Università degli Studi di Firenze

Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

---

Corso di Laurea Magistrale in Ingegneria Informatica
Parallel Computing - Professor Marco Bertini

Mean Shift Clustering Algorithm con OpenMP e CUDA

Lorenzo Meoni
Alessio Corsinovi

Anno Accademico 2022/2023

Contents

Introduzione **i**

1 Mean shift **1**

2 Algoritmi implementati e test **3**

2.1 Algoritmi implementati 3

2.1.1 Versione sequenziale 3

2.1.2 Versione parallela con OpenMP 5

2.1.3 Versione parallela con CUDA 7

2.2 Risultati dei test 10

2.2.1 OpenMP 10

2.2.2 CUDA 15

3 Conclusioni **17**

Bibliografia **18**

Introduzione

Il Mean-Shift Clustering è un algoritmo di raggruppamento a scorrimento di finestra di lettura (sliding-window-based), basato sui centroidi (centroid-based) con un costo computazionale $O(n^2)$. La sua struttura lo rende adatto al calcolo parallelo essendo il problema stesso imbarazzantemente parallelo. In questo elaborato è stata realizzata un'implementazione dell'algoritmo sequenziale, una con OpenMP e una con CUDA, misurando i tempi di esecuzione di ciascuna versione, e valutando gli speedup ottenuti con le versioni parallele all'aumentare della dimensione del dataset e al variare del numero di thread.

Chapter 1

Mean shift

Mean shift è un metodo non parametrico per la ricerca delle mode di una funzione di densità di probabilità. Introdotto nel 1975 da Fukunanga e Hostetler, è equivalente all'applicazione della discesa del gradiente alla stima kernel di densità della distribuzione. L'algoritmo non richiede assunzioni sulla forma dei cluster e ha un singolo parametro, l'ampiezza di banda (bandwidth), la cui determinazione è tuttavia non banale in generale. Mean shift ha applicazioni in analisi dei cluster, elaborazione digitale delle immagini e visione artificiale. [1].

È un algoritmo iterativo per determinare il massimo locale di una funzione di densità di probabilità a partire da un dataset di campioni. Data una funzione kernel K e una stima iniziale della moda x, ad ogni iterazione viene calcolata la media pesata della stima kernel di densità:

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x) x_i}{\sum_{x_i \in N(x)} K(x_i - x)}$$

dove N(x) è l'insieme di campioni x_i per i quali $K(x_i) \neq 0$. Il vettore $m(x) - x$ è detto mean shift. Il punto x viene aggiornato con uno sposta-

mento verso la media, nella direzione indicata dal vettore mean shift $x \leftarrow m(x)$ e il procedimento viene iterato fino a convergenza, quando lo shift diventa irrilevante.

Esistono molti tipi di funzione kernel, in questo elaborato utilizziamo la funzione di kernel gaussiana: $K(x) = e^{-\frac{x^2}{2\sigma^2}}$

La deviazione standard σ è il parametro ampiezza di banda.

L'idea alla base dell'algoritmo mostra che Mean Shift è imbarazzantemente parallelo, perché ciascun punto può essere elaborato indistintamente dagli altri. Questo lavoro presenta due differenti implementazioni parallele: una con OpenMP e l'altra con CUDA.

Chapter 2

Algoritmi implementati e test

2.1 Algoritmi implementati

Sono stati implementati un algoritmo sequenziale e 2 versioni parallele. La versione sequenziale è stata realizzata in C++. Una prima versione parallela è stata realizzata sempre in C++ con l'utilizzo della libreria OpenMP. Infine, è stata realizzata una versione parallela con l'utilizzo in CUDA.

