Daniele Verde 2066523

Kmeans

Per il mio progetto implemento l'algoritmo KMEANS in CUDA e MPI + OPENMP

MPI + OPENMP

Inizializzazione: per usare le due librerie ho deciso di utilizzare MPI per suddividere i punti sui vari processi, e poi parallelizzare il core dell'algoritmo con OpenMP, cioè il ciclo principale do-while con i due step iterativi dell'algoritmo (assignment e update).

Le variabili vengono inizializzate su tutti i rank, ma solo il rank 0 alloca lo spazio per i dati e legge il file di input. Sempre sul rank 0 vengono scelti K punti random che fungono da centroidi iniziali.

Successivamente faccio un broadcast sul comunicatore MPI per far condividere tra tutti i processi il numero totale di punti e delle relative dimensioni del dataset.

Per distribuire meglio il carico di lavoro, ogni processo calcola il numero di punti che dovrà gestire: prima viene calcolato un numero "base" di punti da elaborare dividendo il numero totale di punti per il numero di processi. Poi, per gestire il resto della divisione, aggiungo 1 punto ai processi il cui rank è inferiore al resto. In questo modo il carico è più bilanciato e al massimo ci saranno dei processi che gestiranno 1 solo punto in più rispetto agli altri.

Infine alloco la memoria per il vettore local_data che conterrà i punti assegnati localmente.

Per poter usare MPI_Scatterv e distribuire i dati, nel rank 0 calcolo il numero di punti che devo inviare a ciascun processo e gli offset per accedere correttamente al vettore principale dei dati. Faccio la stessa cosa anche per il vettore classMap, che contiene l'assegnamento dei punti locali ai cluster. Sempre nel rank 0 riempio il vettore dei centroidi con i punti scelti prima.

A questo punto distribuisco i punti sui processi con MPI_Scatterv e faccio il broadcast dei centroidi. Poi libero la memoria usata dal rank 0 per la posizione dei centroidi e per i dati originali, che ormai sono stati distribuiti.

Per rimanere coerente con la versione sequenziale, qui inserisco una MPI_Barrier e poi avvio il timer per il computation time.

Dopo questa fase continuo con l'inizializzazione delle variabili operative ed inoltre alloco lo spazio per pointsPerClass su tutti i processi, mentre i vettori oldCentroids e distCentroids li alloco solo sul rank 0. Dopo queste operazioni preliminari inizia il ciclo do while.

Assignment / Update step: come prima cosa apro la sezione parallela: questa ottimizzazione nasce da un errore che avevo in una versione precedente. All'inizio aprivo e chiudevo blocchi di thread OpenMP solo ed esclusivamente nelle sezioni da parallelizzare, pensando che fosse più corretto visto che poi dovevo comunque passare i dati fra i processi con MPI (esempio assignmentstep su local_data). Ma misurando i tempi risultavano pari o

superiori al tempo della versione solo MPI, come se la parallelizzazione con OpenMP non avvenisse. L'errore stava nel fatto che non stavo usando MPI_Wtime() ma la funzione clock() (come nel sequenziale) per registrare il computation time e così facendo stavo, in realtà, misurando la somma dei tempi dei vari thread.

Prima di capire questo ho provato varie soluzioni, tra cui la versione attuale, in cui i thread vengono aperti una sola volta e lasciati attivi per tutto il ciclo, e dopo aver capito perché il tempo non diminuiva quando aprivo thread OpenMP, ho notato che i tempi erano nettamente migliori nella versione attuale rispetto alle altre versioni testate.

La struttura attuale prevede quindi che uso #pragma omp master ogni volta che ho sezioni che non voglio parallelizzare (come collettive MPI, operazioni globali, ecc.), seguito da una barriera per sincronizzare i thread.

Nell'assignment step uso due variabili per i cambiamenti: una locale (changes) per i cambiamenti fra i thread OpenMP e una globale (globalChanges), che raccoglie il totale dei cambiamenti su tutti i rank. Per chiarezza e leggibilità, ho scelto di parallelizzare solo il ciclo esterno, dato che ogni punto viene processato indipendentemente dagli altri. Ogni thread openmp ha delle variabili private (class, minDist, dist) e uso una riduzione su changes con operatore di somma alla fine del ciclo. Il risultato con static mi è sembrato migliore rispetto alle altre modalità di scheduling.

Dopo la riduzione locale, faccio una barriera (per sicurezza anche se la reduction è già bloccante) e sul thread master faccio una riduzione MPI su globalChanges (con operatore somma su rank 0).

Dopo passo all'update step: prima copio i centroidi attuali su oldCentroids (solo su rank 0), poi azzero i vettori centroids e pointsPerClass. Riutilizzo centroids per calcolare la somma parziale dei punti locali e dei punti per classe, in pratica gli uso come dei "centroidi locali".

Per aggiornare i nuovi centroidi lavoro sulla classMap locale, parallelizzando il ciclo esterno. Dato che ci sono dipendenze RAW (read-after-write) tra la classe assegnata e la scrittura nei centroidi, per evitare race condition ho usato le istruzioni #pragma omp atomic sulle scritture in centroids e pointsPerClass. Come scheduling ho scelto static perché dai test fatti ha mostrato un miglior bilanciamento del carico fra i thread.

