Relazione Sequence

Relazione e progetto prodotti da: Giordana Foglia (2046738), Matteo Sperini (1987495)

OpenMP

L'algoritmo descritto sfrutta il parallelismo fornito dalla libreria OpenMP per effettuare un allineamento genetico tramite la strategia brute-force, migliorando le prestazioni rispetto alla versione sequenziale

Distribuzione del lavoro

Uno degli aspetti chiave è la gestione del carico tra i thread. Nel ciclo principale, l'uso di `#pragma omp parallel for reduction(+:pat_matches)` permette a ciascun thread di accumulare i risultati in una variabile locale, evitando la sincronizzazione continua e migliorando le prestazioni.

- Senza reduction, sarebbe necessaria un'operazione `#pragma omp atomic` a ogni incremento, causando un elevato overhead dovuto alla sincronizzazione tra thread, specialmente all'aumentare del numero di thread utilizzati.
- Con `reduction(+:pat_matches)`, i risultati vengono accumulati localmente per ogni thread e combinati solo alla fine, riducendo la necessità di accessi concorrenti alla memoria e migliorando l'efficienza complessiva.

Inoltre, l'uso della clausola `schedule(dynamic,4)` permette di assegnare le iterazioni ai thread in blocchi di 4 alla volta, anziché in modo fisso. Questo è particolarmente utile perché, essendo i pattern di lunghezza diversa, il tempo di elaborazione delle iterazioni varia. Se si utilizzasse `schedule(static)`, alcune iterazioni potrebbero richiedere più tempo di altre, lasciando alcuni thread inattivi con uno squilibrio nel carico di lavoro. Con `dynamic`, invece, ogni thread riceve nuove iterazioni solo dopo aver completato quelle assegnate, garantendo una distribuzione più bilanciata.

Gestione delle strutture condivise

Un elemento critico è l'aggiornamento di `seq_matches`, l'array che traccia il numero di pattern trovati per ogni posizione della sequenza.

- **Soluzione iniziale**: Creazione di **buffer locali** per ogni thread, poi combinati alla fine. Tuttavia, questo introduceva un overhead aggiuntivo.
- **Soluzione definitiva**: L'uso di `#pragma omp atomic` garantisce aggiornamenti corretti senza buffer aggiuntivi.
  - Tuttavia, aggiornando celle **vicine** nell'array, si verifica **false sharing**, dove più thread accedono a linee di cache condivise, riducendo l'efficacia della cache e causando rallentamenti.
  - L'overhead di `increment_matches` è minimo (**0.83%** del tempo totale), quindi non rappresenta un collo di bottiglia significativo.

Sezione più costosa: Ricerca dei pattern

La ricerca brute-force dei pattern nel ciclo parallelo principale è la parte computazionalmente più intensa, occupando **96.18%** del tempo totale di esecuzione

> 📌 OpenMP ha quindi garantito un buon **load balancing**, distribuendo equamente il lavoro tra i thread. Tuttavia, l'accesso non sequenziale alla memoria, in particolare nell'aggiornamento di `seq_matches` dentro `increment_matches()`, è causa di **cache misses**, riducendo così l'efficienza della CPU. Inoltre, l'uso di `#pragma omp atomic` introduce contesa tra i thread, aumentando l'overhead quando più thread aggiornano posizioni adiacenti della sequenza.

MPI

Per affrontare il problema della ricerca dei pattern in sequenze genetiche, questa volta è stata utilizzata la libreria MPI, distribuendo quindi il lavoro tra più processi. Ogni processo effettua la ricerca di un sottoinsieme di pattern localmente e al termine, i risultati vengono aggregati tramite operazioni collettive quali:

- `MPI_Reduce` per sommare i match locali e ottenere il totale globale.
- `MPI_Reduce` per aggregare le corrispondenze per posizione.
- `MPI_Gather/MPI_Gatherv` per raccogliere le posizioni dei pattern trovati.

Nonostante l'aumento del numero di pattern, i tempi di MPI_Gather (**0.039s**) e MPI_Reduce (**0.0029s**) sono rimasti contenuti rispetto alla computazione principale (input di **700.000** nucleotidi), confermando che le operazioni di riduzione e raccolta non rappresentano un collo di bottiglia.

