Progetto di High Performance Computing 2024-2025

Andrea Monaco, matr. 0001070845 07/02/2025

Introduzione

La seguente relazione descrive il modo in cui è stato parallelizzato l'algoritmo per il calcolo della skyline evidenziando le ottimizzazioni implementate e l'analisi delle prestazioni ottenute. L'obiettivo è quello di mostrare le differenze implementative e prestazionali tra la versione Seriale, OpenMP e CUDA.

Versione OpenMP

La funzione da parallelizzare è `skyline`, che si divide su 2 cicli.

- Il primo è molto veloce e inizializza a `1` ogni valore dell'array `s`. Il ciclo è *embarrassingly parallel* ed è facilmente parallelizzabile utilizzando la direttiva: `#pragma omp parallel for`.
- Il secondo ciclo contiene due cicli annidati. Il ciclo esterno dipende dalle operazioni del ciclo interno, che azzera i valori di `s[i]`. Sebbene quest'ultimo sia embarrassingly parallel, presenta un problema di concorrenza sulla variabile `r`, poiché più thread potrebbero decrementarla simultaneamente.

Una possibile soluzione consiste nel creare una variabile temporanea (locale a ciascun thread) che, al termine del ciclo interno, dovrà essere sottratta al valore di `r` nel ciclo esterno, dove non sono presenti problemi di concorrenza.
Dopo queste modifiche i cicli della funzione `skyline` si presentano in questo modo:

```
#pragma omp parallel for
    for (int i = 0; i < N; i++) {
        s[i] = 1;
    }

    for (int i = 0; i < N; i++) {
        if (s[i]) {
            int a = 0;
#pragma omp parallel for reduction(+ : a)
            for (int j = 0; j < N; j++) {
                if (s[j] && dominates(&(P[i * D]), &(P[j * D]),
D)){
                    s[j] = 0;
                    a++;
                }
            }
            r -= a;
        }
    }
    return r;
```

Testando questa soluzione sull'*i5 11600K* otteniamo uno speedup di ~4 volte su 12 threads. Su questo risultato incide il ciclo esterno eseguito in modo seriale.
A riguardo, si era detto che il ciclo esterno è dipendente da quello interno. I problemi derivano dalle letture su `s`:

- La prima lettura è nell'`if` sul ciclo esterno, questa serve solo per migliorare le prestazioni della versione seriale, quindi la lettura di uno sbagliato valore di `s[i]` in questo caso non cambia il risultato.
- Il secondo problema è invece dovuto alla lettura di `s[j]` nel ciclo interno, infatti è possibile che nel mentre si valutano le condizioni dell'`if` un altro thread assegni `0` a `s[j]`. Questo in sé non rappresenta un problema, tuttavia entrare nell'`if` del ciclo interno comporta anche un decremento `r`, quindi senza cambiare questo dettaglio non è possibile parallelizzare ulteriormente il codice.

Dato che ogni elemento di `s` assume il valore `1` o `0` e `r` rappresenta il conteggio dei valori uguali a `1` (ovvero, il numero di punti nella skyline), è possibile calcolare `r` in un ciclo separato, una volta definito `s`.
Considerando questo, con qualche ottimizzazione la versione definitiva della funzione `skyline` si presenta nel modo seguente:

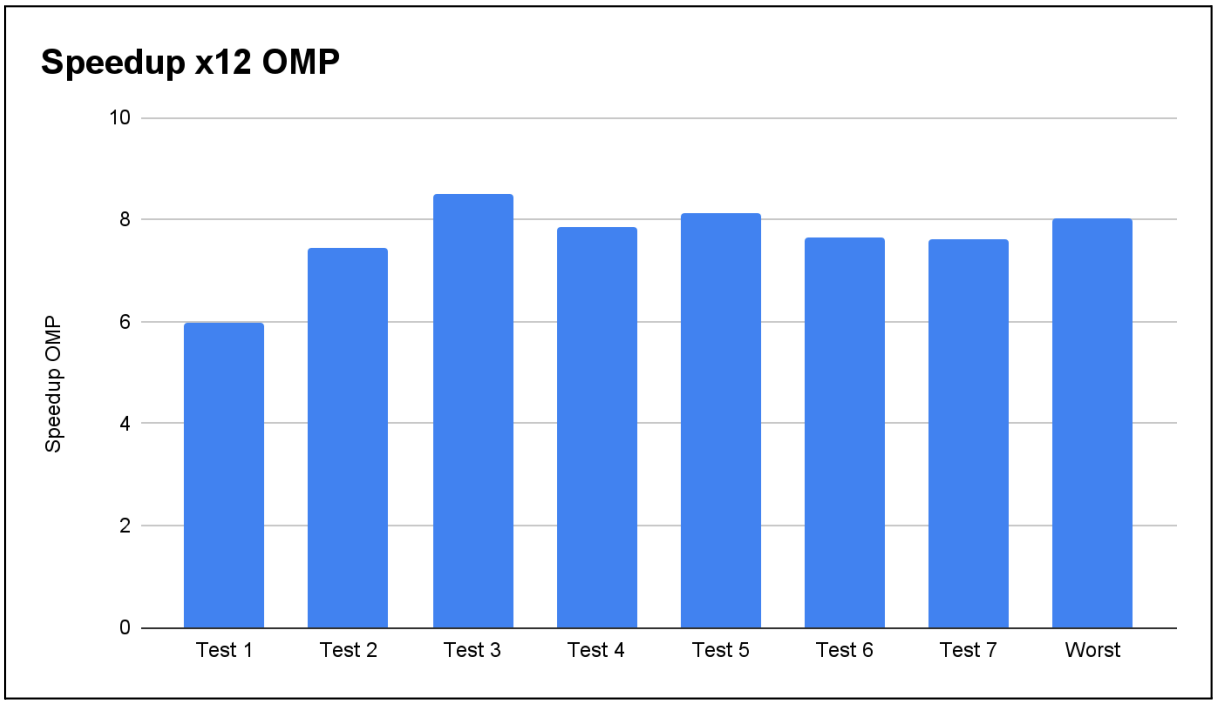
```
int skyline(const points_t *points, int *s) {
    const int D = points->D;
    const int N = points->N;
    const float *P = points->P;
    int r = 0;

#pragma omp parallel
    {
#pragma omp for
        for (int i = 0; i < N; i++) {
            s[i] = 1;
        }
#pragma omp barrier

#pragma omp for
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N && s[i]; j++) {
                if (s[j] && dominates(&(P[i * D]), &(P[j * D]),
D)){
#pragma omp atomic write
                    s[j] = 0;
                }
            }
        }
#pragma omp barrier

#pragma omp for reduction(+ : r)
        for (int i = 0; i < N; i++) {
            r += s[i];
        }
    }
    return r;
}
```

Adesso le condizioni per interrompere il ciclo interno sono diverse perché anche altri thread possono annullare il valore degli elementi di `s`, dunque in alcuni casi è possibile uscire anticipatamente dal ciclo interno. L'istruzione `s[j] = 0;` può essere eseguita più volte, ma non altera il valore finale di `s` o di `r`. Lo schedule utilizzato è statico in quanto il carico di lavoro è ben bilanciato. Si possono rimuovere i controlli su `s[i]` e usare `collapse(2)`, cambiando i tipi di `i` e `j`, ma questo porta a un peggioramento delle prestazioni.

Valutazione delle prestazioni OMP

Le prestazioni sono state valutate in Ubuntu LTS 24.04 su un i5 11600k con 6 core e 12 threads a 4.9 Ghz (sia single che multicore), tutti i test sono quelli contenuti nel dataset.

Speedup:
Lo speedup è calcolato su tutti e 12 thread usando l'hyper-threading, si consideri quindi che le prestazioni per thread sono inferiori.



Come si vede dal grafico lo speedup è ~8, ad esempio nel test 7 è 7,6.

Strong scaling efficiency:
La strong scaling efficiency è stata calcolata misurando il tempo di esecuzione del programma OMP nel test "worst":

- con un solo thread risulta essere `42.09`
- con 6 thread (quindi uno per core) risulta essere `7.46`.

Di conseguenza si ottiene: $Speedup = t(1) \div t(N) = 5,64$. Un ottimo risultato, vicino al numero effettivo di core.

Weak scaling efficiency:
La weak scaling efficiency è stata calcolata sempre rispetto a 6 core nel test "worst".

Il numero di istruzioni eseguito nella funzione `skyline` è di ordine $o(N^2)$; pertanto, visto che il caso "worst" prevede `N = 100000` è stato generato un nuovo input di test worst con $N = 100000 \times \sqrt{6} \simeq 244949$ in questo modo il numero di operazioni aumenta di 6 volte. In single-thread con `N = 100000` il tempo di esecuzione è, dalla misurazione precedente, `42.09`, invece con 6 thread e `N = 244949` il tempo di esecuzione è `46,31`. Dunque $Efficiency = t(1) \div t(N) = 0,91$.

