Contents

K-means

Elia Belli 2006305, Federico Fantozzi 2047034

1 Introduzione

K-means è un algoritmo iterativo per la suddivisione di un insieme di oggetti in K gruppi, detti cluster, in questo documento mostriamo 4 implementazioni parallele dell'algoritmo realizzate con le seguenti tecnologie:

- MPI
- OpenMP
- CUDA
- MPI + OpenMP

2 Algoritmo

Analizzando la versione sequenziale dell'algoritmo, si distinguono 3 sezioni principali all'interno di ogni iterazione:

1. **Classificazione dei punti** : per ogni punto si identifica il cluster di appartenenza in base al centroide a distanza minore, e si tiene conto del numero di cambiamenti dall'iterazione precedente (condizione di terminazione).
2. **Ricalcolo dei centroidi** : in base alla classificazione dei punti appena svolta, si aggiornano le coordinate dei centroidi come media delle posizioni dei punti appartenenti al loro cluster.
3. **Calcolo MaxDist** : si calcola lo spostamento di ogni centroide rispetto all'iterazione precedente e si prende il massimo (condizione di terminazione).

2.1 Possibili modifiche

Riportiamo di seguito alcune modifiche all'algoritmo di base che abbiamo considerato ma infine scartato:

- Nella classificazione dei punti ci interessa calcolare il centroide più vicino, ma per fare ciò è sufficiente applicare il teorema di Pitagora $a^2 = b^2 + c^2$ e confrontare i quadrati delle distanze, infatti se $d_1 < d_2 \Rightarrow d_1^2 < d_2^2$, questo ci permette di evitare il calcolo di $sqrt()$ che in genere è un'operazione costosa: tale modifica risultava in un output differente dalla versione base a causa di approssimazioni differenti.

- Prendendo ispirazione dalla variante K-means++ abbiamo pensato di applicare un'euristica alla scelta dei centroidi iniziali per convergere in meno iterazioni, la modifica è stata scartata poiché l'algoritmo base che abbiamo preso in considerazione utilizza sempre lo stesso insieme di centroidi iniziali.

3 Implementazioni Parallele

L'algoritmo ha complessità $O(lines \cdot samples \cdot K)$ ad ogni iterazione, con $lines$ generalmente $>> K$ e $samples$, quindi il focus principale è stato scalare sul numero di punti.

3.1 MPI

3.1.1 Partizione dell'input

I punti vengono partizionati tra i processi in modo che ogni processo lavori su un blocco di punti contiguo di dimensione $|punti|/|processi|$, se ci sono punti rimanenti vengono distribuiti tra i processi in modalità round-robin.
Ogni processo lavorerà sul blocco a lui assegnato per tutte le iterazioni, senza bisogno di conoscere l'assegnazione degli altri blocchi. Solo al raggiungimento della condizione di terminazione, tramite la collective `Allgather` il processo con rank 0 recupererà le porzioni di $classMap$ dai vari processi e le scriverà sul file di output.
Anche i centroidi sono partizionati allo stesso modo, quindi ogni processo svolge i passi (2) e (3) solamente per i propri.

Partizionare i dati in questo modo consente ad un processo di proseguire con il passo successivo dell'algoritmo utilizzando solo i dati calcolati da lui stesso nel passo precedente. Inoltre, questa organizzazione permette di trasmettere in modo asincrono i dati necessari per i successivi punti di sincronizzazione durante il passaggio da un passo all'altro.

3.1.2 Comunicazione

Terminato il passo (1) un processo esegue due `Iallreduce`, una su $pointsPerClass$ necessario per completare il calcola della media, e una su $changes$, nel frattempo può procedere a sommare le coordinate dei propri punti in $auxCentroids$.

Adesso abbiamo un punto di sincronizzazione: viene eseguita una `Allreduce` affinché ogni processo possieda i dati di *auxCentroids* e si attende che la `Iallreduce` su *pointsPerClass* precedentemente iniziata sia completata, in modo che ogni processo possa procedere con la divisione delle cordinate dei centroidi a lui assegnati.

