Relazione: DNA Sequence Alignment

Team di 1 persona: Francesco Pennacchietti 1934929

1. Scelta delle implementazioni da utilizzare

Per rispettare la consegna e mostrare competenza in tutti e tre i modelli di programmazione (CPU con memoria condivisa, GPU, e memoria distribuita), ho seguito le seguenti implementazioni:

- **OpenMP:** Consente di ottenere una parallelizzazione efficace su una singola macchina con CPU multicore.
- **MPI + CUDA:** Offre un'architettura scalabile, permettendo di sfruttare sia la potenza delle GPU che la capacità di calcolo distribuito su cluster.

2. Workflow iniziale

Il mio lavoro è cominciato da una fase di analisi del codice sequenziale per comprendere al meglio il flusso di esecuzione, le strutture utilizzate e le parti potenzialmente parallelizzabili.

3. OpenMP

All'inizio, ho adottato un approccio basato sul principio del "tutto e subito", cercando di ottimizzare e parallelizzare il codice il più possibile fin da subito. Tuttavia, il risultato non è stato soddisfacente, nonostante l'uso di 32 thread, lo speedup ottenuto si è fermato a circa 2×.

Riconosciuto questo errore, ho deciso di ripartire da zero, adottando una strategia che mi permettesse di isolare le aree di sviluppo in modo da essere cosciente delle conseguenze in termini di tempo di ogni mia scelta implementativa.

3.1 Ciclo principale

3.1.1 Parallelizzazione del Ciclo

Ho deciso di parallelizzare il loop più esterno, in modo che ogni thread lavori su un pattern diverso ed esegua i cicli interni su *start* e *lind* in modo sequenziale. Questo approccio presenta diversi vantaggi. Innanzitutto, i pattern sono completamente **indipendenti** tra loro, eliminando la necessità di sincronizzazione tra thread, eccetto per gli aggiornamenti condivisi come *pat_matches* o *seq_matches*.

Un altro vantaggio derivante dall'indipendenza dei pattern è la **deterministicità** del risultato: la sequenza di elaborazione dei pattern rimane invariata indipendentemente dal numero di thread utilizzati. Inoltre, con un numero elevato di pattern (*pat_number* grande), ogni thread ha un carico di lavoro sufficiente, riducendo l'overhead di OpenMP e garantendo un buon **bilanciamento del carico**.

3.1.2 Scheduling

Nel codice, ho utilizzato lo scheduling statico, che distribuisce equamente i pattern tra i thread in fase di compilazione. Questa scelta è motivata dai seguenti fattori:

1. **Indipendenza dei pattern**
   Poiché ogni pattern viene elaborato indipendentemente dagli altri, non ci sono dipendenze tra iterazioni. Questo permette di assegnare il lavoro in blocchi fissi ai thread senza il rischio di squilibri causati da sincronizzazioni o dipendenze.
2. **Bilanciamento del carico accettabile**
   Anche se il tempo di ricerca può variare tra i pattern (a seconda della loro lunghezza e della presenza di match nella sequenza), nel caso di un numero elevato di pattern (*pat_number* grande), la distribuzione statica garantisce che ogni thread abbia comunque un numero sufficiente di iterazioni da eseguire, riducendo l'overhead di gestione del carico.
3. **Minore overhead rispetto allo scheduling dinamico**
   Lo scheduling statico assegna le iterazioni ai thread in fase di compilazione, evitando l'overhead di gestione dinamica del carico in fase di esecuzione.

3.1.3 Utilizzo della Variabile Locale *local_found*

Per ottimizzare le prestazioni, ho introdotto la variabile locale *local_found*, che viene inizializzata a *NOT_FOUND* all'inizio di ogni iterazione del ciclo for parallelo. Questo approccio ha diversi vantaggi:

1. **Riduzione degli accessi alla memoria globale**
   Ogni thread mantiene una propria copia della variabile *local_found*, evitando accessi ripetuti a strutture dati condivise. Questo riduce il traffico sulla memoria principale e il rischio di contention tra i thread.
2. **Miglioramento della *cache locality***
   Lavorare con una variabile locale permette di sfruttare meglio la *cache* del processore, riducendo i tempi di accesso ai dati e migliorando le prestazioni generali.
3. **Minimizzazione della necessità di sezioni critiche**
   Poiché ogni thread aggiorna solo la propria variabile locale, non è necessario sincronizzare l'accesso a *pat_found[pat]* finché non viene trovato un match. Questo riduce l'uso di sezioni critiche, migliorando il parallelismo.

