K-means clustering

Programmazione di sistemi embedded e multicore

A.A. 2024/2025

Autori:

Francesco Carboni
2058986

Alessandro Gautieri
2041850

Introduzione

L'algoritmo di K-means clustering è stato implementato con 4 diverse soluzioni:

- MPI
- OMP
- CUDA
- MPI+OMP

Versione MPI

1.1 Suddivisione del Carico di Lavoro

L'implementazione MPI inizia dividendo gli N punti tra i P processi disponibili. Ogni processo riceve una porzione distinta del dataset calcolata usando una distribuzione bilanciata. Per fare ciò, utilizziamo la macro `DISTRIBUTION`, che calcola la dimensione dei blocchi assegnati a ciascun processo, e la macro `OFFSET`, che determina gli indici iniziali di ciascun blocco. Questo garantisce che i dati siano distribuiti equamente e che ogni processo conosca esattamente quale parte del dataset elaborare.

1.2 Classificazione dei Punti

Ogni processo esegue la classificazione dei punti nella propria porzione di dati. La distanza euclidea tra ogni punto e i centroidi viene calcolata utilizzando la funzione `euclideanDistance` o la versione ottimizzata `UNROLLEDeuclideanDistance` la scelta tra le funzione viene effettuata in base al numero di dimensioni (*samples*):

`UNROLLEDeuclideanDistance` viene scelta quando il numero di dimensioni è pari per sfruttare al massimo le ottimizzazioni basate su unrolling del loop, il punto in seguito viene assegnato al cluster con la distanza minima.

Il risultato della classificazione viene memorizzato in un array locale `localClassMap`. Durante questa fase:

Ogni modifica alla classificazione incrementa il contatore locale `changes`.

Un array `pointsPerClass` viene aggiornato per contare quanti punti appartengono a ciascun cluster.

1.3 Ottimizzazioni della Distanza Euclidea

Per ottimizzare il calcolo della distanza euclidea, sono stati introdotti diversi miglioramenti basati su un approccio a blocchi per sfruttare al meglio la gerarchia di memoria della CPU:

Eliminazione di `sqrt()`:

- Durante il confronto tra distanze per assegnare i punti ai cluster, la radice quadrata non è necessaria, poiché il confronto si basa sui valori relativi. Abbiamo quindi evitato di calcolare la radice quadrata durante questa fase.

Calcolo a blocchi:

- Il dataset è suddiviso in blocchi di dimensione fissa (ad esempio, 32 elementi), che vengono processati separatamente. Questa tecnica sfrutta la località della cache per ridurre i tempi di accesso alla memoria principale e migliorare le prestazioni complessive.

Accesso sequenziale alla memoria:

- I dati dei punti e dei centroidi sono organizzati in array monodimensionali contigui, ottimizzando l'accesso alla memoria e riducendo i cache miss.

Accumulatori temporanei per blocchi:

- Durante il calcolo delle distanze, ogni blocco utilizza un accumulatore temporaneo locale per sommare le differenze quadratiche. Questo approccio minimizza gli accessi alla memoria globale e migliora l'efficienza computazionale.

Riduzione degli accessi alla memoria:

- Le coordinate dei punti e dei centroidi vengono caricate in variabili temporanee all'interno del blocco, riducendo gli accessi ripetuti alla memoria principale.

Questa combinazione di ottimizzazioni ha permesso di migliorare significativamente le prestazioni del calcolo della distanza, specialmente su dataset di grandi dimensioni con un elevato numero di campioni e dimensioni.

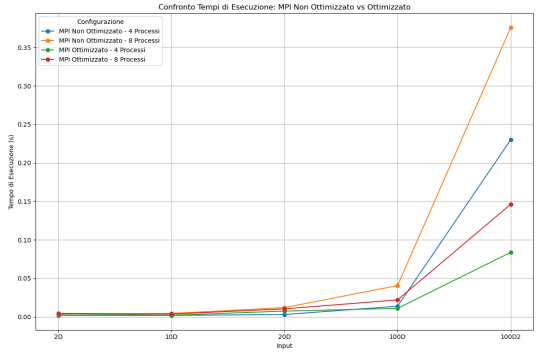


Figure 1: In questo grafico mostriamo la **differenza** di tempo di esecuzione tra le versioni **MPI non ottimizzata e MPI ottimizzata sulla distanza**. I tempi per questo grafico sono stati ottenuti sul nostro PC.

1.4 Comunicazione e Sincronizzazione

La comunicazione tra i processi è stata ottimizzata per ridurre i tempi di inattività:

L'operazione `MPI_Iallreduce` è stata scelta per sommare i dati tra i processi in modo asincrono, sovrapponendo le operazioni di comunicazione al calcolo locale. Questo approccio consente di ridurre i tempi morti, poiché i processi non rimangono bloccati in attesa dei risultati.

Viene utilizzata per aggregare i valori di `pointsPerClass` (conteggio globale dei punti per cluster) e `changes` (numero totale di cambiamenti).

Riduzione globale delle coordinate:

Una seconda `MPI_Iallreduce` viene impiegata per combinare le somme parziali delle coordinate dei centroidi calcolate da ciascun processo. Questo garantisce che i centroidi vengano aggiornati correttamente utilizzando tutti i dati disponibili.

1.5 Aggiornamento dei Centroidi

Ogni processo contribuisce all'aggiornamento globale dei centroidi calcolando localmente le somme delle coordinate dei punti assegnati a ciascun cluster. Questi risultati parziali vengono combinati globalmente tramite MPI. Dopo la riduzione globale:

Ogni processo divide le somme totali per il numero di punti nel cluster corrispondente (`pointsPerClass`), ottenendo i nuovi centroidi.

