UNIVERSITÀ DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica

Kmeans, Comparing Sequential and Parallel version

Docente:

Marco Bertini

Candidato:

Daniele Morganti

ANNO ACCADEMICO 2024/2025

Indice

1 Introduzione

1.1 Contesto e obbiettivi

L'analisi di grandi quantità di dati richiede algoritmi efficienti sia in termini di tempo che di risorse computazionali. Tra questi, l'algoritmo di clustering k-means si distingue come uno dei più utilizzati per la sua semplicità e versatilità in vari ambiti, come la visione artificiale, l'elaborazione del linguaggio naturale e la segmentazione delle immagini. Tuttavia, l'elevato costo computazionale dell'algoritmo, soprattutto per dataset di grandi dimensioni, ha portato alla necessità di ottimizzarne l'esecuzione, sfruttando le potenzialità della computazione parallela.

Questo progetto si propone di confrontare l'implementazione sequenziale dell'algoritmo k-means in C++ con una versione parallela sviluppata utilizzando CUDA, un'architettura di calcolo parallelo progettata per le GPU NVIDIA. Attraverso questo confronto, analizzeremo le differenze in termini di prestazioni, velocità e scalabilità tra le due implementazioni, evidenziando come l'elaborazione parallela possa accelerare il processo di clustering e migliorare le performance in contesti ad alto volume di dati.

L'obiettivo di questo progetto è quindi duplice: da un lato, comprendere i benefici e i limiti dell'implementazione parallela rispetto a quella sequenziale e, dall'altro, esplorare le tecniche di ottimizzazione offerte dalla piattaforma CUDA per massimizzare l'efficienza dell'algoritmo.

1.2 Approccio Generale

L'obiettivo principale di questo progetto è il confronto tra le implementazioni sequenziale e parallela dell'algoritmo k-means, in termini di performance e scalabilità. Per realizzare questo confronto, abbiamo seguito un approccio che parte dallo studio dell'algoritmo k-means, ne analizza le caratteristiche computazionali, e poi si concentra sull'ottimizzazione della sua esecuzione attraverso l'uso della programmazione parallela su GPU con CUDA.

Nell'implementazione sequenziale in C++, l'algoritmo è stato sviluppato seguendo una struttura tradizionale che itera sull'intero dataset per ciascuna delle due fasi principali. Questa implementazione è stata realizzata in modo da essere il più possibile ottimizzata, senza però introdurre parallelismi, per ottenere un riferimento di base con cui confrontare le performance.

Per l'implementazione parallela, abbiamo sfruttato CUDA per distribuire i calcoli su una GPU NVIDIA RTX A2000. Le GPU, grazie alla loro architettura basata su migliaia di core, sono particolarmente adatte per eseguire operazioni ripetitive in parallelo, come quelle richieste nell'algoritmo k-means. Le principali modifiche apportate all'implementazione sequenziale riguardano la distribuzione dei compiti tra i thread della GPU e l'ottimizzazione dell'accesso alla memoria.

2 Implementazione

2.1 Versione sequenziale

L'implementazione sequenziale di kmeans in C++ è un ottimo punto di partenza per comprendere le basi dell'algoritmo e le sue operazioni fondamentali. In questo capitolo, analizzeremo la struttura del codice fornito e il funzionamento dell'algoritmo.

L'implementazione sequenziale dell'algoritmo K-means è contenuta nella funzione kmeans, che accetta come input un vettore di centroidi, un vettore di punti, il numero di cluster k e un numero massimo di iterazioni. La funzione restituisce un vettore di etichette che indicano a quale cluster appartiene ciascun punto.

La funzione inizia definendo un vettore labels, inizialmente vuoto, che verrà utilizzato per memorizzare l'etichetta di cluster per ciascun punto. Dopo l'inizializzazione l'algoritmo esegue un ciclo che continua fino a un numero massimo di iterazioni specificato (di default 100). Ogni iterazione crea un vettore di vettori clusters, dove ogni sotto-vettore rappresenta i punti assegnati a un centroide specifico. Questo è essenziale per raggruppare i punti in base alla loro vicinanza ai centroidi. Per ogni punto nel vettore points, l'algoritmo chiama la funzione findClosestCentroid per determinare il centroide più vicino. Il punto viene quindi aggiunto al cluster corrispondente e l'etichetta viene aggiornata nel vettore labels. Una volta completata l'assegnazione, la funzione computeNewCentroids calcola i nuovi centroidi sulla base dei pun-

ti assegnati a ciascun cluster. Questo passaggio è cruciale, poiché i centroidi devono essere aggiornati per riflettere la nuova configurazione dei punti.