Strategie di assegnazione del carico

Durante la progettazione sono state valutate diverse strategie:

1. **Assegnazione statica dei pattern**
   Ogni processo riceveva un sottoinsieme fisso di pattern senza ulteriori scambi durante l'esecuzione. Sebbene semplice ed equilibrata, questa strategia presentava problemi di sincronizzazione e comunicazione.
2. **Assegnazione dinamica dei pattern**
   Si è valutata un'assegnazione dinamica dei pattern, ma questa introduceva un overhead significativo, poiché ogni processo doveva richiedere nuovi pattern una volta concluso il lavoro sui pattern assegnatogli, aumentando il numero di messaggi e causando rallentamenti.

Problemi riscontrati

- **Gestione incoerente del processo con rank 0**: Il processo con rank 0 dopo aver distribuito i dati non elaborava i propri, portando a dati non inizializzati e risultati errati.
- **Inefficienza della comunicazione**: Lo scambio dei dati non era gestito in modo ottimale.
- **Allocazione errata dell'array** `send_requests`: Inizialmente allocato con una dimensione fissa **2 * (size - 1)**, assumeva che ogni processo ricevesse un solo pattern. In realtà, nella maggior parte dei casi i processi ne ricevevano più di uno, causando buffer overflow ed errori `MPI_ERR_COUNT`. Il problema è stato corretto calcolando correttamente la dimensione corretta per l'array di send, cioè **2 * (end - start)** per ogni processo.
- **Gestione della reduction**: A differenza di OpenMP, dove i thread condividono la memoria, MPI utilizza processi distinti con memoria separata. Questo ha causato un problema nell'inizializzazione degli array. Inizializzando ogni array locale a -1, la riduzione con **MPI_Reduce** sommava questi valori su tutti i processi, alterando la rispettiva checksum, rispetto a OpenMP. Per risolvere il problema, il processo con rank 0 mantiene l'inizializzazione a -1, mentre gli altri inizializzano gli array a 0, garantendo il corretto calcolo della checksum

Soluzione implementata: Suddivisione statica con comunicazioni asincrone

La divisione dei pattern rimane statica, ma è stata migliorata con comunicazioni asincrone tramite **MPI_Isend**/**MPI_Irecv**, per garantire una distribuzione efficiente:

- **Processo 0:** Invia i pattern agli altri processi in due fasi (prima la lunghezza, poi il contenuto).
- **Processi remoti:** Ricevono la lunghezza e i pattern con **MPI_Irecv**, attendono la conclusione con **MPI_Waitall** ed eseguono la ricerca.

Il tempo di questa fase è risultato estremamente ridotto, tra **0.0063** e **0.0075** secondi per processo con una sequenza di **700.000** nucleotidi, confermando che la comunicazione iniziale non è un collo di bottiglia.

MPI + OpenMP

📌 La terza implementazione è una **parallelizzazione ibrida**, che combina MPI per la distribuzione del lavoro tra i processi e OpenMP per accelerare le operazioni all'interno di ogni processo.

Strategia di parallelizzazione

L'algoritmo è strutturato su due livelli:

1. **Distribuzione dei pattern tra i processi MPI**
   - Il numero totale di pattern viene suddiviso equamente tra i processi, gestendo eventuali resti.
   - Come la versione MPI pura, i pattern vengono distribuiti in modo asincrono con **MPI_Isend**, inviando prima la lunghezza e poi il contenuto di ogni pattern.
   - I risultati vengono combinati con **MPI_Reduce** e **MPI_Gatherv** per aggregare i match trovati.
2. **Parallelizzazione con OpenMP all'interno di ciascun processo**
   - **Aggiornamento di** `seq_matches` : `increment_matches` usa `#pragma omp parallel for` con `#pragma omp atomic` per evitare race conditions nell'aggiornamento di `seq_matches`, come nella versione OpenMP.
   - **Parallelizzazione dell'inizializzazione delle strutture dati**.
   - **Ricerca parallela dei pattern**: I thread OpenMP eseguono il confronto tra i pattern e la sequenza in parallelo, riducendo il tempo computazionale.

In questo caso, l'uso della clausola `schedule(dynamic,64)` garantisce una distribuzione efficiente del carico tra i thread OpenMP all'interno di ciascun processo MPI. Poiché il lavoro è già suddiviso tra i processi MPI, ogni processo ha un sottoinsieme dei pattern da elaborare (**local_pat_number**), ma all'interno di ogni processo il carico può comunque essere sbilanciato a causa delle **differenze di lunghezza dei pattern** e del numero di confronti richiesti per ciascuna iterazione.