3.1.4 La Sezione Critica

L'utilizzo della sezione critica nel codice è stato progettato per ridurre al minimo la sincronizzazione, evitando overhead inutili e garantendo comunque la correttezza dei risultati.

1. **Assegnamento di *pat_found[pat] = local_found* (Fuori dalla Sezione Critica)**
   - Ogni thread lavora su un pattern diverso (*pat* è un indice univoco per ciascuna iterazione del ciclo parallelo).
   - Una volta trovato un match per un determinato pattern, il valore *local_found* viene scritto in *pat_found[pat]* senza rischio di sovrascritture da parte di altri thread.

    - Questo permette di mantenere l'operazione fuori dalla sezione critica, eliminando qualsiasi necessità di sincronizzazione.
2. **Incremento di *pat_matches* (Dentro la Sezione Critica)**
    - *pat_matches*++ è una variabile condivisa tra tutti i thread.
    - Senza una sezione critica, si potrebbe verificare una *race condition*, con più thread che aggiornano il valore simultaneamente senza che gli incrementi vengano registrati correttamente (*lost updates*).
    - La sezione critica garantisce che l'incremento avvenga in modo atomico e corretto.
3. **Chiamata a *increment_matches()* (Dentro la Sezione Critica)**
    - La funzione *increment_matches()* aggiorna *seq_matches*, una struttura dati condivisa.
    - Poiché più pattern possono condividere match nella sequenza *(seq_matches* potrebbe essere aggiornato da più thread contemporaneamente), è necessario proteggerne l'aggiornamento con una sezione critica.
    - Questo previene aggiornamenti concorrenti che potrebbero portare a valori inconsistenti o sovrascritture errate.

3.2 generate_rnd_sequence

Per passare da una versione sequenziale a parallela di questa funzione ogni thread deve generare una sequenza indipendente senza interferire con gli altri, garantendo che l'ordine dei numeri casuali rimanga deterministico e riproducibile.

Per farlo, adotto tre strategie fondamentali:

- Dividere il lavoro equamente tra i thread (chunk_size).
- Creare uno stato indipendente per ogni thread (thread_state = *random).
- Far avanzare lo stato del generatore casuale per evitare sovrapposizioni (rng_skip(&thread_state, start)).

3.3 Checksum

Per ottimizzare la sezione del codice in cui calcolo le checksum combino due strategie:

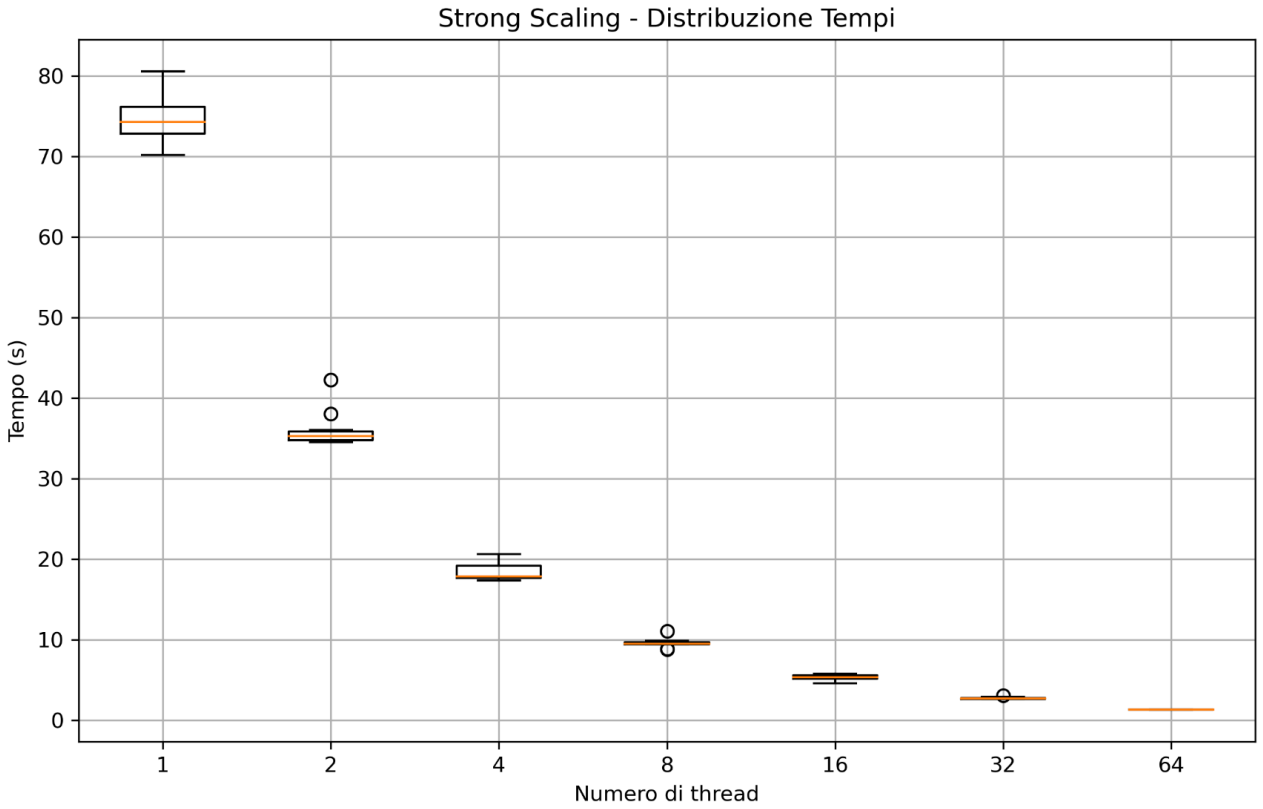
Proprietà del Modulo con Somma: (a+b)%m=((a%m)+(b%m))%m, questo significa che il modulo dell'operazione di somma può essere applicato sia dopo ogni somma parziale che dopo che tutta la somma è stata completata. Questa strategia mi permette di "staccare" l'operazione di modulo dal ciclo risparmiando sia in termini di costo dell'operazione che in termini di complessità nell'accumulare un espressione che combinava l'operazione di somma e di modulo.

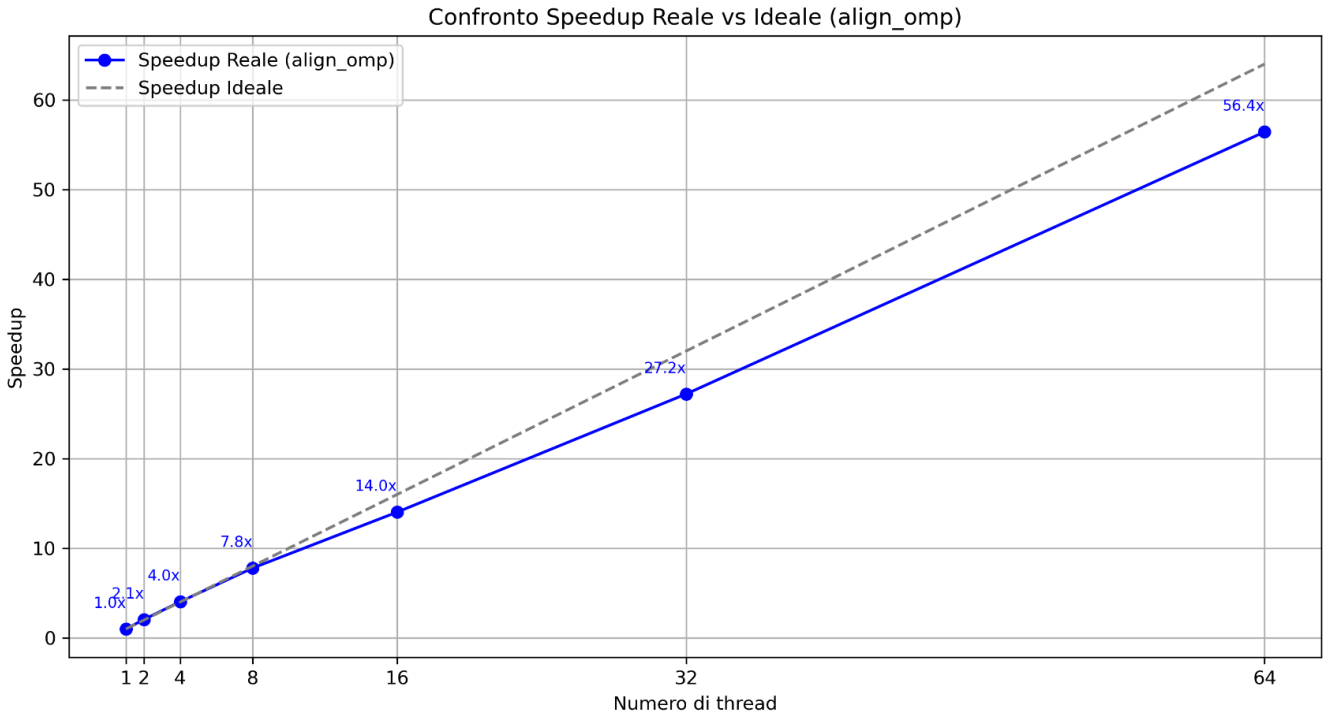
Nowait: Usando *#pragma omp for nowait*, la barriera viene rimossa, quindi ogni thread:

1. Esegue la sua porzione del ciclo for.
2. Appena termina la sua parte, può subito eseguire *#pragma omp atomic checksum_found += local_sum;*, senza aspettare gli altri. Questo significa che **i thread non restano bloccati** alla fine del ciclo for e possono eseguire più velocemente la fase successiva.

Strong Scaling

Input utilizzato: ./align_omp 1048576 0.1 0.3 0.35 8192 32 2 8192 32 2 524288 262144 M 609823

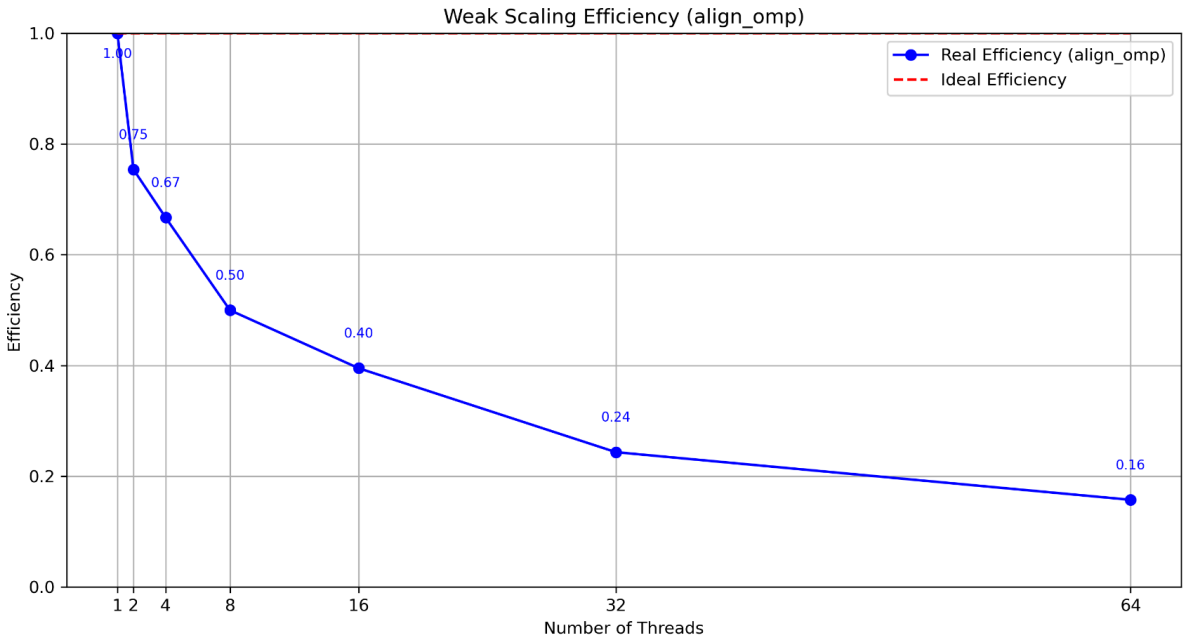




Weak Scaling

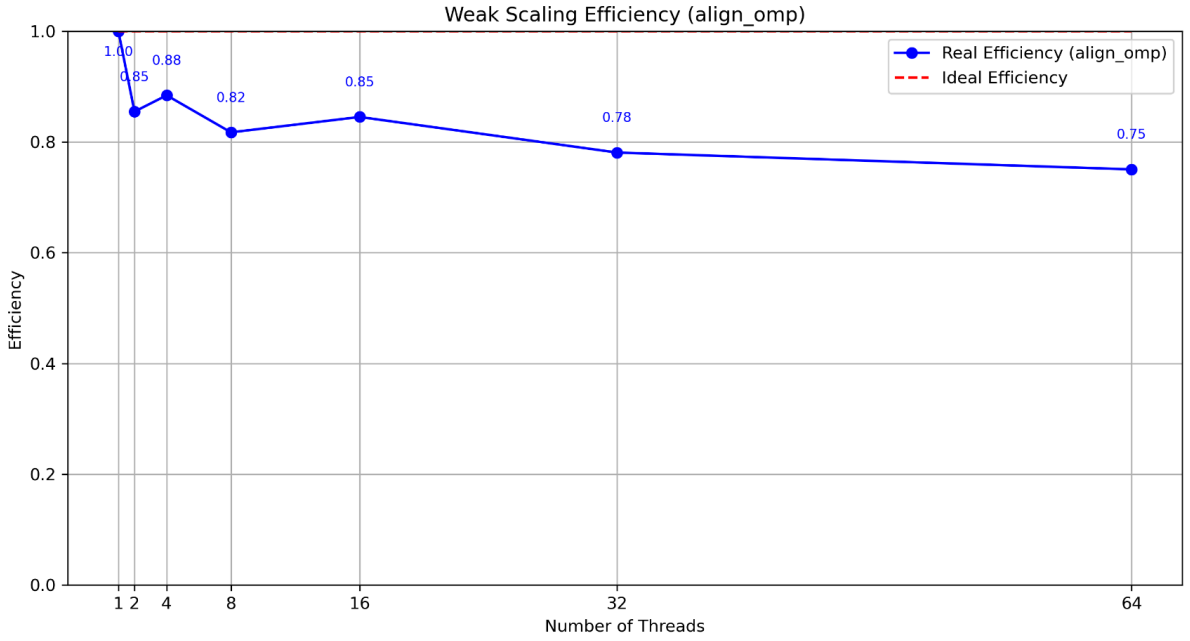
Incremento sia del numero di pattern che della lunghezza della sequenza

./align_omp **1048576** 0.1 0.3 0.35 **512** 8 2 **512** 8 2 524288 262144 M 609823 #1TH
./align_omp **2097152** 0.1 0.3 0.35 **1024** 8 2 **1024** 8 2 524288 262144 M 609823 #2TH
./align_omp **4194304** 0.1 0.3 0.35 **2048** 8 2 **2048** 8 2 524288 262144 M 609823 #4TH
./align_omp **8388608** 0.1 0.3 0.35 **4096** 8 2 **4096** 8 2 524288 262144 M 609823 #8TH