Versione CUDA

Come nella versione omp il primo ciclo che inizializza a 1 tutti i valori di `s` è facilmente parallelizzabile in questo caso con un una funzione `__global__`.

```
__global__ void init_s(int *s, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        s[i] = 1;
    }
}
```

Invece nel secondo ciclo bisogna rimuovere il decremento su `r` anche nella versione cuda e parallelizzando il ciclo esterno si ha

```
__global__ void skyline_kernel(float *P, int *s, int N, int D) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= N) return;

    for (int j = 0; j < N && s[i]; j++) {
        if (dominates(&(P[i * D]), &(P[j * D]), D)) {
            s[j] = 0;
        }
    }
}
```

La funzione `dominates` è rimasta invariata ma le è stato aggiunto il qualificatore `__device__`, invece `r` viene calcolato sulla cpu con una riduzione. Sul computer `isi-raptor03`, con una 1070 il tempo richiesto per eseguire il test 7 adesso è di circa 4 secondi, quindi abbiamo un ottimo miglioramento anche rispetto alla versione OMP che impiega ~29 secondi.
Si può parallelizzare anche il ciclo interno per migliorare ulteriormente le prestazioni:

```
__global__ void skyline_kernel_2(float *P, int *s, int N, int D) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    if(i >= N || j >= N) return;

    if (dominates(&(P[i * D]), &(P[j * D]), D)) {
        atomicExch(&s[j], 0);
    }
```

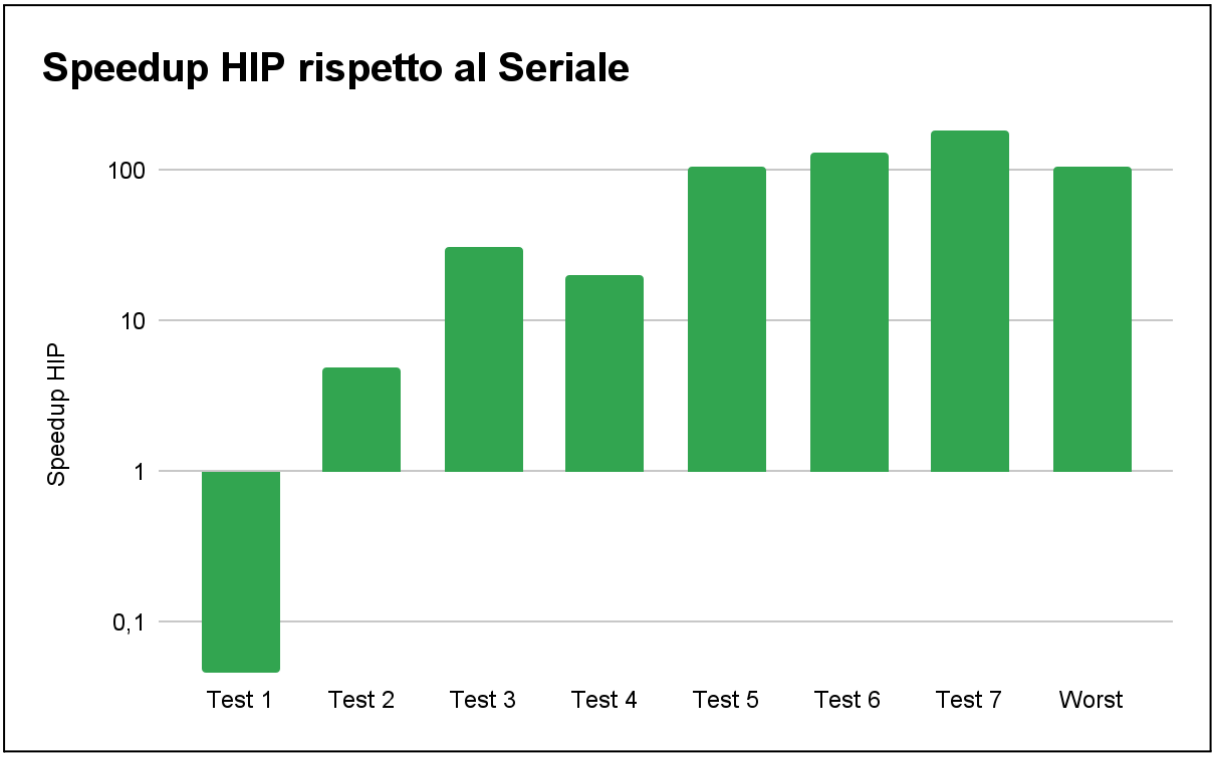
```
}
```

Nell'usare CUDA sono state cambiate anche delle parti del programma svolte dalla CPU, infatti nella versione definitiva il codice eseguito sulla CPU chiama la funzione `skyline`. In questa versione è stata rimossa la verifica su `s[i]` (poiché, su GPU, tale controllo penalizzerebbe le prestazioni) e l'assegnazione `s[j] = 0` è stata sostituita da un'operazione atomica, dato che può essere eseguita più volte contemporaneamente da thread differenti.
La cpu chiama 2 funzioni `__global__`, una monodimensionale che inizializza `s` e una bidimensionale che assegna i corretti valori `1` e `0` all'array, infine calcola il valore di `r`.
Questo è un ottimo miglioramento, infatti `isi-raptor03` ora svolge il test 7 in ~1,5 secondi.

