Progetto di High Performance Computing

Luca Fabri, Matricola: 0000892878

27 luglio 2021

1 Introduzione

L'obiettivo di questo progetto era la realizzazione di due implementazioni dell'algoritmo per il calcolo dello Skyline: una in OpenMP e l'altra utilizzando MPI o CUDA (a scelta). Per la versione a scelta ho realizzato un'implementazione in CUDA.

Prima di descrivere le due versioni volevo far notare che in entrambi i sorgenti sono presenti delle funzioni in comune: `read_points()`, `dominance()` e `print_skyline()`. La prima legge i punti da un file descriptor e li memorizza in una matrice / array, la seconda dati due punti determina se il primo domina il secondo, mentre la terza stampa, utilizzando sempre un file descriptor, l'insieme Skyline.

Inoltre, per memorizzazione dei punti ho utilizzato il tipo double perciò la dimensione dell'input è pari a `N * D * sizeof(double)`.

2 Versione OpenMP

2.1 Parallelizzazione e funzionamento

Il programma in OpenMP è frutto di molteplici implementazioni intermedie che ho realizzato per individuare le performance migliori. Nella versione finale proposta, ogni thread:

1. Calcola `local_start` e `local_end` che rappresentano gli indici di inizio e fine dei punti su cui deve eseguire il confronto di dominanza;
2. Inizializza una porzione di un array condiviso tra i threads, in particolare l'insieme di elementi dell'array compresi tra `local_start` e `local_end`. Lo scopo di questo array è quello di indicare se un punto è appartenente o meno all'insieme Skyline;
3. Per ogni punto compreso tra `local_start` e `local_end` deve determinare se e quali punti dell'insieme complessivo sono dominati da questi;
4. Indicare nell'array condiviso del punto 2) quali punti non appartengono allo Skyline (calcolati al punto 3);
5. Dopo che tutti i thread sono allineati, contare i punti tra `local_start` e `local_end` che appartengono allo Skyline.

L'array al punto 2) è condiviso perchè ogni thread deve poter leggere e scrivere al suo interno, perciò, le scritture sono protette da `#pragma omp atomic write`, che permette di evitare eventuali race conditions. L'"allineamento" dei threads, al punto 5) è effettuato tramite la direttiva `#pragma omp barrier`, che blocca il flusso di esecuzione di ogni thread finchè i threads non si sicronizzano. Dopo la terminazione della sezione parallela, per calcolare la cardinalità dell'insieme Skyline, il processo master somma tutte le cardinalità parziali dei threads.

2.2 Altre versioni

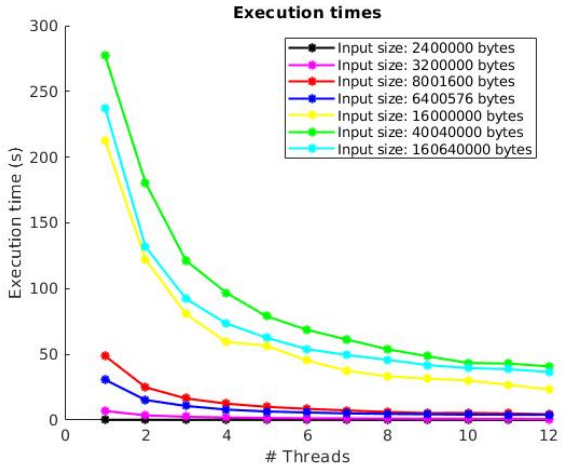
Una versione precedente quella finale evitava di scrivere in stessi elementi dell'array del punto 2. Per far ciò, ogni thread determina solo se punti tra `local_start` e `local_end` appartengono o meno allo Skyline: in questo modo non c'è la necessità di gestire race conditions tramite direttive atomic.

