Relazione HPC: Operatore Skyline

Palazzini Luca
Matricola: 0001070910

19 giugno 2025

1 Introduzione

In questo progetto ho implementato e valutato l'operatore *skyline*, un algoritmo che individua i punti non dominati in uno spazio a D dimensioni. Vengono presentate due versioni parallele del codice: una basata su OpenMP per CPU multicore e l'altra su CUDA per GPU. L'obiettivo è descrivere le strategie di parallelizzazione adottate attenendosi a i pattern di programmazione concorrente.

Tutti i test sono stati eseguiti su un sistema con:

- **CPU**: Intel i5 9400F (6 cores, 6 logical cores, 2.90 GHz)
- **GPU**: GTX 1660ti (1536 CUDA cores, 1140-1455 MHz)

2 Strategie di Parallelizzazione

2.1 Versione OpenMP

Analisi delle dipendenze L'algoritmo originale esegue un doppio ciclo annidato per confrontare ogni punto del dataset con tutti gli altri. Il ciclo esterno su i itera sui punti candidati allo skyline, mentre il ciclo interno su j controlla se i domina j. Dal punto di vista delle dipendenze, le iterazioni del ciclo esterno su i presentano *loop-carried dependencies*, in quanto ogni iterazione potrebbe modificare lo stato dei flag dello skyline mentre un'altra iterazione del ciclo potrebbe controllarla. Al contrario il ciclo interno su j non presenta alcuna *loop-carried dependency*.

Implementazione Per ridurre l'overhead di creazione dei thread, la *parallel region* è stata posta all'esterno del ciclo su i, in modo da inizializzare una sola volta il team di thread.

La variabile r, che rappresenta il numero di punti rimanenti nello skyline, è aggiornata con una *reduction* in OpenMP. Questo evita condizioni di gara e garantisce coerenza senza introdurre blocchi espliciti o sincronizzazioni costose. La riduzione è effettuata con l'operatore sottrazione, in quanto ogni volta che un punto viene rimosso, si decrementa r.

La parallelizzazione vera e propria è applicata al ciclo interno su j tramite la direttiva `#pragma omp for` con `schedule(guided, 4096)`. Questo tipo di scheduling è stato scelto perché la quantità di lavoro per ogni iterazione può variare significativamente: quando un punto viene escluso dallo skyline, non sarà più confrontato nelle iterazioni successive, rendendo il carico irregolare.