Se i nuovi centroidi calcolati sono identici ai centroidi precedenti, l'algoritmo si interrompe, poiché non ci sono ulteriori miglioramenti nell'assegnazione dei punti. Alla fine dell'algoritmo, la funzione restituisce il vettore labels, che fornisce l'assegnazione finale di ogni punto a un cluster.

Complessità Computazionale L'implementazione sequenziale è semplice e chiara, ma ha alcune limitazioni in termini di efficienza, specialmente quando il numero di punti o di dimensioni dei dati cresce. Ogni iterazione comporta un confronto tra ogni punto e tutti i centroidi, il che porta a una complessità computazionale di O(n * k * d), dove n è il numero di punti, k è il numero di cluster e d è la dimensione dei dati. Questo può risultare inefficiente con dataset molto grandi.

```
vector<int> kmeans(vector<Point>& centroids, const vector<Point>& points, int
    k, int maxIterations = 100) {
    vector<int> labels(points.size());

    for (int iteration = 0; iteration < maxIterations; ++iteration) {
        vector<vector<Point>> clusters(k);

        for (int i = 0; i < points.size(); ++i) {
            int closestIdx = findClosestCentroid(points[i], centroids);
            clusters[closestIdx].push_back(points[i]);
            labels[i] = closestIdx;
        }

        vector<Point> newCentroids = computeNewCentroids(clusters);

        if (newCentroids == centroids) break;

        centroids = newCentroids;
    }

    return labels;
}
```

L'implementazione sequenziale dell'algoritmo K-means in C++ rappresenta una base solida per comprendere il funzionamento di questo metodo di clustering. Sebbene sia facile da seguire e implementare, per affrontare dataset di grandi dimensioni è utile esplorare varianti più avan-

zate. Nle priossimo capitolo analizzeremo l'implementazione parallela in CUDA, che possono sfruttare le architetture hardware moderne per migliorare notevolmente le prestazioni.

2.2 Versione parallela

L'implementazione parallela dell'algoritmo K-means in CUDA rappresenta un approccio efficace per ottimizzare le prestazioni rispetto alla versione sequenziale in C++. Questa sezione esamina come il codice utilizza la parallelizzazione per gestire l'assegnazione dei punti ai cluster e il ricalcolo dei centroidi, sfruttando l'architettura delle GPU.

La funzione kmeans accetta come argomenti il numero di punti N, il numero di cluster K, e il numero di thread per blocco tpb. All'interno della funzione, vengono allocate le memorie necessarie sia sulla GPU che sulla CPU, e vengono inizializzati i centroidi e i punti.

Allocazione della Memoria Viene allocata memoria per i punti (d_datapoints), per le assegnazioni dei cluster (d_clust_assn), per i centroidi (d_centroids) e per le dimensioni dei cluster (d_clust_sizes). Queste allocazioni sono fondamentali per il calcolo parallelo, consentendo l'accesso rapido ai dati necessari.

Viene inoltre allocata memoria anche per i dati e le variabili di controllo sulla CPU, inclusi i centroidi e le assegnazioni dei cluster.

Inizializzazione dei Centroidi e dei Punti I centroidi e i punti vengono inizializzati con valori casuali. Questo è un passo importante, poiché i centroidi iniziali possono influenzare notevolmente il risultato finale dell'algoritmo K-means. Le dimensioni dei cluster vengono inizializzate a zero. Utilizzando cudaMemcpy, i dati vengono copiati dalla memoria della CPU a quella della GPU, preparandoli per l'elaborazione parallela.

