Progetto di High Performance Computing

Marco Sternini, 0000971418

Maggio 2023

Introduzione

Il progetto consiste nella realizzazione di due programmi che sfruttano la programmazione parallela a partire da un programma seriale. Il programma in questione è scritto nel linguaggio C e simula il comportamento dei fluidi utilizzando l'algoritmo SPH (Smoothed Particle Hydrodynamics). Le due versioni descritte successivamente sfruttano rispettivamente: OpenMP per la programmazione a memoria condivisa e Cuda per la programmazione GPU. Le due versioni sono state eseguite e valutate sul server *isi-raptor03* con le seguenti specifiche: CPU Intel Xeon E5-2603 con 6 core fisici e 6 core logici e GPU Nvidia GTX1070.

Analisi generale

Il programma seriale è composto principalmente da quattro funzioni che compongono un passo della simulazione. Queste funzioni vanno eseguite in uno specifico ordine, e non possono essere quindi eseguite parallelamente. Le prime due funzioni "*compute_density_pressure*" e "*compute_forces*" si presentano molto simili: due cicli annidati che, presa una particella, modifica lo stato di questa osservando lo stato delle altre particelle.

```
for i in [0, n] do
   for j in [0, n] do
        p[i].x <- p[j].y
   end for
end for
```

Figura 1: Pseudocodice riassuntivo delle funzioni *compute_density_pressure* e *compute_forces*.

In tutte e due le funzioni l'elemento i aggiorna parametri differenti dai parametri letti in j. Non esistono quindi dipendenze nei cicli.
La terza funzione "*integrate*" presenta un unico ciclo che aggiorna lo stato della particella, segue una struttura **embarassingly parallel** in quanto ogni modifica all'elemento è indipendente dagli altri elementi. La quarta funzione "*avg_velocities*" consiste in un unico ciclo che calcola un valore statistico del modello. La struttura permette l'utilizzo del pattern **reduction** con l'operatore somma.

```
var value
for i in [0, n] do
    value <- value + function(p[i])
end for
```

Figura 2: Pseudocodice riassuntivo della funzione *avg_velocities*.

Versione OpenMP

La versione a memoria condivisa è stata sviluppata nel seguente modo:

- parallelizzando i cicli esterni nelle due funzioni *compute_density_pressure* e *compute_forces*

- parallelizzando l'unico ciclo nella funzione *integrate*
- parallelizzando con clausola "reduction" il ciclo all'interno della funzione *avg_velocities*

Osservando le funzioni con due cicli annidati, nei cicli interni la fase di computazione principale non è sempre eseguita perché delimitata da un controllo. L'esecuzione è quindi irregolare. Nel grafico è riportato lo speedup relativo alla versione con scheduling dinamico e alla versione con scheduling statico nelle due funzioni *compute_density_pressure* e *compute_forces*.

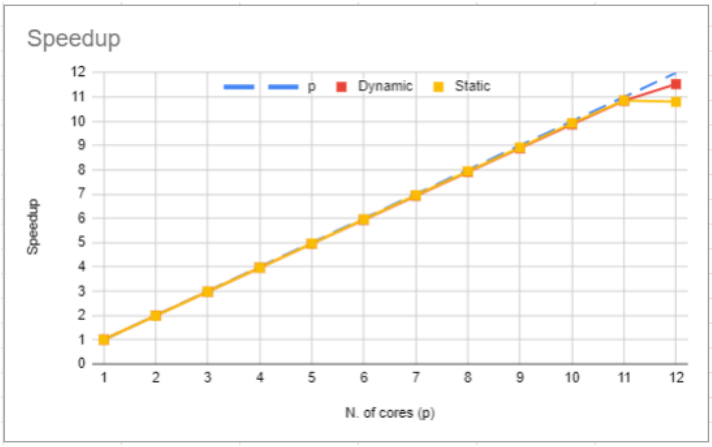

Figura 3: Confronto speedup tra scheduling dinamico e statico con dimensione del problema di 15000 particelle

L'utilizzo dello scheduling dinamico ha permesso di raggiungere risultati migliori per istanze del problema maggiori al costo di prestazioni leggermente inferiori nelle istanze più contenute (da 2000 a 10000 particelle).

Valutazione prestazioni

La misurazione del tempo di esecuzione non considera la fase di inizializzazione delle strutture dati. Il codice eseguito serialmente è molto poco, infatti si osserva come lo speed-up misurato (con dimensione opportuna) si discosta pochi decimi o centesimi da uno speed-up ideale lineare.

