

Progetto Programmazione su Architetture Parallelle

K-Means & DB-SCAN

Andrea Mansi UniUD - 137857
Christian Cagnoni UniUD - 137690

II° Semestre 2020/2021

Contents

1	Introduzione	3
2	Soluzione	5
2.1	Rappresentazione dei dati	5
2.2	K-Means	5
2.3	DB-SCAN	6
3	Risultati	8
3.1	Tempi d'esecuzione	9
4	Compilazione	15
5	Conclusione	15
6	Mismatch dei risultati nelle versioni parallele	17
7	Possibili migliorie	17

1 Introduzione

Lo scopo del progetto è stato quello di sperimentare l'utilizzo di tecniche di programmazione parallela nell'ambito degli algoritmi di clustering. Con clustering ci si riferisce a quell'insieme di tecniche di analisi dei dati volte alla selezione e raggruppamento di elementi omogenei in uno o più insiemi di dati. Le emergenti necessità di applicare tali algoritmi a quantità di dati sempre più grandi (Big Data) rende necessario lo sviluppo di algoritmi paralleli che permettano di abbatterne i tempi di esecuzione. In questo progetto si sono presi in esame due dei più classici algoritmi di clustering: **k-means** e **db-scan**.

Segue lo pseudocodice dei due algoritmi.

K-Means:

```
1 KMEANS(Points, k){
2     // inizializzazione random dei cluster
3     clusters[k]
4     for i = 0 in k {
5         clusters[i] := random()
6     }
7     convergence := false
8     while(!convergence){
9         convergence := true
10        for each point P in Points{
11            calculateCentroid(clusters,P) // assegnamento centroide
12            if (P.Centroid != oldCentroid) then {
13                convergence := false
14            }
15        }
16        for each cluster C in clusters{
17            updateClusterCoords(Points,C) // aggiornamento coords. centroide
18        }
19    }
20 }
21
22 calculateCentroid(clusters,P){
23     for each cluster C in clusters{
24         newDistance := calculateDistance
25         if(newDistance < oldDistance){
26             oldDistance := newDistance // aggiornamento distanza
27             P.cluster := C // aggiornamento cluster
28         }
29     }
30 }
31
32 updateClusterCoords(Points,C){
33     C_Points = [] // lista di punti con cluster = C
34     for each point P in Points{
35         if(P.Cluster == C){
36             C_Points += P // aggiungo P alla lista
37         }
38     }
39 }
```

```

40 // Aggiorno le coordinate di C calcolando la media delle
41 // coordinate dei punti con cluster = C
42 C := meanCoordinates(C_Points)
43 }

```

DB-Scan:

```

1 DBSCAN(Points, distFunc, eps, minPts) {
2     C := 0 // Contatore dei cluster
3     for each point P in Points {
4         // Punto computato precedentemente
5         if label(P) != NULL then continue
6         Neighbors N := RangeQuery(Points, distFunc, P, eps) // Ricerca vicini
7         if |N| < minPts then { // Controllo densita' (< soglia)
8             label(P) := Noise // Etichetta come NOISE
9             continue
10        }
11        C := C + 1 // Prossima etichetta dei cluster
12        label(P) := C // Etichetta il primo punto
13        SeedSet S := N \ {P} // Lista vicini da espandere
14        for each point Q in S { // Processa ogni punto in Q
15            if label(Q) = Noise then label(Q) := C // Cambia da NOISE a C
16            if label(Q) != NULL then continue // Precedentemente processato
17            label(Q) := C
18            Neighbors N := RangeQuery(Points, distFunc, Q, eps) // Ricerca vicini
19            if |N| >= minPts then { // Controllo densita' (< soglia)
20                S := S + N // Aggiungi i nuovi vicini alla lista da espandere
21            }
22        }
23    }
24 }
25
26 RangeQuery(Points, distFunc, Q, eps) {
27     Neighbors N := empty list
28     for each point P in database Points { // Per ogni punto
29         if distFunc(Q, P) <= eps then { // Calcola la distanza e controlla la soglia
30             N := N + {P}
31         }
32     }
33     return N
34 }

```

2 Soluzione

La soluzione proposta mira a confrontare le versioni seriali degli algoritmi con quelle implementate tramite tecniche di programmazione parallela. Nello specifico sono state implementate, per entrambi gli algoritmi, una versione che fa uso di OpenMP e una versione che fa uso di CUDA.

Verranno ora descritte brevemente le varie versioni e le relative metodologie adottate per la loro implementazione.

2.1 Rappresentazione dei dati

I dati in input per entrambi gli algoritmi consistono in un insieme arbitrario di punti di dimensione maggiore o uguale a 2. Per rappresentare tali dati si è deciso di utilizzare una matrice di dimensione 2. Le righe rappresentano i singoli punti e le colonne le singole dimensioni. Si è inoltre aggiunta una colonna per il salvataggio dell'informazione relativa al cluster assegnato per ciascun punto.

Ad esempio, 5 punti di dimensione 3 verrebbero salvati in una matrice come da immagine:

	<i>x</i>	<i>y</i>	<i>z</i>	<i>cluster</i>
1	5.66	1.42	0.92	1
2	7.23	9.12	6.33	2
3	2.88	1.22	7.99	1
4	5.34	6.22	14.55	2
5	1.93	3.78	6.00	2

Per l'implementazione della versione CUDA si è deciso di vettorizzare tale matrice. La vettorizzazione è stata eseguita per righe.

