Progetto GPU Computing

Alessandro Di Liberti

July 2024

1 Introduzione

In questo progetto l'obiettivo è quello di scrivere un algoritmo che sfrutta la grande potenza computazionale di una gpu, per aumentare l'efficienza dell'algoritmo di clustering K-Means.

KMeans è un algoritmo di clustering utilizzato per partizionare un insieme di dati in k cluster distinti. Ogni punto dati viene assegnato al cluster con il centroide più vicino, determinato calcolando la distanza euclidea. L'algoritmo itera tra l'assegnazione dei punti ai cluster e l'aggiornamento dei centroidi dei cluster, calcolati come la media dei punti assegnati, fino a quando i centroidi non cambiano significativamente.

Il progetto si compone quindi di due parti: una prima parte che effettua la generazione del dataset, e la seconda che esegue k-means. Per ogni parte vi è una versione eseguita su cpu e un'altra che fa uso della gpu.

Oltre all'algoritmo di clustering viene effettuato un confronto di tempi per la generazione di un dataset contente delle cloudpoint, effettuata con un algoritmo su cpu e con uno su gpu.

1.1 Motivazione

La generazione del dataset ha richiesto una fase dedicata poichè si è cercato di creare un dataset contente dei cluster che possano poi essere da identificati k-means. Lavorando con un elevato numero di punti, anche in questa fase si è utilizzata la potenza di una gpu per la generazione, in particolare si è sfruttata la librearia cuRAND.

La scelta di cercare di parallelizzare k-means non è casuale: infatti un miglioramento di efficienza quando si utilizza una gpu per il calcolo, lo si ha quando l'algoritmo computato può essere parallelizzato, e quindi non richiede troppi passaggi eseguiti sequenzialmente. K-means, nella sua prima fase, esegue la ricerca del centroide più vicino per ogni punto del dataset. Questa operazione non richiede di essere eseguita sequenzialemente, e quindi può essere migliorata.

1.2 Descrizione progetto

Come detto precedentemente il progetto si compone di due fasi: una generazione del dataset, e l'algoritmo k-means.

Per ognuna delle due fasi è stata creata una versione sequenziale ed una parallela. Il comportamento della versione parallella cerca di essere il più simile possibile a quello della

versione sequenziale. In questo modo si è in grado di confrontare i tempi ed osservare se l'utilizzo di CUDA possa effettivamente portare ad un significativo miglioramento delle prestazioni.

Le due fasi sono implementate in due file *.cu* differenti: *dataGeneration.cu* e *cluster.cu*. I dati generati nella prima fase vengono scritti su un file *.csv* che viene letto da *cluster.cu* per caricare i dati in memoria.

2 Data

Essendo l'intero progetto basato su k-means si è utilizzata una struttura dati che potesse essere efficientemente utilizzata dall'algoritmo: ovvero un array di *Point*. *Point* è una struct composta da due valori float: *x,y*. La codifica dei punti in questo modo permette un'eventuale estensione ad una terza dimensione. Ovviamente essendo k-means un algoritmo che soffre di *curse of dimensionality* non è conveniente aumentare a più di tre dimensioni il datapoint; inoltre la formula della distanza che viene utilizzata è quella euclidea che diminuisce di efficacia all'aumentare di dimensioni.

3 Data Generation

La prima fase del progetto consiste nel generare un dataset che contenga dei cluster globulari, in modo da creare un setting ideale per k-means. Per fare questo si fa utilizzo di una variabile *roots*, che è implementata mediante un array di *Point*, e che rappresenta dei punti virtuali da cui generare i cluster. Per rendere più naturale possibile il cluster si è generato il cluster con una distribuzione normale all'interno di un raggio.

Per poter generare questo tipo di dati si utilizza una tecnica composta di 4 fasi:

1. **Scelta di una root**: viene randomicamente selezionata una root tra quelle generate in precedenza.
2. **Generazione di un angolo**: si genera un valore randomico e che segue una distribuzione uniforme in $[0, 2\pi]$ radianti.
3. **Generazione di una distanza**: si procede generando una distanza randomica e che segue una distribuzione normale a media nulla in $[-radius, +radius]$ (per il 99.7% dei punti). Per ottenere questo risultato si è diviso il valore del raggio per 3 in modo tale che esso rappresenti una deviazione standard della distribuzione. In questo modo, si garantisce che il 99.7% dei punti generati cadrà entro un cerchio di raggio pari al valore originario del raggio. Questo rende la distribuzione dei punti più concentrata e conforme alle proprietà della distribuzione normale, evitando che i punti si disperdano troppo lontano dal centro e mantenendo la maggior parte dei punti entro i limiti desiderati.
4. **Generazione del punto**: avendo cluster di origine, angolo e distanza si applica la formula per generare le coordinate del punto:
   - $x = root.x + distance * cos(\theta)$
   - $y = root.y + distance * sin(\theta)$

Al termine della generazione entrambi i dataset generati in modo sequenziale e parallelo vengono scritti su due file CSV: *points-CPU.csv* e *points-GPU.csv*.