Questo aggiornamento è effettuato in modo sincrono per garantire la coerenza tra i processi.

1.6 Ricostruzione globale

Una volta soddisfatte le condizioni di arresto, i risultati locali memorizzati in `localClassMap` vengono raccolti nel processo con rank `0` utilizzando `MPI_Gatherv`. Questa operazione consente di ricostruire la mappa delle classificazioni globali e di salvarla in un file di output.

1.7 Note aggiuntive sul programma

L'implementazione risulta particolarmente efficiente nel caso di un numero di cluster contenuto, ogni processo infatti conserva una copia dell'intero array di centroidi, e non ha bisogno di fare alcuna comunicazione con altri processi per classificare un punto. Si inizia a perdere di efficienza solo nel caso in cui il numero di cluster diventa particolarmente grande, e quindi il programma passerà la maggior parte del tempo nella fase di **classificazione**. Un' alternativa potrebbe essere quella di dividere i centroidi tra i vari processi, in questo caso ridurremmo il carico del singolo processo, ma allo stesso tempo il numero di comunicazioni per ciclo aumenterebbe notevolemente (almeno 1 comunicazione per ogni punto).

In modo sperimentale abbiamo voluto aggiungere anche in questa versione come nelle due successive, il **loop unrolling**, registrando un incremento di prestazioni non trascurabile.

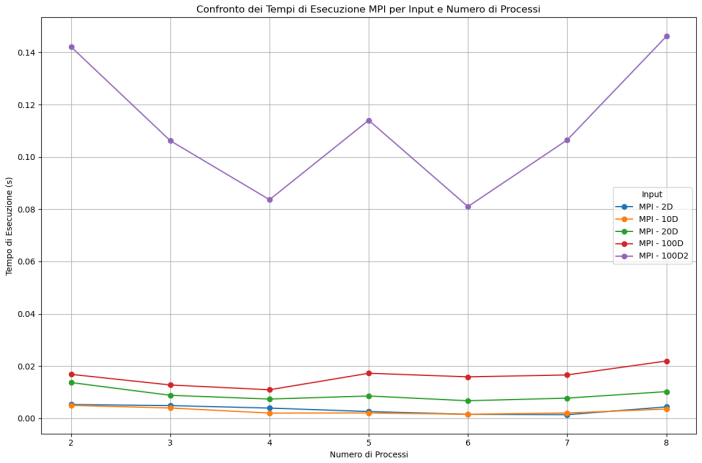


Figure 2: Confronto tra esecuzioni MPI con numeri di processi da 2 a 8.

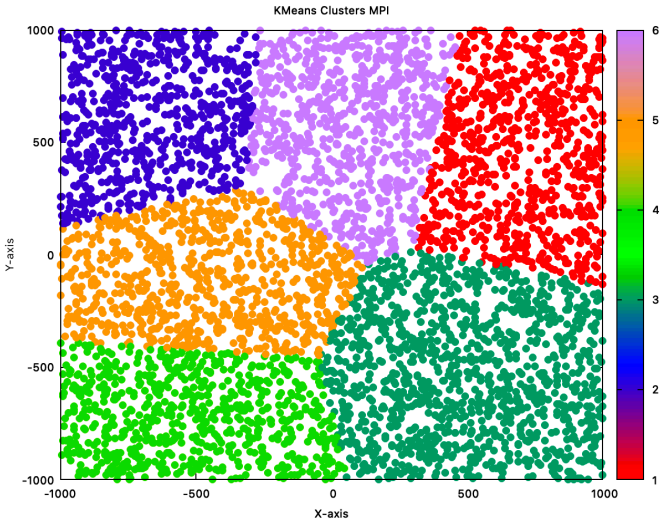


Figure 3: Visualizzazione grafica della correttezza dell'algoritmo, input2D.

Versione OpenMP

2.1-Suddivisione dei compiti

Nella versione OpenMP, il dataset viene suddiviso tra i thread disponibili per sfruttare il parallelismo a livello di dati. L'approccio seguito utilizza il costrutto `#pragma omp parallel for` per distribuire equamente i punti tra i thread. La suddivisione del carico è gestita dinamicamente con l'uso di diversi tipi di `scheduling` (static, guided), che vengono scelti a seconda del tipo di operazione:

- `guided`: Utilizzato per la classificazione dei punti, poiché il costo computazionale varia in base alla complessità delle operazioni sui centroidi.
- `static`: Utilizzato per operazioni con costo computazionale uniforme, come l'aggiornamento dei centroidi.

Questa tecnica garantisce che il carico di lavoro sia distribuito uniformemente tra i thread, minimizzando gli squilibri.

2.2 Classificazione dei punti

Ogni thread calcola la distanza tra i punti e i centroidi per la propria porzione di dati. La funzione `euclideanDistance` o la versione ottimizzata `UNROLLEDeuclideanDistance` la scelta tra le due funzioni viene effettuata in base al numero di dimensioni (*samples*):

`UNROLLEDeuclideanDistance` viene scelta quando il numero di dimensioni è pari per sfruttare al massimo le ottimizzazioni basate su unrolling del loop, abbiamo notato come l'uso dell' `UNROLL` ci permette di dimezzare i tempi di computazione del programma, il punto in seguito viene assegnato al cluster con la distanza minima.

I risultati della classificazione sono salvati nell'array condiviso classMap. La riduzione parallela (*reduction*) è utilizzata per calcolare il numero di cambiamenti di cluster (*changes*) in modo efficiente, garantendo coerenza tra i thread.

