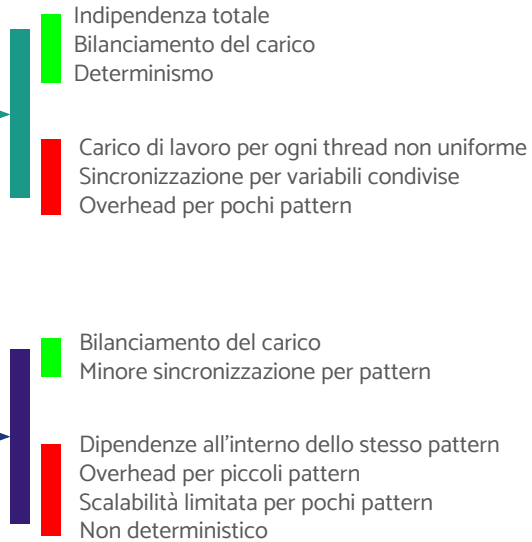
Strategie di Parallelizzazione: OpenMP e CUDA+MPI – Scelte, Tentativi e Ottimizzazioni

Francesco Pennacchietti
Programmazione di Sistemi Embedded e Multicore
Sapienza University of Rome

Versione OpenMP > Ciclo principale

```
/* For each pattern */
for( pat=0; pat < pat_number; pat++ ) {

    /* 5.1. For each possible starting position */
    for( start=0; start <= seq_length - pat_length[pat]; start++) {

        /* 5.1.1. For each pattern element */
        for( lind=0; lind<pat_length[pat]; lind++) {
            /* Stop this test when different nucleotides are found */
            if ( sequence[start + lind] != pattern[pat][lind] ) break;
        }
```



Versione OpenMP > Ciclo principale

```
#pragma omp parallel for schedule(static)
for (pat = 0; pat < pat_number; pat++) {
        unsigned long local_found = NOT_FOUND;      // Dichiarandola all'interno del ciclo parallelo ogni thread avrà la propria copia
        // Ciclo intermedio (sequenziale) (itera su tutte le possibili posizioni di partenza della sequenza per il pattern corrente)
        for (start = 0; start <= seq_length - pat_length[pat]; start++) {
                // Ciclo interno (sequenziale) (itera sugli elementi del pattern e verifica la corrispondenza con la sequenza)
                for (lind = 0; lind < pat_length[pat]; lind++) {
                        if (sequence[start + lind] != pattern[pat][lind]) {
                                break;
                        }
                }
                if (lind == pat_length[pat]) {
                        local_found = start;
                        break;
                }
        }
        // Se il pattern è stato trovato, aggiorna l'indice e incrementa i match
        if (local_found != NOT_FOUND) {
                pat_found[pat] = local_found;
                #pragma omp critical
                {
                        pat_matches++;
                        increment_matches(pat, pat_found, pat_length, seq_matches);
                }
        }
}
```

Serve a tenere traccia della posizione iniziale (start) nella sequenza dove un pattern (pat) viene trovato per la prima volta durante il ciclo su start. È una variabile temporanea locale al threads usata per **evitare accessi concorrenti alla variabile globale pat_found[pat]** durante la ricerca, riducendo il rischio di race condition e migliorando le prestazioni.

L'assegnamento **pat_found[pat] = local_found non ha bisogno di stare in una sezione critica** poiché "pat" sarà sempre diverso e quindi una volta che salvo lo starting point di un pattern questo non verrà più toccato.

pat_matches ha bisogno di essere in una sezione critica poiché è una variabile che viene aggiornata da tutti i threads.

La funzione **increment_matches** sta in una sezione critica poiché diversi pattern potrebbero andare ad aggiornare le stesse entry di seq_matches.

Versione OpenMP > Ciclo principale > Tentativi

```
#pragma omp parallel
        {
                unsigned long *seq_matches_private = calloc(seq_length, sizeof(unsigned long));
                #pragma omp for schedule(static) reduction(+:pat_matches)
                for (int pat = 0; pat < pat_number; pat++) {
                        unsigned long local_found = NOT_FOUND;
                        ...
                        if (local_found != NOT_FOUND) {
                                pat_found[pat] = local_found;
                                pat_matches++;

                                unsigned long base_index = local_found;
                                for (unsigned long ind = 0; ind < pat_length[pat]; ind++) {
                                        seq_matches_private[base_index + ind]++;
                                }
                        }
                }
                #pragma omp critical
                {
                        for (unsigned long i = 0; i < seq_length; i++) {
                                seq_matches[i] += seq_matches_private[i];
                        }
                }
                free(seq_matches_private);
        }
```

Sostituisco la chiamata ad increment_matches con l'effettivo codice ed elimino l'incremento di ogni elemento della sequenza dalla sezione critica.