2.2 K-Means

Si noti come il numero dei cluster (k) su k-means è fornito come input; inoltre l'inizializzazione dei cluster al primo ciclo è eseguita randomicamente.

2.2.1 K-Means Open-MP

K-means permette di applicare il parallelismo a livello dei dati. Il processamento di uno specifico punto (assegnamento di un cluster ad un dato punto) è indipendente dal risultato/processamento degli altri punti. Ogni singolo punto può quindi essere valutato in parallelo senza causare alterazioni sul risultato finale.

Nello specifico si è deciso di parallelizzare il ciclo for relativo all'iterazione dei singoli punti; assegnando quindi ad ogni thread della CPU il processamento di un singolo punto.

La suddivisione del carico avviene in maniera statica. Tale scelta è giustificata dal fatto che le singole iterazioni del ciclo for consistono nella stessa mole di lavoro; pertanto è improbabile che un thread si trovi ad eseguire meno operazioni rispetto ad un altro thread.

La fase di aggiornamento dei centroidi è stata a sua volta parallelizzata ma con schedulazione dinamica. Tale scelta è giustificata dal fatto che l'aggiornamento (ri-calcolo delle

coordinate) di due diversi centroidi potrebbe richiedere tempi molto differenti. Questo è dovuto dal fatto che in una determinata iterazione ai due cluster potrebbero venir assegnati un numero di punti molto differente.

2.2.2 K-Means CUDA

Nell'implementazione CUDA di K-means si è deciso di utilizzare la configurazione di allocazione in memoria globale di tipo pageable (per la copia dell'input su device). La scelta è giustificata dal fatto che l'utilizzo di altre configurazioni ha prodotto dei degradi delle performance dovuti all'overhead lato host.

Le due fasi parallelizzate su Open-MP sono state a sua volta parallelizzate in CUDA. Nello specifico sono stati implementati due specifici kernel per l'assegnamento del cluster per ciascun punto e l'aggiornamento di ciascun centroide. Tali kernel vengono lanciati su un numero calcolato a run-time di blocchi da 1024 thread ciascuno.

Nel kernel relativo all'assegnamento dei centroidi, ciascun thread si occupa di un singolo punto; di conseguenza il numero totale di thread coinciderà con il numero di punti totali (più i thread in eccesso dell'ultimo blocco). Nel caso del kernel di aggiornamento dei centroidi verrà lanciato un numero di thread pari alla quantità di coordinate totali (numero punti * (dimensione punti + 1)). Distinguiamo due classi di thread: la prima che fa riferimento ai thread il cui id è pari a $(dim + i * (dim + 1))$ con $i \geq 0$: che si occupa di eseguire il conteggio del numero dei punti assegnati a ciascun cluster. La seconda classe (che consiste dei thread rimanenti) si occupa di eseguire il calcolo dell'aggiornamento delle coordinate dei cluster.

2.3 DB-SCAN

Si noti come il parallelismo sui dati non sia possibile (se non tramite implementazioni alternative) in DB-SCAN. Per tale algoritmo, infatti, il risultato del processamento di un punto è dipendente dai risultati dei punti precedenti, di conseguenza non è stato possibile computare in parallelo i singoli punti. L'iterazione tra i punti deve essere eseguita serialmente ma è comunque possibile eseguire in parallelo le funzioni di supporto utilizzate nella valutazione dei singoli punti. Si ricorda anche, che DB-SCAN per sua natura risulta molto inefficiente (a confronto di algoritmi più semplici, come k-means) su una quantità elevata di punti.

2.3.1 DB-SCAN Open-MP

Le funzioni/fasi parallelizzate sono la ricerca dei vicini e il calcolo delle distanze. In entrambi i casi si è deciso di schedulare la mole di lavoro in modalità statica. La scelta è motivata dal fatto che ciascuna iterazione svolge la stessa mole di lavoro.

2.3.2 DB-SCAN CUDA

Analogamente all'algoritmo k-means si è deciso di utilizzare la configurazione di allocazione in memoria globale di tipo pageable. L'implementazione consiste in un unico kernel nel quale un singolo thread svolge il compito di gestire il parallelismo dinamico. Nello specifico, tale thread lancia i kernel che svolgono in parallelo le funzioni di supporto. Tra questi kernel, il principale è quello relativo alla ricerca dei vicini di un punto P, che viene lanciato con un numero di blocchi calcolato a run-time di dimensione 1024; ciascun thread si occupa di

un singolo punto (controlla se il proprio punto è un vicino del punto P). Ciascuno di questi thread esegue il lancio di un ulteriore kernel dedito al calcolo delle distanze (in questo caso il numero di thread è pari al numero delle dimensioni dei punti; quindi ciascun thread si occuperà di una singola coordinata).

3 Risultati

La fase di test e verifica della correttezza è svolta tramite un opportuno modulo chiamato tester.cu, implementato al fine di semplificare tali fasi. Il modulo permette di lanciare una batteria di test su uno o più algoritmi con diverse dimensioni e lunghezze in input.

Tale modulo esegue inoltre la misurazione dei tempi e permette inoltre il loro salvataggio su file.