2.3 Ottimizzazioni della distanza euclidea

L'implementazione della distanza euclidea è stata notevolmente ottimizzata per migliorare le prestazioni come nel paragrafo **(1.3)**. In aggiunta bisogna fare alcune osservazioni sull'utilizzo del loop unrolling:

- l'`UNROLL` dimezza i tempi di esecuzione quando il suo valore è 2, e può essere utilizzato per dataset con `samples` pari. Nel caso di punti con dimensione dispari utilizzeremo la distanza euclidea senza fare l'unroll.
- si posso utilizzare `UNROLL`>2 ma bisogna tenere conto di due fattori:
  1. la dimensione dei punti del dataset, che potrebbe portare ad accessi errati nel caso in cui non sia divisore di `UNROLL`
  2. "unrollando" troppo un for loop le performance degradano.

2.4 Parallelismo e sincronizzazione

La sincronizzazione tra i thread è gestita in modo efficiente grazie all'uso di:

- **reduction**: Per sommare i valori globali come changes e maxDist in modo thread-safe.
- **Barriere implicite**: Le operazioni parallele sono sincronizzate automaticamente alla fine di ciascun ciclo `#pragma omp parallel for`.

2.5 Aggiornamento dei centroidi

Le somme delle coordinate dei punti appartenenti ai cluster vengono calcolate in modo seriale. Questo non rallenta il programma, infatti un singolo processo impiega poco tempo nell' accumulo delle coordinate, al contrario più processi dovrebbero coordinarsi con delle `#pragma omp atomic`, una per ogni punto p, e una per ogni dimensione $(p_1, ..., p_n)$ di p.

2.6 Ricostruzione globale

Dopo ogni iterazione:

- I centroidi vengono aggiornati dividendo le somme calcolate per il numero di punti appartenenti al cluster.
- Questo aggiornamento è effettuato in parallelo per ogni dimensione del centroide

2.7 Note aggiuntive sul programma

A differenza dell'implementazione tramite MPI, l'uso di **omp** per data set con dimensioni contenute risulta inefficiente (e.g. input2D.inp), eccelle invece quando il numero di cluster e punti cresce (e.g. input100D2.inp). Qui sotto si riassumono le principali ottimizzazioni:

- **Unrolling**: Utilizzato per processare più dimensioni contemporaneamente.
- **Riduzione dei Cache Miss**: Suddivisione in blocchi e accesso sequenziale alla memoria.
- **Riduzione Parallela**: Uso di reduction per operazioni globali come changes e maxDist.
- **Distribuzione Dinamica del Carico**: Scheduling guided per operazioni non uniformi.

Versione MPI+OpenMP

3.1 Introduzione

In questa sezione descriviamo come è stata realizzata la versione del k-Means che sfrutta sia la parallelizzazione a livello di **processi** tramite `MPI` sia la parallelizzazione a livello di **thread** tramite `OpenMP`. L'obiettivo è massimizzare l'utilizzo di risorse computazionali disponibili su architetture multicore e cluster di calcolatori.

3.2 Struttura generale

L'idea di base è suddividere il dataset tra i processi MPI, così che ognuno gestisca un sottoinsieme dei punti. All'interno di ogni processo MPI, abilitiamo OpenMP per elaborare ulteriormente in parallelo i punti assegnati. Questa strategia consente di:

- Ridurre la comunicazione tra nodi (poiché i dati sono partizionati).
- Sfruttare meglio i core di ciascun nodo tramite il multithreading.

3.3 Distribuzione iniziale dei dati con MPI

Come avviene nella versione puramente MPI, si calcola per ogni processo il numero di punti da gestire, basandosi su una strategia di bilanciamento. Si determina l'offset iniziale che indica il punto del dataset da cui ciascun processo inizia a leggere.

3.4 Suddivisione del carico e parallelismo con OpenMP

All'interno di ciascun processo MPI, attiviamo OpenMP su alcuni passaggi critici, in particolare:

- Assegnazione dei punti al cluster (calcolo della minima distanza rispetto ai centroidi).
- Accumulazione delle coordinate nei vettori che servono per aggiornare i centroidi.

Questo significa che, per il sottoinsieme di punti locale, ogni processo fa lavorare più thread su un for parallelo con reduction. Ad esempio, la clausola `reduction(+: ...)` permette di sommare correttamente i contributi di tutti i thread, evitando problemi di concorrenza tipici delle variabili globali.

3.5 Cambiamenti introdotti rispetto alla versione MPI

Inserimento di `#pragma omp parallel for` con reduction.

- Prima, la variabile `changes` o l'array `pointsPerClass[]` potevano subire incrementi non protetti da più thread, generando risultati sbagliati. Ora, ogni thread ha una copia locale da sommare automaticamente a fine ciclo.

Gestione accurata di `changes`.

- Abbiamo introdotto una variabile locale (local_changes) all'interno di ogni thread. Solo dopo la riduzione si ottiene il numero totale di cambiamenti effettivi (changes = local_changes;). Questo impedisce aggiornamenti **concorrenti** su changes++.

Evita doppio conteggio.

- In questa implementazione ci sono due fasi distinte: assegnazione cluster (con conteggio dei cambiamenti) e accumulazione coordinate. Entrambe ricorrono a riduzioni OpenMP separate e, all'occorrenza, si azzerano le strutture dati prima di ciascun uso.

Cooperazione con MPI.

- I risultati locali, il totale globale di `changes`, la somma delle coordinate `auxCentroids[]` e il conteggio di punti `pointsPerClass[]` vengono comunicati tra processi con `MPI_Allreduce`. In questo modo la fase di aggiornamento dei centroidi resta corretta a livello globale.