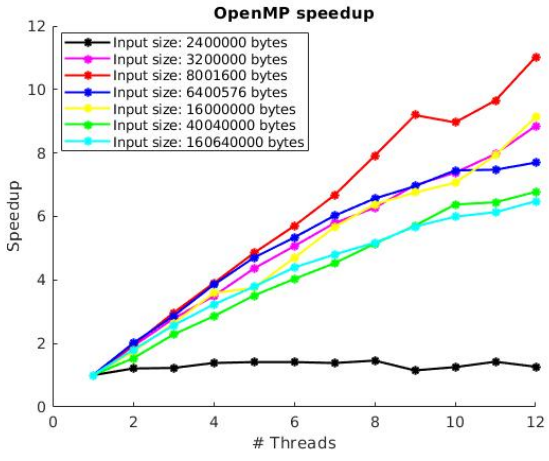
Questa versione risulta tuttavia, per N molto "grandi", computazionalmente meno efficiente rispetto a quella consegnata, poichè ogni thread può al massimo calcolare `local_end - local_start` punti, mentre nella versione finale, dato un punto, ogni thread può determinare se al più N punti non appartengono all'insieme Skyline, facendo risparmiare di conseguenza calcoli agli altri threads del pool.

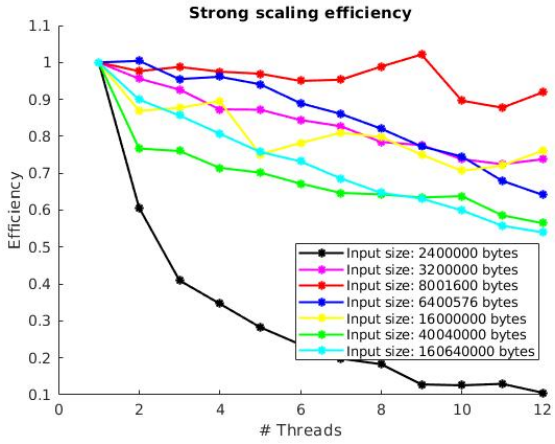
2.3 Performance

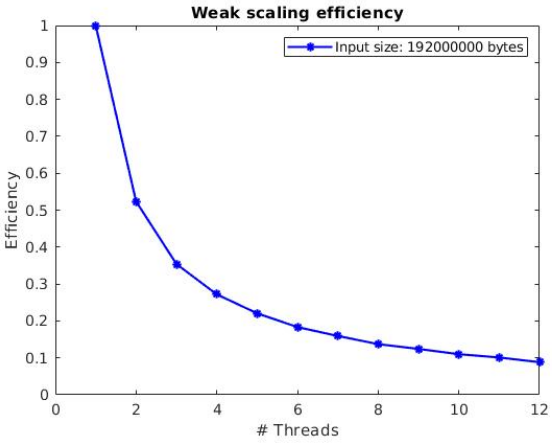
Di seguito vengono mostrati i grafici che ho realizzato con MATLAB per visualizzare le performance del programma. Nella legenda dei primi quattro grafici, anzichè scrivere il numero del test (test1, ..., test7) ho preferito inserire la dimensione dell'input in bytes.

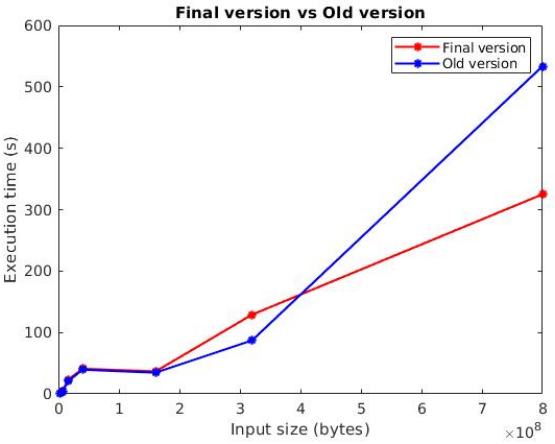
Nell'ultimo grafico viene eseguito un confronto dei tempi di esecuzione della versione finale vs versione vecchia. Anche qui nell'asse delle ascisse ho utilizzato la dimensione dell'input in bytes e ho generato due test aggiuntivi di dimensione maggiore per marcare ancora di più la differenza (test8: N = 200000, D = 200, test9: N = 500000, D = 200).