3.1 Sequential version

Le versione sequenziale, è implementata mediante la trasformata di Box-Muller [1]. Questa operazione viene compiuta per generare dei valori normalmente distribuiti partendo da generazioni uniformi. In seguito alla generazioni di due valori uniformemente distribuiti $u1, u2$, la distanza viene calcolata con:

$$\text{distance} = \sqrt{-2 * \ln(u1)} \cos(2\pi * u2) \tag{1}$$

3.2 Parallel version

La versione parallela in CUDA sfrutta la libreria cuRAND. CuRAND offre la possibilità di generare dati randomici accessibili direttamente lato host, usando una semplice chiamata di funzione; oppure lato device. In questo progetto si è utilizzato l'approccio lato device, quindi si scrive un kernel che usa la libreria per generare un angolo e una distanza. In questo caso la generazione della distanza è semplificata poichè cuRAND offre direttamente un metodo *curand-normal()*.

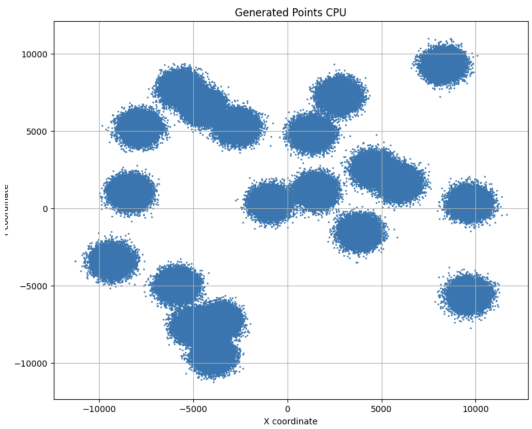
L'idea di popolare un array in memoria device nasce in previsione della fase di clustering. In questo modo infatti l'array *d-points* (che rappresenta l'array di punti caricato su memoria device) non richiede nessun trasferimento di memoria aggiuntivo.

Per la fase intermedia di visualizzazione, questo array viene trasferito lato host per essere scritto sul file csv, ma se si volesse considerare queste due parti come un unico file, la soluzione di generare i dati direttamente sul device risulta molto più efficiente.

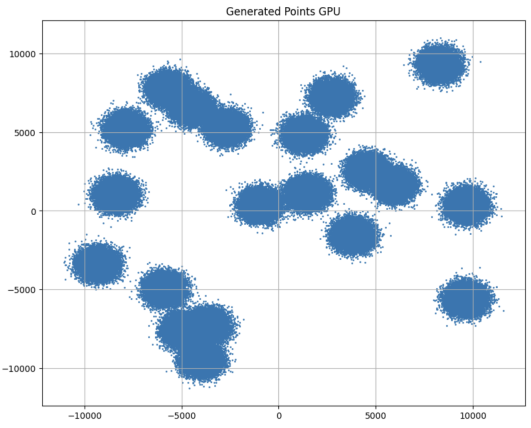
3.3 Visualizzazione

Di seguito la visualizzazione dei dataset generati con i seguenti parametri:

1. dimensione x e y del piano: 20000
2. numero di punti: 1 milione
3. numero di radici: 20
4. raggio dei cluster: 1400



Dataset generated with CPU



Dataset generated with GPU

Come è possibile osservare il risultato è lo stesso, ovviamente le coordinate dei punti non sono uguali, ma i cluster e la loro struttura è identica.

4 K-means

K-means è uno degli algoritmi di clustering più popolari e ampiamente utilizzati, sviluppato da James MacQueen nel 1967 [2], si basa su un approccio di partizionamento iterativo per suddividere un insieme di dati in k gruppi distinti, minimizzando la varianza all'interno di ciascun cluster.