Avevo pensato di evitare le atomic e usare delle variabili private per ogni thread per provare ad avere performance migliori. Però questo avrebbe comportato il dover allocare vettori molto grandi: K * int * num_threads per pointsPerClass e K * features * double * num_threads per i centroidi. Secondo me è un overhead troppo pesante, soprattutto quando K è grande, quindi usare le atomic mi è sembrato un buon compromesso.

A questo punto apro un blocco omp master che lascio aperto fino alla fine dei passaggi. In versioni precedenti avevo provato a parallelizzare anche i cicli all'interno di questa sezione, ma avevo registrato un peggioramento delle prestazioni.

In questa parte faccio due MPI_Allreduce: uno per i centroidi e uno per i punti per classe. In questo modo ottengo su tutti i processi la somma globale dei centroidi e il totale dei punti per classe. Dopo normalizzo i centroidi e ottengo i nuovi centroidi su ogni processo.

Poi, solo su rank 0, dove ho salvato i vecchi centroidi, calcolo la distanza massima tra vecchi e nuovi centroidi.

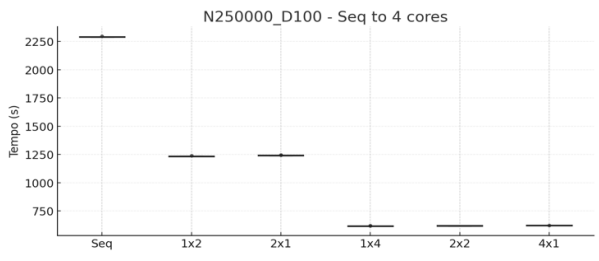
Infine aggiorno la variabile isExit: se una delle 3 condizioni di uscita è soddisfatta (numero minimo di cambiamenti, massimo numero di iterazioni, soglia sulla distanza dei centroidi), isExit viene settato a 0. Qui finisce la sezione master e dopo broadcasto isExit a tutti i processi chiudo la sezione master. Se isExit = 0, tutti i thread e processi usciranno dal ciclo do while.

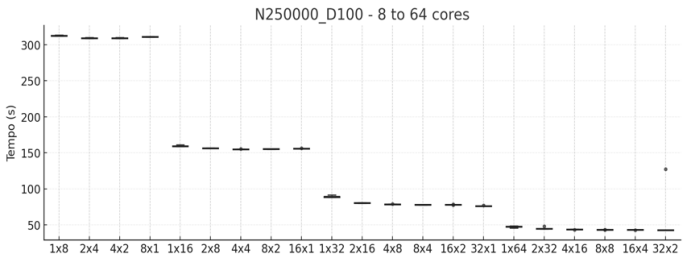
(Anche in questa versione ho provato a rimuovere il calcolo della radice quadrata sulla computazione di ogni distanza come ho fatto in CUDA ma qui non ho misurato nessun beneficio quindi ho deciso di usare la funzione del calcolo della distanza euclidea originale)

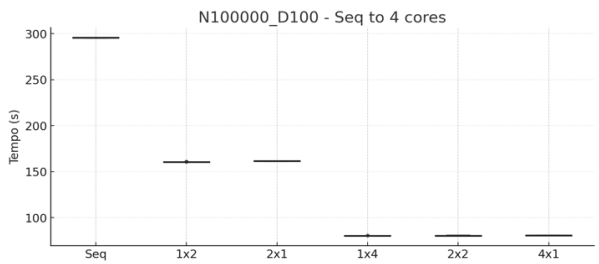
Fase finale: ora che ogni processo MPI ha terminato di elaborare i propri punti, vado a prendere il tempo di calcolo effettivo. Successivamente, sul processo 0 raccolgo tutte le classMap locali dai vari processi utilizzando MPI_Gatherv. Gli argomenti della funzione (ovvero i vettori con il numero di elementi per processo e i relativi offset) li avevo già calcolati nella fase di inizializzazione.

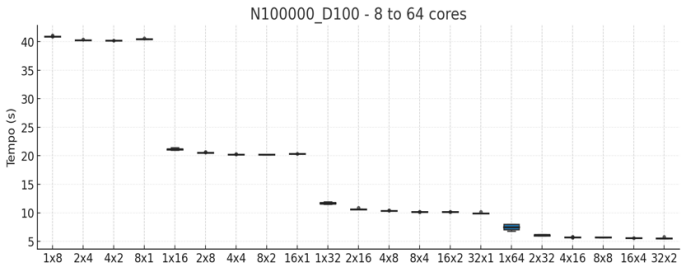
Dopo aver completato la Gatherv, scrivo sul file di output la classificazione finale dei punti e come ultimissimo passo, libero tutte le aree di memoria allocate durante l'esecuzione.