Ciclo Principale dell'Algoritmo L'algoritmo opera all'interno di un ciclo che si ripete fino a raggiungere un numero massimo di iterazioni (definito da MAX_ITER). All'interno di questo ciclo, vengono eseguiti diversi kernel CUDA:

- Assegnazione dei Cluster: Il kernel kMeansClusterAssignment assegna a ciascun punto il cluster più vicino, utilizzando la distanza ai centroidi. Ogni thread elabora un punto, riducendo significativamente il tempo rispetto all'implementazione sequenziale.

- Reset dei Centroidi: Il kernel resetCentroids azzera i valori delle dimensioni dei cluster, preparando il sistema per il ricalcolo dei centroidi nella prossima iterazione.
- Accumulazione dei Centroidi: Il kernel accumulateCentroid calcola i nuovi centroidi sommando i punti appartenenti a ciascun cluster e aggiornando le dimensioni dei cluster. Anche in questo caso, il parallelismo consente di elaborare più punti contemporaneamente.
- Finalizzazione dei Centroidi: Infine, il kernel finalizeCentroids completa il ricalcolo dei centroidi sulla base delle somme accumulate e delle dimensioni dei cluster.

Sincronizzazione e Pulizia Dopo il completamento del ciclo, il codice attende il termine di tutte le operazioni sulla GPU tramite cudaDeviceSynchronize(), assicurando che tutte le assegnazioni e i calcoli siano stati completati. Viene quindi misurato il tempo di esecuzione e le risorse allocate vengono liberate sia sulla GPU che sulla CPU.

```
float kmeans(int N, int K, int tpb){

   Point *d_datapoints;
   int *d_clust_assn = 0;
   Point *d_centroids;
   int *d_clust_sizes=0;

   cudaMalloc(&d_datapoints, N*sizeof(Point));
   cudaMalloc(&d_clust_assn,N*sizeof(int));
   cudaMalloc(&d_centroids,K*sizeof(Point));
   cudaMalloc(&d_clust_sizes,K*sizeof(int));

   Point *h_centroids = (Point*)malloc(K*sizeof(Point));
   Point *h_datapoints = (Point*)malloc(N*sizeof(Point));
   int *h_clust_assn = (int*)malloc(N*sizeof(int));
   int *h_clust_sizes = (int*)malloc(K*sizeof(int));

   srand(time(0));

   for(int c=0;c<K;++c)
   {
      h_centroids[c].x = rand() % 1000;
      h_centroids[c].y = rand() % 1000;
      h_clust_sizes[c]=0;
   }

   for(int d = 0; d < N; ++d)
   {
      h_datapoints[d].x = rand() % 1000;
```

```
    h_datapoints[d].y = rand() % 1000;
  }

    cudaMemcpy(d_centroids,h_centroids,K*sizeof(Point),cudaMemcpyHostToDevice);
     cudaMemcpy(d_datapoints,h_datapoints,N*sizeof(Point),cudaMemcpyHostToDevice);
     cudaMemcpy(d_clust_sizes,h_clust_sizes,K*sizeof(int),cudaMemcpyHostToDevice);

  int cur_iter = 0;

  auto start = std::chrono::high_resolution_clock::now();

  while(cur_iter < MAX_ITER)
  {
    kMeansClusterAssignment<<<(N+tpb-1)/tpb,tpb>>>(d_datapoints,d_clust_assn,d_centroids
        N, K);

    resetCentroids<<<(N + tpb - 1)/tpb, tpb>>>(d_centroids, d_clust_sizes,
        K);
    accumulateCentroid<<<(N + tpb - 1)/tpb,
        tpb>>>(d_datapoints,d_clust_assn,d_centroids,d_clust_sizes, N, K);
    finalizeCentroids<<<(K + tpb - 1)/tpb, tpb>>>(d_centroids,
        d_clust_sizes, K);

    cur_iter+=1;
  }

  cudaDeviceSynchronize();
  auto end = std::chrono::high_resolution_clock::now();
  auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end -
     start);
  std::cout << "Tempo di esecuzione: " << ((float)duration.count())/1000 << "
     millisecondi" << std::endl;

  cudaFree(d_datapoints);
  cudaFree(d_clust_assn);
  cudaFree(d_centroids);
  cudaFree(d_clust_sizes);

  free(h_centroids);
  free(h_datapoints);
  free(h_clust_sizes);

  return ((float)duration.count())/1000;
}
```

3 Risultati

In questo capitolo vengono presentati e analizzati i risultati sperimentali ottenuti eseguendo sia la versione sequenziale che la versione parallela dell'algoritmo Kmeans. L'obiettivo è valutare le prestazioni delle due versioni in termini di tempo di esecuzione, efficienza e scalabilità.