L'assegnazione dinamica in blocchi di **64 iterazioni** bilancia il carico e riduce l'overhead gestionale. Se si utilizzasse `schedule(static)` , ogni thread riceverebbe un numero fisso di iterazioni, senza considerare che alcune potrebbero essere più lunghe e richiedere più tempo.Con `schedule(dynamic,64)` , invece, i thread ricevono blocchi di 64 iterazioni alla volta e ne richiedono di nuove solo dopo aver completato quelle assegnate.

Combinazione dei risultati

- **Operazioni di reduction con MPI**
  - Per ottenere il numero totale di match trovati, viene utilizzata **MPI_Reduce**.
  - L'aggregazione delle corrispondenze nelle posizioni della sequenza avviene tramite una riduzione elemento per elemento.
- **Operazioni di Gather con MPI**
  - Poiché ogni processo può trovare un numero variabile di pattern corrispondenti, viene utilizzata **MPI_Gatherv**, che permette di raccogliere dati di dimensioni diverse senza spreco di memoria.

Gestione della concorrenza tra MPI e OpenMP

L'uso simultaneo di OpenMP e MPI può causare problemi se più thread eseguono operazioni MPI contemporaneamente, poiché MPI non è thread-safe a meno che non venga inizializzato con **MPI_Init_thread** Inoltre, per permettere la chiamata di più operazioni MPI contemporaneamente va specificata la clausola **MPI_THREAD_MULTIPLE**. La soluzione che abbimo adottato è stato invece quella di eseguire le comunicazioni MPI in modo seriale e lasciare che solo i calcoli vengano parallelizzati con OpenMP, evitando possibili errori e garantendo stabilità nell'esecuzione.

Risultati e benefici della parallelizzazione Ibrida

L'implementazione MPI + OpenMP ha ridotto il tempo totale di esecuzione di oltre il **32%** rispetto alla versione MPI pura, grazie a:

- **Migliore utilizzo della memoria**: I thread OpenMP condividono la stessa memoria, riducendo il consumo complessivo di RAM e diminuendo i cache misses rispetto a un approccio puramente distribuito.
- **Bilanciamento del carico migliorato**: OpenMP aiuta a distribuire il lavoro tra i thread, evitando che alcuni processi MPI rimangano inattivi, migliorando così scalabilità ed efficienza dell'algoritmo.

CUDA

Questa implementazione sfrutta le GPU, tramite **CUDA**, per risolvere il problema della ricerca dei pattern in una sequenza genetica. La progettazione ha richiesto la definizione di un livello di parallelismo ottimale e l'ottimizzazione dell'accesso alla memoria per migliorare le prestazioni.

Pattern matching

La prima scelta progettuale è stata definire il livello di parallelismo:

- **Paralellizzare sui pattern**: Ogni thread elabora un pattern diverso.
- **Paralellizzare sulla sequenza**: Ogni thread analizza una porzione della sequenza principale.

Inizialmente, abbiamo valutato **una griglia 1D**, dove ogni thread avrebbe gestito una posizione della sequenza e confrontato tutti i pattern in serie, che ha presentato diversi svantaggi:

- **Basso parallelismo**: Ogni thread avrebbe dovuto iterare su tutti i pattern sequenzialmente invece di processarli in paralello portando ad un carico di lavoro non bilanciato
- **Utilizzo inefficiente della memoria globale**: Ogni thread avrebbe letto tutti i pattern dalla memoria globale, causando ripetizioni inutili e un aumento della latenza

L'idea finale è stata **parallelizzare sia sulla sequenza che sui pattern** attraverso l'utilizzo di una **griglia 2D** dove:

- **L'asse X** rappresenta le posizioni della sequenza suddivisa in chunk in modo tale che ogni blocco ne processi una parte e ogni **thread dentro il blocco** processa una posizione specifica della sequenza.
- **L'asse Y** rappresenta i pattern da confrontare, ogni blocco lavora su un sottoinsieme di pattern permettendo a più thread di elaborare pattern diversi in parallelo

In questa configurazione, ogni blocco è quindi responsabile di una regione specifica della sequenza e di un sottoinsieme di pattern, garantendo un carico di lavoro bilanciato. All'interno di ciascun blocco, i thread lungo l'asse X gestiscono più posizioni della sequenza e i thread lungo l'asse Y si occupano di diversi pattern, riducendo la ripetizione degli accessi alla memoria globale e migliorando le prestazioni complessive.

