Progetto di High Performance Computing 2024/2025

Andrea Baldazzi, matr 0001071149

16/02/2025

1. Introduzione

Il documento descrive il lavoro svolto per la parallelizzazione dell'operatore Skyline, un problema che prevede l'individuazione di uno specifico sottoinsieme di punti nello spazio partendo da una base dati fornita in input.

L'obiettivo del progetto è di individuare e descrivere diverse soluzioni al problema ed analizzare le prestazioni, l'efficienza e la scalabilità rispetto alla versione seriale dell'algoritmo. A tal fine, sono state realizzate due versioni del programma: una basata su OpenMP, per sfruttare il parallelismo a memoria condivisa, e una seconda implementazione basata su CUDA, per sfruttare il parallelismo massivo su una GPU.

1.1 Hardware

Per effettuare i test descritti in seguito è stato utilizzato l'hardware del server isi-raptor03.csr.unibo.it, messo a disposizione dal docente, con le seguenti specifiche:

- **CPU:** Intel Xeon Processor E5-2603 v4
- **GPU:** Nvidia GeForce GTX 1070 8gb

1.2 Testing

I tempi di esecuzione sono stati calcolati come la media di tre misurazioni, al fine di ridurre l'errore. Tutti i risultati dei test e i valori ricavati sono presenti nel file "results.txt".

2. Versione OpenMP

La funzione Skyline da parallelizzare prevede due cicli for annidati per il calcolo dei punti all'interno dello skyline. La soluzione adottata consiste nel parallelizzare il ciclo interno utilizzando la clausola `#pragma omp for reduction(+ : r)`, lasciando quindi l'assegnamento delle iterazioni ai vari thread ad OpenMP e applicando una riduzione sulla variabile r, che tiene conto del numero di punti all'interno dello skyline. È presente all'inizio anche un ciclo for di inizializzazione dell'array s contenente le variabili binarie, parallelizzabile con una semplice `#pragma omp for`.

Inoltre, al fine di riutilizzare più possibile il pool di thread tutto il codice è racchiuso all'interno di un'unica clausola `#pragma omp parallel`. Il partizionamento è statico e a grana grossa in quanto l'insieme dei punti può essere considerato uniforme e la dimensione dell'input può essere scelta indipendentemente dal numero di thread utilizzati.

Si precisa che, inizialmente, era stata considerata una versione del programma che prevedeva la parallelizzazione del ciclo esterno, scartata poi perché presentava prestazioni peggiori di quella sopra descritta. La soluzione consisteva nell'assegnare ad ogni thread un array s locale su cui operare, ed effettuare alla fine dell'algoritmo un'unione dei vari array locali.