In questa fase è inoltre possibile eseguire il controllo della correttezza degli algoritmi paralleli. Per quanto riguarda la correttezza degli algoritmi è necessario fare alcune considerazioni, [contenute nel capitolo 6](#).

Hardware utilizzato:

	Configurazione 1	Configurazione 2
CPU	i7 9700K 8C/8T 4.9Ghz ¹	Ryzen 3 3200G 4C/4T 4.0Ghz
RAM	DDR4 2x8GB 3400Mhz CL16	DDR4 8+16GB 3000Mhz CL16
GPU	RTX 3070Ti (~2040Mhz) ²	GTX 1650 Super (~1725Mhz) ³

¹ La CPU opera a 4.9Ghz sia in modalità single core che multi-core (a differenza delle specifiche di fabbrica).

² La scheda opera a un boost-clock che oscilla tra i 2025 e i 2040Mhz; le memorie (8GB - GDDR6X) operano a 10Ghz contro i 9.5Ghz di fabbrica.

³ La scheda opera a un boost-clock che oscilla tra i 1700 e i 1725Mhz; le memorie sono da 4GB GDDR6.

Si noti che durante la fase di misurazione dei tempi ci si è accertati che non si verificassero eventi di thermal throttling; che avrebbero inevitabilmente invalidato i tempi di esecuzione.

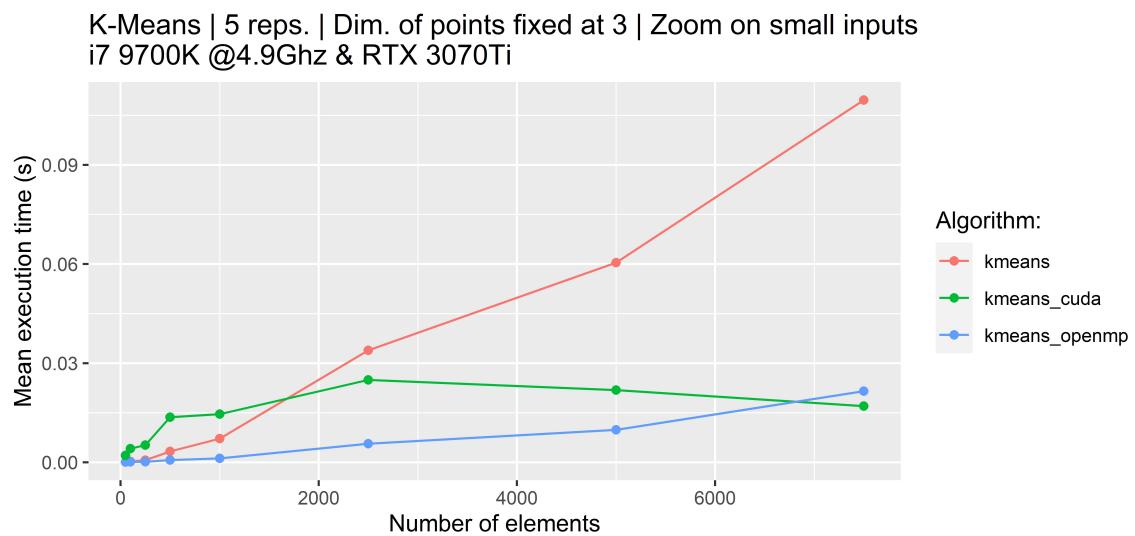
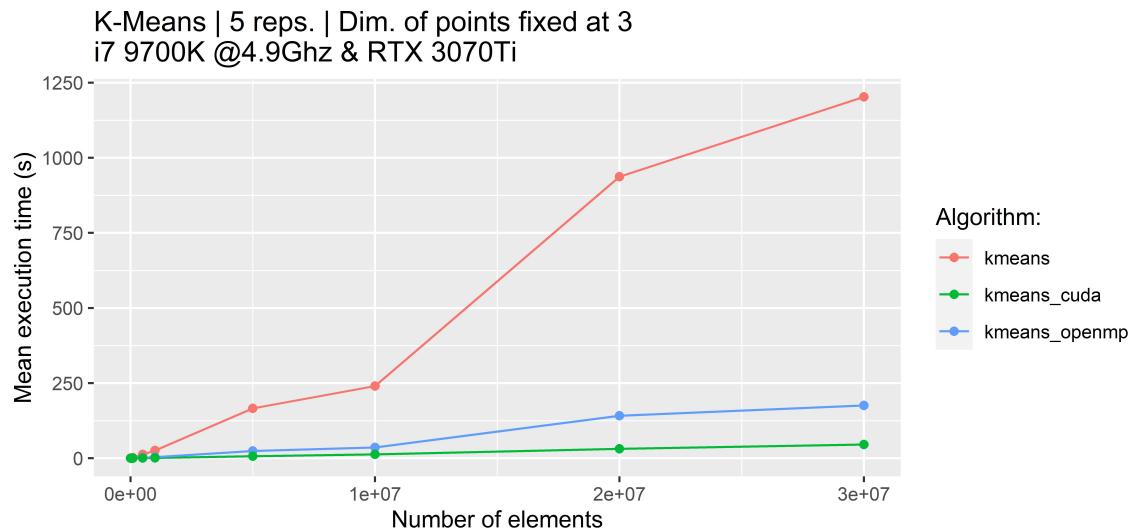
Si noti come le istanze di test sono generate a random, cercando di produrre una nuvola di punti la cui distribuzione sia più uniforme possibile. È importante notare come questi algoritmi possano differire di molto nei tempi di esecuzione nel caso in cui la distribuzione dei punti sia diversa. Ad esempio, con K-Means, una distribuzione totalmente randomica potrebbe richiedere più iterazioni per convergere rispetto a una istanza composta da 2 cluster ben definiti (se k=2).