Ogni thread confronta un pattern con una porzione della sequenza principale. Se la prima lettera del pattern corrisponde alla posizione della sequenza assegnata al thread, vengono verificate anche le lettere successive. Se l'intero pattern è presente, viene considerato un match. A questo punto, il thread registra la posizione iniziale del match in `d_pat_found` utilizzando `atomicCAS`, per garantire che solo il primo thread che trova il pattern possa scriverlo. A questo punto il contatore globale dei pattern trovati viene incrementato di 1 in modo sicuro con `atomicAdd`. Infine, se l'`atomicCAS` è già stato fatto da un'altro thread, viene controllato con un `atomicMin` che il valore in quella posizione di `d_pat_found` sia il più piccolo di tutti, dato che il problema richiede la prima occorrenza da sinistra.

Increment matches

Dopo che il primo kernel ha identificato le posizioni iniziali dei pattern nella sequenza, il secondo kernel `incrementMatchesKernel` aggiorna `d_seq_matches` per tutte le lettere dei pattern trovati. Per massimizzare il parallelismo, utilizziamo una griglia 2D: ogni blocco è responsabile di un insieme di pattern e delle relative lettere all'interno di essi.

- **Asse X (16 thread per blocco)**: Ogni thread gestisce un pattern.
- **Asse Y (16 thread per blocco)**: Ogni thread processa un carattere del pattern.

Se il pattern è stato trovato (`d_pat_found[patId] ≠ ULLONG_MAX`), il thread aggiorna `d_seq_matches` con `atomicAdd()` per evitare conflitti tra thread.

Dopo l'esecuzione del secondo kernel, `d_seq_matches` sulla GPU contiene il numero di volte in cui ogni posizione della sequenza è stata coinvolta in un match. Tuttavia, sulla GPU il conteggio parte da 1 in quanto `atomicAdd()` incrementa il valore ogni volta che un match viene trovato, quindi anche il primo match porta il valore da 0 a 1. Per allinearci al codice CPU originale, dove il conteggio dei match parte da 0, abbiamo decrementato il valore di `seq_matches[i]` quando è maggiore di 0 (altrimenti viene impostato a `NOT_FOUND` per indicare che nessun pattern ha coperto quella posizione) dopo la copia dei dati sulla CPU con `cudaMemcpy()`. Questo assicura che `seq_matches` rappresenti correttamente il numero di pattern che hanno coperto ogni posizione della sequenza, mantenendo la stessa logica del codice sequenziale originale.

Correzione delle Discrepanze tra CPU e GPU

Passando da CPU a GPU, abbiamo riscontrato delle **differenze nei risultati**, dovute a come i dati venivano inizializzati e aggiornati:

1. **Gestione del valore di assenza di match**: Abbiamo notato che `NOT_FOUND` (-1) non era adatto, poiché la variabile `d_pat_found` è di tipo unsigned long long. Assegnare un valore negativo a un tipo non firmato poteva causare comportamenti indesiderati. Per questo motivo, abbiamo scelto di utilizzare `ULLONG_MAX`, che evita conversioni errate e garantisce il corretto funzionamento di `atomicCAS`, permettendo l'aggiornamento di `d_pat_found` solo se il valore iniziale è `ULLONG_MAX`.
2. **Valori iniziali diversi**: L'array delle corrispondenze `seq_matches` veniva inizializzato a -1 su CPU e a 0 su GPU, causando discrepanze nei risultati finali.
    - **Soluzione**: Dopo il trasferimento da GPU a CPU (cudaMemcpy()), abbiamo normalizzato i valori sottraendo 1 ai risultati ottenuti sulla GPU.
3. **Errore nell'accesso ai dati**: I thread nella GPU accedevano sempre all'inizio della sequenza invece che alla posizione corretta nel chunk assegnato.
    - **Soluzione**: Abbiamo modificato gli indici per assicurare che ogni thread accedesse alla sua porzione corretta della sequenza.
4. **Perdita di dati ai bordi dei chunk**: Suddividendo la sequenza in blocchi (chunkSize), alcuni pattern che si trovavano tra due chunk non venivano rilevati.
    - **Soluzione**: Introduzione di una **sovrapposizione (**overlap**)** tra i chunk per garantire che i pattern ai bordi venissero considerati.