2.1 Valutazione delle prestazioni OpenMP

2.1.1 Speedup

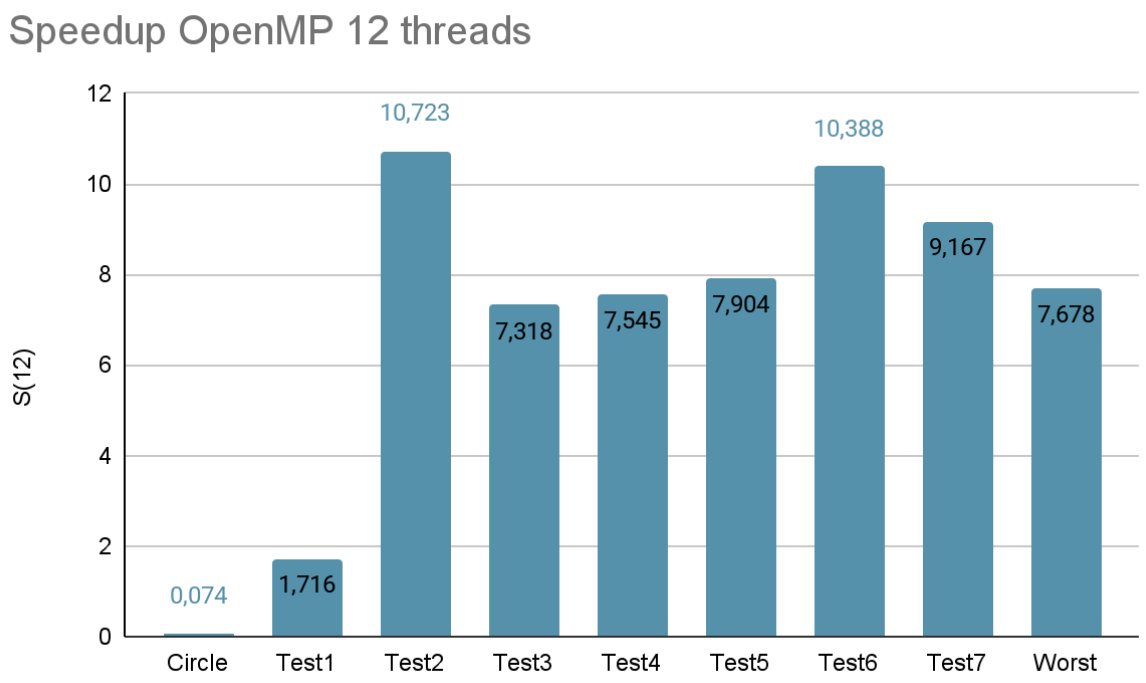

fig. 2.1 Grafico dello speedup con 12 thread, OpenMP

La figura 2.1 mostra lo speedup ottenuto dal programma parallelo, calcolato come S(12) = T(1) ÷ T(12), rispetto alla versione seriale; le misurazioni di quest'ultima, per correttezza, sono state prese utilizzando la versione OpenMP con un solo thread.

Lo speedup medio ottenuto tra tutti i test è di 6.95, con un massimo di 10.72 nel test 2 e un minimo di 0.07 nel test circle. Il motivo di quest'ultimo è da ricercarsi nella scarsità di punti considerati nel test: l'inizializzazione del pool di thread e l'assegnamento ad essi delle varie interazioni comporta infatti un overhead, sottolineando come la parallelizzazione è conveniente quando il volume dei dati è molto elevato.

2.1.2 Strong scaling efficiency

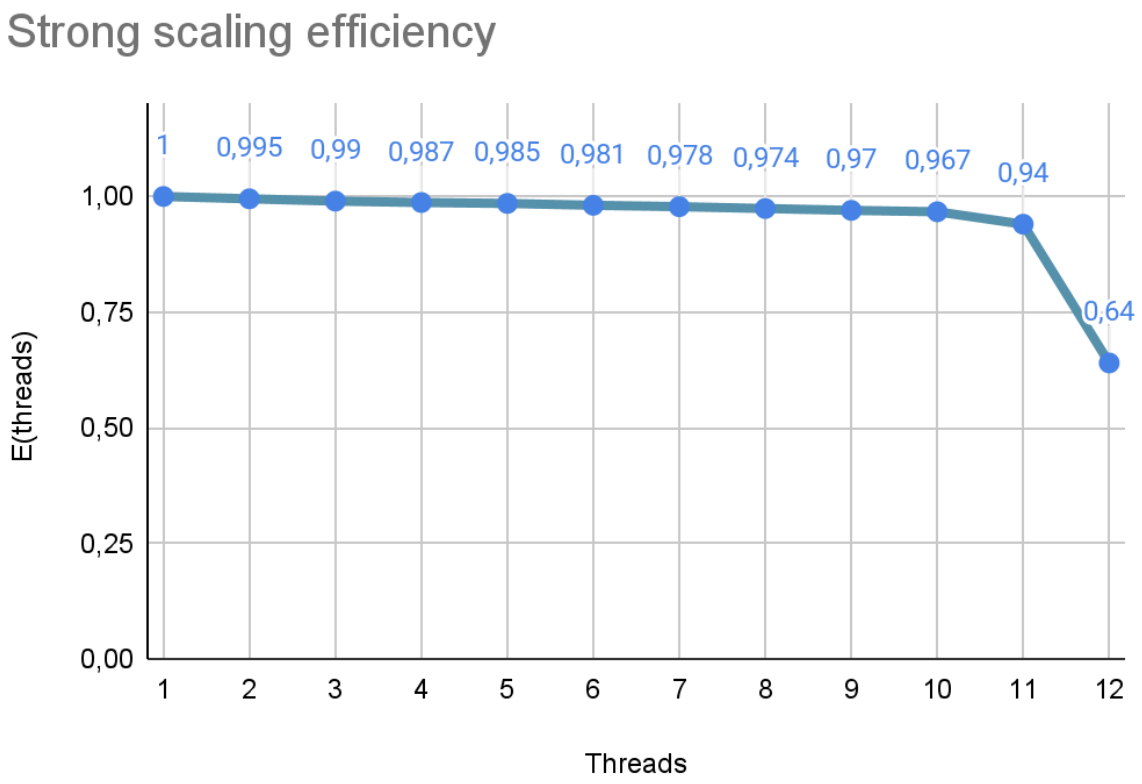

fig. 2.2 Grafico della Strong Scaling Efficiency al variare del numero di thread, OpenMP test “worst”

La figura 2.2 mostra come cambia la Strong Scaling Efficiency al variare del numero di thread. L’efficienza è stata calcolata come E(p) = T(1) ÷ (p • T(p)) utilizzando il test “worst”.

