Progetto di High Performance Computing 2021/2022

Matteo Bambini, matr. 0000916140

20/09/2022

Indice

1 Introduzione

Per la realizzazione di questo progetto ho deciso di iniziare sviluppando la versione parallelizzata con OpenMP e, una volta terminata quella, ho sviluppato la versione parallelizzata con CUDA. Per raccogliere i dati riguardanti le prestazioni ho realizzato alcuni script che automatizzano l'esecuzione del programma e successivamente ho raccolto i dati e li ho inseriti in un foglio excel che mi ha permesso di elaborarli ed estrarre i grafici presenti in questa relazione. Tutti i test che hanno contribuito alle analisi delle prestazioni sono stati eseguiti nello stesso ambiente, ovvero sul server isi-raptor03.csr.unibo.it.

2 Versione OpenMP

2.1 Descrizione Implementazione

La parallelizzazione utilizzando OpenMP è stata molto semplice: è bastato aggiungere la direttiva `#pragma omp parallel for` prima del primo ciclo for all'interno della funzione `step`, ovvero la funzione che esegue il singolo passo dell'elaborazione, per ottenere il risultato desiderato. Questo è stato possibile perché il codice all'interno del ciclo è embarrassingly parallel, ovvero può essere suddiviso in task indipendenti che non richiedono comunicazione tra di loro. Se non fosse stato così sarebbe stato necessario rielaborare il codice all'interno per renderlo parallelo.

Un altro modo per parallelizzare l'algoritmo sarebbe stato quello di inserire la direttiva `#pragma omp parallel for` nel ciclo interno, ma in questo modo ogni task avrebbe elaborato una sola cella, mentre inserendo la direttiva nel ciclo esterno ogni task elabora un'intera riga. In questo modo si hanno partizioni più grandi e una granularità maggiore, che hanno come conseguenza una minimizzazione dell'overhead e una massimizzazione delle prestazioni. Facendo dei test ho scoperto che più aumenta il numero di thread, più aumenta la differenza di prestazioni tra la versione con la direttiva nel ciclo esterno e quella con la direttiva nel ciclo interno.

2.2 Analisi Prestazioni

Dal grafico in figura 1 si può notare che la linea che descrive lo speedup è una retta con una pendenza molto vicina a quella della retta che descrive il numero di processori. Questo indica un'elevata efficienza dell'algoritmo, come si può vedere anche dal grafico della strong scaling efficiency (figura 2).

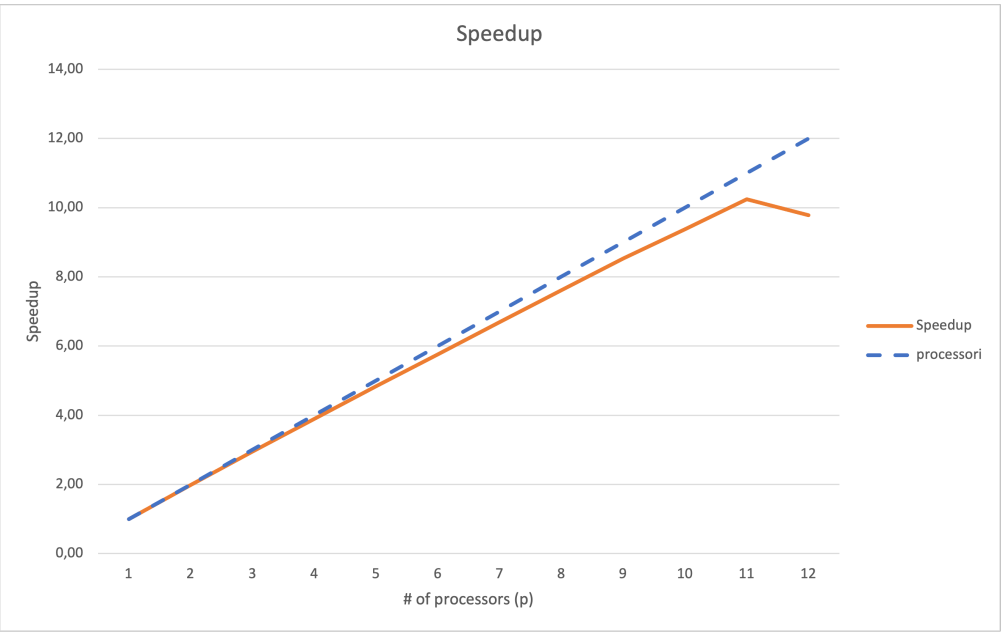

Figura 1: Speedup

Nella parte finale dei grafici però si vede un calo importante di prestazioni. La mia ipotesi iniziale era che questo calo fosse dovuto alla dimensione ridotta del dominio e quindi ad un partizionamento troppo fine che avrebbe causato un overhead eccessivo. Per essere sicuro ho fatto

dei test raddoppiando la dimensione del dominio, ma il risultato è stato sempre lo stesso, quindi sono arrivato alla conclusione che questo calo sia dovuto dal fatto che ho assegnato al programma tutti i processori fisici disponibili nella macchina su cui ho fatto i test, e questo, considerando che oltre al mio programma in esecuzione ci sono tutti i servizi del sistema operativo, ha causato una congestione della CPU che ha portato come conseguenza il calo di prestazioni.