./align_omp **16777216** 0.1 0.3 0.35 **8192** 8 2 **8192** 8 2 524288 262144 M 609823 #16TH
./align_omp **33554432** 0.1 0.3 0.35 **16384** 8 2 **16384** 8 2 524288 262144 M 609823 #32TH
./align_omp **67108864** 0.1 0.3 0.35 **32768** 8 2 **32768** 8 2 524288 262144 M 609823 #64TH



Incremento del numero di pattern (mantenendo fissa la lunghezza della sequenza)

./align_omp 1048576 0.1 0.3 0.35 **512** 8 2 **512** 8 2 524288 262144 M 609823 #1TH
./align_omp 1048576 0.1 0.3 0.35 **1024** 8 2 **1024** 8 2 524288 262144 M 609823 #2TH
./align_omp 1048576 0.1 0.3 0.35 **2048** 8 2 **2048** 8 2 524288 262144 M 609823 #4TH
./align_omp 1048576 0.1 0.3 0.35 **4096** 8 2 **4096** 8 2 524288 262144 M 609823 #8TH
./align_omp 1048576 0.1 0.3 0.35 **8192** 8 2 **8192** 8 2 524288 262144 M 609823 #16TH
./align_omp 1048576 0.1 0.3 0.35 **16384** 8 2 **16384** 8 2 524288 262144 M 609823 #32TH
./align_omp 1048576 0.1 0.3 0.35 **32768** 8 2 **32768** 8 2 524288 262144 M 609823 #64TH