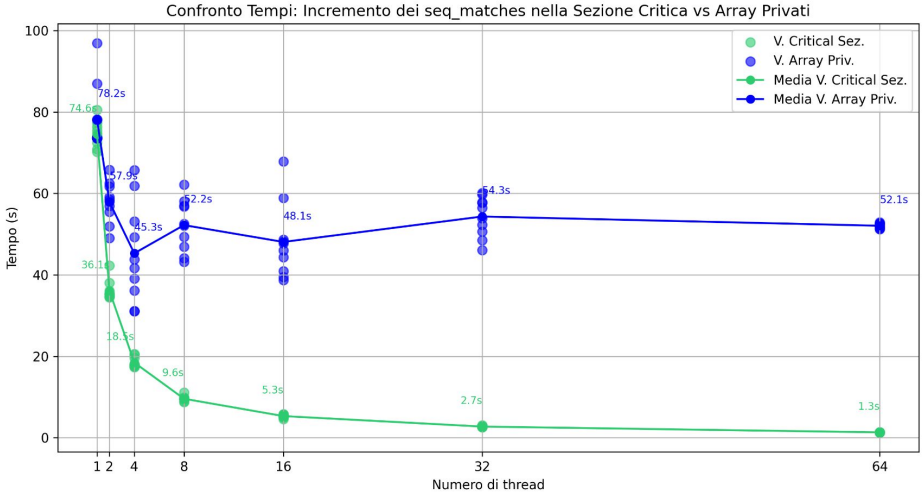
Per farlo creo un array privato per ogni thread in cui esso salva le sue seq_matches. Successivamente, fuori dal for parallelo ma sempre dentro la sezione parallela tutti i threads aggiungeranno la loro seq_matches_private alla seq_matches globale.

Questa operazione la faccio in una sezione critica per evitare race condition ma il collo di bottiglia che questa crea è molto più piccolo: omp_num_threads esecuzioni sequenziali a fronte delle pat_number esecuzioni sequenziali di prima !

Ho abbandonato l'idea degli array privati al thread poiché anche se la memoria heap è più spaziosa, l'overhead che questo comportava superava i benifici di non usare la sezione critica.

Ho provato ad eliminare anche l'ultima sezione critica sostituendola con una reduction su seq_matches ma il costo in termini di memoria (stack) era elevato.

Versione OpenMP > Array privati e Sezione Critica

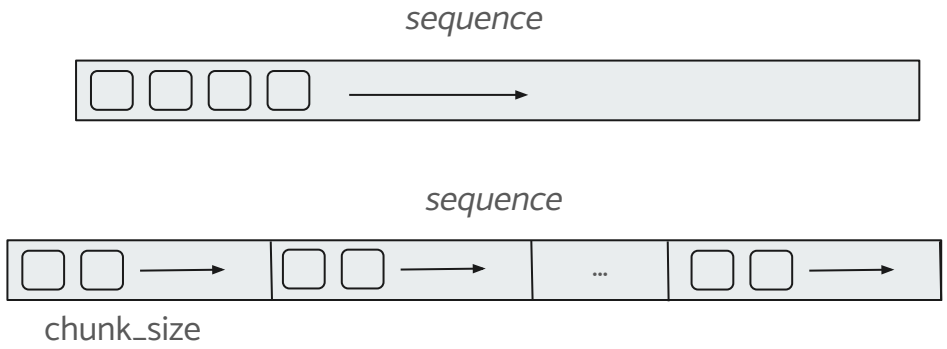


Nella versione con array privati, ogni thread alloca un proprio array seq_matches_priv, aumentando significativamente il consumo di memoria, il che può portare a maggiori **cache misses** ed un **peggior utilizzo della gerarchia di memoria**. Inoltre, la successiva **fase di fusione dei dati in una sezione critica introduce un collo di bottiglia significativo** poiché tutti i thread devono serializzare il loro accesso all'array globale seq_matches, limitando il parallelismo effettivo.

La **maggiore varianza** nei tempi di esecuzione inoltre indica un peggior bilanciamento del carico tra i thread, probabilmente dovuto a un accesso meno efficiente alla memoria e a un costo elevato della fase di merging dei risultati.

Versione OpenMP > generate_rng_sequence

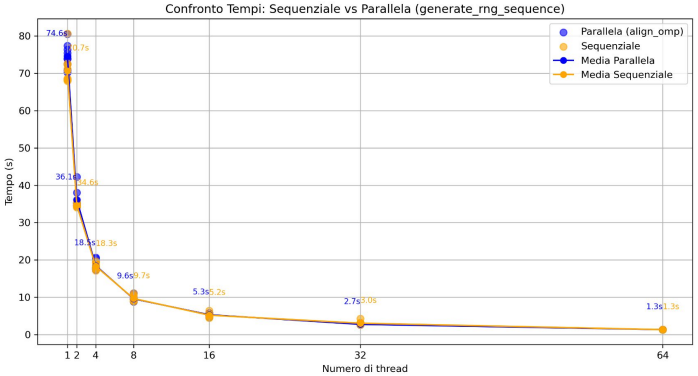
```
void generate_rng_sequence(rng_t *random, float prob_G, float prob_C, float prob_A, char *seq, unsigned long length) {
  unsigned int n_threads = omp_get_max_threads();
  #pragma omp parallel
  {
    int tid = omp_get_thread_num();
    unsigned long chunk_size = (length + n_threads - 1) / n_threads;
    unsigned long start = tid * chunk_size;
    unsigned long end = (start + chunk_size < length) ? start + chunk_size : length;

    rng_t thread_state = *random;
    rng_skip(&thread_state, start);

    for (unsigned long ind = start; ind < end; ind++) {
      double prob = rng_next(&thread_state);
      if (prob < prob_G) seq[ind] = 'G';
      else if (prob < prob_C) seq[ind] = 'C';
      else if (prob < prob_A) seq[ind] = 'A';
      else seq[ind] = 'T';
    }
  }
  rng_skip(random, length);
}
```