2.1.1 Versione sequenziale

L'implementazione sequenziale proposta è una semplice applicazione del principio dietro all'algoritmo Mean Shift in C++. Come kernel è stato utilizzato il kernel gaussiano, mentre per misurare la distanza tra due punti è stata utilizzata le distanza euclidea. Di seguito è mostrato lo pseudocodice della parte saliente dell'algoritmo.

```
1  MEANSHIFT(originalPoints):
2      shiftedPoints ← originalPoints ;
3      while iterationIndex ≤ MAXITERATIONS do
4          foreach p ∈ shiftedPoints do
5              p ← SHIFTPOINT(p; originalPoints) ;

6  SHIFTPOINT(p,originalPoints):
7      shiftedP ← 0 ;
8      weight ← 0 ;
9      foreach o ∈ originalPoints do
10         dist ← dist(p; o) ;
11         weight ← GaussianKernel(dist; BW) ;
12         newPosition ← newPosition + weight ∗ o ;
13         totalWeight ← totalWeight + weight ;
14     return newPosition / totalWeight;
```

Algorithm 1: Mean shift algorithm

La costante MAXITERATIONS rappresenta il numero di volte in cui ciascun punto viene spostato, mentre la costante BW rappresenta la larghezza di banda. L'algoritmo è stato progettato per funzionare specificatamente con punti in tre dimensioni. Le principali strutture dati utilizzate dall'algoritmo sono due array. Il primo array "originalPoints" contiene tutti i punti nelle loro posizioni originali, il secondo array "shiftedPoints" contiene le nuove posizioni assunte dai punti dopo una fase di spostamento. L'array contenente le posizioni originali rimane sempre invariato; ad ogni passaggio viene definita la nuova posizione in cui effettuare lo spostamento calcolato leggendo i punti contenuti in questo array, è quindi importante notare che ogni punto effettua le sue operazioni di spostamento indipendentemente dagli altri punti.

La Figura 2.1 mostra un set di dati con 10.000 punti clusterizzato medi-

ante l'algoritmo.

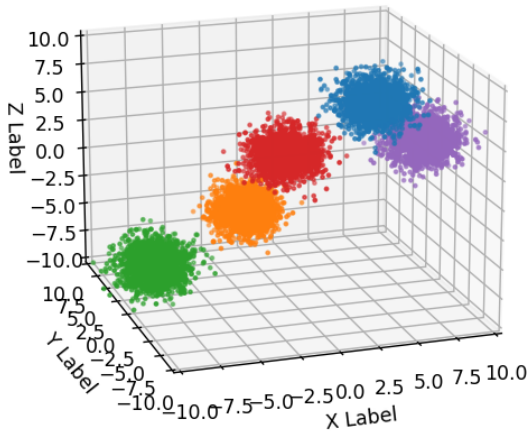

Figure 2.1: Esempio con dataset di 10.000 punti clusterizzato

2.1.2 Versione parallela con OpenMP

L'algoritmo Mean Shift è un problema imbarazzantemente parallelo: ogni punto effettua il suo spostamento indipendentemente dagli altri punti. Mediante la libreria OpenMP è possibile trasformare la versione sequenziale del programma in una versione parallela utilizzando la direttiva pragma. Infatti, come si può vedere nell'algoritmo sotto, l'implementazione con OpenMP differisce da quella sequenziale per la presenza di una singola riga di codice in più.

```
1  OPENMPMEANSHIFT(originalPoints):
2      shiftedPoints ← originalPoints ;
3      while iterationIndex ≤ MAXITERATIONS do
4          #pragmaparallelforschedule(static)
5          foreach p ∈ shiftedPoints do
6              p ← SHIFTPOINT(p; originalPoints) ;

7  SHIFTPOINT(p,originalPoints):
8      shiftedP ← 0 ;
9      weight ← 0 ;
10     foreach o ∈ originalPoints do
11         dist ← dist(p; o) ;
12         weight ← GaussianKernel(dist; BW) ;
13         newPosition ← newPosition + weight ∗ o ;
14         totalWeight ← totalWeight + weight ;
15     return newPosition / totalWeight;
```