È quindi importante marcire il fatto che un determinato tempo di esecuzione non è totalmente rappresentativo in quanto potrebbe rappresentare un caso pessimo o ottimo, sovrastimando o sottostimando il caso medio.

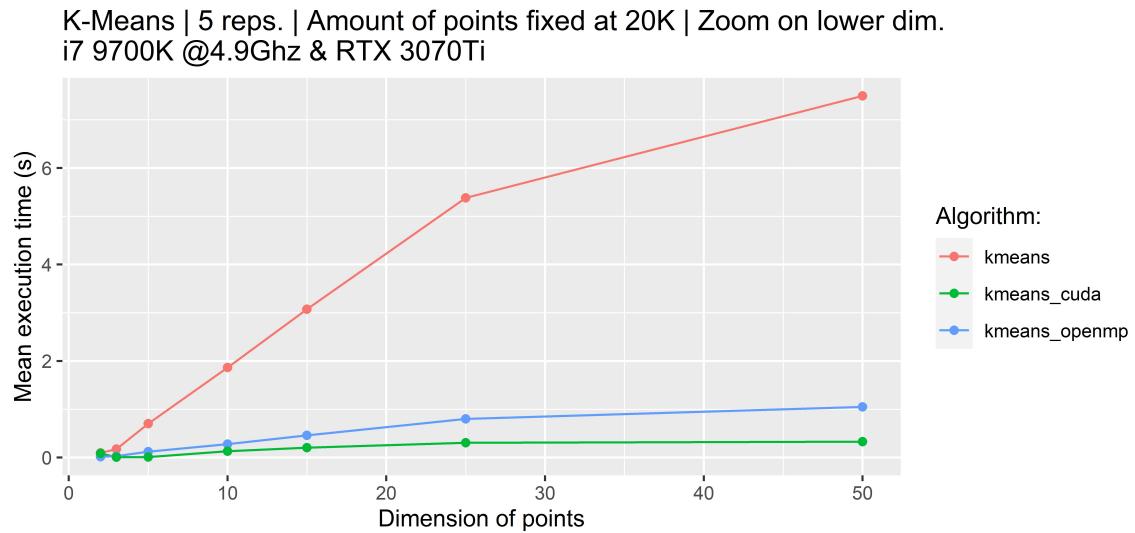
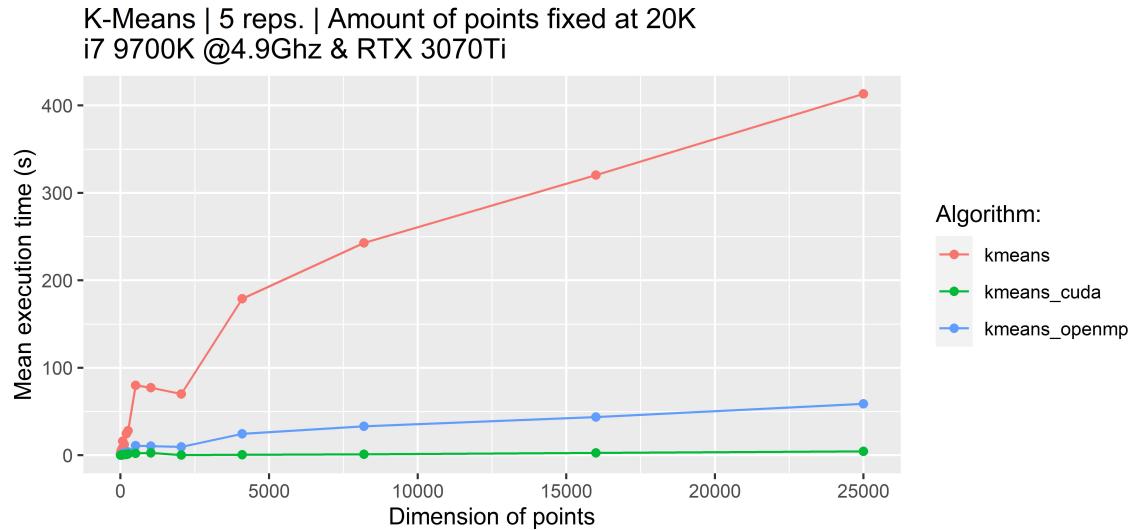
3.1 Tempi d'esecuzione

3.1.1 Tests su K-Means

Il primo test effettuato pone a un valore fisso la dimensione dei punti (punti 3D), e varia il numero di elementi. È possibile notare come la versione seriale sia molto meno efficiente delle due versioni parallele. È comunque importante notare come la versione OpenMP rimanga la più veloce su input di dimensioni contenute (circa inferiori ai 10000 elementi).