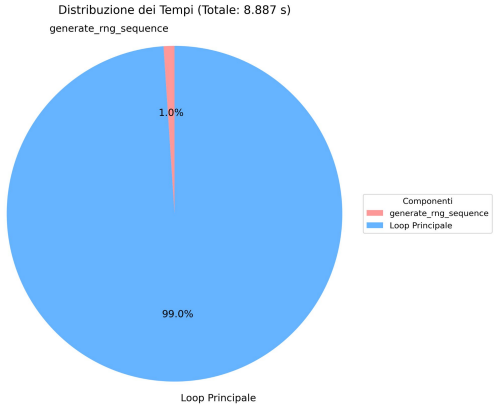


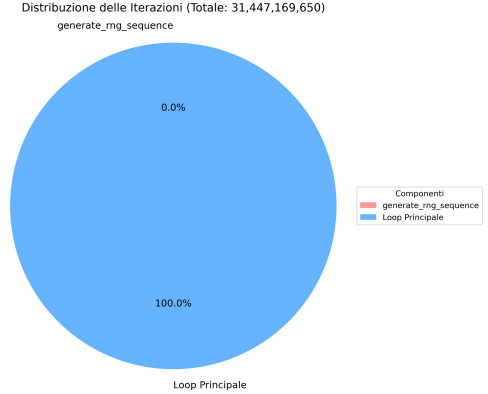
Grazie a rng_skip **mantengo la riproducibilità dei numeri casuali** anche senza un esecuzione sequenziale ordinata.

Avanza lo stato globale del generatore casuale per riflettere l'intero processo. Questo consente di preservare la continuità dei numeri casuali quando la funzione viene chiamata di nuovo. Ricorda che è chiamata dentro un ciclo for!

Versione OpenMP > generate_rng_sequence



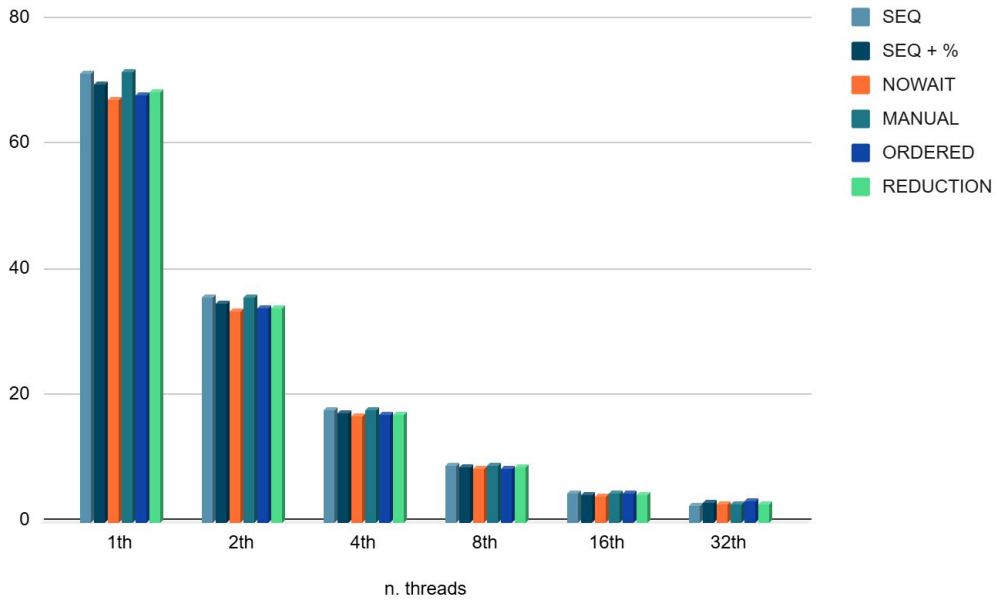




Distribuzione dei Tempi: Il grafico a torta mostra che il loop principale consuma il 99.9% del tempo totale (8.883 s su 8.887 s), Mentre `generate_rng_sequence` contribuisce solo allo 0.1% (0.094 s). Questo indica che il loop principale è il principale bottleneck del programma, con il lavoro di generazione della sequenza che ha un impatto trascurabile sui tempi totali.

Distribuzione delle Iterazioni: Il grafico a torta evidenzia che il loop principale esegue il 99.995% delle iterazioni totali (31,445,604,828 su 31,447,169,650), mentre `generate_rng_sequence` rappresenta solo lo 0.005% (1,564,822 iterazioni). Questo riflette la natura computazionalmente intensiva del loop principale, che itera su milioni di pattern e posizioni, rispetto alla generazione sequenziale di una sequenza di lunghezza fissa.

Versione OpenMP > checksum




```
#pragma omp parallel for ordered
for (int ind = 0; ind < pat_number; ind++) {
        if (pat_found[ind] != (unsigned long)NOT_FOUND) {
                #pragma omp ordered
                checksum_found = (checksum_found + pat_found[ind]);
        }
}
checksum_found = checksum_found % CHECKSUM_MAX;
```

```
#pragma omp parallel
{
        unsigned long local_sum = 0;
        #pragma omp for nowait
        for (int ind = 0; ind < pat_number; ind++) {
                if (pat_found[ind] != (unsigned long)NOT_FOUND)
                        local_sum += pat_found[ind];
                }
        }
        #pragma omp atomic
        checksum_found += local_sum;
}
checksum_found %= CHECKSUM_MAX;
```

```
#pragma omp parallel for reduction(+:checksum_found) schedule(static)
for( ind = 0; ind < pat_number; ind++) {
        if (pat_found[ind] != (unsigned long)NOT_FOUND)
                checksum_found += pat_found[ind];
}
checksum_found = checksum_found % CHECKSUM_MAX;
```