Al termine del passo (2), ogni processo ha già calcolato le coordinate per il propri centroidi. Pertanto, possiamo eseguire una `Iallgather` su *auxCentroids* sovrapponendola al passo (3), anziché eseguirla separatamente alla fine. Poiché il passo (3) dipende sia dal valore di *centroids* che quello di *auxCentroids*, il risultato della `Iallgather` va collezionato su un array di appoggio in modo da non influenzare l'esecuzione del (3).

Al termine del passo (3), i processi vengono sincronizzati e viene calcolata la condizione di terminazione dell'algoritmo. Prima di proseguire, una `wait` rispetto alla `Iallgather` di *auxCentroids* e una `memcpy` garantiscono che i dati relativi alle nuove posizioni dei centroidi siano correttamente calcolati.

3.1.3 Performance

Strong Scaling: abbiamo eseguito le misurazioni per lo strong scaling su vari input, raddoppiando la dimensione ad ogni test, in modo da osservare come il programma si comportasse su piccoli e grandi input.

- Per la legge di **Amdahl** ogni programma ha porzioni che non possono essere parallelizzate, quindi aumentando i processi raggiungiamo un plateau.
- Raggiunto il plateau l'efficienza inizia a calare a causa dell'overhead della comunicazione tra processi, in linea con lo **Universal Scalability Model**[1] che estende Amdahl in modo da considerare i limiti di banda e latenza.

Weak Scaling: lo scaled speedup segue la legge di **Gustafson** fino a 16 processi con efficienza sopra al 90%, poi inizia a calare per i seguenti motivi:

- All'aumentare della dimensione dell'input aumentano i dati da trasferire, quindi la latenza nella comunicazione cresce.
- All'aumentare dell'input l'accesso ai dati avviene in cache di livello inferiore.

[1] Universal Scalability Model

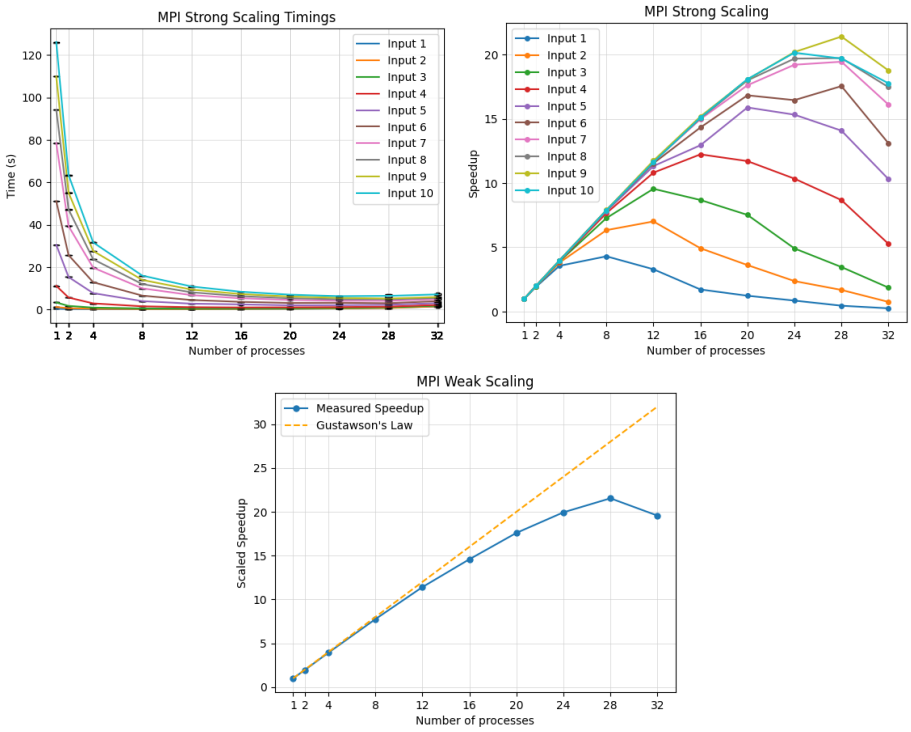

Figure 1: Misurazioni eseguite su input con punti da 100 dimensioni e parametri 100 150 0.01 0.01