Algorithm 2: Mean shift algorithm

La direttiva pragma viene utilizzata subito prima del foreach. In questo modo non c'è bisogno di alcuna sezione critica, poiché in ogni iterazione ogni punto viene spostato indipendentemente dagli altri. Abbiamo eseguito l'algoritmo sia con opzione schedule(static) che con schedule(dynamic) in modo da testare la verisone più efficiente. Nel primo caso il ciclo for viene suddiviso staticamente in pezzi di uguali dimensioni, garantendo che ogni thread riceva lo stesso carico di lavoro. È indicato nel caso in cui il numero di iterazioni è fisso dall'inizio e ogni iterazione richiede la stessa quantità di lavoro. Nel secondo caso il sistema decide come e quando assegnare un lavoro ad un thread, non appena il thread ha eseguito e diventa idle, un

nuovo compito puo' essergli assegnato. Questa seconda modalità è indicata nel caso di un ciclo for dove le iterazioni crichiedono un numero variabile, o anche imprevedibile, di lavoro. Nel nostro algoritmo potrebbe succedere che un thread finisca molto presto le sue iterazioni, ovvero nel caso in cui il punto converga molto velocemente, e poi debba aspettare che gli altri thread completino il suo lavoro, sprecando risorse. L'opzione dynamic potrebbe dunque rivelarsi vincente. In tabella è riportato il confronto tra i tempi di esecuzione di una versione parallela con schedulazione statica e di una versione parallela con schedulazione dinamica. Il test è stato eseguito su un dataset di 10.000 punti su tre dimensioni:

Threads	**Static** (*seconds*)	**Dynamic** (*seconds*)
2	191,381	196,587
3	146,128	145,592
4	119,702	111,925
5	107,128	94,390
6	99,904	88,703
7	90,457	83,733
8	84,119	85,131

Possiamo notare l'adozione della schedulazione Dynamic porti ad un'esecuzione più rapida poiché il carico di lavoro è distribuito in modo migliore tra i thread.

2.1.3 Versione parallela con CUDA

L'implementazione dell'algoritmo Mean Shift con CUDA consente di sfruttare il gran numero di core presenti all'interno della GPU. Infatti l'elaborazione di ciascun punto può essere assegnata ad un thread diverso. Durante i nostri

test abbiamo provato a sftruttare il tiling e la shared memory, ma senza successo in quanto ogni punto per essere shiftato doveva necessariamente leggere tutti gli altri punti e al crescere del numero di punti, lo spazio in shared memory non era sufficiente a caricare un numero significativo di punti. Essendo l'array dei punti di una sola dimensione, anche i blocchi e il grid sono stati definiti con la sola coordinata x. In particolare, blockDim.x è stato impostato uguale ad una costante (BLOCK_DIM), mentre il numero di blocchi è stato impostato come numeroPunti / BLOCK_DIM.

L'algoritmo è stato testato in 2 modalità: organizzando i dati come Structure of Arrays e come Arrays of Structures. La scelta tra "structure of array" (SoA) e "array of structure" (AoS) può influenzare le prestazioni in CUDA a seconda dei modelli di accesso alla memoria e del tipo di operazioni eseguite. In generale, SoA separa i dati in diverse strutture di array, consentendo un accesso parallelo ai singoli componenti, il che può essere vantaggioso in alcuni casi, come ad esempio quando si eseguono operazioni in cui è richiesto l'accesso simultaneo a diverse proprietà separate. In questo modo i punti vengono archiviati in memoria come Structure of Arrays per consentire il coalescing e per essere più amichevoli verso la cache L2: $[x_1, ...x_n, y_1; ...; y_n; z_1; ...; z_n]$. Nel nostro caso è risutlata più performante la struttura "array of structure", probabilmente perché l'algoritmo richiede l'accesso simultaneo a tutti i dati relativi a ciascun punto (x, y, z). Se le operazioni richiedono l'accesso simultaneo a più campi di una struttura (AoS), l'accesso a dati consecutivi nella memoria può essere più efficiente rispetto all'accesso a SoA, che richiede molteplici letture da diverse strutture di array.