Si può notare che all’aumentare del numero di thread utilizzati l’efficienza decresce leggermente a causa dell’overhead di OpenMP. L'utilizzo di 12 thread, mappati sui core della CPU, evidenzia un calo nell'efficienza dovuto al sovraccarico delle risorse hardware, condivise con il sistema operativo e altri processi.

2.1.3 Weak scaling efficiency

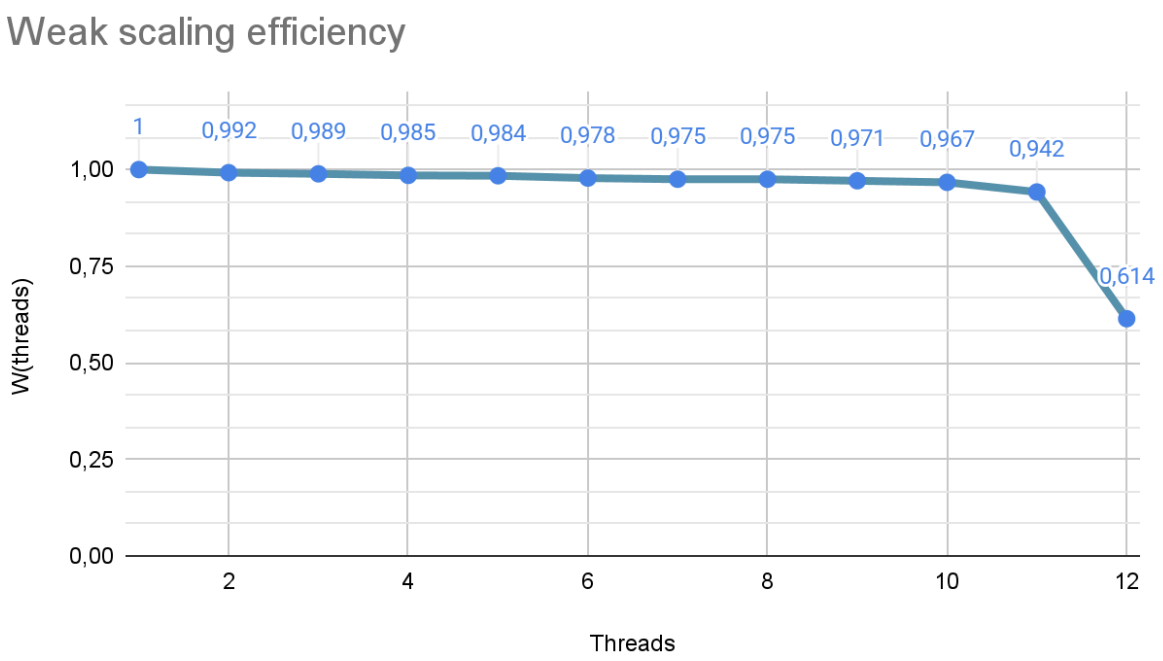

fig. 2.3 Grafico della Weak Scaling Efficiency al variare del numero di thread, OpenMP test “worst”

La figura 2.3 mostra come cambia la Weak Scaling Efficiency al variare del numero di thread. L’efficienza è stata calcolata come W(p) = T(1) ÷ T(p), mantenendo lo stesso carico di lavoro per ogni thread. I problemi sono stati generati partendo dal programma generatore “inputgen”; in particolare, considerando la complessità asintotica $O(D \cdot N^2)$ dell’algoritmo Skyline, ogni unità di esecuzione esegue circa 25000^2 controlli di dominanza tra due punti, scelti mantenendo invariato il numero di dimensioni (10 nel test effettuato).

Si può notare che il comportamento è molto simile a quello della Strong Scaling Efficiency, in cui si misura un’efficienza molto vicino a 1 tranne nel caso in cui vengono utilizzati 12 thread.