	Numero di Punti									
Processi	3.125	6.250	12.500	25.000	37.500	50.000	62.500	75.000	87.5000	100.000
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99
2	0.96	0.98	0.99	0.99	1.00	0.99	1.00	1.00	1.00	0.99
4	0.89	0.95	0.98	0.98	0.99	0.99	0.99	0.99	0.99	0.99
8	0.54	0.80	0.91	0.96	0.98	0.98	0.99	0.99	0.99	0.99
12	0.27	0.58	0.80	0.90	0.94	0.96	0.97	0.97	0.98	0.97
16	0.11	0.31	0.54	0.76	0.81	0.90	0.94	0.94	0.95	0.94
20	0.06	0.18	0.38	0.60	0.80	0.84	0.88	0.90	0.90	0.90
24	0.04	0.10	0.20	0.43	0.64	0.69	0.80	0.82	0.84	0.84
28	0.02	0.06	0.12	0.31	0.50	0.63	0.70	0.70	0.76	0.70
32	0.01	0.02	0.06	0.16	0.32	0.41	0.50	0.55	0.60	0.55

Table 1: Efficienza ricavata dalle misure in 1

3.2 OpenMP

La versione OpenMP è stata sviluppata partendo da una versione ottimizzata del codice sequenziale, seguendo inizialmente le idee utilizzate nella versione MPI.

Abbiamo scelto di utilizzare un'unica regione parallela che racchiude l'intero `do-while`, evitando così la creazione e la distruzione dei threads ad ogni iterazione.

3.2.1 Partizione dell'input

A differenza di MPI, dove il partizionamento dell'input è stato gestito manualmente, in OpenMP la suddivisione del carico di lavoro tra i threads viene automatizzata tramite le direttive `pragma`.
Utilizzando lo scheduling di default, che corrisponde a `static` con un `chunk size` pari a $|punti|/|processi|$, il carico viene distribuito in modo uniforme tra i threads, replicando la stessa strategia di partizionamento adottata nella versione MPI.

3.2.2 Sincronizzazione e reductions

A differenza di MPI, abbiamo il vantaggio della memoria condivisa che ci permette di diminuire i punti di sincronizzazione ed eliminare l'overhead della comunicazione tra processi.

Sincronizzazioni: gli unici punti di sincronizzazione sono presenti tra la somma e la divisione per il calcolo di $auxCentroids$, ovvero quando i threads passano dal lavorare sui propri punti ai propri centroidi, e dopo il calcolo di $maxDist$.
Tramite un `pragma omp single` un'unico thread è incaricato di valutare le condizioni di terminazione e preparare gli array condivisi per la prossima iterazione.

Reductions: all'interno della regione parallela utilizziamo dei `pragma omp for` con clausola `reduction` (e clausola `nowait` se la sincronizzazione è superflua) per effettuare le riduzioni su $auxCentroids$ e $pointsPerClass$.
Abbiamo notato che usando K e $samples$ grandi si ha stack overflow, questo è dovuto dal fatto che OpenMP utilizza lo stack per creare gli array di appoggio privati al thread; per ovviare al problema abbiamo implementato una reduction allocando degli array privati sull'heap dovendo però introdurre delle operazioni atomiche: l'analisi è stata poi condotta su un numero molto inferiore di centroidi quindi abbiamo deciso di tornare alla versione iniziale a causa del rallentamento da parte delle atomiche.

3.2.3 Performance

Per l'analisi delle performance di OpenMP abbiamo deciso di limitarci al numero di core fisici disponibili, senza sfruttare l'hyperthreading.

Utilizzo della Cache: abbiamo analizzato l'utilizzo della cache tramite `perf` su una CPU locale con 4 core fisici, in particolare ci interessava capire quanti cache miss avessimo e la possibile causa.
Abbiamo riscontrato un aumento degli `L1_dcache_load_misses` e `LLC_load_misses` proporzionale all'aumento di dimensione dell'input (con thread fissati a 4), così abbiamo avanzato due ipotesi:

- **False Sharing**: le porzioni di memoria condivisa *data* e *centroids* non possono causare il fenomeno siccome utilizzati in sola lettura e mai modificati, mentre la scrittura su $auxCentroids$ avviene solo privatamente per poi eseguire la reduction, quindi l'ipotesi è stata scartata.