La formula per accedere da un thread ad un elemento dell'array è: $blockDim.x * blockIdx.x + threadIdx.x$

```
1  CUDAMEANSHIFT(originalPoints):
2      shiftedPoints ← originalPoints ;
3      while iterationIndex ≤ MAXITERATIONS do
4          CUDAKERNEL(shiftedPoints; originalPoints) ;
5  CUDAKERNEL(shiftedPts;,originalPts):
6      idx ← blockIdx.x ∗ blockDim.x + threadIdx.x ;
7      if idx ≤ originalPts then
8          newPosition ← 0 ;
9          weight ← 0 ;
10         p ← shiftedPts[idx] ;
11         foreach o ∈ originalPoints do
12             dist ← dist(p; o) ;
13             weight ← GaussianKernel(dist; BW) ;
14             newPosition ← newPosition + weight ∗ o ;
15             totalWeight ← totalWeight + weight ;
16         shiftedPts[idx] ← newPosition/totalWeight ;
```

Algorithm 3: Mean shift algorithm

Ogni thread, prima di calcolare lo spostamento del suo punto corrispondente deve verificare se il suo indice non è maggiore del numero di punti: in questo caso il thread semplicemente non fa niente. Questo succede quando il numero di punti non è multiplo di BLOCK_DIM, e quindi vengono creati più thread del necessario.

2.2 Risultati dei test

La metrica utilizzata per confrontare le prestazioni dell'algoritmo sequenziale con le versioni OpenMP e CUDA è lo speedup, calcolato come: $S = \frac{t_S}{t_P}$

dove t_S e t_P sono rispettivamente il tempo di esecuzione della versione sequenziale e di quella parallela dell'algoritmo. I dataset utilizzati per le diverse implementazioni sono stati generati con la funzione make_blob di sklearn.datasets. [2]

Sono stati generati dataset di distribuzioni gaussiane con 5 centroidi, con deviazione standard uguale a 1 e composti rispettivamente da 100, 1000, 10.000, 20.000 punti in tre dimensioni per la versione sequenziale e con OpenMP, mentre per la versione con CUDA abbiamo generato dataset costituiti anche da 100.000, 250.000, 500.000, 1.000.000, 2.000.000 di punti.

La costante MAXITERATIONS è stata fissata pari a 10 perché è stato empiricamente stimato che 10 iterazioni erano sufficienti per far si che tutti i punti convergessero ai massimi locali, mentre il valore associato alla costante BW (ovvero larghezza di banda) è stato fissato uguale a 2. I test sono stati eseguiti su una macchina con CPU AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx, 2,10 GHz, 4 Cores, 8 Cores Logici. E su GPU NVIDIA RTX A2000 - 12 GB, CUDA 11.8.

Per rendere i risultati più affidabili e rappresentativi ogni tempo di esecuzione è stato ottenuto come la media ottenuta lanciando 10 volte ogni test.

2.2.1 OpenMP

Per valurare lo speedup della versione squenziale rispetto a quella parallela con OpenMP sono stati generati dataset di distribuzioni gaussiane con

5 centroidi, con deviazione standard uguale a 1 e composti rispettivamente da 100, 1000, 10.000, 20.000. I risultati mostrano come OpenMP, grazie all'utilizzo di una singola direttiva pragma, permetta di raggiungere speedup pari anhe a 4,50 dimostrando di avere un ottimo rapporto tra speedup e costi di sviluppo.

Number of points	100	
Sequential time	0,0357650	
Threads	**Time** (seconds)	**Speedup**
2	0,0300927	1,19
3	0,0203536	1,76
4	**0,0176960**	**2,02**
5	0,0185872	1,92
6	0,0208589	1,71
7	0,0193226	1,85
8	0,0182474	1,96

Table 2.1: Speedup con 100 punti al variare del numero dei thread con OpenMP (miglior risultato in grassetto)

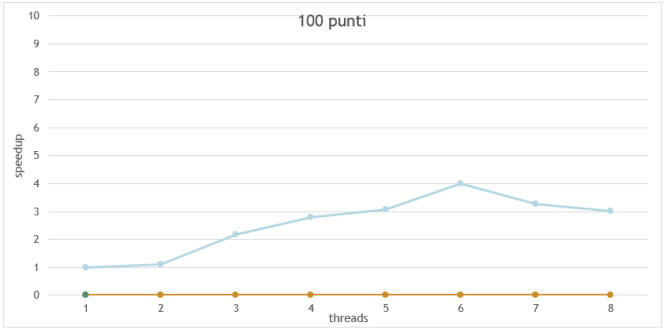

Figure 2.2: Speedup con 100 punti al variare del numere dei thread con OpenMP

Number of points	1.000	
Sequential time	4,0584600	
Threads	**Time** (seconds)	**Speedup**
2	1,9794600	2,05
3	1,3988400	2,90
4	1,2514600	3,24
5	1,3904500	2,92
6	1,0773600	3,77
7	0,8700630	4,66
8	**0,7883690**	**5,15**