4. MPI + CUDA

Per lo sviluppo di questa versione l'approccio di procedere per gradi si è rivelato fondamentale per la corretta riuscita del progetto. Inizialmente ho cominciato a sviluppare un **codice CUDA che fosse ad un solo processo** e che poi nelle chiamate kernel usava n threads. Una volta ottenuta una versione CUDA funzionante ma non ottimizzata sono passato all'**aggiunta del codice MPI** in modo tale da poter usare più processi contemporaneamente. L'ultima fase dello sviluppo l'ho spesa per **ottimizzare il codice** esplorando moltissime alternative e analizzando report e tempi ottenuti.

Nella presentazione mi sono focalizzato sul commentare il processo evolutivo del progetto mentre qua nel report approfondisco di più le scelte finali e i tempi d'esecuzione.

4.1. Cuore Computazionale del programma

La parte del programma computazionalmente più importante è senza dubbio la sezione **5. "Search for each pattern"** quindi il focus principale è stato fin dall'inizio sullo sviluppo di questa parte.
Per avere una panoramica generale sul flusso d'esecuzione di questa parte ecco un breve riassunto:

1. Preparazione del buffer concatenato per i pattern
2. Allocazione memoria sulla GPU e Copia dei dati dalla CPU alla GPU per il kernel findPatterns
3. Configurazione e lancio del kernel findPatterns
4. Elaborazione risultati del kernel findPatterns
5. Configurazione del secondo kernel: updateSeqMatches
6. Post-elaborazione dei risultati
7. Deallocazione memoria

4.1.1. Ruolo dei due kernel: findPatterns e updateSeqMatches

La possibilità di usare la parallelizzazione su GPU apre a delle possibilità di ottimizzazione del codice e dei tempi molto importanti se si mantiene una coerenza tra la struttura delle griglie e dei blocchi e la natura del problema. E' quindi molto importante parlare della struttura del kernel oltre che del suo scopo e delle scelte implementative che lo compongono.

Lo scopo generale di questa sezione del codice è ottenere tre elementi che poi serviranno come input nella sezione dei checksum per produrre i risultati finali. Questi sono:

- **pat_found** = Array che contiene la posizione minima nella sequenza trovata per ogni pattern
- **seq_matches** = Array dei contatori di match per ogni posizione della sequenza
- **matches** = Numero di pattern trovati nella sequenza

Perché due kernel separati? La scelta di usare due kernel separati è puramente tecnica, dobbiamo assicurarci che la ricerca delle occorrenze minime dei pattern nella sequenza sia conclusa prima di elaborare i risultati, che costituiscono l'input per il calcolo di *seq_matches*. La soluzione a questo problema sarebbe quella di avere una barriera globale per sincronizzare tutti i threads ma usando

cudaDeviceSynchronize() bloccherei solo quelli relativi ad un blocco. Per aggirare questo problema mi è venuto in mente di separare il codice in due kernel separati forzando obbligatoriamente questa sincronizzazione globale poiché tutti i threads devono finire prima di ritornare il controllo alla CPU. In questo modo ottengo dei risultati validi dal primo kernel pronti per essere usati come input nel secondo kernel pagando un overhead per il lancio del kernel molto basso rispetto al tempo finale (~20 microsecondi).

findPatterns

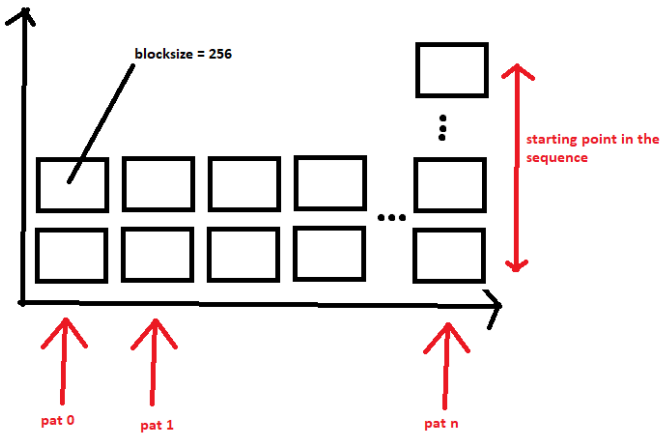
Il kernel findPatterns cerca in parallelo la posizione minima di match per ogni pattern nella sequenza. Genera l'array *pat_found*, dove ogni elemento contiene la posizione iniziale del primo match trovato per il rispettivo pattern o *NOT_FOUND_ULL* se il pattern non è presente.