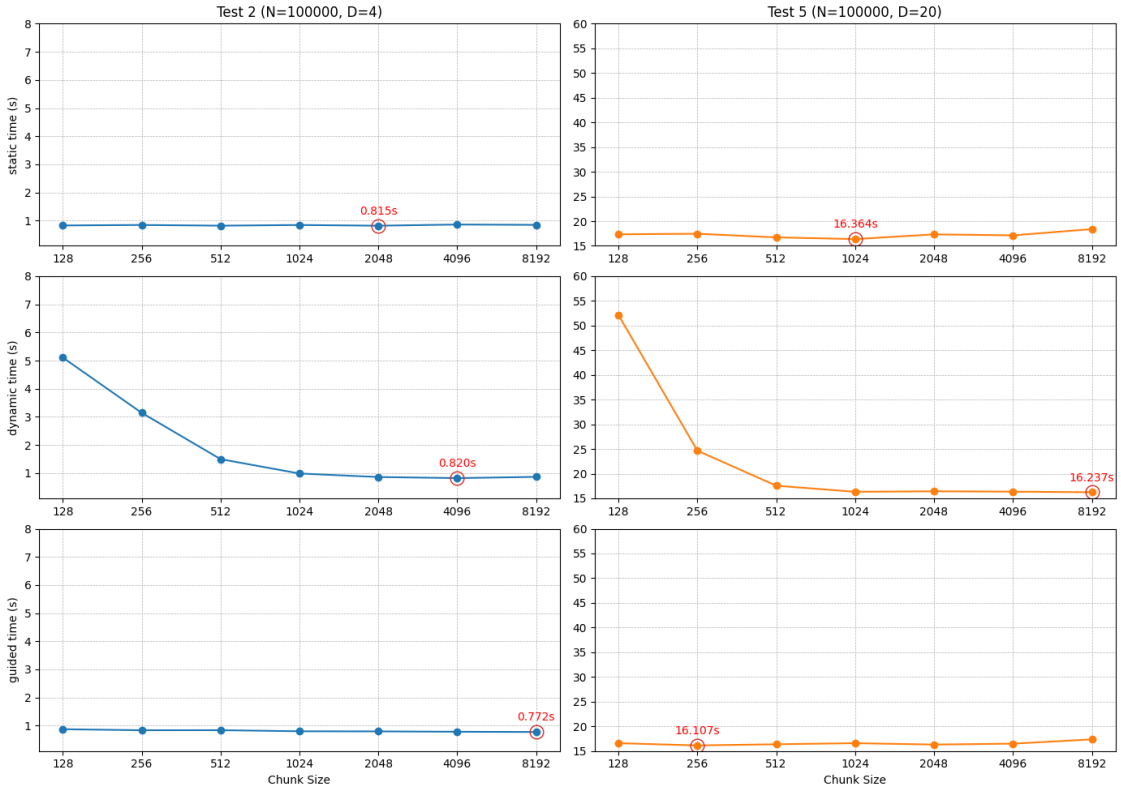

Figura 1: Grafici dei test della granularitá

Come possiamo notare dal grafico, l'utilizzo di una schedule dinamica, con fine granularitá, importa un enorme costo di overhead in performance rispetto a quella statica. Nonostante ció, le schedule di tipo dinamico e guided risultano essere piú

efficienti a granularitá grossolana, perció ho deciso di utilizzare una schedule di tipo *guided*. I test sono stati svolti su:

- **Test 2**: 100000 punti (N) a 4 dimensioni (D);
- **Test 5**: 100000 punti (N) a 20 dimensioni (D).

2.2 Versione CUDA

Mappatura dei thread Per la versione CUDA ho deciso di assegnare ogni thread a un punto i del dataset: questo thread si occupa di confrontare il proprio punto con tutti gli altri j. Questo approccio è piuttosto diretto e semplice da implementare, e funziona bene per sfruttare il parallelismo massivo della GPU. Il codice è stato suddiviso in tre kernel principali che vengono eseguiti uno dopo l'altro.

Implementazione

- **Inizializzazione**: Un primo kernel viene utilizzato per inizializzare tutti i flag dello skyline a 1, tramite la direttiva *cudaMemset*, assumendo inizialmente che ogni punto faccia parte dello skyline.
- **Confronto tra punti**: Un secondo kernel esegue l'algoritmo vero e proprio. Viene utilizzato della *shared memory* per caricare dinamicamente un subset di punti, utilizzati dall'intero gruppo di thread per calcolare lo skyline. In pratica viene cerata una *Sliding window* sulla global memory, caricando i punti nella shared memory, confrontandoli, e passando al blocco di punti successivo. Ogni thread analizza il proprio punto i confrontandolo con tutti i punti appartenenti alla shared memory. Se durante i confronti il thread trova un punto j che domina il suo i, allora aggiorna il flag del punto i nella global memory per indicare che non fa parte dello skyline.
- **Riduzione**: Un terzo kernel esegue una *tree reduction* per calcolare il numero totale di punti che rimangono nello skyline, sommando i flag ancora attivi.

Problema di shared memory Un limite importante è la dimensione massima della shared memory. Se il numero di dimensioni D è troppo alto, ovvero quando $D \geq 1536$, non é possibile caricare nemmeno due punti in *Shared memory*. Per questo vi é un altro kernel, che utilizza esclusivamente global memory per valori di D grandi. La logica rimane la stessa, ma le performance calano in quanto accessi a global memory hanno costo elevato.

3 Analisi delle Performance

3.1 Versione OpenMP

Dai risultati sperimentali riportati si osserva come l'efficienza e lo speedup del programma varino sensibilmente in base alla dimensione del problema, ossia al numero di dimensioni D e al carico computazionale complessivo.

Nel **Test 2** (con $D = 4$), il workload è relativamente leggero e i miglioramenti ottenuti aggiungendo thread sono piuttosto modesti. Ad esempio, passando da 1 a 6 thread, lo speedup cresce da 1 a circa 3, con un'efficienza che cala dal 100% a circa il 50%. Questo calo è dovuto principalmente all'overhead intrinseco nella gestione dei thread e nella sincronizzazione, che pesa più quando la quantità di lavoro per thread è ridotta.

Al contrario, nel **Test 5** (con $D = 20$), il carico di lavoro è molto più elevato e lo scaling è decisamente migliore. Lo speedup con 6 thread raggiunge 4.27, con un'efficienza intorno all'85%, che è un risultato molto più vicino al massimo teorico. Questo indica anche che l'overhead dei thread è meno impattante rispetto al tempo di calcolo, consentendo uno sfruttamento del parallelismo migliore.