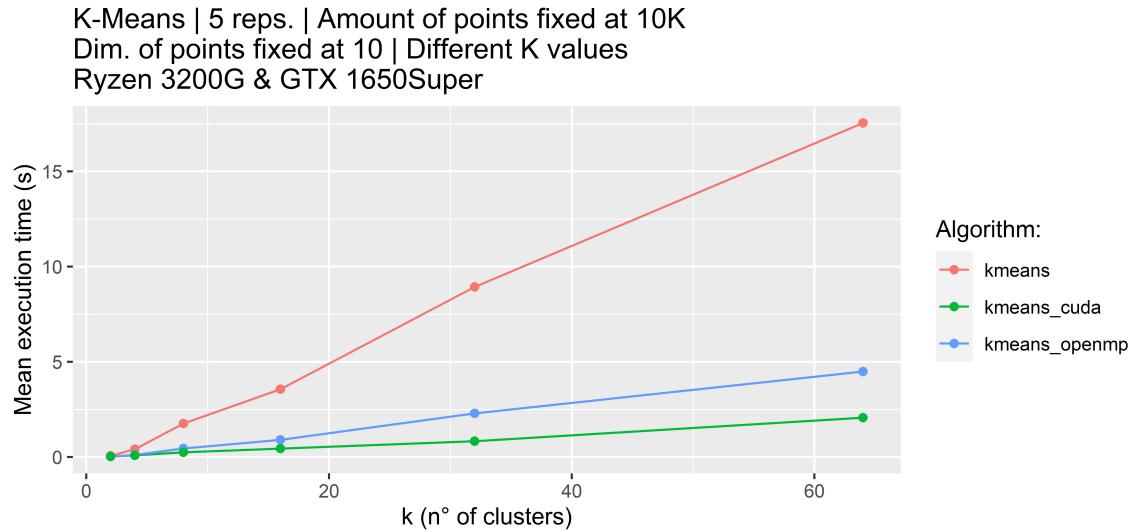


I risultati rimangono molto simili se fissiamo il numero di elementi a 20000 e variamo la loro dimensione. Anche qui è possibile notare come la versione seriale sia sempre la più lenta.



È importante notare come l'elevato throughput del dispositivo CUDA permetta di gestire punti di elevata dimensione in maniera molto più efficiente anche della versione OpenMP.

Un ultimo test dimostra come, sulla stessa istanza di input randomica da 20000 elementi di dimensione 10 un diverso valore di k influenzi i tempi di esecuzione. Valori di k più elevati aumentano il tempo richiesto per la convergenza (si noti come il comportamento potrebbe variare nel caso in cui l'input non sia una nuovola randomica di punti).

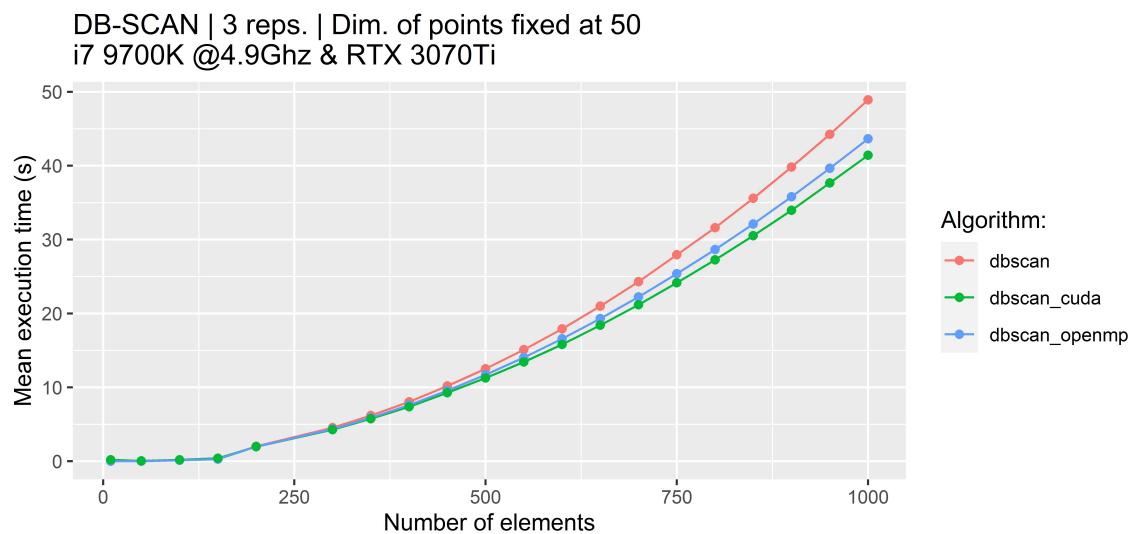
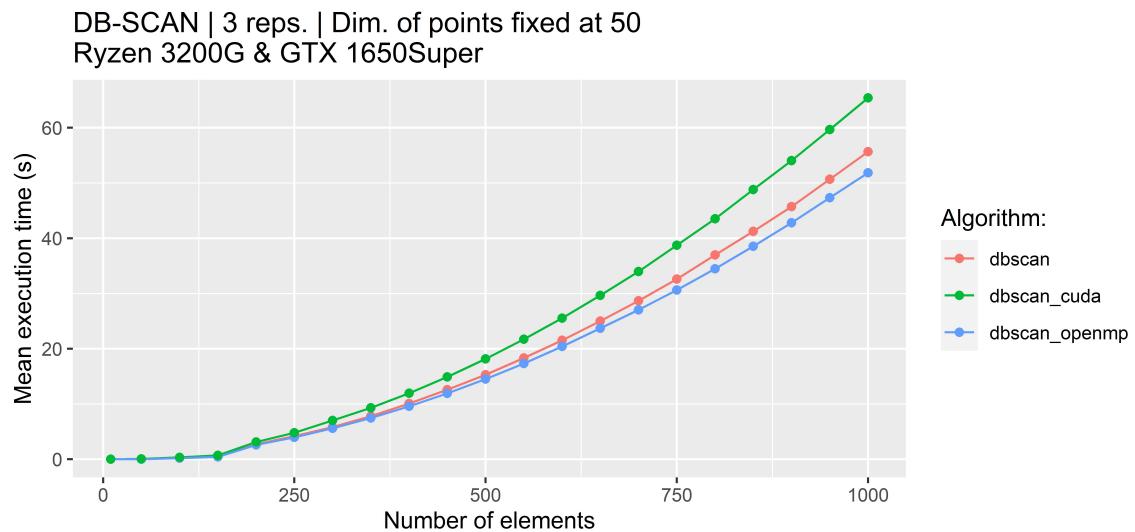