Gli esperimenti sequenziali di seguito sono stati eseguiti su un processore Intel® Core™ i7-12700K con 12 core, quelli paralleli su una scheda NVIDIA RTX A2000 12 GB.

Obiettivi dell'Analisi L'analisi dei risultati sperimentali si propone di rispondere a diverse domande:

- Qual è il tempo di esecuzione dell'implementazione sequenziale rispetto a quella parallela?
- L'implementazione parallela scala meglio con l'aumento del numero di punti (N) o cluster (K)?
- Qual è l'utilizzo delle risorse (memoria e GPU) nelle due versioni?
- Come influiscono parametri come il numero di thread per blocco sulla prestazione dell'implementazione parallela?

Metodologia dei Test I test sono stati eseguiti utilizzando un dataset generato randomicamente prima del test, variando il numero di punti e il numero di cluster per valutare come le due versioni si comportano al crescere del carico computazionale. Inoltre, per la versione parallela, sono stati eseguiti test variando il numero di thread per blocco da 32 fino al numero massimo di 1024, per esaminare l'impatto di varie configuarazione dei grid sul tempo di esecuzione.

Metriche di Valutazione Le principali metriche utilizzate per il confronto tra le due versioni sono le seguenti:

Tempo di esecuzione: tempo necessario per completare l'algoritmo, misurato in millisecondi. Questa metrica permette di osservare la riduzione di tempo ottenuta dalla parallelizzazione. Speedup: definito come il rapporto tra il tempo di esecuzione della versione seriale e il tempo di esecuzione della versione parallela con n thread. Lo speedup ideale è direttamente proporzionale al numero di thread, ma in pratica è influenzato dall'overhead di sincronizzazione.

3.1 Primo Esperimento

Nel primo esperimento effettuato il codice è stato eseguito più volte incrementando ogni volta la lunghezza della lista di punti, partendo da una lunghezza di 50'000 campioni, raddoppiando fino ad arrivare a 25'600'000 campioni. In questo esperimento il numero di clusters è stato mantenuto costante a 10, stessa cosa per il numero di threads per blocco, mantenuto a 512 (che successivamente vedremo essere la configurazione migliore.

Dal grafico in figura 3.1 è possibile osservare come l'andamento del tempo di esecuzione in entrambi i casi sia lineare con il numero di punti in ingresso (coerente con l'analisi computazionale effettuata nel capitolo precedente) e di come lo speedup rimanga piuttosto costante entro un certo range di valori.