Gestione della Memoria

L'ottimizzazione della memoria è stata un aspetto chiave per migliorare le prestazioni della GPU.

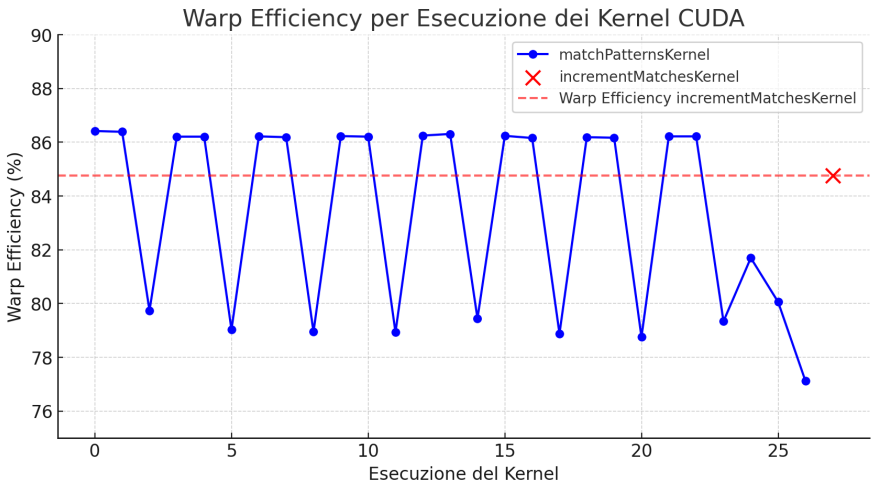
Problemi iniziali con la memoria condivisa:

- **Limiti di capacità**: La memoria condivisa si è rivelata insufficiente per gestire dataset di grandi dimensioni.
- **Overhead di sincronizzazione**: L'uso di `__syncthreads()` introduceva rallentamenti.
- **Riduzione dell'occupancy**: L'uso eccessivo della memoria condivisa limitava il numero di thread per blocco, riducendo l'efficienza della GPU.

Soluzione:

- Eliminazione della memoria condivisa per i dati più grandi.
- Ottimizzazione dell'accesso alla **memoria globale**, garantendo **accessi coalescenti** (letture consecutive da parte di thread adiacenti), riducendo il numero di transazioni con la memoria.

Analisi Warp Efficiency



Abbiamo misurato la **warp efficiency** per valutare l'efficacia dell'uso dei thread nella GPU, ossia la percentuale di **warp attivi** rispetto al massimo teorico supportato dall'hardware.

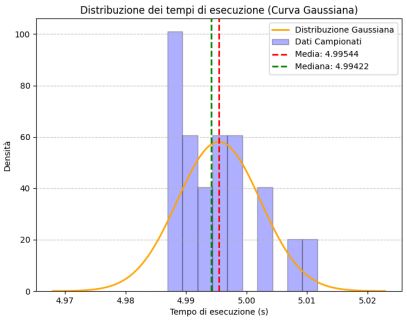
Risultati ottenuti:

- **Kernel principale (** `matchPatternsKernel` **)**
  - Warp efficiency **tra il 77% e l'86%**
  - Ciò indica che in alcune esecuzioni **molti warp non riescono a operare in modo completamente parallelo**, a causa di **branch divergence** (divergenza dei percorsi di esecuzione tra i thread)
- **Kernel di aggiornamento (** `incrementMatchesKernel` **)**
  - Warp efficiency **circa 84.5%**, con una maggiore stabilità
- **Occupancy della GPU**
  - Il profiler NVIDIA ha misurato un'**occupancy del 86%**, indicando che il numero di warp residenti su ogni **Streaming Multiprocessor (SM)** è pari all'**86% del massimo teorico supportato dalla RTX 4070**
  - Questo valore è **inferiore all'occupancy teorica (93.75%)**, quindi ci sono inefficienze dovute all'uso delle risorse o alla gestione dei kernel