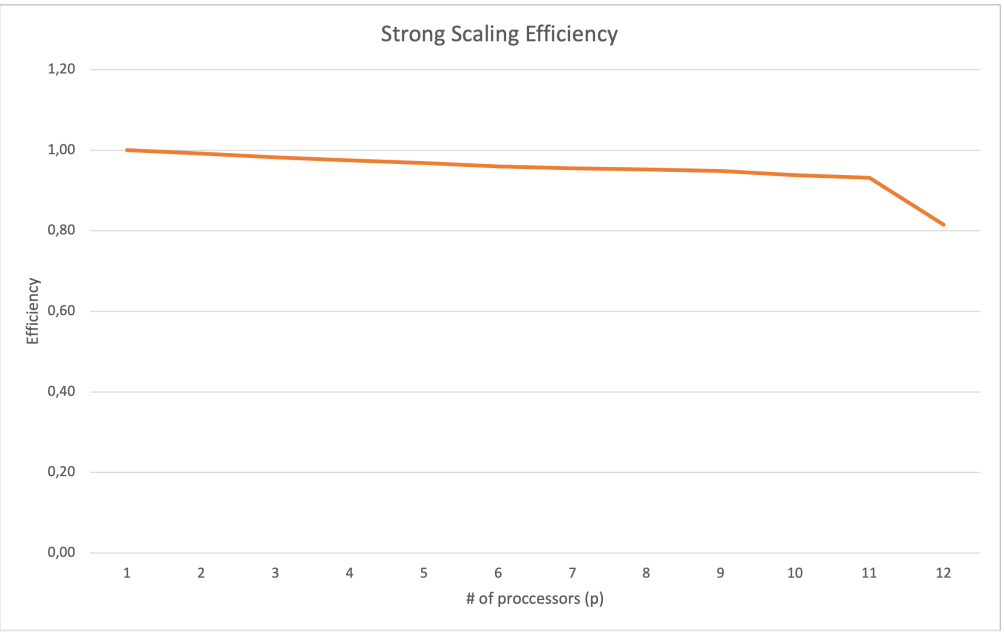

Figura 2: Strong Scaling Efficiency

3 Versione CUDA

3.1 Descrizione Implementazione

La parallelizzazione utilizzando CUDA ha richiesto un po' più di lavoro rispetto a quella con OpenMP ma il processo è stato comunque abbastanza semplice. Per sviluppare il kernel CUDA sono partito dalla funzione `step` che era già implementata: prima di tutto ho rimosso il ciclo `for` che era al suo interno e successivamente ho aggiunto il calcolo degli indici che permette di identificare il pixel da elaborare (calcolo effettuato utilizzando thread id e block id). È stato necessario anche aggiungere la direttiva `__device__` alle funzioni `swap_cells` e `IDX` per renderle visibili anche dalla GPU e poterle utilizzare senza dover fare ulteriori modifiche. Il ciclo che era all'interno l'ho spostato nel main; in questo modo le iterazioni vengono gestite dalla CPU e il kernel CUDA esegue una sola iterazione ogni volta che viene richiamato. Ovviamente è stato anche necessario gestire allocazione e deallocazione della memoria della GPU.

3.2 Analisi Prestazioni

Utilizzando CUDA non è possibile calcolare speedup e strong/weak scaling efficiency, quindi ho dovuto utilizzare altre metriche per calcolare le prestaioni. In particolare ho deciso di utilizzare il throughput, ovvero il numero di elementi processati per unità di tempo. In questo caso ho deciso di utilizzare come unità di misura per il throughput il megapixel per secondo (Mpixel/s).

Nel grafico in figura 3 si può notare che, con una dimensione del dominio ridotta, ci sono importanti oscillazioni, mentre man mano che la dimensione del dominio aumenta, il valore di throughput tende a stabilizzarsi tra 13000 e 14000 Mpixel/s.

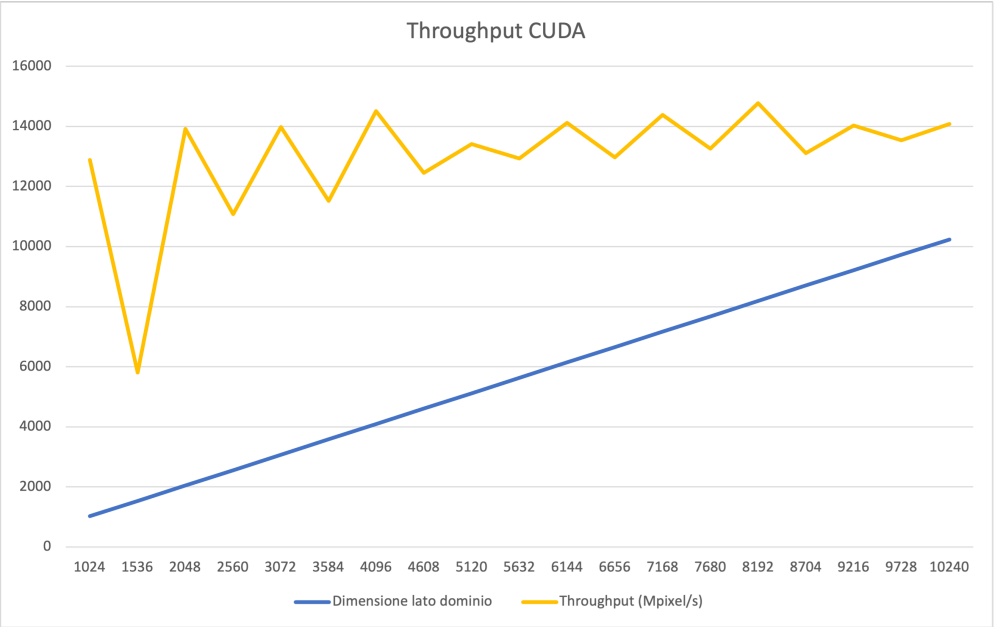

Figura 3: Throughput versione CUDA