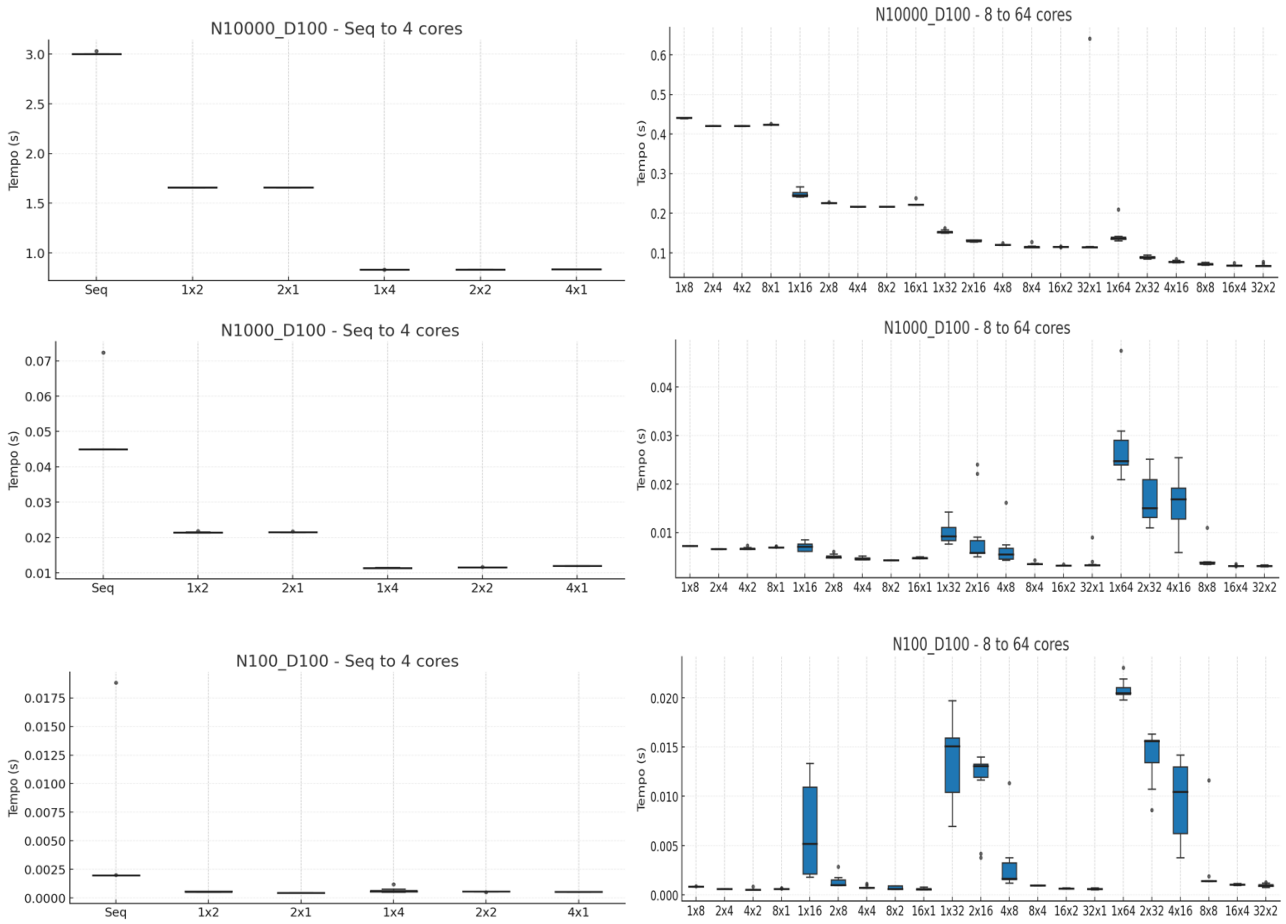
Test: tutti i seguenti risultati provengono da test effettuati sul cluster dell'università. Sono stati utilizzati 5 file di input, ciascuno contenente punti con 100 feature, ma con un numero variabile di punti: 250000, 100000, 10000, 1000, 100 e con K = √N (N=# di punti). Poiché l'account multicore dispone di nodi da 64 CPU, ho eseguito test utilizzando tutte le combinazioni possibili di processi e thread, eseguendo 10 run per configurazione e prendendo il valore mediano. (#MPIx#OpenMp). I parametri di uscita sono stati: maxIt = 500, minChang = 0,01 e Precision = 0,01.











Correttezza: inizialmente ho riscontrato discrepanze tra i risultati delle due implementazioni, che avevo attribuito alla non-associatività delle somme in virgola mobile. In realtà l'incongruenza derivava dal fatto che la versione sequenziale usava il tipo float, mentre quella parallela impiegava il tipo double. Uniformando anche la versione sequenziale a double, le differenze sono scomparse.

Speedup ed efficienza:

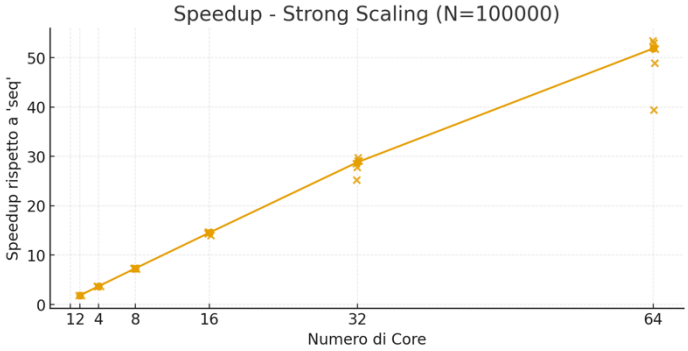
Risultati per N100_D100 (K=10)				Risultati per N1000_D100 (K=32)				Risultati per N10000_D100 (K=100)				Risultati per N100000_D100 (K=316)			
config	runtime	speedup	efficiency	config	runtime	speedup	efficiency	config	runtime	speedup	efficiency	config	runtime	speedup	efficiency
seq	0,00198050	1,000	1,000	seq	0,04498150	1,000	1,000	seq	3,00236350	1,000	1,000	seq	295,53869950	1,000	1,000
1x2	0,00056150	3,527	1,763	1x2	0,02143900	2,098	1,049	1x2	1,65626750	1,812	0,906	1x2	160,55477900	1,840	0,920
2x1	0,00045250	4,376	2,188	2x1	0,02149750	2,092	1,046	2x1	1,65722650	1,811	0,905	2x1	161,44780750	1,830	0,915
1x4	0,00057350	3,453	0,863	1x4	0,01139000	3,949	0,987	1x4	0,83201000	3,608	0,902	1x4	80,35730100	3,677	0,919
2x2	0,00055500	3,568	0,892	2x2	0,01154650	3,895	0,973	2x2	0,83267550	3,605	0,901	2x2	80,42346950	3,674	0,918
4x1	0,00053100	3,729	0,932	4x1	0,01199650	3,749	0,937	4x1	0,83555450	3,593	0,898	4x1	80,78661800	3,658	0,914
1x8	0,00083900	2,360	0,295	1x8	0,00727400	6,183	0,772	1x8	0,44061750	6,813	0,851	1x8	40,91511250	7,223	0,902
2x4	0,00062050	3,191	0,398	2x4	0,00660750	6,807	0,850	2x4	0,42083400	7,134	0,891	2x4	40,25678350	7,341	0,917
4x2	0,00053650	3,691	0,461	4x2	0,00666250	6,751	0,843	4x2	0,42058050	7,138	0,892	4x2	40,21286900	7,349	0,918
8x1	0,00060750	3,260	0,407	8x1	0,00692450	6,495	0,811	8x1	0,42365500	7,086	0,885	8x1	40,46728700	7,303	0,912
1x16	0,00521050	0,380	0,023	1x16	0,00712550	6,312	0,394	1x16	0,24569100	12,220	0,763	1x16	21,10694050	14,001	0,875
2x8	0,00099550	1,989	0,124	2x8	0,00485700	9,261	0,578	2x8	0,22597000	13,286	0,830	2x8	20,53576450	14,391	0,899
4x4	0,00074000	2,676	0,167	4x4	0,00460650	9,764	0,610	4x4	0,21622700	13,885	0,867	4x4	20,20794500	14,624	0,914
8x2	0,00062350	3,176	0,198	8x2	0,00430100	10,458	0,653	8x2	0,21677600	13,850	0,865	8x2	20,19679000	14,632	0,914
16x1	0,00056900	3,480	0,217	16x1	0,00473500	9,499	0,593	16x1	0,22138000	13,562	0,847	16x1	20,32065850	14,543	0,908
1x32	0,01508050	0,131	0,004	1x32	0,00921950	4,878	0,152	1x32	0,15179200	19,779	0,618	1x32	11,71888250	25,219	0,788
2x16	0,01305650	0,151	0,004	2x16	0,00581850	7,730	0,241	2x16	0,13121200	22,881	0,715	2x16	10,63593300	27,786	0,868
4x8	0,00164150	1,206	0,037	4x8	0,00548000	8,208	0,256	4x8	0,11994700	25,030	0,782	4x8	10,34606950	28,565	0,892
8x4	0,00095450	2,074	0,064	8x4	0,00353350	12,730	0,397	8x4	0,11436600	26,252	0,820	8x4	10,16080000	29,086	0,908
16x2	0,00065450	3,025	0,094	16x2	0,00323900	13,887	0,433	16x2	0,11525900	26,048	0,814	16x2	10,16650350	29,069	0,908
32x1	0,00060400	3,278	0,102	32x1	0,00327700	13,726	0,428	32x1	0,11383800	26,374	0,824	32x1	9,92376300	29,780	0,930
1x64	0,02046850	0,096	0,001	1x64	0,02472150	1,819	0,028	1x64	0,13559700	22,141	0,345	1x64	7,49304500	39,441	0,616
2x32	0,01558700	0,127	0,001	2x32	0,01509250	2,980	0,046	2x32	0,08848650	33,930	0,530	2x32	6,03785700	48,947	0,764
4x16	0,01045150	0,189	0,002	4x16	0,01687900	2,664	0,041	4x16	0,07710500	38,938	0,608	4x16	5,71270400	51,733	0,808
8x8	0,00142250	1,392	0,021	8x8	0,00385850	11,657	0,182	8x8	0,07063750	42,503	0,664	8x8	5,68270600	52,006	0,812
16x4	0,00105550	1,876	0,029	16x4	0,00311050	14,461	0,225	16x4	0,06745600	44,508	0,695	16x4	5,57733100	52,989	0,827
32x2	0,00092200	2,148	0,033	32x2	0,00316150	14,227	0,222	32x2	0,06693200	44,856	0,700	32x2	5,53366900	53,407	0,834