Valutazioni delle prestazioni CUDA/HIP

Le prestazioni sono state valutate in Ubuntu LTS 24.04 su una radeon AMD 6600xt con l'utilizzo driver ROCm e del comando `hipify` per convertire i programmi cuda a programmi hip.

Speedup:
In questo caso le prestazioni sono limitate dal dover allocare la memoria sulla gpu e dal dover trasferire gli array `s` e `P` tra host e device. Infatti nel primo test la gpu ha una resa peggiore persino del seriale, visto che N è relativamente piccolo, tuttavia con l'aumentare del numero delle operazioni da svolgere la GPU raggiunge uno $Speedup \simeq 185$ nel test 7.
Il grafico è in scala logaritmica per evidenziare meglio la differenza tra single core e GPU.

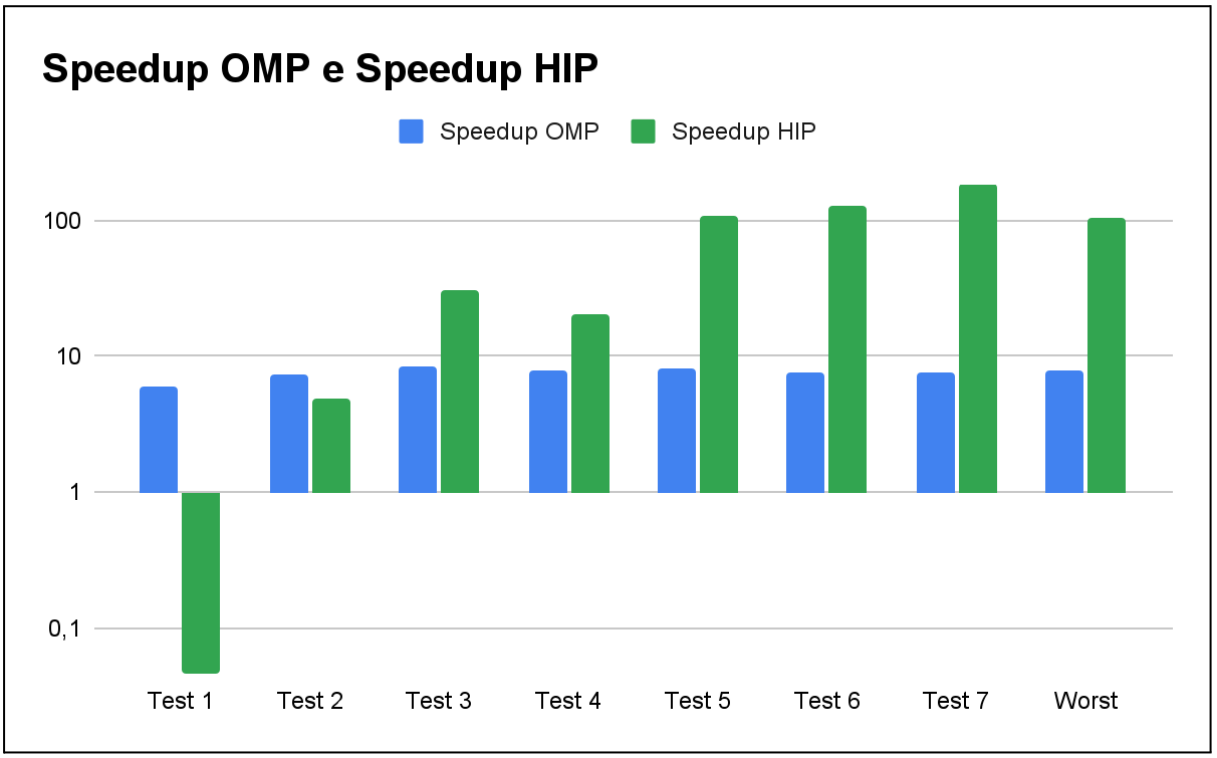


Conclusioni

Viene mostrata una tabella con le misure in secondi ottenute eseguendo i vari test, sempre su Ubuntu 24.04 LTS, i5 11600k e AMD radeon 6600xt.

Test	Seriale	OMP	CUDA/HIP	Speedup OMP	Speedup HIP
Test 1	0,018	0,003	0,400	5,988	0,045
Test 2	1,956	0,262	0,404	7,455	4,848
Test 3	15,072	1,775	0,488	8,490	30,895
Test 4	9,410	1,201	0,463	7,839	20,321
Test 5	53,444	6,584	0,496	8,117	107,752
Test 6	74,181	9,694	0,569	7,653	130,276
Test 7	100,988	13,271	0,546	7,610	185,053
Worst	42,653	5,323	0,403	8,013	105,908

E questo è un grafo logaritmico che mostra lo speedup rispetto alla versione seriale.



Come si evince da quest'ultimo grafico, il codice funziona meglio su CUDA/HIP con l'aumentare del numero di operazioni.