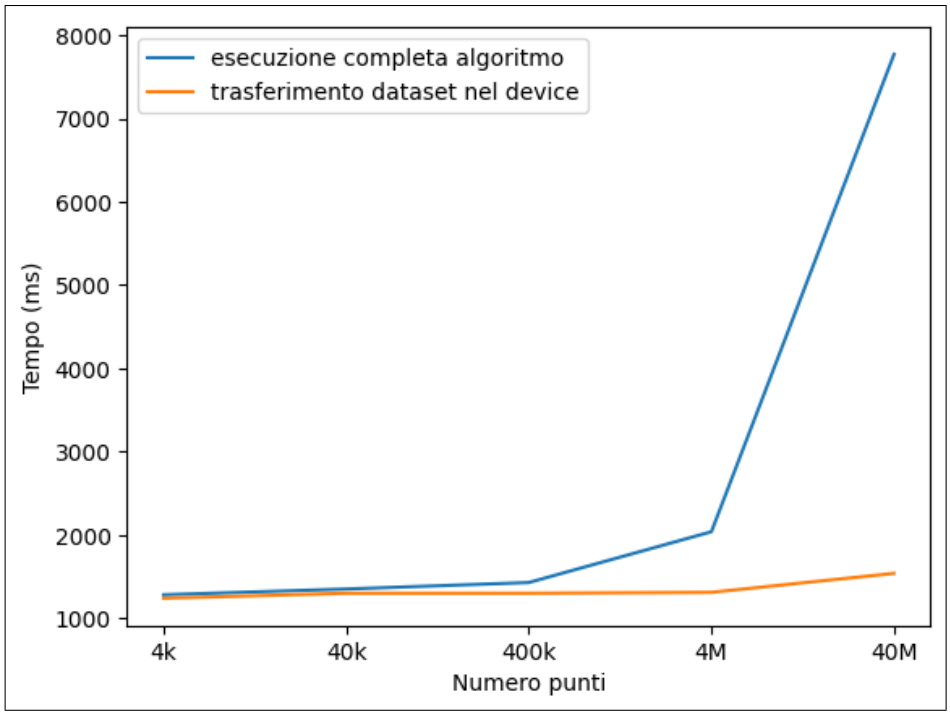

Figura 2: Confronto del tempo necessario per eseguire l'algoritmo rispetto a quello del solo trasferimento del dataset nel device. Il numero di thread per blocco è 256. Il numero di cluster è fissato a 4.

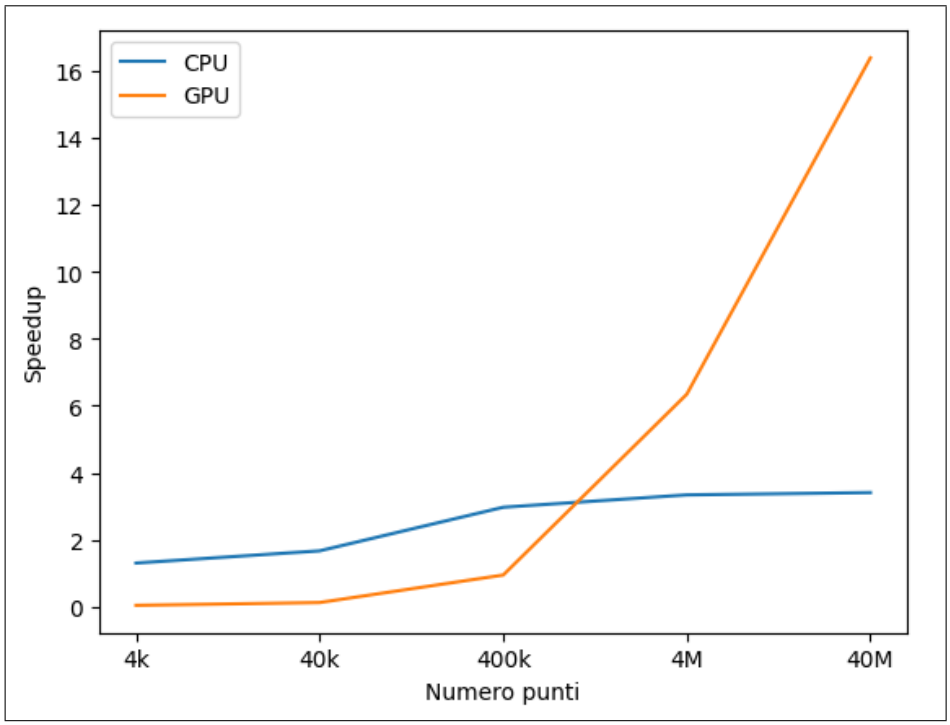

Figura 3: Confronto speedup CPU con 4 thread rispetto GPU con 256 thread per blocco al variare del numero di punti. Il numero di cluster è fissato a 4.