Struttura di griglie e blocchi

Per sfruttare al massimo la parallelizzazione offerta da CUDA, il kernel findPatterns è progettato con una struttura di griglie e blocchi che bilancia il carico di lavoro tra i pattern e le posizioni nella sequenza.

- **Block Size:** Ogni blocco contiene **128 threads**, la scelta di questo numero è il risultato di test sperimentali, che hanno mostrato il miglior tempo di esecuzione rispetto a valori più grandi (es. 256, 512, o 1024). Un block size di 128 consente un'elevata occupancy della GPU, garantendo che ogni Streaming Multiprocessor (SM) esegua più blocchi contemporaneamente, senza saturare le risorse (es. registri o shared memory). Questo valore ottimizza il parallelismo intra-blocco, dove ogni thread verifica una posizione diversa della sequenza per un dato pattern.

- **Grid Size 2D**: La griglia bidimensionale consente di parallelizzare sia sui pattern (asse X) che sulle posizioni nella sequenza (asse Y), massimizzando l'uso delle risorse della GPU.
    - **Asse X:** Ha una dimensione pari a *pat_number*, il numero totale di pattern. Ogni blocco lungo l'asse X è assegnato a un pattern specifico, permettendo di processare tutti i pattern in parallelo.
    - **Asse Y:** Ha una dimensione *gridY*, calcolata come *(my_seq_length + blockSize.x - 1) / blockSize.x*, dove *my_seq_length* è la porzione di sequenza assegnata a ciascun processo MPI. Questo assicura che ogni blocco copra una parte della sequenza, con thread sufficienti per verificare tutte le possibili posizioni di match. Un limite massimo di 65535 blocchi sull'asse Y è imposto per rispettare le specifiche CUDA.



Questa configurazione garantisce che ogni thread verifichi una specifica posizione iniziale per un pattern, con i blocchi che collaborano per trovare la posizione minima di match, memorizzata in *pat_found* tramite operazioni atomiche (descritte nella sezione successiva).

Uso della memoria condivisa (*shared_min*)

Ogni blocco utilizza una variabile di memoria condivisa (*shared_min*) **per memorizzare la posizione minima di match trovata dai thread del blocco per il rispettivo pattern**. La memoria condivisa è veloce e accessibile solo ai thread dello stesso blocco, riducendo la necessità di accessi ripetuti alla memoria globale, che è più lenta. All'inizio, *shared_min* è inizializzata a *NOT_FOUND_ULL* dal primo thread del blocco (*threadIdx.x == 0*), garantendo che tutti i thread partano da una base comune. Dopo che ogni thread verifica una possibile posizione di match, i risultati vengono aggregati in *shared_min* tramite operazioni atomiche, minimizzando i conflitti e preparando il valore finale per la scrittura globale.

Operazioni atomiche per il calcolo del minimo

Per determinare la posizione minima di match in modo sicuro, il kernel utilizza due operazioni atomiche: atomicCAS (Compare-And-Swap) e atomicMin.

- Quando un thread trova un match valido, usa **atomicCAS** per inizializzare *shared_min* con la propria posizione (*global_start*), ma solo **se *shared_min* è ancora *NOT_FOUND_ULL***. Questo evita sovrascritture accidentali.
- **Se *shared_min* è già stato inizializzato**, il thread confronta la propria posizione con il valore corrente e usa **atomicMin** per aggiornarlo solo se la nuova posizione è più bassa.

Lo stesso processo si ripete quando il primo thread del blocco scrive il risultato da *shared_min* a *pat_found* nella memoria globale. Queste operazioni garantiscono che il valore finale in *pat_found* sia il minimo corretto, anche in presenza di più match concorrenti.