- **Cache Limitata**: le performance tendono a deteriorarsi al crescere dell'input perché non è più possibile avere tutti i dati a disposizione in cache, questo è risultato più evidente in locale avendo delle cache L1 da 64 KiB , L2 da 512 KiB ed L3 da 3 MiB che costringevano ad accedere quasi sempre in DRAM.
  Considerando invece una CPU del cluster e l'input 10, dal peso di 40 MB, esso può invece risiedere interamente nella cache L3 da 128 MiB e se partizionato tra 32 threads rientra quasi interamente nella L1 da 1 MiB.

Questa osservazione vale anche per la versione MPI siccome la causa non è la memoria condivisa, inoltre ci fa capire che un fattore limitante per il nostro programma è la quantità di cache disponibile e quindi una versione distribuita del programma fa al caso nostro se vogliamo elaborare input ancora più grandi.

Strong Scaling: il programma lavorando su memoria condivisa non segue lo Universal Scalability Model come MPI, mostrando uno strong scaling migliore; il plateau causato dalla legge di Amdahl risulta poco evidente, ciò dimostra che la porzione sequenziale, in fase di computazione, è molto bassa e che il problema è altamente parallelizzabile.

Weak Scaling: lo scaled speedup segue la legge di **Gustafson** con efficienza superiore al 90% fino a 28 threads.

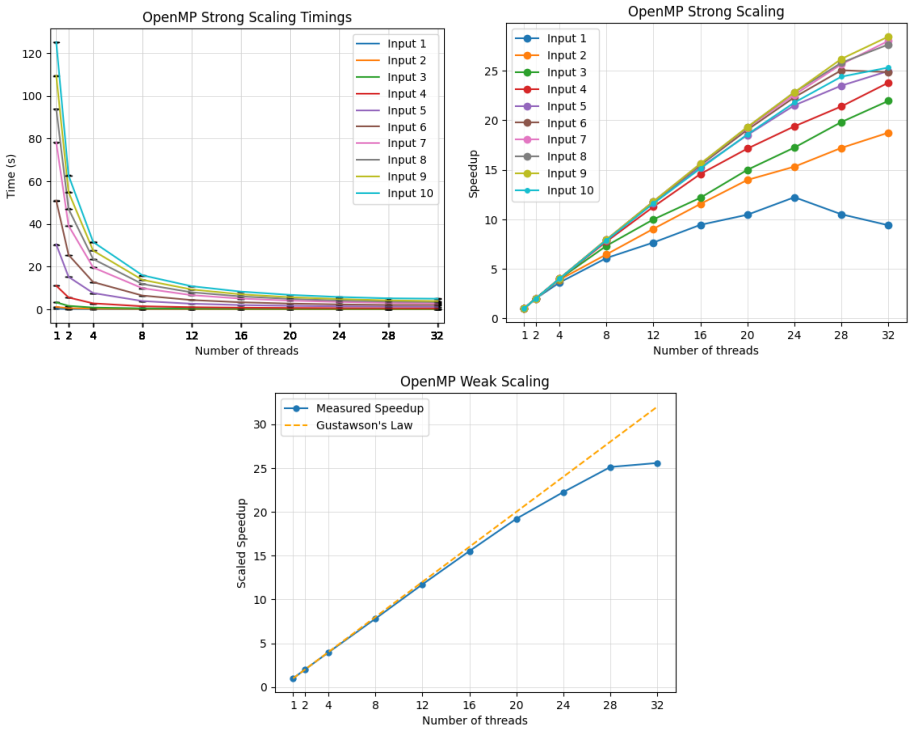

Figure 2: Misurazioni eseguite su input con punti da 100 dimensioni e parametri 100 150 0.01 0.01