3.1.2 Tests su DB-SCAN

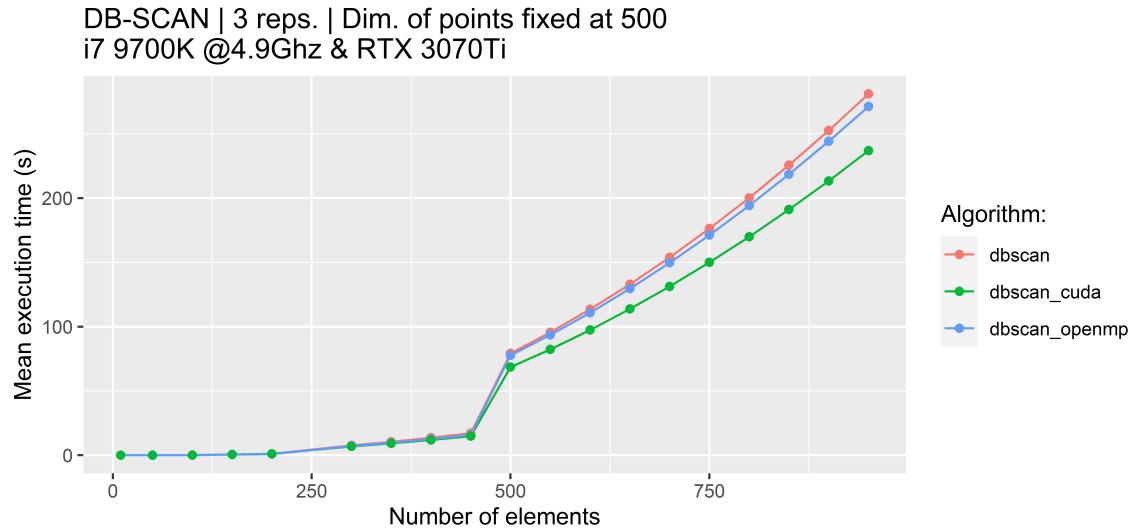
Il primo test effettuato pone la dimensione dei punti al valore fisso 50 e varia il numero di elementi dell'input. È possibile notare come l'implementazione CUDA non risulti particolarmente più efficiente, infatti la differenza risulta minima in entrambe le configurazioni hardware (nel primo caso risulta poco più inefficiente, nel secondo caso il contrario).

Questo risultato è dovuto principalmente al fatto che l'implementazione dell'algoritmo è prevalentemente seriale ad eccezione delle funzioni di supporto che vengono eseguite in parallelo.

L'esecuzione parallela di tali funzioni di supporto rende la versione OpenMP chiaramente sempre più veloce della versione seriale, mentre nella versione CUDA il vantaggio del parallelismo viene soppresso dall'overhead dovuto al lancio dei kernel e dall'inferiore velocità di esecuzione della parte seriale (eseguita da un unico thread CUDA).