Distribuzione dei tempi

Distribuzione codice sequenziale

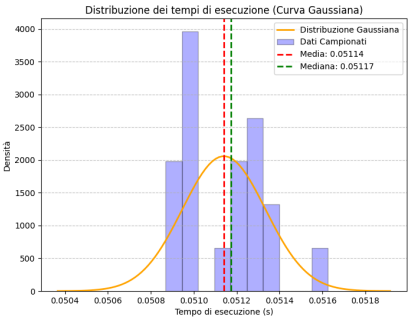
Nella versione **sequenziale** la distribuzione dei tempi ha una media di circa 4,995 secondi e una mediana di 4,994 secondi che sono vicine, il che indica una distribuzione stabile e simmetrica. Inoltre la variazione dei tempi (a parità di condizioni iniziali) è ridotta



Distribuzione sequenziale

Distribuzione codice CUDA

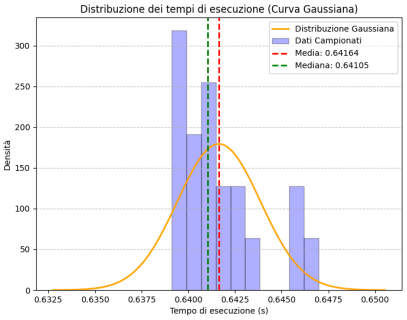
La soluzione basata su **CUDA** ha una media di 0,05114 secondi e una mediana di 0,05117 secondi; Dato chesono estremamente vicine, la distribuzione dei tempi è molto stabile. Questo significa che l'accelerazione hardware della GPU e l'ottimizzazione dei thread permettono di ridurre drasticamente il tempo di calcolo



Distribuzione CUDA

Distribuzione codice MPI

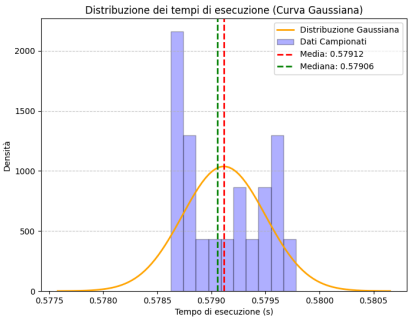
L'implementazione **MPI**, che distribuisce il carico computazionale su più processi operanti in parallelo, raggiunge una media di 0,64164 secondi e una mediana di 0,64105 secondi, con valori anch'essi vicini. Pur mostrando un miglioramento rispetto alla versione sequenziale, i tempi più elevati rispetto a CUDA suggeriscono che gli overhead di comunicazione e la latenza inter-processo (tempo necessario per trasferire un messaggio da un processo a un altro) incidono in maniera non trascurabile sulla performance. La distribuzione è leggermente asimmetrica verso destra, indicando che alcune esecuzioni sono state più lente della media



Distribuzione MPI

Distribuzione MPI+OMP

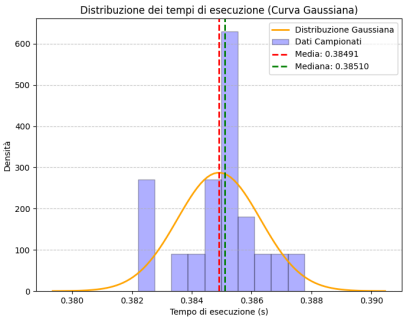
La versione **OMP+MPI**, che combina il parallelismo a livello di processi (MPI) con quello a livello di thread (OpenMP), ottiene una media di 0,57912 secondi e una mediana di 0,57906 secondi. Notiamo che la distribuzione è più stretta e non si osserva una asimmmetria tra ivalori, indicando una maggiore stabilità del programma. Dal grafico della distribuzione si osserva che l'intervallo di variazione è approssimativamente 2 millisecondi.



Distribuzione MPI+OMP

Distribuzione OMP

La versione **OpenMP** che utilizza i thread non deve occuparsi della comunicazione tra processi e questo permette di ridurre i ritardi, ottenendo tempi medi e mediani di circa 0,385 secondi. Nonostante questo non è in grado di competere con CUDA, che sfrutta la potenza delle GPU per eseguire un gran numero di operazioni in parallelo.