3. Versione CUDA

Per la versione CUDA sono state individuate due soluzioni di parallelizzazione. In entrambe viene allocata memoria sulla GPU per ospitare l’array s, il valore di r e l’array P dei punti.

La prima soluzione consiste nell’esecuzione di un unico kernel monodimensionale, con blocchi di dimensione massima pari a 1024 thread. Ad ogni thread viene assegnato un punto e si controlla se esso viene dominato da qualche altro punto, ponendo la sua variabile binaria pari a 0 ed effettuando una sottrazione atomica (`atomicSub`) su r, inizializzata al numero totale di punti. Vengono evitate race condition dal momento che ogni thread accede solo alla variabile binaria dell’array s con indice pari al proprio id (calcolato come `idx = threadIdx.x + blockIdx.x * blockDim.x`). Il kernel descritto è “`skyline_kernel_1d`”.

La seconda soluzione, più veloce della prima, è quella utilizzata per i test. Essa richiama in sequenza tre kernel:

- `init_kernel`: si utilizzano blocchi monodimensionali per l'inizializzazione dell'array s essendo un compito embarrassingly parallel.
- `skyline_kernel`: questo kernel sfrutta blocchi a due dimensioni 32 x 32 assegnando ad ogni thread una coppia di punti. Esso controlla se il primo domina il secondo impostando la sua variabile binaria in s a 0. Dato che a questa zona di memoria potrebbero accedere più thread allo stesso tempo si utilizza una `atomicExch` per evitare race condition.
- `sum_kernel`: quest'ultimo kernel è responsabile di sommare tutti gli elementi dell'array s per ricavare il numero r di punti appartenenti allo skyline. Ogni blocco, con dimensione 1024 x 1, utilizza tutti i thread per copiare la sua parte di array s da sommare nella memoria shared, con l'obiettivo di eliminare accessi continui alla memoria globale. Successivamente, la metà dei thread disponibili esegue il calcolo della somma locale, seguita da un'ulteriore riduzione in cui un quarto dei thread esegue l'operazione, proseguendo iterativamente fino ad arrivare ad un unico thread. A questo punto il thread 0 di ogni blocco esegue una somma atomica (`atomicAdd`) sulla variabile r, inizializzata a 0, completando così il processo di somma.

Entrambe le soluzioni accettano in input un numero arbitrario di punti.

3.1 Valutazione delle prestazioni CUDA

3.1.1 Throughput

Dal momento che non si conosce il numero di CUDA core utilizzati da un programma CUDA, non si può parlare di speedup ma solo di throughput, ovvero il numero di operazioni eseguite al secondo.