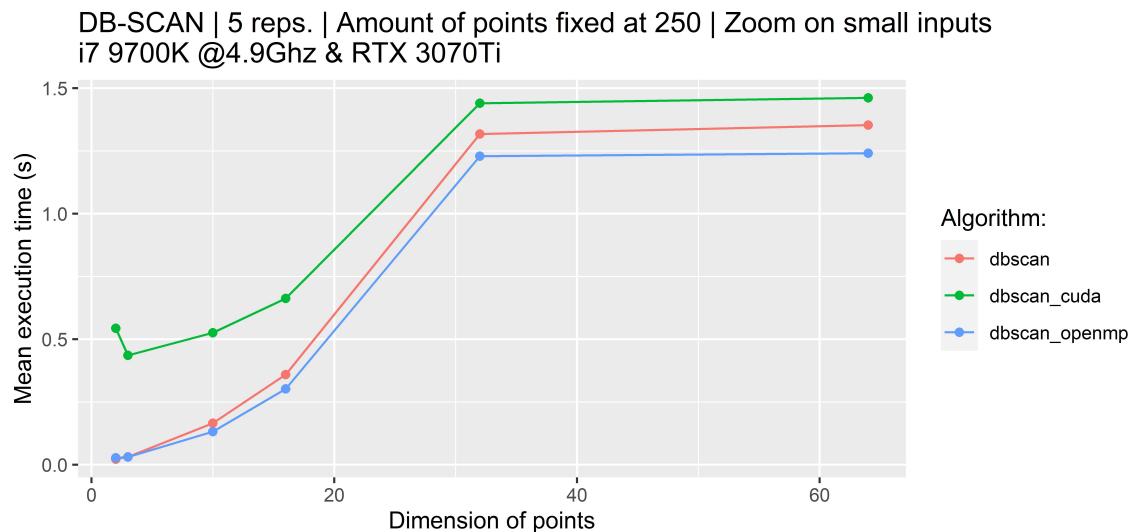
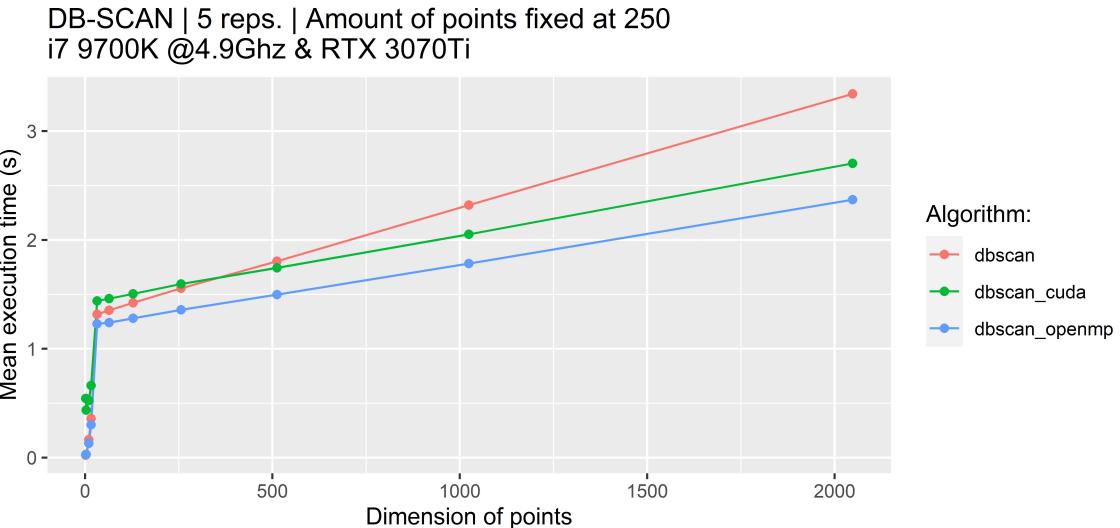


È comunque importante notare come il precedente test sfavorisca parzialmente la versione CUDA in quanto la mole di lavoro eseguibile in parallelo fosse limitata. In un secondo test in cui la dimensione dei punti è stata incrementata a 500 è possibile notare come si accentui la distanza tra la versione CUDA e quella Open-MP (a favore di quella CUDA).

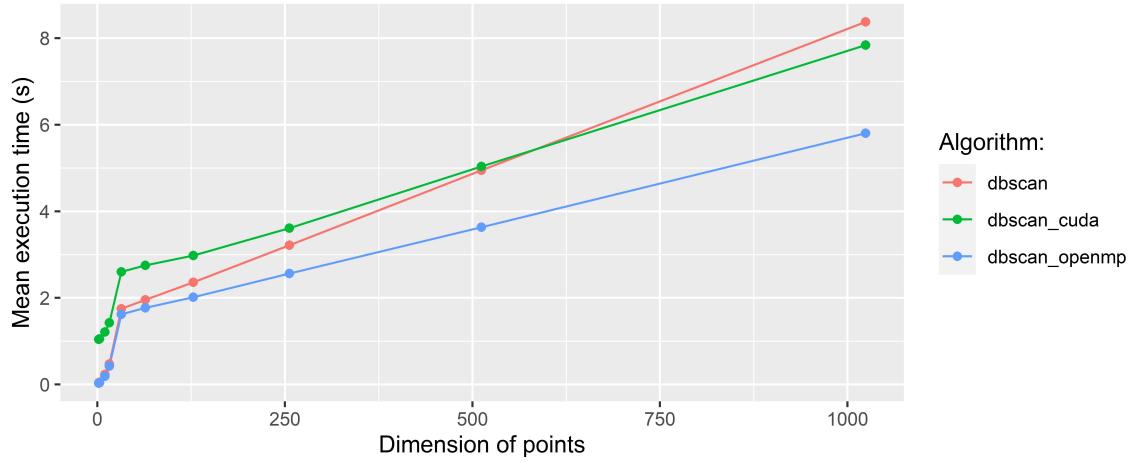


Un secondo test effettuato pone al valore fisso 250 il numero di punti, variando la loro dimensione. È possibile notare che in entrambe le configurazioni di test la versione Open-MP risulti la più veloce.