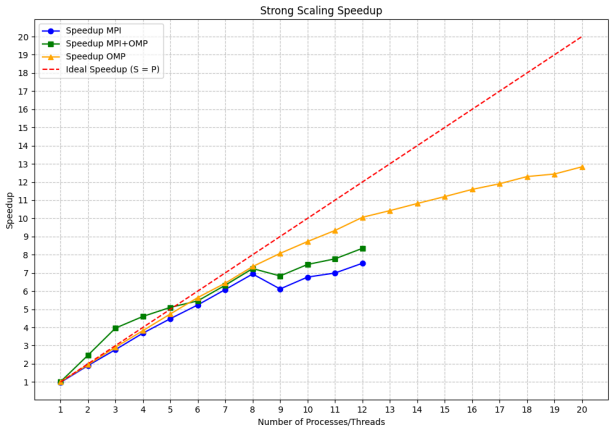


Distribuzione OMP

Strong scaling speedup

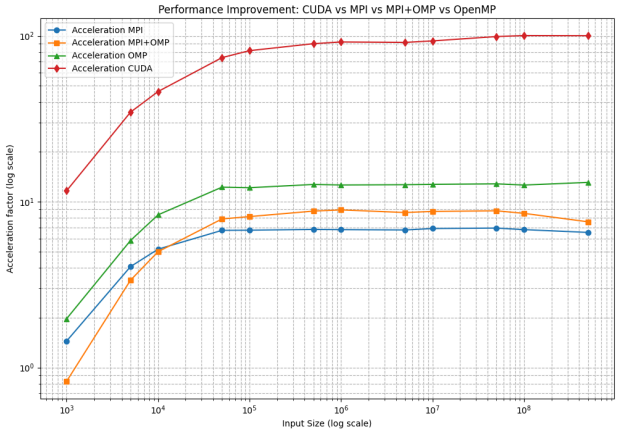
$$S(p) = \frac{T(1)}{T(p)}$$

Nel grafico, la linea rossa tratteggiata indica lo speedup ideale, che aumenta in modo lineare con il numero di processi o thread. La linea blu (**MPI**) parte con uno speedup significativo, ma tende a crescere più lentamente man mano che aumentano i processi, perché la comunicazione tra processi introduce ritardi sempre maggiori. Con input di dimensioni grandi, però, l'impatto di questi overhead risulta meno incisivo rispetto al tempo complessivo di calcolo, consentendo comunque un buon guadagno. La linea verde (**MPI+OMP**) sfrutta meglio le risorse di calcolo, offrendo uno speedup superiore a MPI, soprattutto quando l'input è abbastanza grande da ammortizzare sia la comunicazione tra processi sia la gestione dei thread. Si registra inoltre un super-linear speedup entro i primi 5 thread, probabilmente dovuto ad un migliore utilizzo della cache, oppure minori overhead di sincronizzazione. La linea gialla (**OMP**) non richiede comunicazione tra processi e, per un certo intervallo di thread, mostra una crescita dello speedup più marcata rispetto a MPI e MPI+OMP. Tuttavia, essendo limitata a un singolo nodo, non può scalare oltre la disponibilità di core locali e, per input molto grandi o numeri elevati di processi/thread, può incontrare limiti fisici e di memoria.

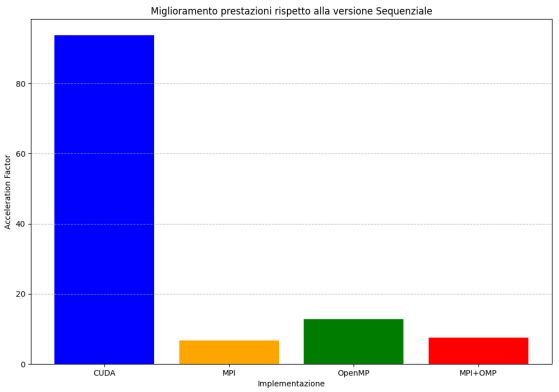


Acceleration factor

Abbiamo riportato l'asse dell'input in scala logaritmica per evidenziare al meglio la crescita esponenziale dei dati e, di conseguenza, mettere in risalto le prestazioni di CUDA anche su ordini di grandezza molto diversi. La linea rossa (**CUDA**) si mantiene nettamente al di sopra delle altre per quasi tutto l'intervallo, indicando che la GPU fornisce il miglior incremento di prestazioni, soprattutto quando l'input diventa grande grazie all'ampio sfruttamento del parallelismo. Confrontando CUDA con le altre implementazioni, si nota che il divario si amplia progressivamente all'aumentare dell'input, sottolineando la capacità delle GPU di accelerare il calcolo in modo più incisivo rispetto a soluzioni basate esclusivamente sulla CPU.