3.6 I problemi riscontrati e soluzioni

Nel passaggio dalla versione puramente MPI a quella ibrida MPI+OpenMP, abbiamo incontrato principalmente:

- Race condition sui contatori: alcune variabili (come changes o pointsPerClass) venivano incrementate in un ciclo parallelo senza protezione. Abbiamo risolto inserendo la clausola `reduction(+: ...)`, che garantisce la somma dei valori locali di ogni thread.
- Doppio incremento di pointsPerClass: in precedenti versioni, veniva incrementato più volte nella stessa iterazione, senza un azzeramento intermedio. Ora abbiamo separato chiaramente la fase di assegnazione da quella di accumulo delle coordinate, eliminando i conteggi duplicati.

Utilizzando variabili **private** nei parallel for, abbiamo risolto i conflitti.

3.7 Vantaggi del doppio livello di parallelismo

Riduzione dei tempi di comunicazione:

- la banda fra processi MPI viene utilizzata in modo più efficiente, perché ciascun processo elabora una parte del dataset in parallelo localmente, scambiando solo le informazioni aggregate.
- Scalabilità: su un cluster di nodi multicore, possiamo aumentare sia il numero di processi MPI che il numero di thread per processo, ottenendo un utilizzo ottimale delle risorse.
- Flessibilità: in fase di esecuzione, è sufficiente impostare la variabile di ambiente `OMP_NUM_THREADS` per cambiare il numero di thread. Si può così testare con facilità l'efficienza su diverse configurazioni (un thread con molti processi, oppure pochi processi con molti thread).

3.8 Conclusioni

La versione MPI + OpenMP del k-Means è stata realizzata aggiungendo costrutti OpenMP nei punti chiave del sorgente MPI. Con la corretta gestione delle variabili in riduzione e l'azzeramento iniziale di vettori condivisi, il calcolo risulta più efficiente e scalabile. L'introduzione del doppio livello di parallelismo consente di sfruttare l'intero potenziale delle architetture parallele, combinando comunicazione inter-nodo (MPI) e concorrenza intra-nodo (OpenMP).

Note: questa implementazione supporta chiamate alle funzioni della libreria MPI solo da parte del master thread (MPI_THREAD_FUNNELED)

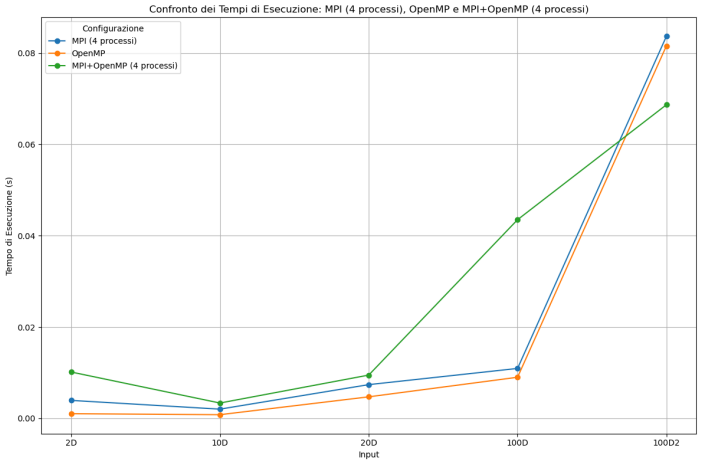


Figure 4: In questo grafico mostriamo il confronto dei tempi di esecuzione di **MPI, OpenMP e MPI + OpenMP** I tempi per questo grafico sono stati ottenuti sul nostro PC.

Versione CUDA

4.0 Premessa

L'implementazione della versione **CUDA** di K-means riprende alcune delle ottimizzazioni utilizzate in precedenza, ma suddivide il carico di lavoro in modo differente. Tale modifica è dovuta alla possibilità di avere un gran numero di thread che eseguono in parallelo.

4.1 Gestione della memoria

4.1.1 __costant__ memory

La **costant memory** risulta particolarmente efficiente per la memorizzazione di dati che non variano durante il programma, essendo read-only e cached, quindi il numero di cluster d_K, il numero di punti d_lines e il numero di dimensioni per punti d_samples sono tutti dichiarati __costant__ nello scope globale. I thread accedono spesso a questi dati durante il calcolo dei centroidi, nello specifico ogni warp accede alle stesse tre locazioni, quindi i dati posso esse mandati in broadcast a tutti i suoi thread. Un idea potrebbe essere quella di caricare l'intero array data nella costant memory, prima però bisogna notare tre cose:

- ha una dimensione limitata a 64kB, perciò non è adatta a memorizzare grandi quantità di dati;
- la dimensione dell'array deve essere nota a priori (e non è il nostro caso);
- nel caso di K-means i thread accedono ognuno a dati diversi ma consecutivi in memoria, per questo motivo i memory burst della **global memory** risultano più efficienti.

4.1.2 __device__ memory

Le variabili dichiarate in device memory risiedono nella memoria globale, allocate con cudaMalloc(), sono persistenti tra un kernel e l'altro, condizioni perfette per memorizzare le seguenti variabili:

- d_auxCentroids → accumula le coordinate dei punti per aggiornare i centroidi;
- d_centroids → contiene le coordinate dei centroidi;
- d_pointsPerClass → contiene i punti appartenenti ad ogni cluster;
- d_classMap → mappa delle classi dei punti;
- d_changes → accumula cambiamenti nei cluster.