Il processo inizia con l'assegnazione casuale di k centroidi, che vengono poi aggiornati iterativamente sulla base della media dei punti assegnati a ciascun cluster. L'algoritmo converge quando non vi sono più cambiamenti significativi nelle posizioni dei centroidi o nelle assegnazioni dei punti ai cluster.

Il vantaggio di questo algoritmo è la sua semplicità, ma presenta anche delle limitazioni, come la sensibilità ai valori iniziali dei centroidi e la difficoltà di determinare il numero ottimale di cluster.

L'algoritmo si compone delle seguenti fasi:

1. **Inizializzazione random**: dati k centroidi, l'algoritmo posiziona randomicamente nello spazio questi punti virtuali.

2. **Ricerca centroide più vicino**: ad punto del dataset viene assegnato il centroide più vicino, utilizzando come distanza quella euclidea.

3. **Aggiornamento centroidi**: per ogni cluster si calcola il punto dei medio dei punti che lo compongono e si posiziona il centroide di quel cluster in quella posizione appena calcolata

4. **Iterazione fino a convergenza**: si ripetono i passaggi 2 e 3 fino alla convergenza (minimo locale) dell'algoritmo.

In questa fase il primo passaggio è quello di leggere il dataset generato nella fase precedente e salvato su un file CSV.

Successivamente terminato il clustering, il dataset con le label per ogni punto viene nuovamente scritto su un altro CSV per poter essere stampato graficamente.

4.1 Sequential version

La versione sequenziale rispetta esattamente il comportamento generale dell'algoritmo, si utilizza l'array di punti, un array che rappresenta la posizione dei centroidi, un array temporaneo per il calcolo della nuova posizione e uno che rappresenta le label per ogni punto.

Lo stopping criteria è implementato mediante una variabile *changed* che tiene traccia se c'è stato un cambio di assegnazione di label, quando questo non accade per ogni punto, allora l'algoritmo ha raggiunto la convergenza.

4.2 Parallel version

Per la versione parallela il vero vantaggio lo si ha nella ricerca del centroide più vicino per ogni punto; mentre per quanto riguarda l'aggiornamento dei centroidi, essendo un'operazione svolta su k elementi (con k parametro relativamente piccolo) il vantaggio della parallizzazione è minimo.

Tuttavia un fattore da tenere a mente è che gli array descritti in precedenza risiedono su memoria device e il loro trasferimento su memoria host per poi essere nuovamente spostati su gpu per il passaggio successivo è costoso a livello di tempo.

La soluzione implementata è quindi la seguente: lato host si utilizza un loop regolato dalla flag *h-changed*, questa variabile simula il comportamento di *changed* della versione sequenziale, ma è utilizzata solo in seguito ad un trasferimento dal device.

All'interno del loop viene chiamato un kernel *kmeansSumPoints* in cui ogni thread lavora su un punto, ricerca il centroide più vicino e somma le coordinate di quel punto con un *atomicAdd()* in un array dei diversi cluster, in posizione del cluster di appartenenza. Inoltre un counter di punti di appartenza ai cluster viene incrementato sempre con un *atomicAdd*.

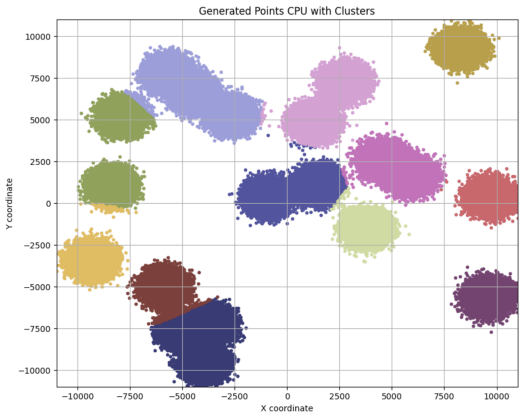
La seconda fase consiste di invocare k kernel per aggiornare la posizione dei centroidi, in questo modo i dati risiedendo già sul device non necessitano di essere trasferiti. L'update consiste in una divisione dell'array contente le somme delle coordinate e del counter di punti di ogni cluster.

La scelta di questa implementazione è dettata dal fatto che la convergenza dell'algoritmo (specie se con k relativamente alto) impiega un importante numero di iterazioni, e se si procedesse ad eseguire l'aggiornamento dei centroidi lato host, questo richiederebbe un alto numero di trasferimenti che comprometterebbero i tempi di esecuzione.

