Relazione di High Performance Computing

Marco Galeri, matr. 0001019991

February 11, 2024

1. Introduzione

L'obbiettivo del progetto, oggetto della relazione, è il miglioramento delle prestazioni del programma seriale fornito; lo scopo del programma è quello di determinare la minima area occupata da un numero di cerchi definito dall'utente che eviti sovrapposizioni.

L'algoritmo itera tre funzioni che risolvono il problema attraverso un approccio *"force-directed"* spostando i vari cerchi a seguito di diverse somme vettoriali, il numero di iterazioni effettuate viene anch'esso definito dall'utente.

Analizzando le funzioni presenti nel codice possiamo definire il costo computazionale dell'algoritmo, considerando n il numero di cerchi definito dall'utente:

- `move_circles` , `reset_displacements`: $O(n)$

  Queste due funzioni vengono utilizzate solo per modificare la posizione di tutti i cerchi presenti scorrendoli una volta, quindi abbiamo un costo computazionale di $O(n)$.

- `compute_forces`: $O(n^2)$

  Questa funzione rappresenta la parte computazionalmente più complessa del programma che determina la somma vettoriale di tutte le forze che agiscono su ciascun cerchio.

  La funzione confronta ogni coppia di cerchi (c_i, c_j) con $i < j$, da questa considerazione possiamo ottenere il costo computazionale esatto di $\theta(n(n+1)/2)$ che può venir scritto come $O(n^2)$.

2. OpenMP

2.1 Soluzione

Nel programma fornito le funzioni `move_circles` e `reset_displacements` non riportano nessuna problematica non avendo alcuna *loop carried dependencies*, cioè non sono presenti dipendenze tra le diverse iterazioni del ciclo. Per questa motivazione la loro porzione di codice è *embarassingly parallel*, ossia completamente parallelizzabile quindi si può usare la direttiva `#pragma omp parallel for` per ottimizarle.

La funzione `compute_forces` invece possiede alcune dipendenze quindi bisogna utilizzare un approccio differente. Le dipendenze presenti sono legate alla scrittura sullo stesso valore durante diverse iterazioni, quindi con l'uso della direttiva `reduction` si risolve questa problematica attraverso il pattern *reduction*[1] implementato internamente alla direttiva.

Data la struttura del problema questa funzione soffre un grave sbilanciamento del carico di lavoro se affrontato con uno scheduling statico, usato di default nelle direttive OpenMP, perciò per risolvere quest'ultima problematica si può cambiare lo scheduling dei processori con la direttiva `schedule:dynamic`.

2.2 Prestazioni

Lo studio delle prestazioni si interessa di due possibili casi di utilizzo del programma; uno ha un numero N di cerchi presenti elevato (10·000) ed un basso numero IT di iterazioni (20), mentre il secondo in modo contrario avrà un numero minore di cerchi (500) ed uno elevato di iterazioni (8·000).

Come possiamo vedere dal grafico in figura 2.1, nel primo caso, segnato in rosso, si ha un andamento ideale rimanendo molto vicini alla retta dello speedup lineare[2] con un accenno di curvatura come descritto nella legge di Amdahl[3]. Il secondo caso invece, segnato in blu, presenta un evidente distaccamento dallo speedup ideale al l'aumentare dei processori utilizzati, questo comportamento è probabilmente dovuto alla impossibilità di parallelizzare la ripetizione dell'intero processo perdendo tempo in ripetute sincronizzazioni dei processi.

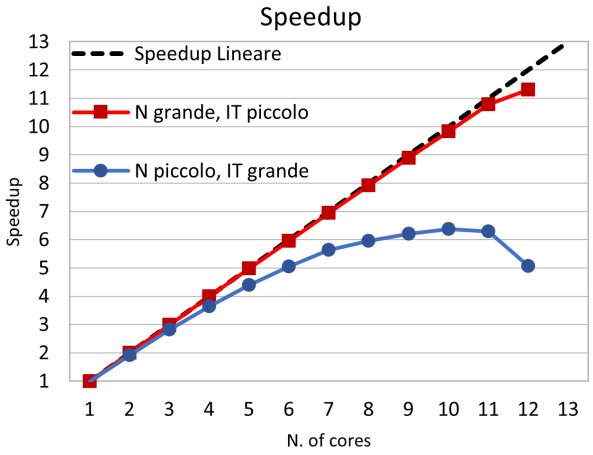
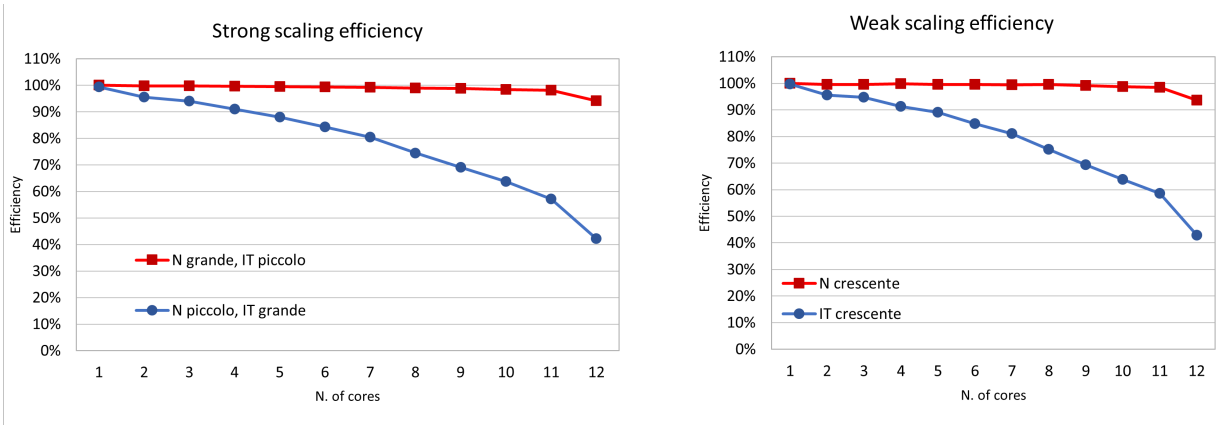