Versione CUDA+MPI > Punti critici emersi dal report nsys

1. CUDA GPU MemOps Summary
    a. Perché i trasferimenti Host-to-Device sono così numerosi?
    b. Tempo significativo in cudaMemcpy Device-to-Host

2. CUDA GPU Kernel Summary (findPatterns cuore computazionale)

3. Ottimizzazione della Comunicazione MPI

1. CUDA GPU MemOps Summary

By Time:

- **[CUDA memcpy Device-to-Host]** (69.0%): 273M ns (~0.27 secondi) su 4 operazioni
- **[CUDA memcpy Host-to-Device]** (30.7%): 122M ns (~0.12 secondi) su 131.080 operazioni
- **[CUDA memset]** (0.3%): 1.2M ns, trascurabile.

By Size:

- **[CUDA memcpy Device-to-Host]** 537.9 MB su 4 operazioni (134.5 MB in media)
- **[CUDA memcpy Host-to-Device]** 137.8 MB su 131.080 operazioni (~0 MB in media)
- **[CUDA memset]** 537.9 MB su 4 operazioni (134.5 MB in media)

1a. Perché i trasferimenti Host-to-Device sono così numerosi?

*Questo è dovuto ad un operazione che viene eseguita in **fase di inizializzazione del programma**, prima ancora di far partire il timer carico tutti i pattern in memoria device.*

```
for( ind=0; ind<pat_number; ind++ ) {
        CUDA_CHECK_FUNCTION( cudaMalloc( &(d_pattern_in_host[ind]), sizeof(char *) * pat_length[ind] ) );
        CUDA_CHECK_FUNCTION( cudaMemcpy( d_pattern_in_host[ind], pattern[ind], pat_length[ind] * sizeof(char), cudaMemcpyHostToDevice ) );
        }
```

Questo spiega le 131.080 operazioni, infatti, **`pat_number = 32,768 + 32,768 = 65,536`** (ho sia i pattern random che i pattern sample). Poi dovrò raddoppiare questo numero poiché il ciclo viene eseguito due volte (una per ogni processo MPI), quindi, **`65,536 * 2 = 131,072`** iterazioni del ciclo. Aggiungo altre 8 operazioni (**`4 altri trasferimenti * 2 processi mpi`**), per un totale di 131.080 operazioni che corrisponde esattamente al numero di `cudaMemcpy Host-to-Device`nel profiling.

1b. Tempo significativo in cudaMemcpy Device-to-Host

Anche se i trasferimenti Host-to-Device dominano il numero di chiamate (131.080), **i 4 trasferimenti Device-to-Host occupano 0.3 secondi e muovono 537.919 MB**. Questo è un costo significativo, soprattutto considerando che rappresentano il **64.5% del tempo totale di MemOps**.

A cosa è dovuto? Dopo il calcolo sul kernel `findPatterns`, il programma riporta i risultati dalla GPU alla CPU in blocchi grandi (media 134 MB).

Questo potrebbe sembrare un punto critico, ma in realtà **l'applicazione è chiaramente GPU-bound**!

- Il kernel **`findPatterns` domina il runtime**, impiegando 125 secondi su 126.
- **Ridurre i 0.3 secondi dei trasferimenti Device-to-Host non cambierà significativamente le prestazioni**.
- **Complicare eccessivamente il codice per ottimizzare questi trasferimenti sarebbe inefficiente rispetto ai reali colli di bottiglia**.

2. CUDA GPU Kernel Summary

Dall'analisi dei tempi di esecuzione, il kernel **`findPatterns` è il chiaro collo di bottiglia** dell'applicazione:

- **Occupa il 100% del tempo di esecuzione della GPU**, con 125.26 secondi su 125.27 totali.
- **Ogni esecuzione del kernel richiede circa 62.6 secondi**, e viene lanciato **due volte**.
- Il kernel **updateSeqMatches**, invece, è del tutto **trascurabile**, impiegando appena 16 microsecondi per chiamata.

Questo conferma che l'applicazione è **fortemente GPU-bound** ⇒ Qualsiasi ottimizzazione che non riguardi direttamente findPatterns avrà un impatto minimo sulle prestazioni generali.

Strategie di Ottimizzazione

1. Ridurre il Numero di Confronti
2. Ottimizzare l'Uso delle Atomiche
3. Aumentare la Località dei Dati
4. Bilanciare il Carico

... > 2.1. > Ridurre il Numero di Confronti

```
// Pre-filtraggio: controlla il primo carattere
if (sequence[start_position] != patterns[pat][0]) return;

for (int i = 1; i < pat_length[pat]; i++) {
        if (sequence[start_position + i] != patterns[pat][i]) return;
}
```

Il pre-filtraggio mira a ridurre il numero di confronti completi tra la sequenza e i pattern controllando il primo carattere (sequence[start_position] == patterns[pat][0]) prima di entrare nel ciclo for.

L'obiettivo è evitare di eseguire il ciclo per quei thread i cui pattern falliscono già al primo carattere, riducendo così il carico computazionale complessivo.

Il pre-filtraggio **funziona bene quando scarta una quantità significativa di pattern**, riducendo gli accessi alla memoria e le iterazioni nel kernel.