4.1.3 __shared__ memory

In una prima implementazione ogni blocco caricava in shared memory tutti i centroidi con le rispettive coordinate, in questo modo ogni thread nello stesso blocco poteva accedervi ad un costo minore rispetto ad un'accesso in memoria globale. Aumentando il numero di cluster ci si rende conto che il kernel che carica in memoria i centroidi non viene nemmeno lanciato, questo perché, come per la costant memory, anche la shared memory ha una dimensione limitata per ciascun SM. Rimuovere shared_centroids non ha un impatto rilevante sulle prestazioni e ci permette di eseguire il codice su un gran numero di cluster. Un discorso simile si può fare per l'array d_pointsPerClass che viene aggiornato con atomicAdd(). Per ottimizzare potremmo creare una copia locale dell'array condivisa nel blocco, aggiornarlo con degli incrementi atomici fatti al livello di shared memory (quindi meno costosi), e poi fare un altra atomicAdd dall'array locale al blocco a quello globale:

```
extern __shared__ int local_pointsPerClass[];
...
atomicAdd(&d_pointsPerClass, local_pointsPerclass);
```

In questo caso la però la versione senza shared memory risulta più efficiente e più tollerante al crescere del numero di cluster.

4.2 gridSize e blockSize

Prima di lanciare il kernel è doveroso decidere come si vuole dividere il lavoro. Dato che non esiste una legge univoca che stabilisce quanti blocchi usare e quanto farli grandi, si può procedere per tentativi, cercando di bilanciare il carico tra i vari SM ed evitando di fare sprechi (di thread). Una buona pratica è quella di avere la dimensione dei blocchi multiplo del warpSize, e cercare il più possibile di distribuire in modo bilanciato i blocchi tra i vari SM, minimizzando l'**imbalance**. Un'altra cosa importante di cui abbiamo tenuto conto è la quantità di registri che il thread utilizza nel kernel (-Xptxas -v), se ne utilizzasse più di quelli che gli si possono allocare la variabile verrebbe mossa in global memory.

4.3 GPU Kernel

4.3.1 GPU_ClassAssignment

Nel seguente kernel viene eseguito l'assegnamento dei punti in d_data ai vari cluster in d_centroids. Ogni thread classifica POINTS_PER_THREAD punti, aggiorna la mappa dei centroidi d_classMap, incrementa il contatore di punti per classe con atomicAdd() insieme alle coordinate in d_auxCentroids. I cambiamenti di cluster sono mantenuti nella variabile local_changes, locale ad ogni thread, che viene aggiornata una volta sola alla fine della classificazione dei punti.

4.3.2 GPU_CentroidsUpdate

Ogni thread che esegue questa funzione prende una coordinata di uno dei cluster in d_auxCentroids[global_id] e la divide per d_pointsPerClass[Class], salva il risultato in d_centroids.

4.4 Note aggiuntive sul programma

Nella seguente implementazione non è stato possibile sfruttare la __shared__ memory (per i motivi citati sopra). Ogni thread inoltre fa molteplici accessi alla memoria globale per ottenere le coordinate dei punti e dei centroidi. Anche nel caso

di dataset con punti in poche dimensioni, risulta comunque conveniente lasciare abilitata la **cache L1** (-Xptxas -dlcm=cg). Questo perché per dataset con dimensioni abbastanza grandi è conveniente caricare a blocchi di 128 byte, rispetto ai 32 caricati dalla L2 (anche se non vengono sfruttati gli accessi coalescenti).

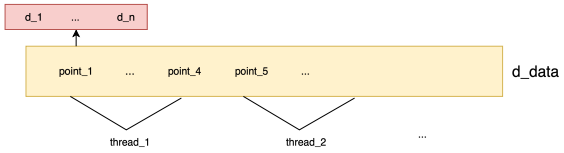


Figure 5: distribuzione dei punti tra i thread.

Come nelle implementazioni precedenti facciamo l'unroll di due iterazioni nella distanza euclidea. L'utilizzo del kernel `GPU_CentroidsUpdate()` è conveniente per due motivi:

- banalmente viene effettuata una sola operazione in parallelo da ogni thread.
- In una prima implementazione veniva utilizzato solo il kernel che assegna le classi, successivamente per ogni iterazione del ciclo più esterno venivano spostati dal device alla memoria dell'host `d_auxCentroids` e `d_pointsPerClass` per fare i calcoli di aggiornamento in locale. In questo modo evitiamo degli spostamenti inutili.

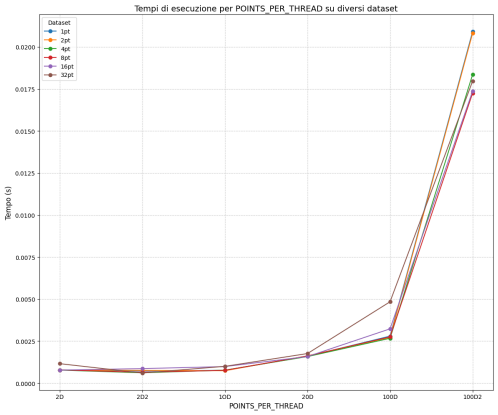


Figure 6: il grafico sopra mostra come sia conveniente, al crescere del dataset, assegnare più di due punti per thread.

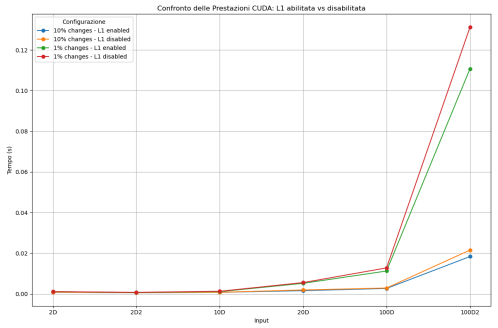


Figure 7: La cache L1 è particolarmente efficiente quando si ha a che fare con dataset grandi e bisogna compiere più iterazioni.