Table 2.2: Speedup con 1.000 punti al variare del numero dei thread con OpenMP (miglior risultato in grassetto)

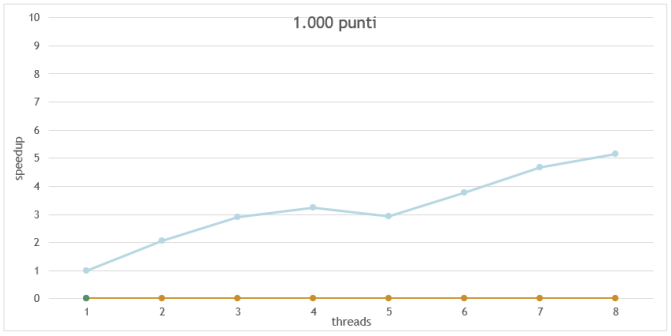

Figure 2.3: Speedup con 1.000 punti al variare del numere dei thread con OpenMP

Number of points	10.000	
Sequential time	350,2970000	
Threads	**Time** (seconds)	**Speedup**
2	196,5870000	1,78
3	145,5920000	2,41
4	111,9250000	3,13
5	94,3906000	3,71
6	88,7030000	3,95
7	**83,7330000**	**4,18**
8	85,1310000	4,11

Table 2.3: Speedup con 10.000 punti al variare del numero dei thread con OpenMP (miglior risultato in grassetto)

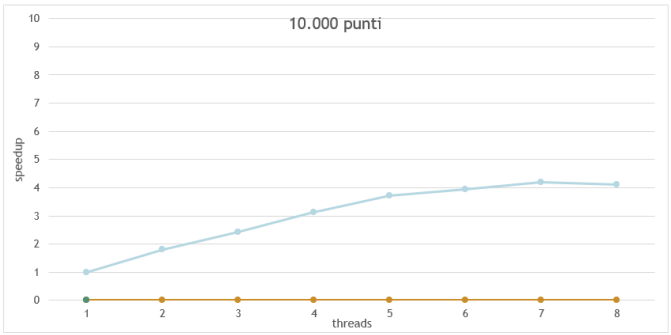

Figure 2.4: Speedup con 10.000 punti al variare del numere dei thread con OpenMP

Number of points	20.000	
Sequential time	1451,1200000	
Threads	**Time** (seconds)	**Speedup**
2	796,3090000	1,82
3	548,3970000	2,65
4	449,7490000	3,23
5	389,3060000	3,73
6	373,0740000	3,89
7	**324,7580000**	**4,47**
8	330,1860000	4,39

Table 2.4: Speedup con 20.000 punti al variare del numero dei thread con OpenMP (miglior risultato in grassetto)

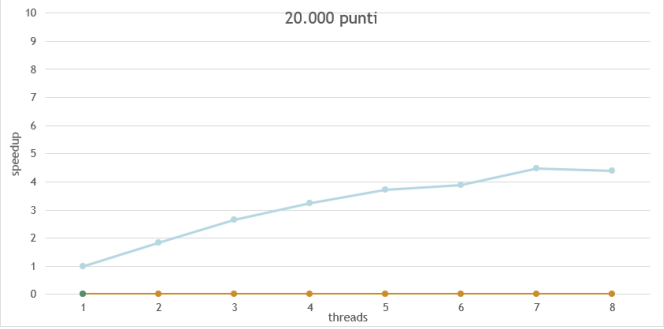

Figure 2.5: Speedup con 20.000 punti al variare del numere dei thread con OpenMP

Nelle tabelle 2.1, 2.2, 2.3, 2.4 e nelle figure 2.2, 2.3, 2.4, 2.5 sono mostrati i riusltati dei test. I test nella versione sequenziale e in quella parallela con OpenMP si sono fermati a dataset con dimensione pari a 20.000 punti, in quanto dataset di dimensioni maggiori avrebbero preso troppo tempo di esecuzione.