Risultati per N250000_D100 (K=500)			
config	runtime	speedup	efficiency
seq	2289,99239450	1,000	1,000
1x2	1235,11250850	1,854	0,927
2x1	1241,36915500	1,844	0,922
1x4	617,95206100	3,705	0,926
2x2	619,01876200	3,699	0,924
4x1	621,93388600	3,682	0,920
1x8	312,53224100	7,327	0,915
2x4	309,28973850	7,404	0,925
4x2	309,31316850	7,403	0,925
8x1	311,20427700	7,358	0,919
1x16	159,28559550	14,376	0,898
2x8	156,44127900	14,638	0,914
4x4	155,03976700	14,770	0,923
8x2	155,18892750	14,756	0,922
16x1	155,95860050	14,683	0,917
1x32	88,83024350	25,779	0,805
2x16	80,31601700	28,512	0,891
4x8	78,62299150	29,126	0,910
8x4	77,82370650	29,425	0,919
16x2	77,81085200	29,430	0,919
32x1	75,92776850	30,160	0,942
1x64	47,30781100	48,406	0,756
2x32	44,62745400	51,313	0,801
4x16	43,67502100	52,432	0,819
8x8	43,41011500	52,752	0,824
16x4	43,05590450	53,186	0,831
32x2	42,73588500	53,584	0,837

Da questi risultati possiamo notare che: all'aumentare del numero di punti (da 100 fino a 250000), l'effetto del parallelismo diventa sempre più evidente. Per i grandi dataset il tempo di esecuzione cala drasticamente all'aumentare del numero di core e lo speedup migliora mentre per i piccoli input notiamo che l'overhead di gestione supera nettamente i vantaggi della computazione parallela non apportando nessun vantaggio significativo, anzi degradandone le prestazioni in determinati casi.

In tutti i test possiamo notare che le configurazioni che prevedo più di 8 thread openMp presentano un calo di prestazioni rispetto alle altre configurazioni con pari numero di core totali. Questo può dipendere dalle atomic nell'update che aumentano con molti thread nello stesso processo, e overhead di sincronizzazione OpenMP. Per questo le migliori configurazioni sono tipicamente MPI "più alto" × OMP "più basso".

Efficienza superiore a 1: Nei primi test ottenevo risultati anomali con efficienza > 1, cosa che in linea teorica, non dovrebbe essere possibile (il calcolo è stato fatto cosi: T_seq / (n_cpu * T_par)). Questo è capitato perché avevo eseguito più job contemporaneamente sullo stesso nodo, introducendo contese che hanno falsato i tempi. Ripetendo i test senza co-scheduling (quest'ultimi sono quelli inseriti nella relazione) l'efficienza > 1 è rimasta solo sul dataset da 100 punti che immagino sia dovuto alla piccola dimensione dell'input che riesce a beneficiare della cache.



Strong scalability: Nel caso del test con 100000 punti, il grafico di strong scaling evidenzia una crescita dello speedup quasi linearmente fino a 32 core, oltre la crescita inizia a essere più lenta a causa dei costi di overhead che aumentano all'aumentare dei thread/processi.

In pratica, raddoppiare da 32 a 64 core non raddoppia più le prestazioni: lo speedup passa da circa 29 a circa 53 (efficienza da 0,93 a 0,83 circa).

Weak scalability: Usando i dati sull'efficienza si vede che all'aumentare del problema e del numero rimane complessivamente alta: a 4 core è 0,973 (quasi ideale), a 8 core scende a 0,892, a 32 core torna intorno a 0,908. A 64 core compare un degrado più visibile, coerente con il maggior peso dei costi di coordinamento come già detto prima. Però va anche notato che nei miei test K cresce con N ($K = \sqrt{N}$), e nonostante il carico cresca di N*K tra un gradino e l'altro della diagonale l'efficienza rimane molto alta!

	N100	N1000	N10000	N100000	N250000
seq	1.0	1.0	1.0	1.0	1.0
4 core (2x2)	0,892	0,973	0,901	0,918	0,924
8 core (4x2)	0,461	0,843	0,892	0,908	0,925
32 core (8x4)	0,064	0,397	0,820	0,908	0,919
64 core (8x8)	0,021	0,182	0,664	0,812	0,824

Daniele Verde 2066523

CUDA

Struttura generale: nell'algoritmo parallelo provo a mantenere la struttura simile a quella del sequenziale, soprattutto come nomenclatura delle variabili, però apporto delle ottimizzazioni. Nella prima fase, quella di inizializzazione, cerco di precalcolarmi tutto quello che posso (allocazione variabili, spazio su device e host, etc) in modo da non dover perdere tempo e ridurre il più possibile i calcoli nelle iterazioni. Ogni kernel è lanciato con una configurazione personalizzata (calcolata in fase iniziale) per ottenere il massimo della parallelizzazione. Nei due kernel più significativi, che sono anche quelli che consultano di più i centroidi, ho deciso di salvare i centroidi nella shared memory. Essendo la shared memory limitata, questi kernel prevedono di fare tiling su K, caricando nella shared solo K elementi alla volta, in modo da poter eseguire l'algoritmo parallelo anche con K molto grandi. Ma comunque questo non mi metteva al sicuro da tutte le casistiche. Per questo ho deciso di fare anche tiling su D (features), in caso 1xD centroide fosse maggiore della shared memory. Nel pratico, all'inizio controllo se (int * k (allineato a multipli di 8) + 1* features * double + 1024) < shared memory, (questo calcolo deriva dal fatto che in un kernel metto anche la classmap nella shared oltre ai centroidi quindi controllo quello come limite massimo), se così non fosse si attivano una serie di configurazioni per i kernel diverse che fanno anche tiling su D (questi kernel portano lo stesso nome ma finiscono con tileD).