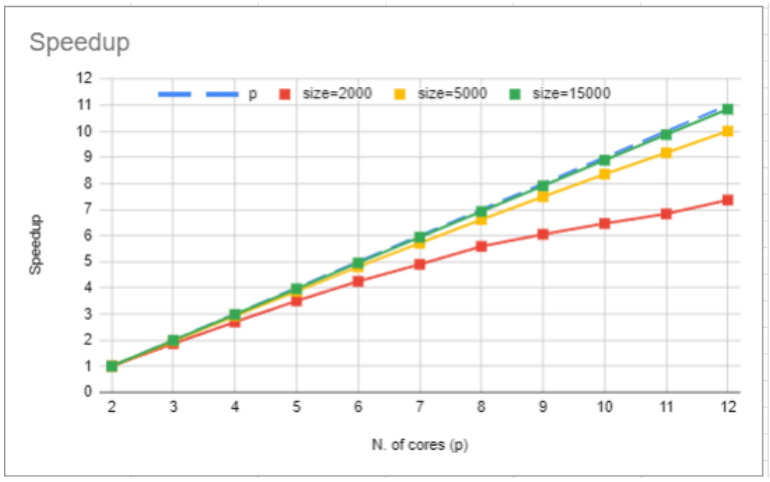

Figura 4: Grafico speedup

L'impatto dello scheduling dinamico si vede nelle istanze del problema con numero di particelle minore. L'overhead della gestione dei thread runtime ne peggiora la strong-scaling efficiency e lo speedup generale.

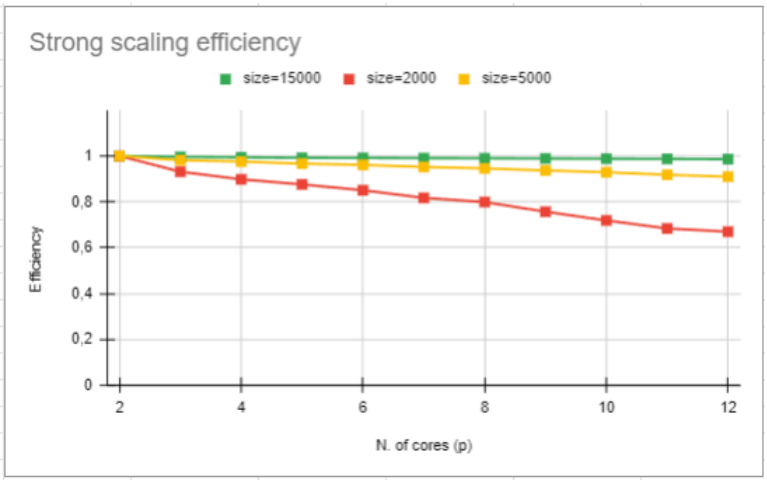

Figura 5: Grafico strong-scaling efficiency

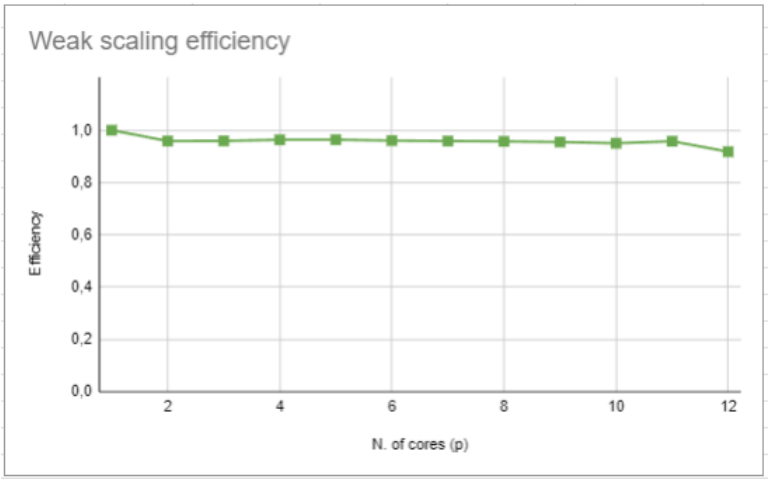

Figura 6: Grafico weak-scaling efficiency

La weak-scaling efficiency è stata calcolata considerando 2000 particelle come unità base di lavoro e una complessità di $O(n^2)$. Il programma presenta un'ottima weak-scaling efficiency compresa tra 1 e 0.9.

Versione Cuda

Le 4 funzioni principali diventano 4 differenti kernel eseguiti in ordine. I primi tre kernel non utilizzano particolari pattern di programmazione parallela. Ogni thread calcola il proprio **id** ed aggiorna lo stato della particella corrispondente al suo identificativo. La griglia dei thread e dei blocchi utilizzata è unidimensionale con una dimensione massima per blocco di 1024 thread. Il quarto kernel ha richiesto più lavoro in quanto la reduction in cuda deve essere eseguita "manualmente". Ogni thread popola la **shared memory** del blocco con il valore di velocità media della particella a cui fa riferimento. Successivamente i thread del blocco cooperano per sommare tutti i valori fino ad ottenere un unico

risultato. Questo risultato è il risultato parziale della riduzione. Ogni blocco produce un risultato parziale che l'host utilizzerà per calcolare il risultato finale. Se la dimensione del problema non è multiplo di 1024 (la dimensione massima del blocco) certi thread saranno associati a particelle non esistenti. Per ovviare a questo problema se il proprio identificativo è superiore al numero di particelle del modello al posto della velocità media viene inserito 0, ovvero l'elemento neutro della riduzione in questione. Un ulteriore modifica è l'utilizzo della **constant memory** della gpu. In fase di inizializzazione del programma l'host calcola le costanti e copia il valore nella constant memory del device.