Per il dataset di 100 punti (vedi tabella 2.1) è interessante notare che la

velocità diminuisce quando il numero dei thread è maggiore di 5. Questo mostra come per un numero così basso di punti l'overhead dei thread sia superiore al guadagno derivante dal loro utilizzo. Questo fenomeno scompare per i dataset con più punti, dove l'utilizzo di più thread diminuisce il tempo di esecuzione (e di conseguenza aumenta lo speedup).

2.2.2 CUDA

Il primo test mirava a trovare la dimensione ottimale per BLOCK_DIM. Sono qundi stati misurati i tempi di esecuzione al variare della dimensione dei blocchi, mantenendo uno stesso resultset con 500.000 punti. Come si vede dai tempi mostrati in tabella 2.5 abbiamo ottenuto le migliori prestazioni con BLOCK_DIM = 128 e quindi questo valore è stato utilizzato per tutti gli altri test.

BLOCK_DIM	**Time** (seconds)
16	41,916332
32	22,085489
64	19,541801
128	**19,185013**
256	19,304701
512	19,386659
1024	19,865561

Table 2.5: Tempo di esecuzione con 500.000 punti al variare del valore di BLOCK_DIM (miglior risultato in grassetto)

Questo si spiega in quanto, fare blocchi più grandi, porta ad avere abbastanza warp e lo streaming multiprocessor riesce ad essere occupato con continuità. Al contrario avere blcocchi troppo grandi porta alla saturazione

dei registri (che sono 32.000) e la gpu lavora male.

L'algoritmo con CUDA è stato eseguito su tutti i dataset e quando possibile è stato valutato lo speedup rispetto alla versione sequenziale dell'algoritmo di Mean shift.

Number of points	**Time** (seconds)	**Speedup**
100	0,000201	61,61
1.000	0,001057	745,86
10.000	0,013681	6.222,57
20.000	0,020681	26.516,95
100.000	0,814327	
250.000	4,817884	
500.000	19,3792	
1.000.000	78,010368	
2.000.000	323,810242	

Table 2.6: Tempo di esecuzione e speedup al variare del numero dei punti con CUDA

In tabella 2.6 si vede chiaramente come l'utilizzo della GPU permetta di ridurre notevolmente i tempi di esecuzione favorendo l'esecuzione dell'algoritmo anche su dataset di dimensioni impensabili nelle versioni seuenziale e parallela con OpenMP. Infatti è importante notare che il tempo di esecuzione con set di dati con oltre 20.000 punti non è stato misurato nelle altre versioni dell'algoritmo perché avrebbe comportato troppo tempo. Si vede come l'utilizzo di CUDA porta ad un aumento dello speedup esponenziale, arrivando anche a 26.516 (caso con dataset di 20.000 punti).

Chapter 3

Conclusioni

In questo lavoro è stato presentato l'algoritmo di clustering Mean Shift ed è stato mostrato come la sua struttura imbarazzantemente parallela lo renda adatto per il calcolo parallelo. È stata sviluppata un'implementazione parallela con OpenMP aggiungendo soltanto una direttiva che ha permesso di ottenere un aumento dello speedup di oltre 5. Successivamente è stata presentata un'implementazione in CUDA e con i suoi tempi di esecuzione ha mostrato come l'utilizzo della GPU renda l'algoritmo di Mean Shift applicabile anche a set di dati con dimensioni intrattabili per le CPU.

Bibliography

[1] Mean shift. https://it.wikipedia.org/wiki/Mean_shift.

[2] F. pedregosa, g. varoquaux, a. gramfort, v. michel, b. thirion, o. grisel, m. blondel, p. prettenhofer, r. weiss, v. dubourg, j. vanderplas, a. passos, d. cournapeau, m. brucher, m. perrot, and e. duchesnay. scikit-learn: Machine learning in python. journal of machine learning research, 12:2825–2830, 2011. https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html.