	Numero di Punti									
Processi	3.125	6.250	12.500	25.000	37.500	50.000	62.500	75.000	87.5000	100.000
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
4	0.90	0.97	0.99	0.99	0.99	1.00	1.00	0.99	0.99	0.99
8	0.76	0.81	0.92	0.97	0.98	0.98	0.99	0.99	0.99	0.99
12	0.64	0.75	0.83	0.94	0.97	0.97	0.98	0.98	0.98	0.98
16	0.59	0.72	0.76	0.91	0.95	0.96	0.98	0.97	0.98	0.98
20	0.52	0.70	0.75	0.86	0.92	0.95	0.96	0.97	0.97	0.97
24	0.51	0.64	0.72	0.81	0.89	0.93	0.94	0.95	0.95	0.95
28	0.38	0.61	0.71	0.76	0.84	0.89	0.92	0.92	0.92	0.94
32	0.29	0.59	0.69	0.74	0.78	0.77	0.88	0.86	0.86	0.89

Table 2: Efficienza ricavata dalle misure in 2

3.3 CUDA

Il programma CUDA è diviso in 3 kernels per garantire la sincronizzazione tra blocchi, infatti per l'esecuzione di ognuno è necessario che tutti thread abbiamo terminato l'esecuzione del precedente.

3.3.1 Kernel 1 : kmeansClassMap

Ad ogni thread è assegnato un punto, su cui esegue le seguenti operazioni:

1. Calcola la classe di appartenenza del punto, aggiornando $classMap$ in global memory e $localPointsPerClass$ in shared memory.

2. Controlla se la classe è cambiata dall'iterazione precedente, in tal caso aggiorna un contatore $localChanges$ in shared memory: aggiornare i dati prima in shared memory, sebbene necessiti comunque di una somma atomica, riduce l'attesa per accedere alla sezione critica siccome solo i threads del blocco concorrono alla scrittura, inoltre l'accesso è più veloce.

3. Somma le coordinate del punto al centroide di appartenenza in $auxCentroids$ in global memory: a causa dell'aleatorietà della classificazione, per effettuare una somma locale al blocco sarebbe necessario avere per intero un $auxCentroids$ in shared, ma ciò è possibile solo per piccoli valori di K e $samples$.

Una volta che tutti i thread del blocco hanno eseguito i passi, si procede con la somma di $localPointsPerClass$ e $localChanges$ rispettivamente in $pointsPerClass$ e $changes$ in global memory.

Memoria Condivisa: la prima versione del programma lavorava su $centroids$ e $data$ in global memory, siccome $centroids$ è acceduto da tutti i threads abbiamo deciso di caricarlo completanente in shared all'inizio del kernel riducendo gli accessi in global memory di $blockSize$ volte : nonostante lo speedup, questo approccio è limitato dalla dimensione della shared memory che si satura velocemente all'aumentare di K e $samples$, per questo motivo siamo passati al **tiling** con lo stesso effetto ma con un'occupazione nettamente minore della shared.
A questo punto avevamo a disposizione la shared per ridurre gli accessi a $data$ in global memory, ogni thread lavora su un solo punto quindi ad ognuno è riservata una porzione della shared in cui caricarlo, dovendo caricare $blockSize$ punti in shared è necessario ridimensionare la dimensione del blocco a dovere, ciò viene fatto da una funzione lato host all'avvio del programma: con questa tecnica abbiamo ridotto di $(K \cdot samples)\times$ gli accessi di ogni thread in global memory, riscontrando uno speedup del $2\times$ per l'intero programma.

Tiling: nella classificazione del proprio punto, ogni thread calcola la distanza da ogni centroide nello stesso ordine, ciò consente ad ogni iterazione (sui centroidi nel kernel) di caricare in shared un centroide da cui tutti devono calcolare la distanza, per poi sostituirlo col successivo alla prossima iter-

azione, ciò rende necessaria la sincronizzazione nel blocco rallentando leggermente l'esecuzione rispetto alla versione precedente ma aggirando la limitazione della shared memory.
Siccome il tiling è realizzato in modo che ogni thread del blocco carichi una coordinata del centroide in shared, si possono verificare casi in cui molti thread non contribuiscono al caricamento (es. se $blockSize = 128$ e $samples = 2$, 126 threads vanno in stallo), per questo motivo tramite un parametro `tileSize` al kernel avevamo pensato di caricare un numero centroidi in shared memory che andasse a massimizzare i threads impiegati, riducendo così anche il numero di sincronizzazioni: questa idea è stata infine scartata in quanto necessitava l'utilizzo dell'operatore modulo, che abbiamo riscontrato essere alquanto inefficiente su GPU.