In questo caso però la memoria globale e le istruzioni atomiche dominano i tempi di esecuzione, quindi **l'ottimizzazione non ha un impatto visibile!**

… > 2.2. > Ottimizzare l’Uso delle Atomiche

L’obiettivo dell’ottimizzazione è ridurre il costo delle operazioni atomiche (`atomicCAS` e `atomicMin`) nel kernel, identificate come un **potenziale collo di bottiglia a causa della contesa tra thread che aggiornano lo stesso elemento di `pat_found`**.

La strategia adottata è una **riduzione locale per blocco**, che minimizza il numero di atomiche trasferendo il lavoro di aggregazione alla memoria condivisa.

Perché è un Collo di Bottiglia?

- **Costo delle Atomiche**: Su GPU, `atomicCAS` e `atomicMin` sono operazioni lente (10-100 cicli), soprattutto con contesa, rispetto ai confronti (1-2 cicli).
- **Numero di Operazioni**: Se un pattern ha molti match (es. 10,000 occorrenze in `sequence`), ci saranno fino a 10,000 atomiche su `pat_found[pat]`, causando serializzazione e ritardi.
- **Evidenza Indiretta**: Il pre-filtraggio non ha ridotto i tempi, suggerendo che i confronti nel ciclo `for` non dominano; le atomiche sono il sospetto principale.

Problema originale: Nella versione non ottimizzata, ogni thread che trovava un match eseguiva `atomicCAS` e, se necessario, `atomicMin` su `pat_found[pat]`. Migliaia di thread per pattern quindi potevano competere per lo stesso elemento, causando contesa e serializzazione.

Soluzione: La riduzione locale limiterebbe le atomiche ad una per blocco relativo ad un pattern. In questo modo trasferisco il lavoro di aggregazione alla memoria condivisa che è molto più veloce e non richiede concorrenza globale.

Algoritmo di Riduzione:

- La riduzione parallela usa un approccio ad albero logaritmico: a ogni iterazione, i thread confrontano coppie di valori a distanza decrescente (`s`), aggiornando il minimo.
- Dopo $\log_2(256) = 8$ passi, `shared_min[0]` contiene il valore minimo del blocco.

Per usare la riduzione locale al blocco devo fare un passo indietro! Cambio l'organizzazione dei threads

Attualmente, il kernel `findPatterns` è configurato in modo tale che ha sull'asse X gli starting point della sequenza da cui partire e sull'asse Y tanti blocchi quanti me ne servono per coprire #pattern considerando un blocco da 256 threads.

Voglio **invertire questa logica** e adottare una nuova configurazione: Avere sull'asse delle X #pattern e per ogni pattern (indice di x) un numero di blocchi tale da coprire seq_length starting point (sempre blocchi da 256 threads).

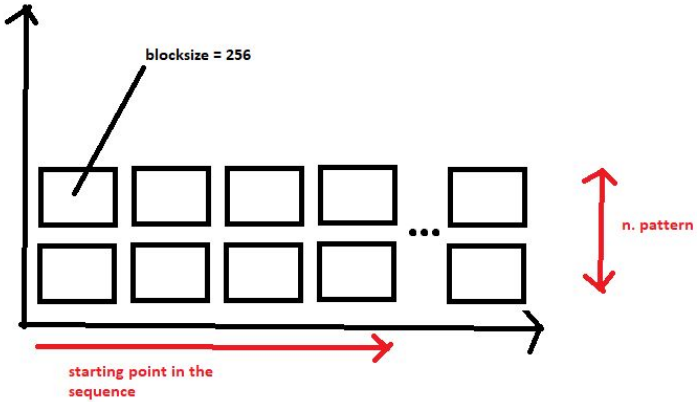
Motivazione del cambiamento

1. **Maggiore efficienza nella gestione della memoria.** Ogni blocco esaminerà lo stesso pattern in posizioni di sequenza diverse, quindi potrà **condividere dati nella shared memory**. In futuro, questo permetterà di introdurre **un'ottimizzazione locale all'interno del blocco**, migliorando le prestazioni.

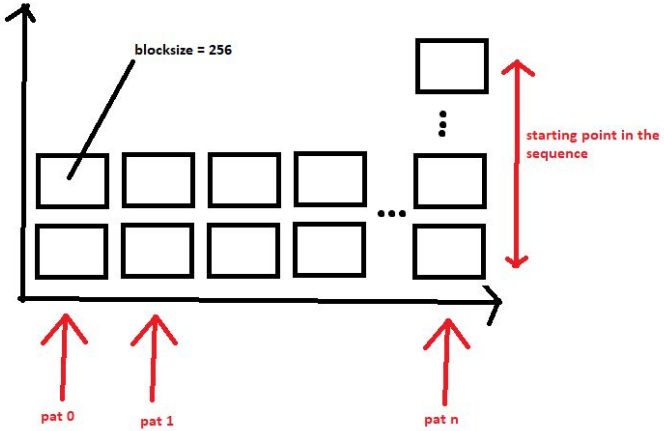
2. **Maggiore parallelismo indipendente.** Poiché ogni pattern è analizzato separatamente, ogni blocco in X sarà indipendente dagli altri, riducendo eventuali conflitti di memoria. La ridistribuzione del carico tra blocchi sarà più bilanciata rispetto alla configurazione attuale.

3. **Scalabilità migliore.** Questa configurazione è più adatta per una futura ottimizzazione **basata sulla memoria condivisa**. In un secondo momento, sarà possibile implementare una **riduzione locale all'interno del blocco** per trovare il match minimo senza dover sincronizzare tutti i blocchi globalmente.