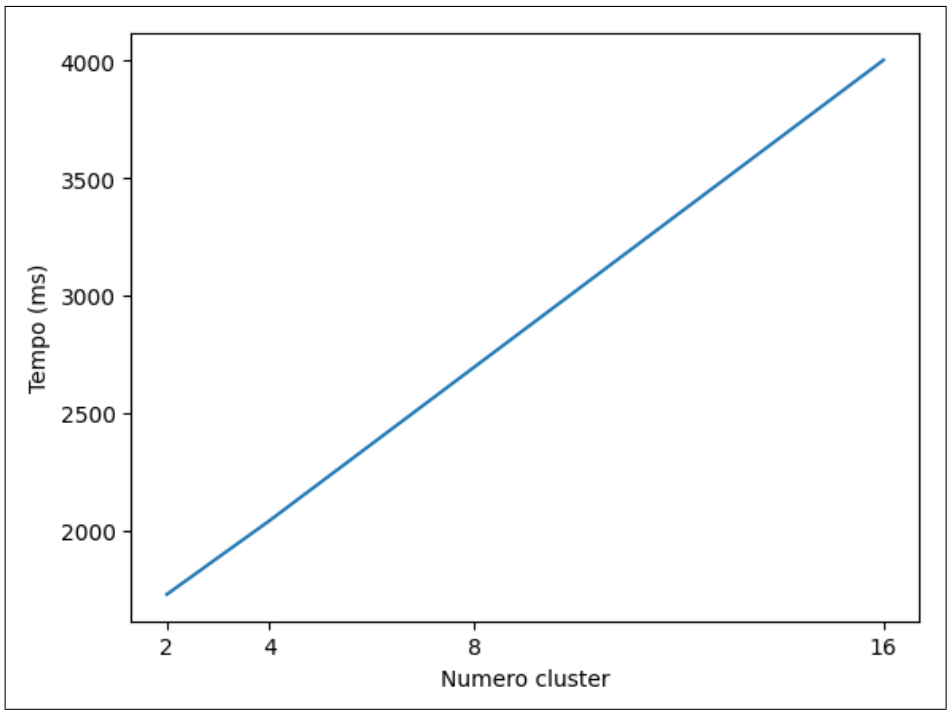

Figura 4: Tempo per eseguire l'algoritmo versione CUDA al variare del numero dei cluster. Il numero di punti è fissato a 4M. Il numero di thread per blocco è 256.

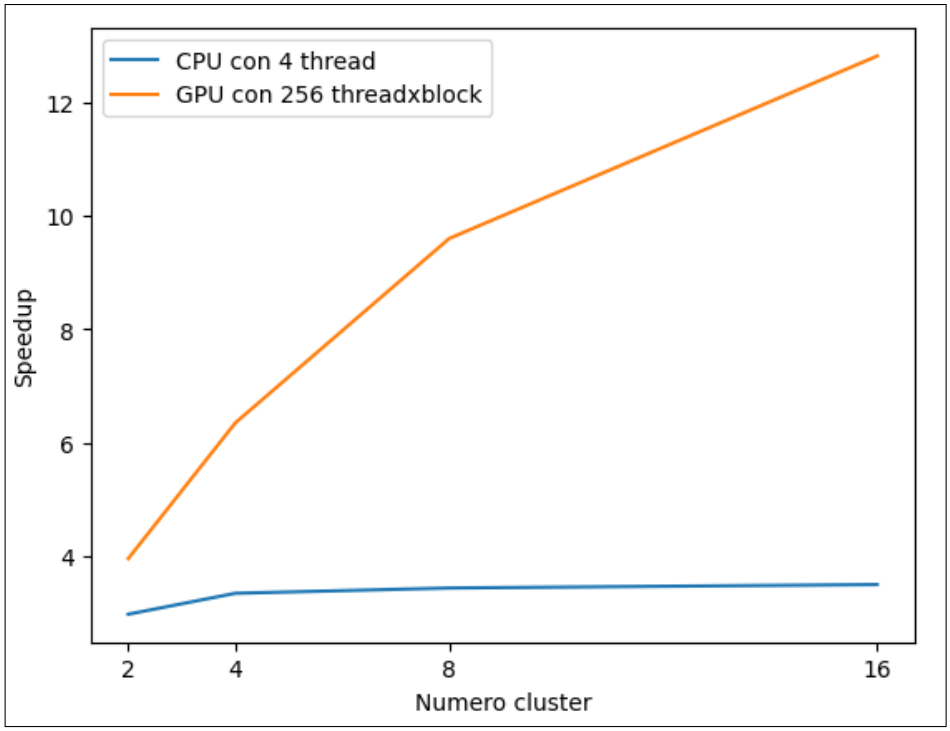

Figura 5: Confronto speedup CPU con 4 thread rispetto GPU con 256 thread per blocco al variare del numero dei cluster. Il numero di punti è fissato a 4M.