3.3.2 Kernel 2 : kmeansCentroidsDiv

Una volta che tutti i blocchi hanno eseguito il Kernel 1 possiamo concludere il calcolo della media eseguendo la divisione delle coordinate dei centroidi con i relativi punti per classe.

Inizialmente abbiamo pensato di assegnare un centroide ad ogni thread, cosi da effettuare un solo accesso in global memory per recuperare il valore in $pointsPerClass$ e salvarlo in un registro riducendo di $samples\times$ gli accessi in global; siccome il numero di centroidi è generalmente basso, ciò portava all'esecuzione di pochi blocchi, così la maggior parte delle SM erano inattive in questa fase.
Quindi si è deciso di distribuire le coordinate anziché i centroidi ai vari thread, in modo che ognuno esegua la divisione delle coordinate a partire dal suo $globID$ con un offset di $gridSize$, ottenendo uno speedup del $3\times$ per il kernel rispetto alla versione precedente.

3.3.3 Kernel 3 : kmeansMaxDist

Calcolate le coordinate dei nuovi centroidi dobbiamo calcolare la condizione di terminazione per lo spostamento minimo dei centroidi.

Ad ogni thread è assegnato un centroide per cui calcola lo spostamento dall'iterazione precedente, a questo punto deve confrontarlo col massimo e aggiornarlo nel caso sia maggiore, ciò richiede l'impiego di un'operazione max atomica che viene effettuata prima su una variabile in shared memory e alla fine nel kernel in global memory.

AtomicMax: CUDA non fornisce un'operazione `atomicMax` su float, ma è possibile realizzarla a partire da `atomicCAS` che permette di verificare se il valore è stato modificato (compare) e sostituirlo col nuovo (swap) nel caso sia rimasto lo stesso, nel nostro caso il valore sostituito è il nuovo massimo valutato sull'ultimo valore letto.

3.3.4 Performance

Di seguito riportiamo lo speedup rispetto al sequenziale misurato su una NVIDIA Quadro RTX6000 del cluster e una NVIDIA GeForce MX920 che abbiamo poi utilizzato per l'analisi dei kernel.

Utilizzando `nvprof` abbiamo osservato che il programma passa il 99.89% del tempo nel Kernel 1, ne abbiamo quindi analizzato le metriche per realizzare il **roofline model**[2], riscontrando che l'applicazione è *compute-bound*; a seguire alcune ipotesi per motivare il risultato ottenuto:

- Utilizzo elevato di operazioni atomiche
- Mancato utilizzo di strided reductions
- FMAD disabilitato

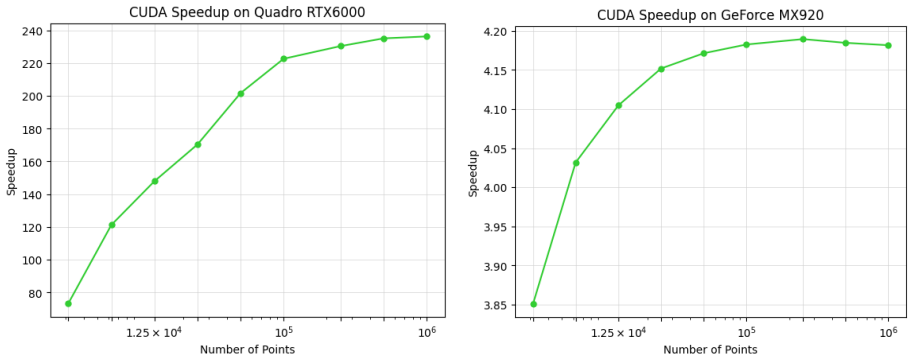

Figure 3: Speedup misurato su input da 3125 a 1.000.000 di punti (100D) con parametri 100 150 0.01 0.01