Versioni a Confronto



Il problema che andrei a riscontrare, introducendo l'uso della memoria condivisa è che la riduzione locale **mescolerebbe i risultati di pattern diversi all'interno dello stesso blocco** cercando un minimo. Il minimo di n pattern che controllano i match a partire dallo stesso starting point non ha una valenza effettiva logicamente. Che cosa rappresenterebbe? Il pattern minimo che fa match a partire da quello starting point? Non ha senso!



Con questo **nuovo approccio**, facendo il minimo del blocco (costo molto più piccolo perché memoria condivisa) ottengo lo starting point più piccolo su cui il pattern_i matcha. In realtà se ho più blocchi per lo stesso pattern dovrò comunque fare un'operazione di confronto tra i minimi di ogni blocco relativo al pattern_i ma i confronti saranno molti di meno che fare seq_length minimi globali.

Versioni a Confronto

NOTA: CUDA GPU MemOps Summary

Il tempo e lo spazio speso nelle operazioni di memoria non ha subito variazioni significative. Questo suggerisce che **il miglioramento non deriva da un minor numero di trasferimenti di dati tra CPU e GPU, ma da un'efficienza maggiore nell'elaborazione interna dei dati nella GPU.**

1. CUDA API Summary

	Nuova Configurazione	Vecchia Configurazione	Differenza
cudaMemcpy (Total Time)	**959 ms (53.2%)**	1447 ms (62.9%)	-34%
cudaMalloc (Total Time)	444 ms (24.6%)	444 ms (19.3%)	≈ uguale
cudaFree (Total Time)	399 ms (22.1%)	409 ms (17.8%)	≈ uguale
cudaLaunchKernel (Total Time)	633 µs	612 µs	≈ uguale

Analisi: La memcpy ha subito un miglioramento significativo, riducendo il tempo totale del 34%. Questo suggerisce che il nuovo schema di lancio dei kernel potrebbe aver migliorato il layout dei dati in memoria. Il tempo di cudaMalloc e cudaFree è rimasto più o meno invariato, il che ha senso perché l'allocazione della memoria non dipende molto dalla disposizione della griglia.

2. CUDA GPU Kernel Summary

Kernel	Nuova Configurazione (Total Time)	Vecchia Configurazione (Total Time)	Differenza
findPatterns	**594 ms**	1079 ms	-45%
updateSeqMatches	56 µs	55 µs	≈ uguale

Analisi: Il kernel findPatterns ha ridotto il suo tempo di esecuzione di circa 45%, il che è un miglioramento netto e significativo. Il kernel updateSeqMatches non ha subito variazioni importanti, il che è normale perché non è stato coinvolto nel cambio di strategia.

Ora la Shared Memory

Dopo aver cambiato la configurazione della griglia posso finalmente introdurre la Shared Memory. Inizialmente percorro la strada progettuale già tracciata utilizzandola per salvare i minimi ma successivamente provo anche una strada alternativa che consiste nel utilizzare la memoria condivisa per memorizzare il pattern su cui lavoro in quel momento.

`start_position minima locale al blocco`

`extern __shared__ unsigned long long shared_min[]:`
La shared memory viene usata per memorizzare il valore minimo locale (`start_position`) trovato dai thread del blocco per un dato pattern. Solo un valore (`shared_min[0]`) viene aggiornato usando operazioni atomiche (`atomicCAS` e `atomicMin`) nella shared memory.

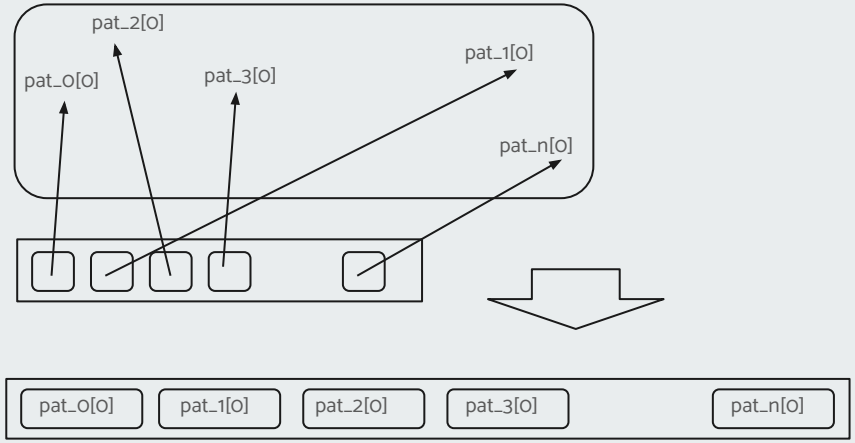
Scopo: Ridurre il numero di operazioni atomiche sulla memoria globale (`pat_found`) calcolando prima un minimo locale nel blocco e aggiornando `pat_found` solo una volta per blocco con il valore minimo.

`pattern su cui il blocco di threads lavora`

`extern __shared__ char shared_pattern[]:`
La shared memory viene usata per caricare il pattern corrente (`patterns[pat]`) all'interno del blocco. Tutti i thread collaborano per copiare i caratteri del pattern dalla memoria globale alla shared memory, e poi usano `shared_pattern` per il confronto con `sequence`.

Scopo: Ridurre gli accessi ripetuti alla memoria globale per leggere `patterns[pat][i]` durante il ciclo di confronto. Ogni carattere del pattern viene caricato una sola volta nella shared memory invece di essere letto da ogni thread dalla memoria globale.