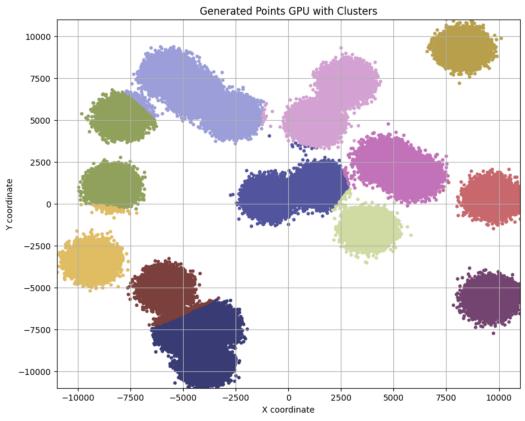
4.3 Visualization

Di seguito la visualizzazione dei cluster individuati da k-mean in entrambre le versioni

1. nCluster: 12



Clustering with CPU



Clustering with GPU

Anche in questo caso l'algoritmo parallelo si comporta allo stesso modo della sua controparte sequenziale.

5 Speedup e profiling

In questa sezione vengono mostrati i tempi e gli speedup dell'esecuzione con i parametri mostrati precedentemente. Per calcolare lo speedup si è utilizzata la formula:

$$\textbf{speedup} = \frac{\text{GPUTime}}{\text{CPUTime}} \tag{2}$$

Per queanto riguarda il profiling, invece, si è utilizzato ncu (Nsight compute CLI) per mostrare le statistiche sull'esecuzione.

Di seguito un elenco delle caratteristica della scheda:

Dispositivo:	Tesla T4
Versione del driver CUDA / Runtime:	12.2 / 12.2
Nome dell'architettura della GPU:	Turing
Versione principale/minore della capacità CUDA:	7.5
Quantità totale di memoria globale:	15102 MB (15835660288 bytes)
Multiprocessori (MP):	40
CUDA Cores per MP:	64
Totale CUDA Cores:	2560
Frequenza massima della GPU:	1590 MHz (1.59 GHz)
Frequenza della memoria:	5001 MHz
Larghezza del bus della memoria:	256-bit
Dimensione della cache L2:	4194304 bytes

Table 1: Caratteristiche tecniche della GPU Tesla T4

5.1 Data generation

In questa prima fase i tempi comprendono solo l'esecuzione dell'algoritmo per la generazione dei dati, escludendo i trasferimenti di memoria.

Con un kernel strutturato come:

- **blockSize**: 64.
- **numBlock**: *(nPoints + blockSize - 1) / blockSize* (15625 in questo caso).

I tempi di esecuzione sono stati:

	CPU	GPU
time [s]	**0.1147**	**0.0416**
speedup	/	**2.8**

Nella generazione pura l'algoritmo parallelizzato con CUDA risulta più veloce rispetto a quello sequenziale.

5.1.1 Profiling

GPU Speed Of Light Throughput		
Nome della metrica	Unità di misura	Valore della metrica
DRAM Frequency	cycle/nsecond	4.99
SM Frequency	cycle/usecond	584.35
Elapsed Cycles	cycle	23,861,514
Memory Throughput	%	97.09
DRAM Throughput	%	0.37
Duration	msecond	40.83
L1/TEX Cache Throughput	%	97.68
L2 Cache Throughput	%	1.37
SM Active Cycles	cycle	23,715,349
Compute (SM) Throughput	%	97.09

Table 2: GPU Speed Of Light Throughput

Il throughput di memoria è del 97.09%, e il throughput di calcolo degli SM è altrettanto alto, indicando un uso ottimale delle capacità di calcolo e di memoria.

Occupancy		
Nome della metrica	Unità di misura	Valore della metrica
Block Limit SM	block	16
Block Limit Registers	block	16
Block Limit Shared Mem	block	16
Block Limit Warps	block	16
Theoretical Active Warps per SM	warp	32
Theoretical Occupancy	%	100
Achieved Occupancy	%	90.81
Achieved Active Warps Per SM	warp	29.06

Table 3: Occupancy

In questa tabella è possibile osserva che l'occupancy raggiunta è del 90%, suggerendo che la GPU è sfruttata molto vicino al suo potenziale massimo.

5.2 K-means

In questa seconda fase la misurarazione delle performance e il profiling risultano più complessi visto che l'algoritmo è implementato da un loop lato host che richiama più volte i kernel. Inoltre solo il primo kernel è utilizzato in modo da essere efficiente, il secondo esiste solo per evitare dei trasferimenti aggiuntivi.