Input	Punti	Dimensioni	Occupancy	GFLOP/s
1	100.000	100	0.05	15.1
2	200.000	50	0.15	30.4
3	400.000	25	0.68	45.4
4	5.000.000	2	0.99	25.1

Table 3: Input utilizzati per il roofline model

[2]Roofline Performance Model

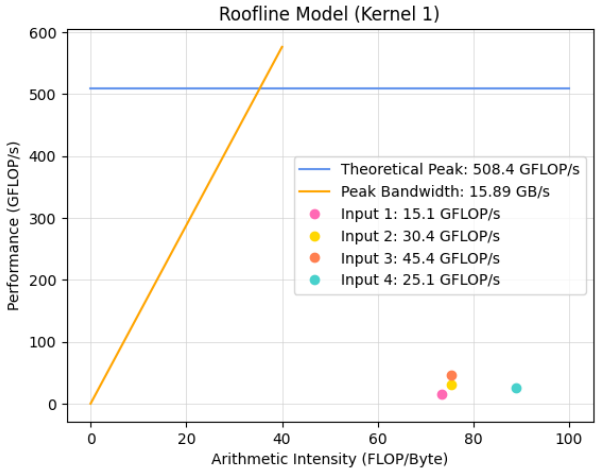

Figure 4: roofline model per il Kernel 1 su NVIDIA GeForce 920MX

3.4 MPI + OpenMP

Questa versione è seguita direttamente dalle versioni di MPI e OpenMP precedenti, che basandosi sulle stesse idee si sono prestate bene alla fusione.

3.4.1 Distribuzione dei processi

Per ogni nodo utilizzato abbiamo un processo MPI e 32 threads in modo da utilizzare completamente i core, sfruttando la memoria condivisa per il calcolo e utilizzando MPI solamente per la comunicazione.
Abbiamo utilizzato la modalità `SERIALIZED` e introdotto degli `omp single` nei punti di sincronizzazione in modo che ad eseguire la chiamata MPI sia solamente il primo thread che vi arriva, mentre gli altri si fermano fino al termine della comunicazione nel caso sia bloccante.
Per il resto la struttura del programma è rimasta quella della versione OpenMP.

3.4.2 Performance

Abbiamo eseguito i test raddoppiando il numero di nodi, da 1 a 32.

Nonostante distribuire i dati tra più nodi ci permetta di sfruttare più potenza computazionale, risolvendo il problema sul limite della cache, il nostro bottleneck diventa la comunicazione tra i nodi, che può essere migliorato aumentando la banda o modificando la topologia.

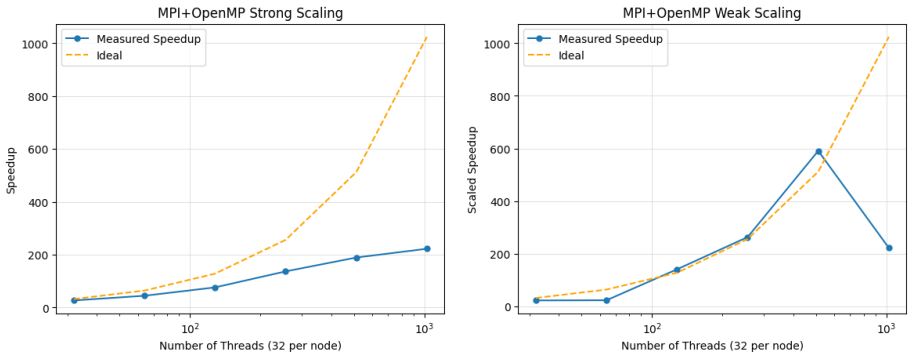

Figure 5: Strong Scaling misurato su input da 1.000.000 punti (100D)
Weak Scaling misurato su input da 31.250 a 1.000.000 punti (100D)
con parametri 100 150 0.01 0.01