5 Analisi delle performance

In questa sezione mostreremo i risulati delle diverse implementazioni. Alcuni dati sono stati ottenuti con esecuzioni sul nostro portatile personale, un Apple Macbook Air del 2024 con processore M3. Altre esecuzioni sono state invece eseguite sul cluster della Sapienza.

5.1 Parametri di esecuzione

Le seguenti specifiche (al di fuori degli argomenti) sono le specifiche migliori paragonando i vari tempi di ciascuna prova.

Argomenti utilizzati in tutte le esecuzioni:

- 6 300 1 0.0001 (Num Cluster, Max Iterations, Min Changes, Threshold)

5.1.1 Parametri sul Cluster

MPI:
- cpu=8
- processi=16

OMP:
- cpu=8
- threads=8

MPI+OMP:
- cpu=8
- threads=4
- processi=4

CUDA:
- gpu=1

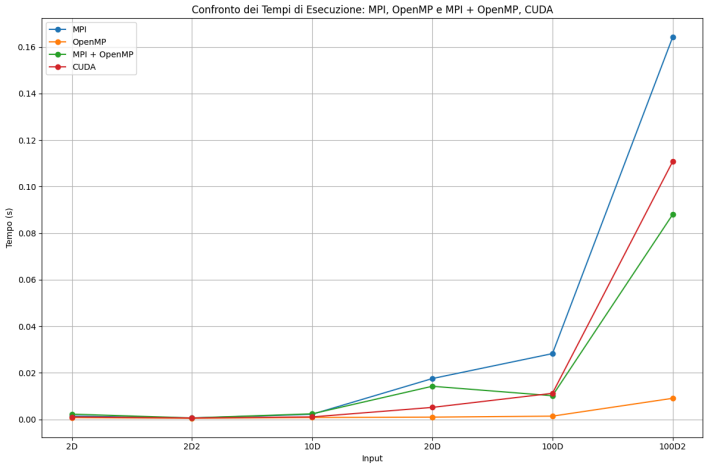


Figure 8: In questo grafico mostriamo il confronto dei tempi di esecuzione di **MPI, OpenMP e MPI + OpenMP, CUDA**. I tempi per questo grafico sono stati ottenuti sul cluster.

5.2 Altre esecuzioni su MPI

In questa tabella mostriamo i tempi di esecuzione con argomenti diversi da quelli sopracitati, abbiamo deciso di fare anche queste simulazioni per capire l'andamento dei programma con argomenti di input diversi da quelli standard, il file di input per ogni esecuzione era input100D2.inp.

Esecuzione	Tempo su PC	Tempo su Cluster
1° esecuzione	2.461747	0.554725
2° esecuzione	5.804244	5.297486
3° esecuzione	21.353719	23.219704
4° esecuzione	30.266113	13.144850
5° esecuzione	115.531722	35.844395

5.2.1 Specifiche argomenti esecuzioni

I dati forniti saranno nell ordine:

Num Cluster, Max Iterations, Min Changesm Threshold.

1° esecuzione:
• 100,300,1,0.0001

2° esecuzione:
• 1000,300,1,0.0001

3° esecuzione:
• 1000,300,0.001,0.0001

4° esecuzione:
• 10000,300,1,0.0001

5° esecuzione:
• 10000,300,0.001,0.0001

5.3 Speedup ed efficienza

L'implementaziome MPI, su input di grandi dimensioni (100D2), ha uno speedup lineare fino a 4 processi, successivamente c'è un calo delle performance a 5, con un successivo picco a 6. Per dataset di dimensione intermedia non abbiamo un grande speedup, ma al variare del numero di processi la performance rimane comunque migliore della versione ad un processo.

Su insiemi di dati più piccoli l'overhead dovuto alla comunicazione tra processi non viene compensato dalla distribuzione del carico di lavoro, risultando in uno speedup < 1.

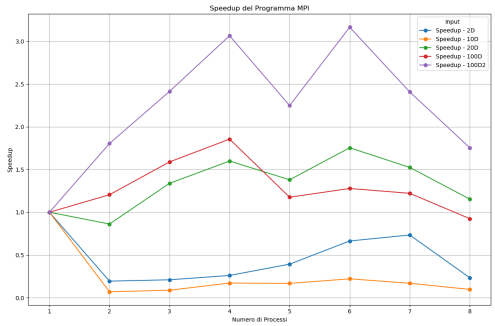


Figure 9: speedup di MPI.

Discorso simile si può fare per l'efficienza, che ha una decrescita più lenta per dataset grandi e molto più rapida per insiemi di dati come il 2D e 10D.

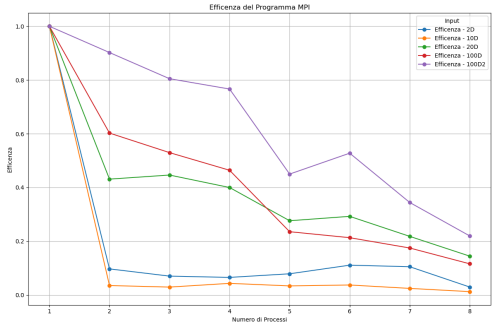


Figure 10: efficienza di MPI.

L'implementazione MPI non è strong scalable, in quanto aumentando il numero di processi e mantenendo lo stesso input l'efficienza non rimane stabile.