3 Versione CUDA

3.1 Limiti hardware e scelta dimensione dei thread block

Per l'implementazione in CUDA ho cercato di sfruttare al più possibile le peculiarità hardware che permettono di ottimizzare i tempi di esecuzione. In particolare:

- Ho cercato di evitare la divergenza dei flussi di esecuzione poichè il codice diventa più adatto ad essere eseguito in warps SIMT (Single Instruction, Multiple Threads);
- Ho cercato di coalizzare gli accessi alla memoria globale, facendo accedere ai threads blocchi di memoria consecutivi.

D'altra parte vanno considerati anche i limiti hardware, come il massimo numero di threads per thread block e la dimensione della memoria shared: a causa di questi è difficile variegare di implementazioni e quindi mi è risulato abbastanza facile individuare quella migliore.

Per quanto riguarda la scelta del numero di threads per thread block ho eseguito vari test che mi hanno sottolineato come la dimensione di 32 thread / thread block, sia meno efficiente rispetto a dimensioni multiple di 32 e strettamente maggiori di 32.

Ricercando una motivazione, una delle possibili cause potrebbe essere legata alla CUDA Occupancy.

La **CUDA Occupancy** è il rapporto tra il numero dei warp attivi per SM (Streaming Multiprocessor) e il numero massimo di warp che possono essere attivi in un istante in un SM. Per nascondere le latenze dovute alle dipendenze come ad esempio letture nella memoria globale, ogni warp scheduler dovrebbe avere almeno un warp eleggibile ad ogni ciclo di clock. Se manteniamo tanti warp attivi quanti ne possiamo inserire in un SM, raggiungendo la massima Occupancy, possiamo evitare situazioni nella quale tutti gli warp sono in stallo e nessuna istruzione viene eseguita [1].

Calcolando la CUDA Occupancy [2] ho riscontrato che utilizzando la dimensione di 32 thread / thread block questa è pari al 50% mentre una dimensione di 64 o maggiore è pari al 100%. Questa motivazione è tuttavia difficile da dimostrare ma volevo mostrare una possibile causa della differenza dei tempi di esecuzione. Per la dimensione dei thread blocks ho perciò scelto 64 threads (2 warps).

3.2 Parallelizzazione e funzionamento

I passi che vengono eseguiti per l'esecuzione del programma sono i seguenti:

- Copiamo i punti dall'host alla memoria globale del device;
- Creiamo un array di bool di lunghezza N all'interno della memoria del device, dove la funzione Kernel indicherà i punti appartenenti allo Skyline;
- Creiamo una variabile intera all'interno della memoria del device dove ogni thread incrementerà la cardinalità dell'insieme.

La funzione Kernel che calcola i punti appartenenti allo Skyline è molto simile alla vecchia versione di OpenMP che ho precedentemente descritto:

- Lanciamo tanti threads quanti sono i punti forniti in input;
- Ogni thread determina se un punto appartiene o no allo Skyline;
- Viene incrementato di 0 od 1 la cardinalità dell'insieme Skyline attraverso una scrittura atomica (per evitare race conditions).

Utilizzando quest'approccio ogni thread esegue scritture all'interno delle proprie porzioni di memoria, perciò evitiamo il sovraccarico di scritture che necessitano la sincronizzazione per gestire le race conditions.

Inoltre, durante il confronto di dominanza, ogni thread di un warp accede a locazioni adiacenti in memoria globale e perciò gli accessi risultano coalizzati.

Infine, il flusso di esecuzione di un warp non diverge poichè se un thread determina che il suo punto non fa parte dello Skyline, questo rimane inattivo finchè tutti i threads del warp sono usciti dal ciclo for. Tutti i threads attivi di un warp eseguono perciò la stessa istruzione.

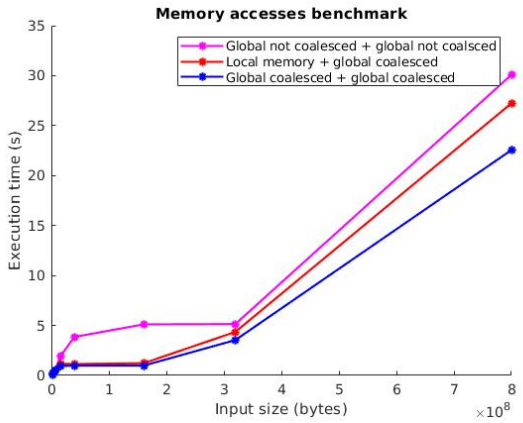
L'`atomicAdd()` per l'incrementazione della cardinalità ho riscontrato che è più efficiente della versione seriale su CPU.