… > 2.3. > Aumentare la Località dei Dati



Evitare l'uso di array di puntatori nella memoria globale

Attualmente, i pattern sono memorizzati come un array di puntatori `char **patterns`, dove ogni puntatore riferisce a una stringa separata. Questo è inefficiente su GPU perché:

- Gli accessi ai pattern risultano **non coalescenti**, poiché ogni thread accede a un indirizzo arbitrario.
- A ogni accesso `patterns[pat][i]`, la GPU deve prima recuperare l'indirizzo e poi il valore, introducendo **latenza aggiuntiva**.

Soluzione: Allocare un buffer lineare per tutti i pattern

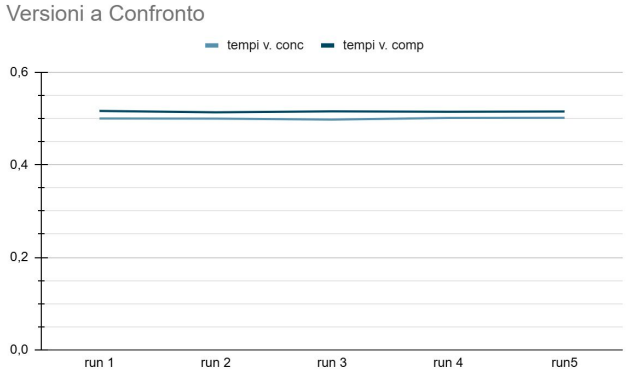
Un'ottima strategia per migliorare la località è **concatenare tutti i pattern in un unico array continuo di memoria globale** e usare un array di **offset** per accedere ai singoli pattern.

… > Aumentare la Località dei Dati > Valutazione cambiamento

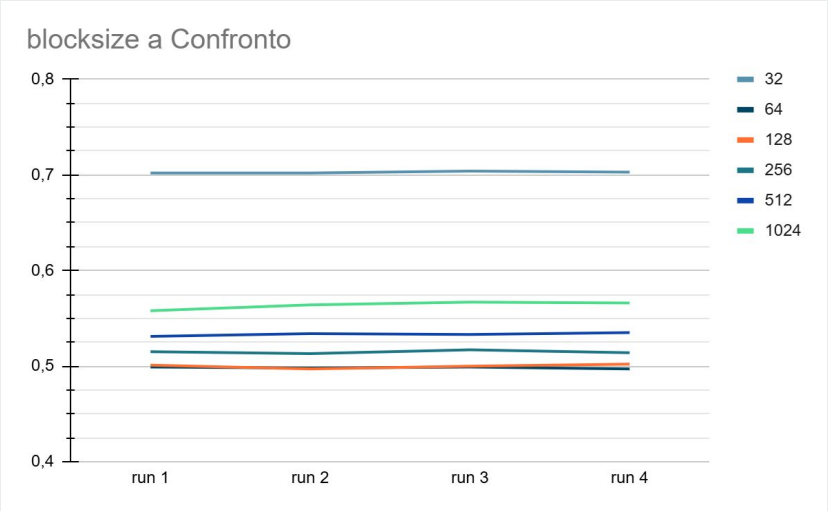
1. Tempo di Esecuzione delle Chiamate CUDA API: La **cudaMemcpy** è leggermente più veloce nella versione con pattern continui (961 ms contro 1025 ms), riducendo il tempo di trasferimento dei dati. La **cudaMalloc** e **cudaFree** rimangono comparabili tra le due versioni, con variazioni minime. Il **cudaLaunchKernel** ha un tempo inferiore del **10%** nella versione con pattern continui, segnalando una minore latenza nell'avvio del kernel.

2. Tempo di Esecuzione del Kernel. Il kernel principale **findPatterns** ha un tempo ridotto dell'**8.6%**, il che suggerisce che l'approccio con pattern continui sia più efficiente nel calcolo.

3. Operazioni di Memoria GPU. Il trasferimento **Host-to-Device (HtoD)** ha un carico di dati maggiore del 42.5% nella versione con pattern continui, questo non incide negativamente sul tempo totale perché il collo di bottiglia principale è il calcolo GPU e non la memoria. Il tempo del kernel findPatterns è di 594 ms (nella versione con pattern continui), mentre il tempo di HtoD è solo 33 ms.



… > 2.4. > Bilanciare il Carico



1. Commento su NVTX Range Summary. I due kernel hanno i seguenti tempi, 365 ms e 397 ms, con una differenza di circa 33 ms. **La variabilità tra le due esecuzioni è moderata: 33 ms su un tempo medio di 381 ms (~8.5% di differenza).** Questo suggerisce che le due esecuzioni hanno carichi di lavoro simili, ma c'è una certa disomogeneità che potrebbe derivare dal fatto che i blocchi devono lavorare su pattern di complessità diverse.

2. Commento sui tempi di esecuzione con diverse blockSize

Andamento dei tempi:

- **32 → 64**: Forte miglioramento (da 0.70275 s a 0.49825 s, -29%). Passare da 32 a 64 thread per blocco aumenta significativamente l'efficienza, probabilmente grazie a una migliore occupazione della GPU e a una riduzione degli overhead di gestione dei blocchi.
- **128 → 256 → 512 → 1024**: I tempi aumentano progressivamente (0.5 s → 0.51475 s → 0.53325 s → 0.56375 s). Con blocchi più grandi, le prestazioni peggiorano, con un incremento totale del 13% da 64 a 1024.