6 Analisi dell'efficienza con perf

Abbiamo svolto un'analisi dei nostri programmi tramite il tool linux **Perf**, eseguito su un computer diverso dal nostro PC e dal Cluster, i dati sono riportati in questo ordine:

Versione MPI | Versione OpenMP | Versione MPI + OpenMP

I dati che abbiamo deciso di riportare dall'analisi **Perf** sono i seguenti:

1. **Cache Misses**: Mancati accessi alla cache; indicano inefficienza nell'uso della memoria locale, utile per identificare **colli di bottiglia** nella memoria.
2. **Cache References**: Numero di accessi alla cache; misura il carico sulla gerarchia della memoria, utile per valutare **l'intensità** delle operazioni di memoria.
3. **Instructions per Cycle (IPC)**: Istruzioni eseguite per ciclo CPU; indica quanto **efficientemente** viene utilizzata la CPU, utile per confrontare il parallelismo tra versioni.
4. **Branch Misses**: Predizioni errate dei salti condizionali; identificano **inefficienze** nel controllo del flusso, utili per ottimizzare il branching del codice.

6.1 Dati

Input 2D:
- Gestione Cache:
  - cache-misses 17,91% | 21,75% | 18,66%
  - cache-references 31,782 M/sec | 11,109 | 11,050
- Efficienza CPU:
  - instructions 1,02 insn per cycle | 1,98 | 0,86
  - cycles 2,642 GHz | 1,634 | 3,152
- Branch Mispredictions
  - branch-misses 2,27% | 0,78% | 0,74%

Input 2D2:
- Gestione Cache:
  - cache-misses 18,07% | 34,39% | 23,45%
  - cache-references 31,208 M/sec | 2,884 | 16,807
- Efficienza CPU:
  - instructions 1,00 insn per cycle | 0,52 | 0,86
  - cycles 2,437 GHz | 3,139 | 3,144
- Branch Mispredictions
  - branch-misses 2,52% | 0,25% | 0,86%

Input 10D:
- Gestione Cache:
  - cache-misses 17,34% | 33,32% | 17,15%
  - cache-references 32,158 M/sec | 2,212 | 6,476
- Efficienza CPU:
  - instructions 0,96 insn per cycle | 0,52 | 0,86
  - cycles 2,580 GHz | 3,217 | 3,160
- Branch Mispredictions

- branch-misses 2,46% | 0,25% | 0,46%

Input 20D:
- Gestione Cache:
  - cache-misses 19,23% | 32,32% | 18,23%
  - cache-references 22,292 M/sec | 3,592 | 3,735
- Efficienza CPU:
  - instructions 1,47 insn per cycle | 1,78 | 1,11
  - cycles 2,807 GHz | 2,623 | 3,276
- Branch Mispredictions
  - branch-misses 1,48% | 0,69% | 0,35%

Input 100D:
- Gestione Cache:
  - cache-misses 23,23% | 54,91% | 21,38%
  - cache-references 10,311 M/sec | 2,530 | 3,728
- Efficienza CPU:
  - instructions 2,15 insn per cycle | 1,63 | 1,09
  - cycles 3,276 GHz | 3,407 | 3,317
- Branch Mispredictions
  - branch-misses 0,72% | 0,68% | 0,40%

Input 100D2:
- Gestione Cache:
  - cache-misses 42,18% | 54,34% | 36,67%
  - cache-references 4,427 M/sec | 2,668 | 2,887
- Efficienza CPU:
  - instructions 2,32 insn per cycle | 2,16 | 1,76
  - cycles 3,369 | 3,416 | 3,351
- Branch Mispredictions
  - branch-misses 0,62% | 0,70% | 0,55%

6.2 Analisi dei dati riscontrati

Dall'analisi dei dati raccolti con Perf per le tre versioni del programma (MPI, OpenMP, MPI+OpenMP) su diversi input, emergono alcune osservazioni generali sulle prestazioni:

- **Cache Misses**: La versione **OpenMP** mostra generalmente un numero più elevato di mancati accessi alla cache (cache-misses), in particolare per gli input di dimensioni maggiori (ad esempio, 100D e 100D2, con valori superiori al 50%). Al contrario, **MPI** e **MPI+OpenMP** tendono a mantenere valori più stabili e ridotti, con una gestione della memoria più prevedibile.
- **Efficienza della CPU**: La metrica instructions per cycle (*IPC*) evidenzia che la versione **MPI** è generalmente più efficiente nell'uso della CPU, superando in alcuni casi le altre versioni (ad esempio, 100D2 con 2,32 insn per cycle). **OpenMP** risulta invece meno performante per input complessi, mentre **MPI+OpenMP** mostra un comportamento intermedio, combinando in parte i vantaggi delle due tecnologie.
- **Branch Mispredictions**: La percentuale di branch-misses è più alta nella versione **MPI** per input più semplici (2D e 2D2), ma diminuisce significativamente per gli input più complessi (100D e 100D2). **OpenMP** e **MPI+OpenMP** mantengono valori generalmente più bassi, il che indica una migliore gestione del controllo di flusso.
- **Scalabilità e Input Complessi**: Per input ad alta dimensionalità (100D e 100D2), la versione **MPI** si distingue per un equilibrio tra cache misses e IPC, dimostrando una buona scalabilità. La versione **OpenMP** soffre maggiormente, evidenziando difficoltà a sfruttare pienamente i core disponibili per input molto grandi.