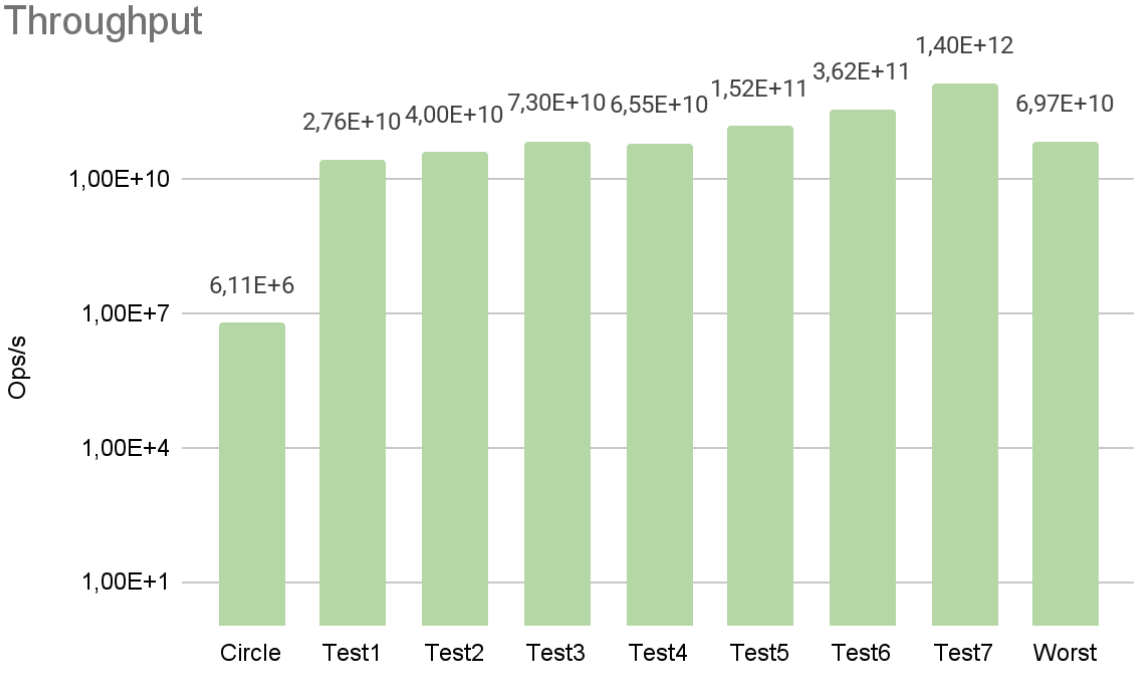

fig. 3.1 Grafico del throughput, CUDA (scala logaritmica)

La figura 3.1 mostra il throughput ottenuto dal programma CUDA. Il valore è stato calcolato come Thr = operations ÷ wall-clock time, in cui il numero di operazioni è dato dalla complessità asintotica $O(D \cdot N^2)$.

Si può notare che il programma riesce a sfruttare molto bene la dimensione crescente dell'input grazie alla presenza di un numero molto elevato di CUDA core della GPU. Nel test *circle,* invece, si osserva un throughput molto ridotto, con tempi di esecuzione addirittura superiori alla versione seriale, imputabile all'overhead d'inizializzazione della GPU.

4. Conclusioni

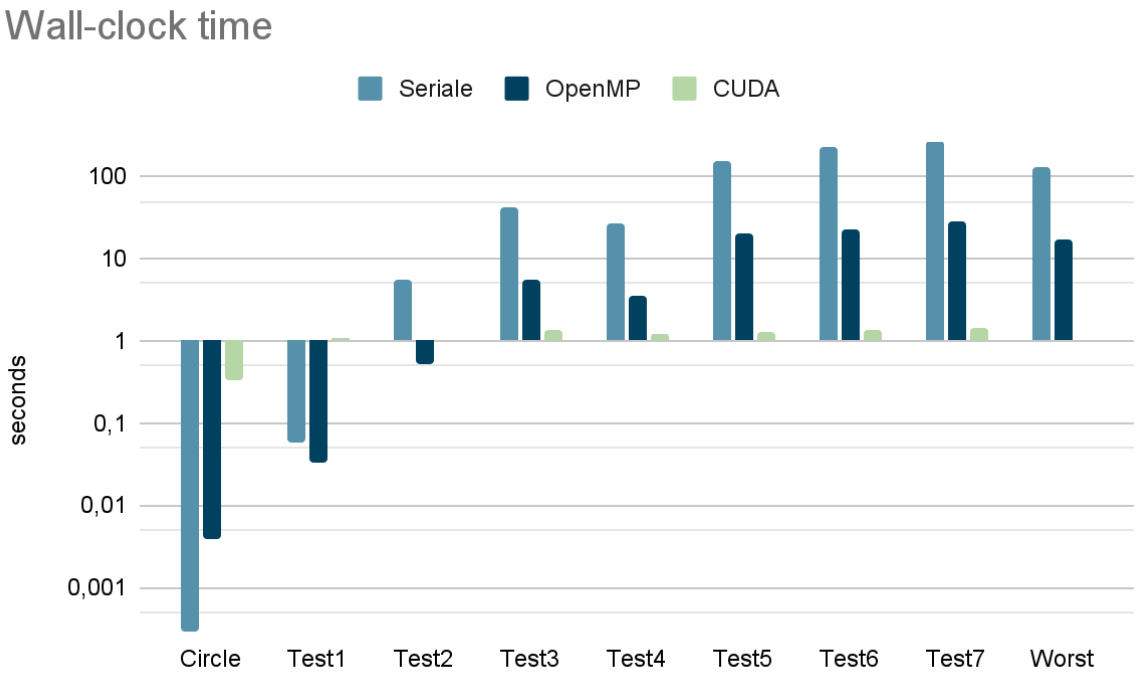

fig. 4.1 Grafico dei tempi di esecuzione (scala logaritmica)

In conclusione, come si può notare dalla figura 4.1, si può dire che la parallelizzazione di un programma è conveniente qualora i dati siano molti, in modo da rendere trascurabile l'overhead introdotto. Questo aspetto diventa ancora più rilevante nei programmi basati su CUDA, dove l'overhead è maggiore a causa della necessità di trasferire i dati tra diverse memorie.