Buffer concatenato per i pattern

Invece di gestire ogni pattern come un array separato, il kernel utilizza un buffer continuo (*pattern_buffer*) che concatena tutti i pattern, con un array di offset (*pat_offsets*) per indicare l'inizio di ciascun pattern. Questa scelta riduce la complessità di gestione della memoria sulla GPU, poiché un **unico trasferimento di dati (cudaMemcpy)** è sufficiente per copiare tutti i pattern. Inoltre, consente un **accesso lineare alla memoria durante il confronto** dei caratteri, migliorando le prestazioni grazie alla coalescenza degli accessi.

Integrazione con MPI

Il kernel è progettato per operare su una porzione della sequenza assegnata a ciascun processo MPI (*my_seq_length = seq_length / size*). La posizione globale di ogni thread (*global_start*) è calcolata aggiungendo l'offset del processo *(rank * my_seq_length)* alla posizione locale (*local_start*). Questo permette al kernel di lavorare in parallelo non solo tra i thread CUDA, ma anche tra i processi MPI, con i risultati locali in *pat_found* successivamente ridotti tramite *MPI_Allreduce* per ottenere il minimo globale. Questa scelta assicura che il programma scali bene su più nodi.

updateSeqMatches

Il kernel updateSeqMatches aggiorna in parallelo i contatori di match nella sequenza, incrementando l'array *seq_matches* per ogni posizione coperta dai pattern trovati. Utilizza le posizioni minime memorizzate in *pat_found* per identificare dove incrementare i contatori, garantendo che ogni carattere della sequenza rifletta il numero di match che lo coinvolgono.

Struttura di griglie e blocchi

A differenza di findPatterns, che usa una griglia 2D per coprire pattern e sequenza, **updateSeqMatches adotta una griglia 1D**, poiché opera solo sui pattern locali trovati.

- **Block Size:** Ogni blocco contiene 256 thread, questo valore è stato scelto per bilanciare il parallelismo con l'efficienza, considerando il carico di lavoro relativamente leggero di ogni thread.

- **Grid Size 1D:** La griglia è unidimensionale, con un numero di blocchi pari a *(local_pat_number + 255) / 256*, dove *local_pat_number* è il numero di pattern assegnati al processo MPI corrente. Questo assicura che ogni pattern locale sia processato da un thread dedicato, coprendo tutti i pattern con il minimo numero di blocchi.

Il kernel opera su una porzione di *pat_found* specifica per ogni processo, incrementando atomicamente i contatori in *seq_matches* per le posizioni valide, e si integra con la successiva riduzione MPI per aggregare i risultati globali.

4.2. Checksum

Per ottimizzare la sezione del codice in cui calcolo le checksum non mi conviene lavorare sulla parallelizzazione MPI e/o CUDA ma semplicemente sulla CPU. Come per la versione OpenMP uso la proprietà del modulo con somma che mi permette di “staccare” l’operatore di modulo dal ciclo riducendo di ⅔ il tempo di esecuzione.

4.3. Tempi

Per valutare le prestazioni del programma, non è stato possibile condurre analisi di strong scaling o weak scaling, come fatto nella versione OpenMP, a causa del vincolo hardware che limita l'esecuzione a non più di due processi MPI, ciascuno associato a una GPU. Lo strong scaling richiede di misurare il tempo di un problema fisso con un numero crescente di processori, ma con solo due GPU disponibili, si possono confrontare al massimo una e due configurazioni, insufficienti per una curva di scaling significativa. Analogamente, il weak scaling necessita di aumentare la dimensione del problema proporzionalmente ai processori, ma il limite di due processi non consente di esplorare configurazioni scalabili.

1. ./align_cuda **32768** 0.1 0.3 0.35 **512** 8 2 **512** 8 2 524288 262144 M 609823
2. ./align_cuda **65536** 0.1 0.3 0.35 **1024** 8 2 **1024** 8 2 524288 262144 M 609823
3. ./align_cuda **131072** 0.1 0.3 0.35 **2048** 8 2 **2048** 8 2 524288 262144 M 609823
4. ./align_cuda **262144** 0.1 0.3 0.35 **4096** 8 2 **4096** 8 2 524288 262144 M 609823
5. ./align_cuda **524288** 0.1 0.3 0.35 **8192** 16 2 **8192** 16 2 524288 262144 M 609823
6. ./align_cuda **1048576** 0.1 0.3 0.35 **16384** 32 2 **16384** 32 2 524288 262144 M 609823