In generale, la versione **MPI** risulta più bilanciata in termini di efficienza CPU e gestione della memoria, mentre **OpenMP** mostra maggiore sensibilità agli input complessi. La versione combinata **MPI+OpenMP** rappresenta un compromesso, ma non eccelle in modo significativo rispetto alle altre due.

7 Analisi delle prestazioni CUDA

7.0 Premessa

I report che seguono sono stati effettuati su una Nvidia RTX 2060 super (arch: Turing, mem. bandwidth: 448 GB/s, SM: 34, CC=7.5). Per valutare le performance delle applicazioni scritte in cuda abbiamo diverse alternative:
- Nsight-System (nsys), strumento che permette l'analisi generale delle prestazioni di un programma.
- Nsight-Compute (ncu) si focalizza in modo più specifico sul kernel dell'applicazione, dandone una valutazione approfondita.
- nvprof profiler simile ad nsys, ma non è più compatibile con CC $\geq$ 8.0.

7.1 Nsight-System

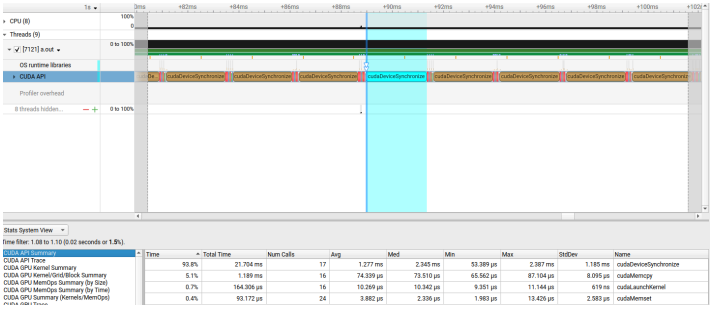
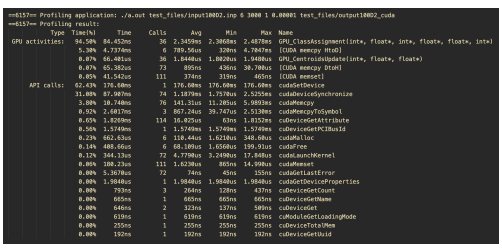


Figure 11: Report su input 100D2 con parametri 6 3000 1 0.00001 e 16 punti per thread

Come si può notare anche da questa primo report il collo di bottiglia dell'applicazione risulta essere la cudaDeviceSynchronize(), con un tempo massimo di 2.387ms e minimo di 53.389µs. Per capire il motivo della grande differenza di prestazioni possiamo servirci di *nvprof*:



Il report con *nvprof* fornisce un immagine più chiara, in particolare notiamo che

il kernel più lento (come si poteva intuire) è GPU_ClassAssignment(), al contrario GPU_CentroidsUpdate() risulta quasi trascurabile considerando l'impatto sul tempo totale (0.07%). Questo suggerisce, che nel caso volessimo apportare ulteriori ottimizzazioni, dovremmo focalizzare la nostra attenzione sul primo kernel. Per dataset più piccoli non possiamo considerare trascurabile l'overhead introdotto dalle **prime chiamate** alla libreria CUDA (cudaMalloc/Memcpy…), quindi i trasferimenti dalla memoria dell'host a quella del device.

7.2 Nsight-Compute

Con lo scopo di analizzare a fondo l'esecuzione dei kernel, possiamo utilizzare *ncu* sulla nostra applicazione e *ncu-ui* per visualizzarne i risultati.

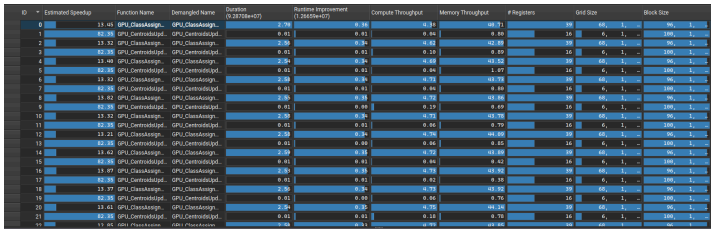


Figure 13: report ncu sull' esecuzione con parametri come in **fig.11**

Dalla fig.12 deduciamo che l'implementazione di entrambi i kernel può avere ampio margine di miglioramento. Nel caso di GPU_CentroidsUpdate(), notiamo un basso **compute** (in GFLOPS) e **memory throughput** dovuto alla semplicità del kernel:

- due letture dalla memoria,
- una operazione in virgola mobile,
- una scrittura in memoria.

GPU_ClassAssignment() invece, ha un elevato compute throughput, memory throughput ed utilizzo di registri. Entrambi i kernel hanno un *estimated speedup* elevato, questo significa che esiste la possibilità di avere un incremento significativo di prestazioni. Il kernel di aggiornamento dei centroidi viene eseguito con dimensione della griglia e dei blocchi inadatta ad un uso bilanciato degli SM e dell'esecuzione in blocchi con la suddivisione in warp, e.g. la gridSize=6 in GPU_CentroidsUpdate() a fronte di 34 SM disponibili.

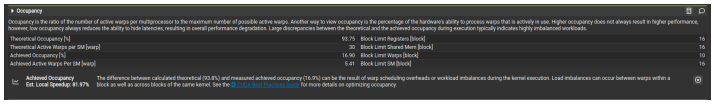


Figure 14: occupancy tra gli SM

Nonostante questo il principale collo di bottiglia rimane il kernel di classificazione e accumulo, in cui l'occupancy ottenuta è solo del 16.90% a fronte dei 93.75% teorici, dovuto probabilmente a blocchi di dimensione ridotta, dato che ogni thread esegue la classificazione di più punti (16 per questa esecuzione).