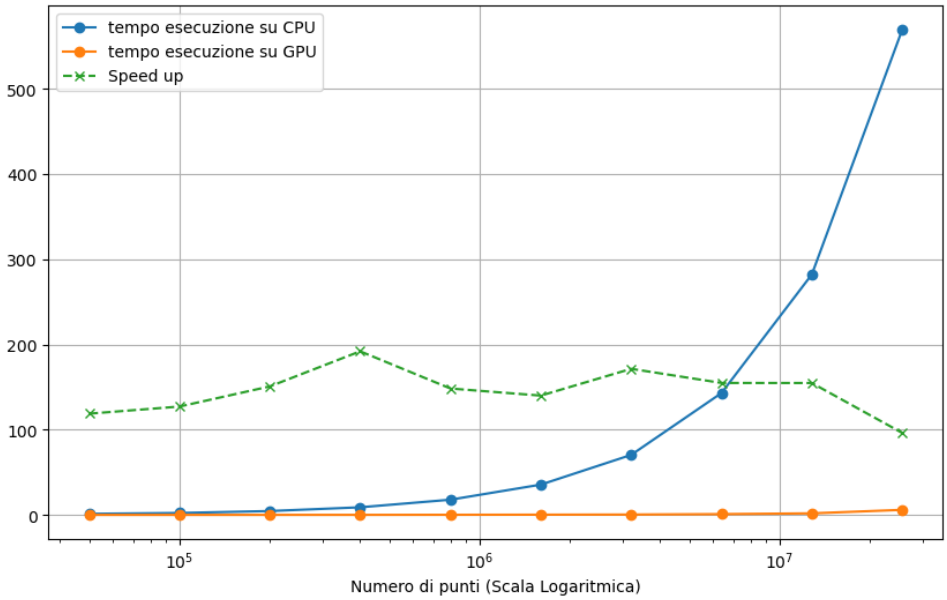

Figura 3.1: Tempo di esecuzione in funzione del numero di punti

N	C	TPB	TimePar (s)	TimeSeq (s)
50000	10	512	0.010676	1.26854
100000	10	512	0.0177	2.25095
200000	10	512	0.030052	4.53194
400000	10	512	0.046483	8.92533
800000	10	512	0.120401	17.861
1600000	10	512	0.253927	35.5569
3200000	10	512	0.410767	70.4271
6400000	10	512	0.923785	143.066
12800000	10	512	1.82352	282.278
25600000	10	512	5.9163	569.381

Tabella 3.1: Tempo di esecuzione in funzione del numero di punti

Possiamo quindi dedurre da questo primo esperimento che il numero di punti non influisca in modo rilevante sullo speedup della parallelizzazione. Tuttavia possiamo osservare un notevole aumento delle prestazioni dell'algoritmo.

3.2 Secondo Esperimento

In questo secondo esperimento sono state testate le due versioni dell'algoritmo aumentando progressivamente il numero di cluster, andando ad osservare come si comporta lo speedup aumentando il carico computazionale complessivo.

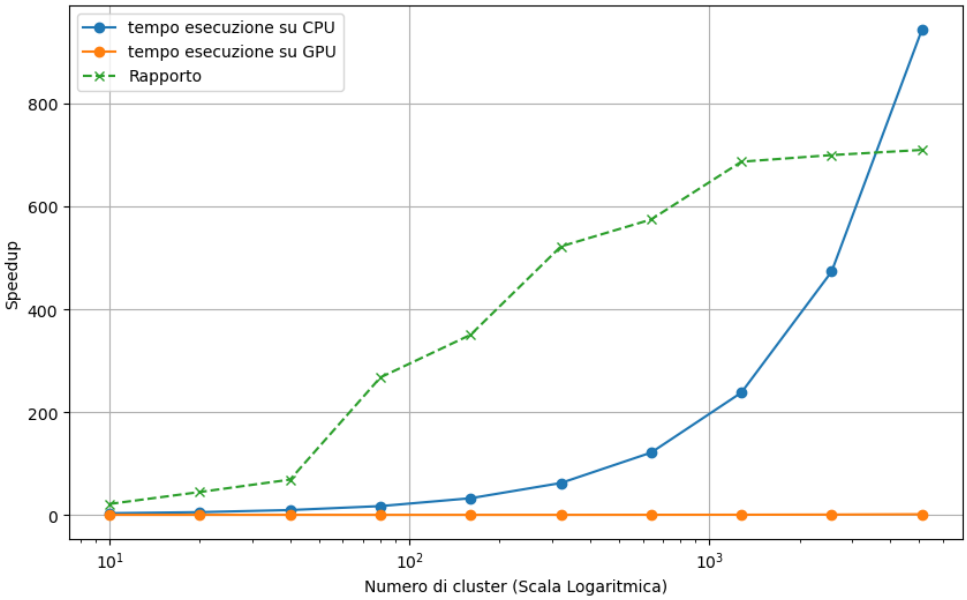

Figura 3.2: Tempo di esecuzione e speedup in funzione del numero di cluster a parità di numero di punti

N	C	TPB	TimePar	Time Seq
500000	10	512	0.154257	3.29514
500000	20	512	0.120754	5.38575
500000	40	512	0.138204	9.49251
500000	80	512	0.064153	17.1599
500000	160	512	0.92733	32.4988
500000	320	512	0.118892	62.0931
500000	640	512	0.211003	121.273
500000	1280	512	0.346312	237.965
500000	2560	512	0.676217	473.387
500000	5120	512	1.32911	943.759

Tabella 3.2: Tempo di esecuzione delle due versioni in funzione del numero di cluster

Come si può vedere in figura 3.2 abbiamo un significativo incremento dello speedup all'aumentare dei cluster.