Figure 2.1: Grafico rappresentante lo speedup all'aumentare dei core

[1] pattern di programmazione parallela per l'uso di operatori binari associativi in una lista di elementi
[2] retta che indica le massime prestazioni teoriche al variare del numero di processori
[3] legge per il calcolo del miglioramento massimo atteso di un programma parzialmente parallelizzato



(a) (b)

Figure 2.3: Grafici rappresentanti rispettivamente la Strong scaling efficency e la Weak scaling efficency del programma

Un altro metodo per visualizzare questo comportamento è lo studio della *Strong scaling efficency*, mostrato in figura 2.3, che mostra l'efficenza dell'uso dei processori mantenendo la grandezza del problema costante.

Infine il grafico *b* mostra la *Weak scaling efficency*, ossia la misura delle prestazioni del programma mantenendo costante il numero di operazioni eseguito da ogni singolo processore. Come si vede dal grafico i risultati non presentano grandi differenze rispetto a quelli ottenuti dallo studio della *Strong scaling efficency*.

3. CUDA

3.1 Soluzione

L'approccio di ottimizzazione con CUDA si incentra sull'utilizzo dei CUDA-threads nel metodo più efficente possibile.

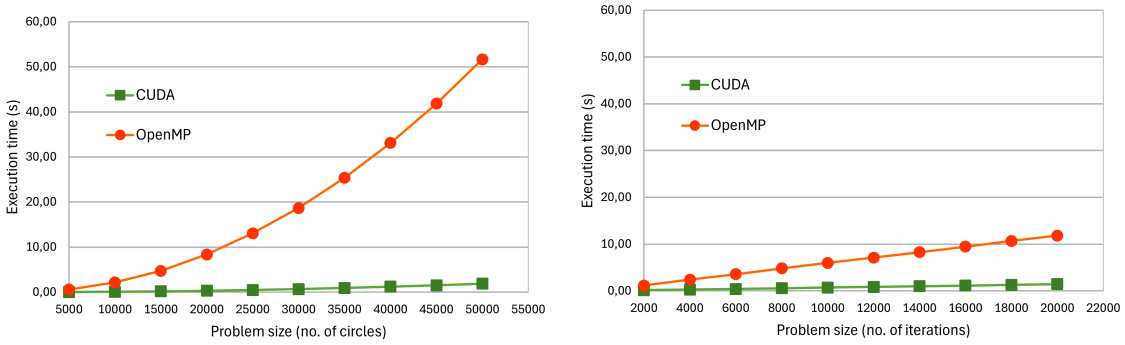
Le funzioni `move_circles` e `reset_displacements` vengono ottimizzate semplicemente eseguendo ogni operazione attraverso thread differenti in modo da appiattire effettivamente il costo di queste funzioni ad una singola operazione. Questa considerazione è valida senza tenere conto delle operazioni di comunicazione tra la *host memory* [1] e la *device memory* [2].

La funzione `compute_forces` invece viene ottimizzata sfruttando la multi-dimensionalità delle griglie presenti nelle GPU. In particolare per eseguire il confronto delle coppie di cerchi viene mappata una matrice bidimensionale di thread rappresentante tutte le coppie di cerchi (c_i, c_j) possibili e poi vengono usati solo i thread che rispettano la condizione $i < j$ come definito dal programma. Inoltre le problematiche derivanti dalle dipendenze presenti nella funzione `compute_forces` possono venir risolte con l'uso delle funzioni atomiche, in particolare `atomicAdd()`, messe a servizio dal compilatore dedicato alla compilazione di file CUDA.