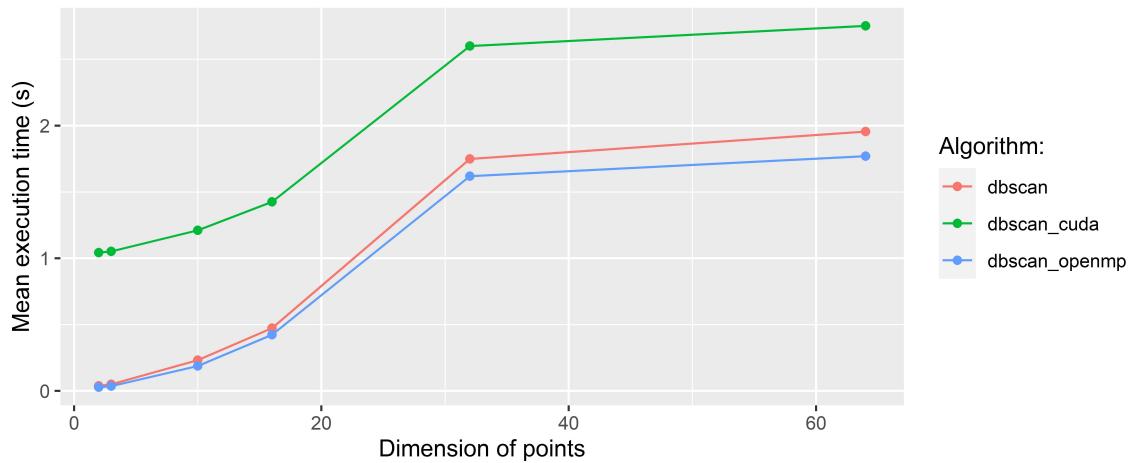
La versione CUDA risulta più veloce di quella seriale solo con dimensioni dei punti elevate. Tale comportamento è giustificato dal fatto che il vantaggio (in termini di velocità) di esecuzione della porzione seriale del codice da parte dell'host viene compensato dall'elevata mole di lavoro delle porzioni parallele (che diventano sempre più significative con l'aumento della dimensione dei punti).



DB-SCAN | 5 reps. | Amount of points fixed at 250
Ryzen 3200G & GTX 1650Super



DB-SCAN | 5 reps. | Amount of points fixed at 250 | Zoom on small inputs
Ryzen 3200G & GTX 1650Super



4 Compilazione

Al fine di semplificare la fase di compilazione viene fornito un opportuno makefile. Per poter compilare ed eseguire il codice è necessario che sulla macchina siano installati CUDA, un compilatore C++ e WindowsKit per piattaforme Microsoft Windows. È opportuno notare come potrebbe risultare necessario specificare (modificando il makefile) il path (della propria macchina) di alcune specifiche librerie CUDA etc.

5 Conclusione

I risultati ottenuti hanno sottolineato gli evidenti vantaggi nello sfruttare architetture parallele nell'implementazione di algoritmi che, come k-means, sono di natura parallelizzabili.

Allo stesso modo è risultato evidente come questa parallelizzazione non sia sempre semplice e diretta, DB-SCAN è risultato essere un esempio di algoritmo che possiede parti intrise-

camente seriali, e dunque la sua parallelizzazione non è sempre possibile (se non parzialmente) o richiede modifiche nella logica di esecuzione.

È inoltre emerso come l'utilizzo di OpenMP, richieda uno sforzo (in termini di tempo) effimero per rendere parallelo un algoritmo, ottenendo un notevole guadagno nei tempi di esecuzione. Contrariamente ad OpenMP, lo sviluppo e l'implementazione di un algoritmo parallelo su CUDA richiede un notevole sforzo di progettazione, il quale non sempre garantisce risultati migliori (a causa del fatto che gli overhead di comunicazione host-device e il lancio dei kernel potrebbero rendere l'algoritmo meno veloce).

Pertanto, è sempre ragionevole valutare un'ipotetica implementazione parallela di un algoritmo, tenendo comunque in considerazione il tempo a disposizione e le eventuali caratteristiche dell'algoritmo in questione, che potrebbe favorire l'utilizzo di alcune tecniche/architetture piuttosto che altre, o addirittura non essere parallelizzabile.

6 Mismatch dei risultati nelle versioni parallele

La natura non deterministica dell'esecuzione in parallelo nell'implementazione CUDA di k-means può portare a risultati discordanti rispetto alla versione seriale. L'algoritmo CUDA non garantisce la produzione dello stesso risultato se eseguito più volte sullo stesso input per via del fatto che i punti potrebbero venir computati in ordine diverso (i risultati discordanti dalla versione seriale non sono a loro volta costanti, ovvero il numero di "errori" può variare di volta in volta).

Inoltre, anche nelle funzioni di supporto non è sempre garantito il determinismo (ad esempio nelle somme eseguite in parallelo relative all'aggiornamento delle coordinate dei centroidi la proprietà commutativa non è garantita).

Un'ultima problematica riscontrata (che accentua il mismatch) è dovuta alla differente precisione macchina tra host e device; che in alcune particolari istanze causa la propagazione di errori (tra le coordinate) che potrebbero comportare l'assegnamento di un punto a un diverso cluster.

Per sopperire al problema è stato deciso (in fase di convalida sperimentale della correttezza) di considerare due esecuzioni corrette se i due risultati differiscono di poco (a tal proposito è stata introdotta una soglia).

7 Possibili migliorie

Entrambe le versioni CUDA dei due algoritmi sono state analizzate tramite il tool di profiling Nvidia Nsight Compute. Tale fase di analisi ha portato alla luce come i differenti kernel non sempre producessero un throughput soddisfacente. A tal proposito si potrebbe analizzare più a fondo i singoli kernel nel tentativo di migliorare le performance.