(Per effettuare i test di strong scaling ho usato un'altra versione quasi uguale, la v8, dove faccio striding sui punti in modo da limitare i thread che creo).

Inizializzazione: siccome nel kernel updateKernelSumAndCount, utilizzo shared memory per mantenere sia i contatori dei punti per cluster (int) sia i valori intermedi dei centroidi (double). Per evitare accessi non allineati alla parte in double, allineo il numero di K interi a un multiplo di 8 (uso la shared anche con tipi di dato diversi).

Per sfruttare al meglio la parallelizzazione, ho optato, per gestire i dati e anche i centroidi, con un pattern SoA con padding, invece del formato AoS come nel codice sequenziale. Questo perché con SoA i thread accedono in modo coalescente alla memoria globale. Ho inoltre applicato padding sul numero di punti in modo che il caricamento dei dati su device sia allineato a multipli di warp.

Nella fase di inizializzazione per adattarsi meglio a qualsiasi GPU, il codice utilizza funzioni dedicate (makeGridBlock1D, makeGridBlock2DTileK, makeGridBlock3DTileKTileD) che calcolano dinamicamente i parametri di lancio (numero di thread per blocco e dimensione della griglia), in base alle caratteristiche hardware del dispositivo, ai vincoli sui registri per thread, e alle esigenze specifiche di ciascun kernel (che illustro più avanti). Tutte le funzioni creano sempre blocchi monodimensionali, la cosa che varia è la dimensionalità della griglia

perché la uso per rappresentare dimensioni logiche diverse (punti, cluster, features). Ho reputato che fosse migliore per le mie esigenze aggiungere assi sulla griglia e non nei blocchi in modo da essere libero dal vincolo di massimo di thread per blocco.

La funzione makeGridBlock1D costruisce una griglia 1D scegliendo un numero di thread per block come multiplo di warp e rispettando i limiti di thread per blocco/SM e di registri per blocco adattandoli dinamicamente in base al device, limitando ad un massimo di 512 thread per block (questa ottimizzazione mi ha dato degli ottimi risultati sul cluster).

Se non devo fare tiling anche su D per il kernel updateKernelSumAndCount uso makeGridBlock2DTileK, che genera una griglia 2D in cui l'asse Y viene suddiviso in tile da tileK centroidi. Altrimenti uso makeGridBlock3DTileKTileD, che estende il tiling anche sull'asse delle dimensioni, aggiungendo alla griglia 2D una terza dimensione Z che rappresenterà il numero di tileD. Questa strategia consente di mantenere i dati caricati in shared memory gestibili, migliorando gli accessi alla memoria globale anche in presenza di dataset ad alta dimensionalità.

Assignment / Update step: Nel do while principale uso i seguenti kernel che ho usato per parallelizzare le operazioni:

assignmentKernel: utilizzo un thread per ogni punto, carico una mattonella di K alla volta e stabilisco a quale cluster appartiene il punto. Alloco i centroidi nella shared memory, in maniera collaborativa tra i thread, e confronto le distanze senza fare la radice quadrata in modo da risparmiare operazioni. Anche usare gli operatori ternari mi ha fatto risparmiare tempo computazionale. Alla fine faccio un'atomic per aggiornare changed.

updateKernelInit: utlizzo K*D thread che collaborano per settare a 0 auxCentroid e pointPerClass

updateKernelSumAndCount: utilizzo P*n_tileK thread, e il kernel deve essere lanciato in una griglia 2D dove l'indice X indica i punti e l'indice Y indica il numero di tileK (nel caso della versione tileD utilizza anche la terza dimensione moltiplicando i thread *n_tileK *n_tileD). Qui nella shared ho sia le mattonelle di centrodi che tutta, o una parte, della classmap. Ogni thread somma tramite atomic le feature dei punti nel centroide corrispondente e il punto in pointsperclass. Dopo di che un solo thread per blocco, sempre tramite atomic, somma su centroidi e pointsperclass globali.

updateKernelDivideKbyN: utilizzo K thread, questo kernel lo uso per normalizzare i centroidi ed ottenerne la media.

updateKernelDistanceK: utilizzo K thread per calcolare la distanza tra i vecchi ed i nuovi centroidi, i parametri passati sono array con vecchi centroidi, array con nuovi centroidi, lunghezza dei vettori e vettore dove salvare la distanza.

La funzione blockReduceMax ha all'interno due kernel: questa funzione mi fa ottenere la distanza massima dal vettore delle distanze in tempo logaritmico. Per farlo utilizzo due kernel

e un vettore d'appoggio di dimensione K/maxthreadperblocco. Prima riduco all'interno dei blocchi e salvo i massimi per blocco nel vettore d'appoggio, poi riduco nuovamente e ottengo il risultato finale. Il limite della seconda riduzione è che il numero di elementi da ridurre (quindi K) deve essere minore o uguale al numero massimo di thread per blocco supportato dal device, in quanto il kernel finale esegue la riduzione completa in un singolo blocco. Per questo progetto ho lasciato questo limite non prevedendo test con K cosi elevati (considerando ambienti di test con maxThreadsPerBlocks elevati).I kernel all'interno sono:

blockReduceMax: utilizzo K thread, questo kernel effettua la riduzione intrablocco shiftando a sinistra i risultati in modo da far lavorare i warp in maniera ottimizzata

finalReduceMax: utilizzo la potenza di 2 minima che racchiude il numero di elementi della prima riduzione e riduco sempre shiftando a sinistra