3.3 Terzo esperimento

Nel terzo esperimento, è stato aumentato progressivamente il numero di thread per blocco (tpb) mantenendo costante il numero di punti (N) e il numero di cluster (K), è possibile osservare vari effetti sulle prestazioni dell'implementazione parallela dell'algoritmo K-means.

Dai risultati è possibile osservare che se tpb è troppo basso, la GPU potrebbe non essere completamente utilizzata, portando a tempi di esecuzione più lunghi. Questo perchè una GPU ottimizza l'occupancy distribuendo i blocchi di thread su diversi SM, cercando di mantenere i core attivi il più possibile. Tuttavia, se il numero di thread per blocco è troppo basso, la GPU potrebbe non riuscire a raggiungere un'occupancy elevata. Una bassa occupancy significa meno istruzioni in esecuzione in parallelo, causando un peggioramento delle prestazioni.

Un altro motivo è che ogni blocco richiede un certo overhead per la gestione e la sincronizzazione. Con pochi thread per blocco, il numero totale di blocchi necessari per coprire il dataset aumenta, incrementando questo overhead amministrativo. Questo impone un carico aggiuntivo alla GPU per il coordinamento dei blocchi, riducendo le prestazioni complessive.

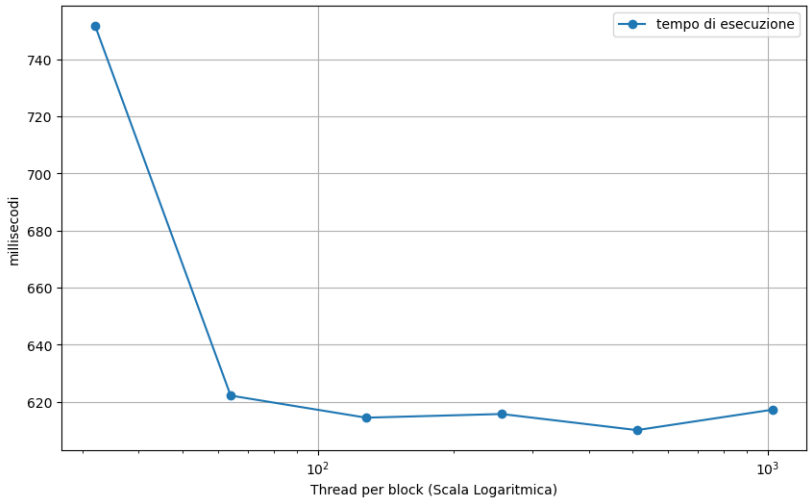

Figura 3.3: Tempo di esecuzione in funzione del numero di thread per blocco

numPoints	numClusters	tpb	time (ms)
5000000	100	32	736
5000000	100	64	621
5000000	100	128	617
5000000	100	256	618
5000000	100	512	610
5000000	100	1024	619

Tabella 3.3: Tempo di esecuzione in funzione del numero di thread per blocco

Dai grafici è possibile vedere come la configurazione peggiore è con 32 thread per blocco, mentra la migliore è ottenuta a 512 thread per blocco, ciò è in linea con le osservazioni fatte in precedenza.

4 Conclusioni

L'implementazione parallela dell'algoritmo K-means offre notevoli vantaggi rispetto all'approccio sequenziale, specialmente quando il numero di punti e cluster aumenta. Utilizzando i thread CUDA, l'algoritmo è in grado di gestire grandi volumi di dati in parallelo, riducendo significativamente il tempo di esecuzione. Tuttavia, è importante considerare anche il tempo necessario per la copia dei dati tra la CPU e la GPU, che può influenzare le prestazioni complessive.

L'implementazione parallela dell'algoritmo K-means in CUDA evidenzia come la parallelizzazione possa migliorare drasticamente le prestazioni di algoritmi di clustering su grandi dataset. Questa versione sfrutta le potenzialità delle GPU per elaborare simultaneamente più dati, rendendo l'algoritmo molto più efficiente rispetto alla sua controparte sequenziale. Le ottimizzazioni nella gestione della memoria e nei calcoli paralleli sono essenziali per ottenere risultati rapidi e scalabili.