Variabilità: La variabilità è minima con `blockSize` 32 e 64 (2 ms), ma cresce con blocchi più grandi, raggiungendo 9 ms con 1024. Questo suggerisce che **blocchi più grandi introducono una maggiore instabilità nell'esecuzione**, forse per squilibri nel carico o sincronizzazioni più complesse.

3. Ottimizzazione della Comunicazione MPI > Sostituzione di Operazioni Bloccanti con Non Bloccanti

Analizziamo come sostituire le operazioni bloccanti con le loro controparti **non bloccanti** (`MPI_Ireduce`, `MPI_Iallreduce`) per migliorare l'**overlapping tra computazione e comunicazione**, ridurre il tempo di inattività dei processi e aumentare la scalabilità del sistema.

- Dopo il primo kernel (`findPatterns`), ogni processo ha una copia locale di `pat_found` con le posizioni dei pattern trovate nella sua porzione di sequenza. **`MPI_Allreduce`** con `MPI_MIN` calcola il minimo tra tutte le copie locali per ogni elemento di `pat_found`, producendo un array globale `h_pat_found` condiviso tra i processi. Questo è necessario perché il secondo kernel (`updateSeqMatches`) richiede i risultati globali.

- Dopo il secondo kernel (`updateSeqMatches`), ogni processo ha una copia locale di `seq_matches` con i conteggi dei match nella sua porzione di lavoro. **`MPI_Reduce`** somma questi conteggi in `h_seq_matches` sul rank 0 che poi servirà per calcolare i checksum.

… > MPI_Allreduce

```
MPI_Request request;
MPI_Iallreduce(pat_found, h_pat_found, pat_number,
        MPI_UNSIGNED_LONG_LONG, MPI_MIN,
        MPI_COMM_WORLD, &request
);
// Lavoro utile qui (es. preparazione per updateSeqMatches)
MPI_Wait(&request, MPI_STATUS_IGNORE);
```

`MPI_Allreduce` è un'operazione collettiva che coinvolge tutti i processi in `MPI_COMM_WORLD` (nel mio caso, 2 processi). Combina i dati locali con un'operazione (qui `MPI_MIN`) e distribuisce il risultato a tutti.

Algoritmo tipico:

1. **Riduzione**. I dati locali vengono aggregati in un processo "radice" (non esplicito in `MPI_Allreduce`, ma scelto internamente dall'implementazione).
2. **Broadcast**: Il risultato aggregato viene trasmesso a tutti i processi. Con solo 2 processi, un algoritmo comune è lo **scambio bidirezionale**: Rank 0 e Rank 1 si scambiano i loro array. Entrambi calcolano localmente `MPI_MIN`.

L'operazione non bloccante funziona avviando la riduzione in background e permettendo al programma di continuare con altro lavoro che non dipende dal risultato dell'operazione. Completa infine l'operazione con `MPI_Wait`.

… > MPI_Reduce

```
MPI_Request request;
MPI_Ireduce(seq_matches, h_seq_matches, seq_length,
        MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD, &request);

// Lavoro utile qui (deallocazione della memoria GPU e prima parte della sezione checksum)
MPI_Wait(&request, MPI_STATUS_IGNORE);
```

`MPI_Reduce` è un'operazione collettiva che coinvolge tutti i processi in `MPI_COMM_WORLD` (nel mio caso, 2 processi). Aggrega i dati locali di ciascun processo e invia il risultato a un processo **radice** specificato (qui Rank 0).

Con soli **2 processi**, un algoritmo comune è il semplice **scambio di dati**: Rank 1 invia il suo array a Rank 0, che calcola la somma (`MPI_SUM`) e memorizza il risultato.

L'operazione non bloccante funziona avviando la riduzione in background. **Sovrapponendo comunicazione e computazione** permette ai processi di eseguire altre operazioni mentre il dato viene ridotto. L'operazione viene completata con `MPI_Wait`.

… > Risultati sostituzione

Versione con MPI non bloccante = 69.901006 s

Versione con MPI bloccante = 66.912015 s.

Possibili spiegazioni della differenza

Overhead di `MPI_Ireduce` e `MPI_Iallreduce`

Le operazioni non bloccanti introducono un overhead per la gestione delle richieste (`MPI_Request`) e la sincronizzazione successiva con `MPI_Wait`. Con un dataset relativamente piccolo e 2 processi, questo overhead combinato potrebbe superare il beneficio della sovrapposizione.

Lavoro utile troppo breve

Il lavoro tra `MPI_Iallreduce` e il suo `MPI_Wait` (preparazione del secondo kernel) e tra `MPI_Ireduce` e il suo `MPI_Wait` (deallocazione strutture in GPU e una parte di checksum) è rapido. Ad esempio, la deallocazione GPU richiede 0.79 sec (da `cudaFree` nel report), e la preparazione del kernel `updateSeqMatches` è minima (0.033 sec). Se la comunicazione MPI combinata di `MPI_Iallreduce` e `MPI_Ireduce` è più lunga (es. 1-2 sec), non c'è abbastanza lavoro da sovrapporre, e l'attesa esplicita di `MPI_Wait` in entrambe le chiamate aggiunge ritardo rispetto alla versione bloccante con `MPI_Allreduce` e `MPI_Reduce`.

Comportamento del cluster

Il cluster potrebbe non ottimizzare bene le operazioni non bloccanti con solo 2 processi. `MPI_Reduce` e `MPI_Allreduce` potrebbero sfruttare implementazioni più efficienti riducendo l'overhead.

Thank you for the attention!