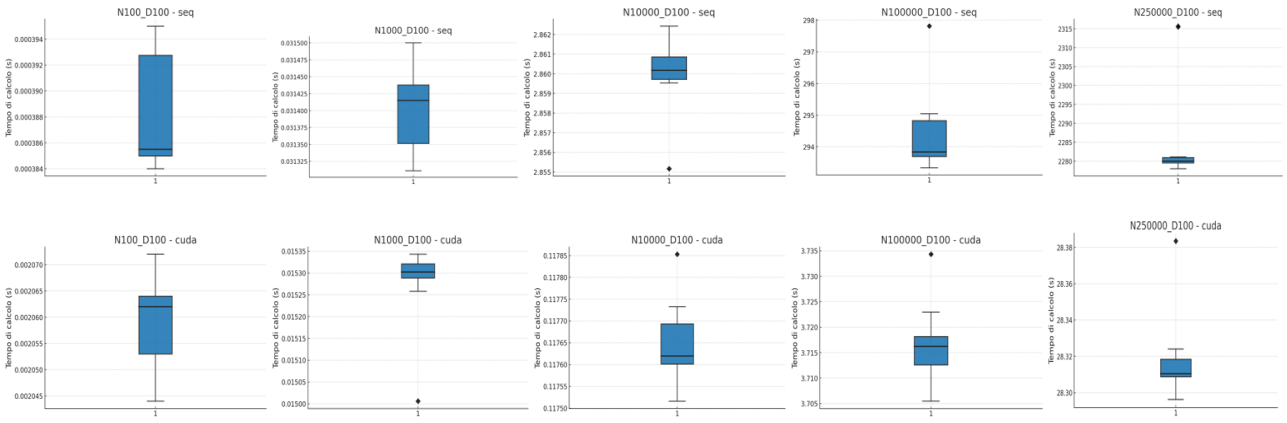
Dal do-while uscirò se vengono raggiunte le condizioni di uscita, a quel punto trasferisco dal device la classmap finale, la scrivo su un file e libero la memoria

Test: Sono stati utilizzati 5 file di input, ciascuno contenente punti con 100 feature, ma con un numero variabile di punti: 250000, 100000, 10000, 1000 e 100 con K = √N (N=numero di punti). I test sono stati effettuati sul cluster dell'università, ho considerato il numero di punti come il massimo della paralizzazione nelle parti più impegnative dell'algoritmo, ogni test è stato eseguito 10 volte e poi ho preso il mediano. (Nei kernel piu complessi lancio 1 thread per ogni punto) I parametri di uscita sono stati: maxIt = 500, minChang = 0,01 e Precision = 0,01.

KMEANS_cuda_par_v9.cu è la versione "normale".

KMEANS_cuda_par_v8.cu è la versione che include un argomento aggiuntivo che limita la creazione di thread.

Risultati esecuzioni dei test:



Dopo vari test mi accorgo che i nodi presentano delle differenze in termini di prestazioni.

Tutti i test sono stati rifatti ed eseguiti solo sul nodo 110 del cluster, i risultati riportati si riferiscono a questi ultimi.

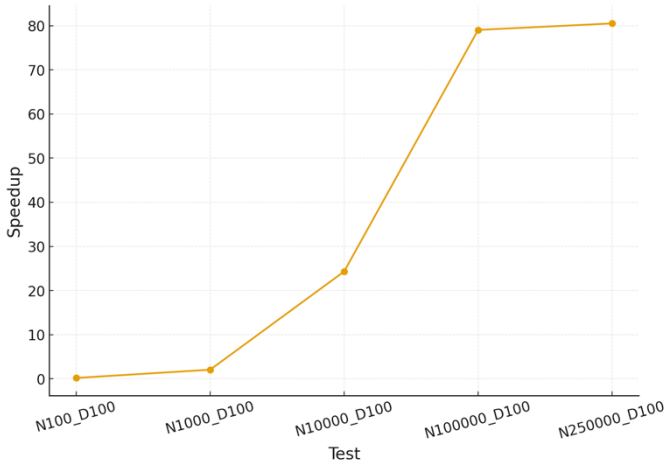
Correttezza: utilizzando il tipo double non si osservano divergenze tra gli output della versione sequenziale e quella parallela. In precedenza, il codice sequenziale fornito usava il tipo float e, mantenendo tale tipo, per dataset di grandi dimensioni si verificavano discrepanze nelle assegnazioni(circa l'1.5/2% su 100k punti), sia con la versione sequenziale sia tra le varie run cuda, con un incremento proporzionale alla dimensione del dataset.

Queste differenze derivano dalla gestione della virgola mobile e, in particolare, dalla non associatività della somma: l'ordine con cui vengono eseguite le operazioni influenza il risultato, poiché parte della componente decimale può essere troncata e propagare piccoli errori numerici. L'esecuzione parallela su GPU introduce inoltre un certo grado di non determinismo, dato che i thread non sommano i contributi sempre nello stesso ordine.

L'uso del tipo double riduce drasticamente questo problema aumentando la precisione numerica, ma per dataset ancora più grandi potrebbero comunque presentarsi differenze dovute alle stesse ragioni.