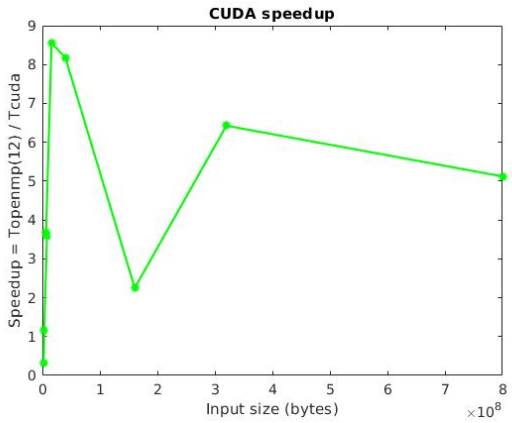
Al termine dell'esecuzione del Kernel:

- Dalla memoria del device a quella dell'host vengono copiati l'array di lunghezza N e il valore della variabile che memorizza la cardinalità dell'insieme;
- Vengono stampati in standard output i punti dello Skyline.

3.3 Performance

Nei grafici seguenti vengono mostrati dei test che ho eseguito per notare la differenza dei tempi di esecuzione.





Nel **primo grafico** vengono mostrate le differenze nell'applicazione di accessi coalizzati o non e differenze nell'utilizzo della memoria globale o locale.

Il compito di ogni thread è quello di calcolare se il numero incaricato appartiene allo Skyline. Per farlo, ad ogni iterazione del ciclo for deve leggere le dimensioni del suo punto per eseguire il confronto di dominanza: possiamo quindi provare a memorizzarlo nella memoria locale e leggere tutti gli altri dalla memoria globale o non memorizzarlo nella memoria locale e leggere quindi tutti i numeri dalla memoria globale.

Utilizzando la memoria locale ed accessi agli altri punti in maniera coalizzata si può notare che il tempo di esecuzione è comunque maggiore rispetto al caso di soli accessi coalizzati nella memoria globale. Questo perchè la memoria locale è locale al thread e perciò è impossibile eseguire accessi coalizzati.

L'utilizzo quindi di soli accessi coalizzati nella memoria globale risulta più veloce.

Per eseguire gli accessi in modo coalizzato la matrice dei punti viene ruotata e quindi la sua dimensione diventa D x N. In questo modo, nel confronto di dominanza, ogni thread di un warp accede a memoria consecutiva poichè in una stessa riga si trova la dimensione i di ciascun punto.

Tutti i tempi di esecuzione della funzione Kernel sono stati misurati tramite l'API di CUDA [3] ma può essere utilizzato anche il Visual Profiler [4].

Nel **secondo grafico** viene mostrato lo speedup dell'implementazione in CUDA in relazione al tempo di esecuzione in OpenMP utilizzando 12 threads. In questo caso viene tenuto conto anche delle porzioni seriali dei programmi.

4 Conclusioni

Nell'individuazione di possibili implementazioni, la vecchia versione di OpenMP mi è risultata utile per la versione in CUDA.

In quest'ultima, per input limitati, l'overhead per la copia dei punti e l'allocazione della memoria nel device supera di gran lunga il tempo di esecuzione della funzione Kernel, perciò, in tali casi, è preferibile disporre di una CPU con core più veloci che di una GPU più performante.

Riferimenti bibliografici

[1] *Achieved Occupancy*. URL: `https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm`.

[2] Mark Harris. *CUDA Pro Tip: Occupancy API Simplifies Launch Configuration*. URL: `https://developer.nvidia.com/blog/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/`.

[3] Mark Harris. *How to Implement Performance Metrics in CUDA C/C++*. URL: `https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/`.

[4] Mark Harris. *CUDA Pro Tip: nvprof is Your Handy Universal GPU Profiler*. URL: `https://developer.nvidia.com/blog/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/`.