La misurazione dei tempi e il calcolo dello speedup sono meno rilevanti rispetto alla prima fase poichè dipendono molto da come vengono inizializzati i centroidi: se la posizione è favorevole al clustering la versione CPU impiegerà poco tempo, altrimenti può richiedere molte operazioni prima di convergere.

Un'osservazione importante emersa durante i test è che il tempo richiesto dalla gpu è sempre nell'intorno di centesimi di secondo, indipedendentemente dall'inizializzazione.

La struttura di *kmeansSumPoint()* è la seguente:

- **blockSize**: 64.
- **numBlock**: *(nPoints + blockSize - 1) / blockSize* (15625 in questo caso).

	CPU	GPU (no memory transfer)	GPU (with memory transfer)
time [s]	**1.2941**	**0.0173**	**0.0218**
speedup	/	**74.8**	**59.2**

5.2.1 Profiling

Come precedentemente accennato la parte di profiling risulta meno significativa. Le statistiche prese in considerazione sono quelle di *kmeansSumPoint*, ad una iterazione. Il numero di chiamate dipende da quante iterazioni richiede l'algoritmo per convergere.

GPU Speed Of Light Throughput		
Nome della metrica	Unità di misura	Valore della metrica
DRAM Frequency	cycle/nsecond	5.00
SM Frequency	cycle/usecond	584.98
Elapsed Cycles	cycle	1,374,987
Memory Throughput	%	3.97
DRAM Throughput	%	2.28
Duration	msecond	2.35
L1/TEX Cache Throughput	%	7.28
L2 Cache Throughput	%	3.12
SM Active Cycles	cycle	1,368,070.55
Compute (SM) Throughput	%	4.38

Table 4: GPU Speed Of Light Throughput

I motivi del basso valore di throughput sono probabilmente legati alle operazioni di *atomicAdd()* necessarie per calcolare successivamente la posizione aggiornata dei centroidi. Tuttavia questo non compromette eccessivamente i tempi di esecuzione.

Occupancy		
Nome della metrica	Unità di misura	Valore della metrica
Block Limit SM	block	16
Block Limit Registers	block	32
Block Limit Shared Mem	block	16
Block Limit Warps	block	16
Theoretical Active Warps per SM	warp	32
Theoretical Occupancy	%	100
Achieved Occupancy	%	89.99
Achieved Active Warps Per SM	warp	28.80

Table 5: Occupancy

Per quanto riguarda l'occupancy i valori ottenuti sono elevati e in linea con quelli raggiunti con il precedente kernel.

Queste statistiche sono simili per ogni invocazione del kernel.

6 Conclusioni

L'obiettivo del progetto era stabilire se fosse conveniente un'implemntazione parallelizzata dell'algoritmo di k-means, includendo anche una generazione di un dataset con delle caratteristiche precise.

I risultati mostrano che con un alto numero di dati la soluzione abbatte notevolmente i tempi d'esecuzione, specie per la parte di k-means.

Sono tuttavia emerse delle problematiche legate alla natura dell'algoritmo:

- **Risultato dipendente da inizializzazione**
- **Numero di k cluster non noto a priori**

Il numero di k cluster in questo progetto è stato mitigato offrendo uno step intermedio che mostra la visualizzazione del dataset e permette di scegliere un valore abbastanza ideale.

Per quanto riguarda il problema principale, ovvero l'inizializzazione, nell'implementazione semplice di k-means non è possibile risolvere il problema senza cambiare l'implementazione o usare versioni migliori dell'algoritmo (es: *k-means++*).

E' tuttavia possibile sfruttare una soluzione che in approccio sequenziale risulterebbe troppo lenta, ma che se eseguita su GPU potrebbe essere vantaggiosa: ovvero partire da un k relativamente basso, ottenere una prima soluzione ed eseguire nuovamente l'algoritmo solo sui cluster che non soddisfanno le condizioni richieste. In questo modo si esegue molte più volte l'algoritmo, ma visto che i tempi sono molto bassi, si potrebbe idealmente ottenere un ottimo risultato in tempi accettabili.

Questa soluzione fa leva sul fatto che i tempi della GPU non "esplodono" in caso di cattiva inizializzazione, a differenza di quanto accade su CPU.

References

[1] G. E. P. Box and Mervin E. Muller. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics*, 29(2):610 – 611, 1958.

[2] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.