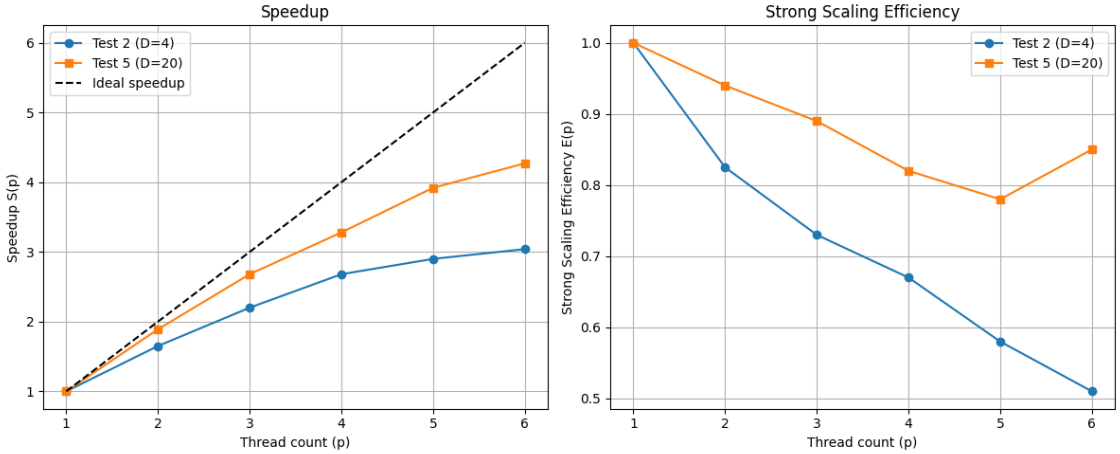

Figura 2: Scaling OpenMP

3.2 Versione CUDA

L'implementazione CUDA ha ottenuto prestazioni decisamente superiori rispetto a quella su CPU, in particolare con dataset di grandi dimensioni e alta dimensionalità.

L'uso della *shared memory* nei kernel ha portato benefici significativi in termini di throughput, riducendo l'accesso alla *global memory* che rappresenta il collo di

bottiglia principale. Tuttavia, come discusso in precedenza, con D molto grandi (oltre 1500) non è più possibile usare shared memory e il kernel deve ripiegare su global memory, causando un peggioramento nelle performance.

I dati dei calcoli seguenti sono tutti stati svolti con test7 ($N = 100000, D = 200$)

Kernel	**Speedup (stima)**	**Throughput Computazionale**	**Throughput Memoria**
shared_skyline	5.13	14.06 GFLOP/s	48.74 GB/s
r_reduction	8.69	53.30 GFLOP/s	53.30 GB/s

Tabella 1: Performance secondo *NVIDIA Nsight Compute*

La tabella descrive valori di performance dei kernel CUDA ottenuti tramite l'utilizzo di *NVIDIA Nsight Compute*, un tool che permette di profilare kernel computazionali su GPU.

Oltre alla profilazione, è stato calcolato lo speedup rispetto alla versione OpenMP:

$$\text{Speedup} = \frac{t_{\text{OpenMP}}}{t_{\text{CUDA}}} = \frac{41.204}{1.5683} \approx 26.27$$

Questo conferma l'efficienza superiore della GPU nel gestire carichi massicci, anche se il risultato è influenzato negativamente dai colli di bottiglia della memoria globale.

Dato che la complessità computazionale dell'operatore skyline è asintoticamente $O(N^2)$, si può stimare un throughput globale in termini di operazioni al secondo come segue:

$$\text{Throughput} = \frac{\text{time complexity}}{\text{wall-clock time}} = \frac{100000^2}{17.32} \approx 577.37 \times 10^6 \text{ operazioni/sec}$$

Questa stima conferma un'ottima performance in termini assoluti, anche se non massimale.

4 Conclusioni

La versione OpenMP, sebbene relativamente semplice da implementare, soffre su problemi piccoli o poco computazionalmente intensi. Quando il carico cresce, invece, riesce a scalare in modo decente fino al limite dei core disponibili. La scelta di uno scheduling guided si è dimostrata efficace nel bilanciare il carico riducendo l'overhead.

La versione CUDA ha mostrato performance migliori, grazie al parallelismo massivo e alla gestione fine delle risorse di memoria (shared vs global). Tuttavia, non è priva

di limiti: la dimensione della shared memory è un vincolo rigido, e quando viene superato si entra in uno scenario meno ottimale.

Nonostante ció si sarebbero potute migliorare le implementazioni utilizzando un approccio divide and conquer, dove viene suddiviso il problema in problemi piú piccoli, ed ogni unitá computazionale potrebbe calcolarne lo skyline, per poi unire i risultati facendo un ultimo operatore skyline sui minori punti rimasti.