[1] la memoria presente in una CPU
[2] la memoria presente in una GPU

3.2 Prestazioni

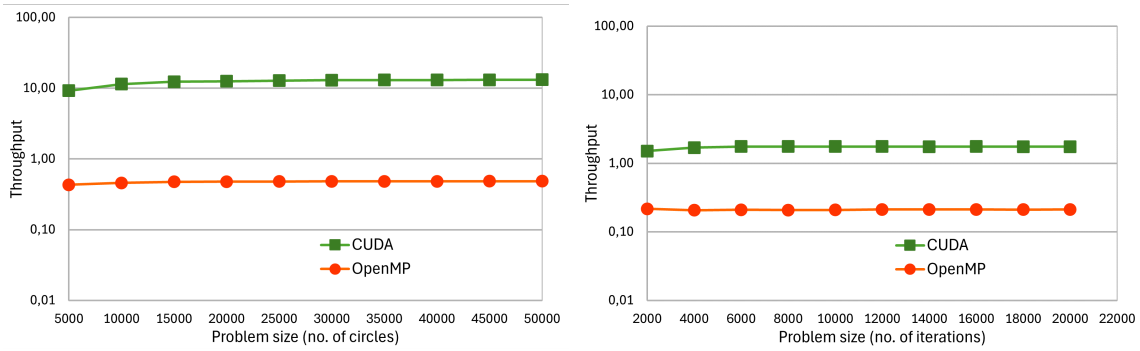
Lo studio delle prestazioni si interessa sempre dei due casi citati precedentemente nel capitolo dedicato alla versione OpenMP in cui variano le dimensioni di input del programma.



(a) (b)

Figure 3.2: Grafici rappresentanti i tempi di esecuzione di CUDA e di OpenMP

Come mostrato nei grafici in figura 3.2 si nota subito la netta differenza tra i tempi di esecuzione della versione OpenMP in confronto alla versione in CUDA, questo divario è prodotto dalla differente struttura hardware delle GPU rispetto alle CPU. Infatti le GPU dispongono di un numero di thread migliaglia di volte superiore a quello delle CPU, questo rende i programmi CUDA ottimi per problemi di grandi dimensioni. Si può notare inoltre una differenza tra il grafico a ed il grafico b, dovuta a come gli input inseriti agiscono sul numero di operazioni da svolgere; il numero di cerchi N aumenta quadraticamente il numero di operazioni da svolgere, mentre il numero di iterazioni IT incrementa linearmente.



(a) (b)

Figure 3.4: Grafici in scala logaritmica rappresentanti il *throughput* in *Gops/sec* (miliardi di operazioni al secondo)

Dal momento che non si ha la possibilità di conoscere il numero di thread utilizzati da un programma CUDA, una metrica importante da prendere in considerazione è il *throughput* ossia la quantità di operazioni che un programma riesce a elaborare al secondo; questo valore riportato nei grafici in figura 3.4 rimarca l'efficacia di CUDA con problemi di grandi dimensioni. Inoltre si vede, comparando i due grafici, una disparità sostanziale nel throughput di entrambe le versioni. Questo comportamento è dovuto alla struttura del programma,

poiché ogni iterazione presenta un discreto *overhead*[3], perciò all'aumentare delle iterazioni aumenterà anche il tempo di overhead del programma, specialmente se la grandezza di esso, come in questo caso, è di piccole dimensioni, diminuendo così il *throughput* ottenuto.

4. Conclusioni

Le ottimizzazioni proposte al programma fornito di entrambe le versioni hanno ottenuto risultati notevoli raggiungendo le prestazioni attese in partenza.

In conclusione si può costatare che l'uso di CUDA sia il più performante tra i due metodi proposti ottenendo uno speedup fino ad oltre 25 volte maggiore rispetto a quello ottenuto da OpenMP.

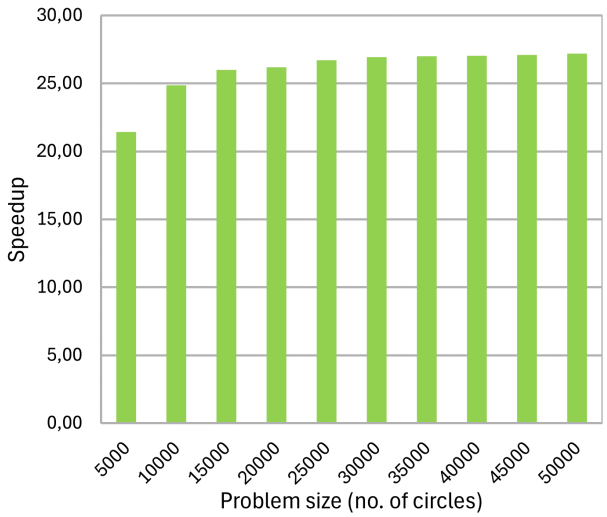


Figure 4.1: Grafico rappresentante lo speedup di CUDA relativamente a quello di OpenMP

[3]tutte quelle operazioni di comunicazione e sincronizzazione usate nella programmazione parallela