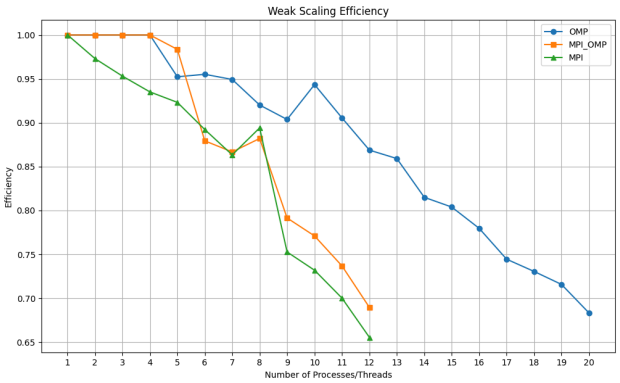
📌 Anche il grafico del fattore di accelerazione conferma questi risultati: **CUDA raggiunge un'accelerazione superiore a 90**, distanziandosi nettamente dalle altre implementazioni. Al contrario, le soluzioni basate su CPU offrono miglioramenti più contenuti, con OpenMP che si distingue leggermente rispetto a MPI e alla combinazione MPI+OMP



Efficiency weak scaling

$$E_{\text{WEAK}} = \frac{T(1)}{T(p)} \text{ (con problema } N \cdot p)$$

L'implementazione basata su OpenMP mantiene un'efficienza vicina a 1 per circa 6-7 thread quindi il parallelismo gestisce bene l'aumento del carico di lavoro. Dopo 8-10 thread cala progressivamente indicando che forse l'overhead e la contesa delle risorse impatta sulle prestazioni. Nonostante questo OpenMP scala in modo molto pià graduale rispetto a MPI+OpenMP e MPI indicando che scala meglio. L'implementazione basata su MPI descresce in modo costante fin dall'inizio e dopo 8 processi l'efficienza scende molto rapidamente indicando un significativo overhead di comunicazione tra I processi. MPI+OpenMP mantiene un'efficienza alta fino a 6-7 processi ma dopo 8 cala bruscamente indicando che inizialmente utilizzare sia MPI che OpenMP aiuta molto ma dopo un certo numero di processi diventa troppo complicato da gestire e invece di andare più veloce, rallenta.



Efficiency strong scaling

$$E_{\text{STRONG}}(p) = \frac{S(p)}{p} = \frac{T(1)}{p \cdot T(p)}$$

Nell'implementazione basata su **MPI**, i valori di efficienza sono sempre inferiori a 1. Questo comportamento è atteso poiché l'overhead introdotto dalla comunicazione tra i processi di cui abbiamo già discusso in precedenza riduce i benefici della parallelizzazione. Dal grafico, si osserva che fino a circa 6-7 processi, l'efficienza rimane relativamente alta, ma aumentandoli, si verifica un progressivo calo dovuto alla sincronizzazione e alla latenza nella trasmissione dei dati. In particolare, dopo 8 processi, l'efficienza dell'implementazione **MPI** subisce un crollo molto rapido. Diversamente, l'implementazione **MPI+OpenMP** mostra valori di efficienza superiore a 1 peralcuni casi con pochi processi. Per esempio, con 2, 3, 4 e 5 processi e thread (5 processi, 5 thread) l'efficienza è rispettivamente pari a 1.237, 1.320, 1.151 e 1.021. L'efficienza risulta maggiore di 1 in **MPI+OpenMP** perché nel calcolo classico si divide lo speedup per il numero di processi, ma il carico di lavoro non è sempre bilanciato perchè alcuni processi MPI possono terminare prima o rimanere in attesa di comunicazione, mentre alcuni thread OpenMP potrebbero non essere completamente utilizzati. Questo porta a un numero **effettivo** di unità di calcolo minore rispetto a $p_{MPI} \cdot T_{MPI}$ facendo apparire in alcuni casi l'efficienza superiore a 1. L'implementazione **OpenMP** mantiene un'efficienza vicina a 1 inizialmente, ma all'aumentare dei thread tende a calare gradualmente.