Speedup:

Lo speedup evidenzia come l'algoritmo parallelo in CUDA raggiunga performance significative solo a partire da dataset di dimensioni medio-grandi. Nei test con un numero ridotto di punti, l'overhead di lancio dei kernel riduce l'efficacia della parallelizzazione (nel caso N100 la versione parallela è addirittura più lenta).

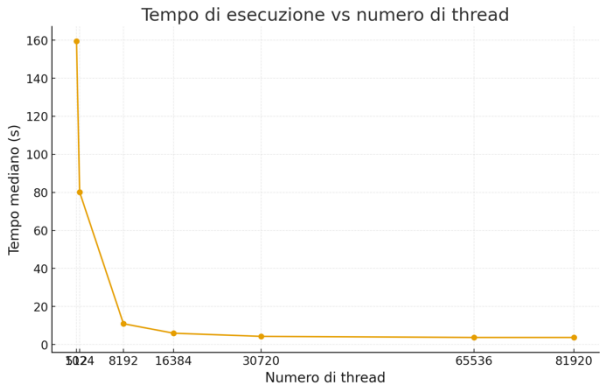


Quando N è grande lo speedup si stabilizza intorno agli 80x. Aumentare N non mi da più miglioramenti(nonostante significhi aumentare i thread nel mio algoritmo): questo significa che sono in un caso di memory-bound, il tempo non è limitato "da quanti calcoli faccio", ma da quanto velocemente riesco a spostare i dati da/verso la memoria globale, altrimenti avrei avuto uno speed-up ancora crescente all'aumentare di N. Questo comportamento (plateau) è tipico di un collo di bottiglia di memoria e il test successivo mi indica che inizio ad esserlo già da 45000 punti.

test	sequenziale (s)	cuda (s)	speedup
N100_D100	0.0003855	0.002062	0.1869544131910766
N1000_D100	0.031415	0.015302	2.052999607894393
N10000_D100	2.8601739999999998	0.11761949999999999	24.317175298313632
N100000_D100	293.8373795	3.7162595	79.06804664744213
N250000_D100	2279.9815815	28.310456000000002	80.5349649436943

In questa tabella sono riportati i mediani delle varie run

Strong scaling: in questi test ho usato KMEANS_cuda_par_v8.cu limitando i thread sul test file N100000_D100.inp, questi sono i risultati ottenuti su 10 run per ciascuna configurazione:



Dal grafico si vede che la versione CUDA mostra un comportamento di strong-scaling quasi ideale diminuendo i costi computazionali all'aumentare dei thread, superata però la soglia dei 30000 thread la pendenza della curva cambia nettamente. A 30720 il tempo scende solo a circa 4,30 secondi, e addirittura raddoppiando di nuovo i thread fino a 65 536 il guadagno si riduce a qualche decimo di secondo, con un plateau che si estende fino ai 100 000 thread.

Questo plateau può essere spiegato dal fatto che ci sono limiti inevitabili: i vincoli fisici della GPU come la banda di memoria. Infatti nell'assignment kernel, devo leggere molti byte ma poi faccio poche operazioni su quei byte e ne carico subito altri per scorrere i punti, avendo così pochi FLOP per byte (bassa intensità artmetica).

(Grazie a questo test, che voleva simulare un test di strong scaling anche se non vado a modificare le risorse hardware vere e proprie, mi sono accorto che il limite di thread per blocco che avevo impostato a 512 per questa versione per limitare il numero di registri che mi faceva fallire il kernel mi dava un notevole speedup e quindi ho deciso di adottare lo stesso limite anche nella versione v9. Tra le differenze ho notato che con questa config un SM riesce a gestire 2 blocchi e non soltanto 1 nei kernel più pesanti). Variando da 1024 a 512 thread per blocco ho avuto un ulteriore riprova di questo, se le ALU fossero state già occupate al 100%, aggiungere più blocchi/SM non avrebbe aumentato il throughput. Con 2 blocchi in esecuzione mentre uno attende per sincronizzarsi o per caricare dalla memoria l'altro può essere eseguito nascondendo la latenza.

Thread	Tempo mediano (s)	Speedup
512	159.4638175	1.8426586300682284
1024	80.099324	3.668412725930122
8192	10.9375475	26.865015169076976
16384	5.9656355	49.255000494079795
30720	4.3021985	68.29935427200766
65536	3.7093315000000002	79.21572377664276
81920	3.7071955	79.26136603801984

Altri test: per curiosità ho eseguito anche delle prove forzando l'algoritmo a entrare nel branch con inShared = false. In questo caso ho osservato una differenza di circa 0,088 s sull'input da 100000 punti e di circa 1,241 s sull'input da 250000 punti. Però, considerando le variazioni di tempi dovute all'esecuzione su nodi diversi, non è semplice stabilire se questi guadagni siano effettivamente significativi e dovuti al branch o al nodo. (questi test gli ho condotti prima di decidere di usare solo il nodo110 per i test). Provando invece a limitare ancora il massimo di thread per blocco a 256 ho ottenuto un degrado delle prestazioni di circa 0,5 s su 100000, segnale che il limite non è l'occupancy e che con il limite a 512 thread, con 2 blocchi per SM, ha già abbastanza warp da eseguire.