Valutazione prestazioni

La valutazione comprende due parametri: il tempo di esecuzione e il throughput. La formula di calcolo del throughput è la seguente:

$$T = \frac{n_p \cdot s}{t_a} \tag{1}$$

Dove n_p è il numero di particelle in input, s il numero di step di simulazione e t_a il tempo medio di esecuzione.

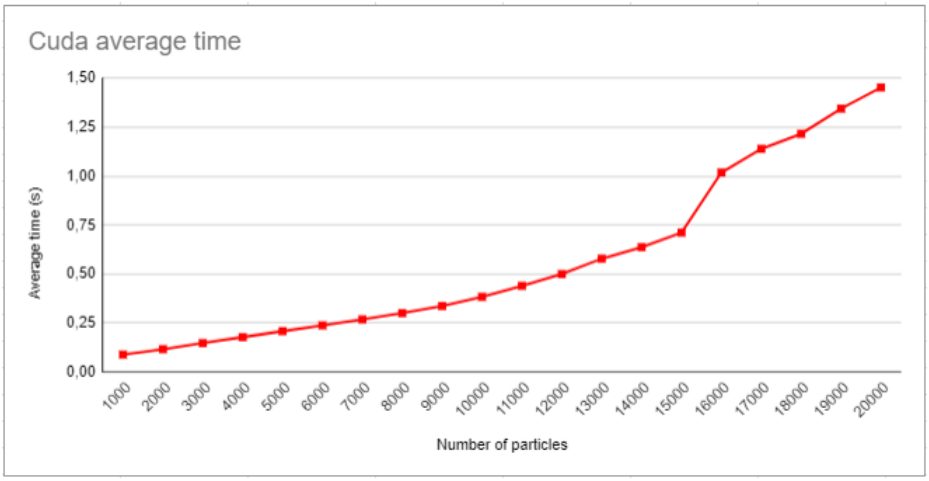

Figura 7: Grafico tempo medio di esecuzione per numero di particelle

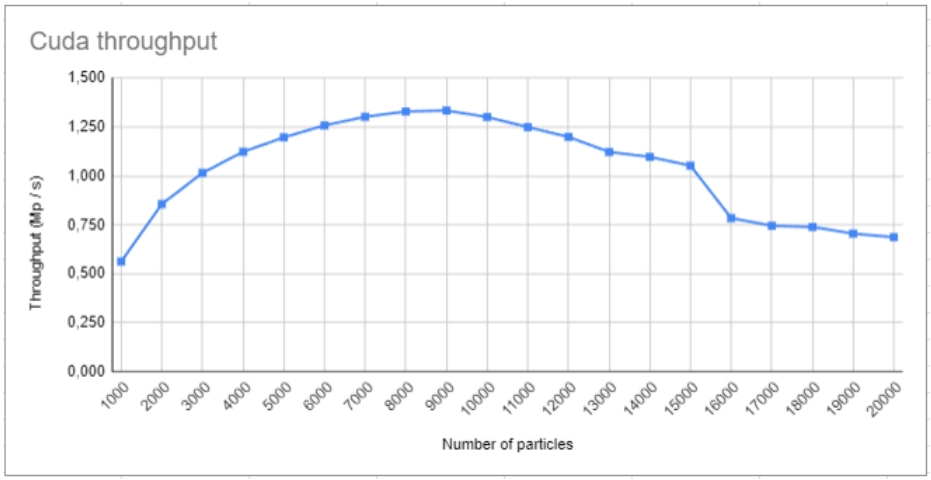

Figura 8: Grafico throughput in milioni di particelle al secondo

In entrambi i grafici si nota una ripida salita o discesa rispettivamente nel grafico del tempo medio o del throughput, da 15000 a 16000 particelle in input. Osservando le specifiche della scheda video in

questione questa presenta 15 streaming multiprocessor che possono gestire ognuno un massimo di 32 blocchi o 2048 thread. Considerando una dimensione dell'input di 15000 particelle la computazione sarà divisa in $nBlocks_1 = \lceil 15000/1024 \rceil = 15$. Lo scheduler dei blocchi cerca di mantenere un carico di lavoro bilanciato tra i multiprocessor e quindi assegna un blocco ad ognuno di essi[1]. Considerando invece un input di 16000 particelle $nBlocks_2 = \lceil 16000/1024 \rceil = 16$ per cui un multiprocessor avrà 2 blocchi assegnati. Questo aumenta il tempo di esecuzione e quindi diminuisce il throughput calcolato. Ho effettuato ulteriori test per verificare se il comportamento si ripresentasse in altre istanze del problema. Aumentando il limite del problema lo stesso fenomeno si ripropone nell'intervallo da 30000 a 31000 particelle ovvero 30/31 blocchi confermando l'ipotesi precedente. In tal caso ogni multiprocessor avrà 2 blocchi assegnati mentre uno solo 3.

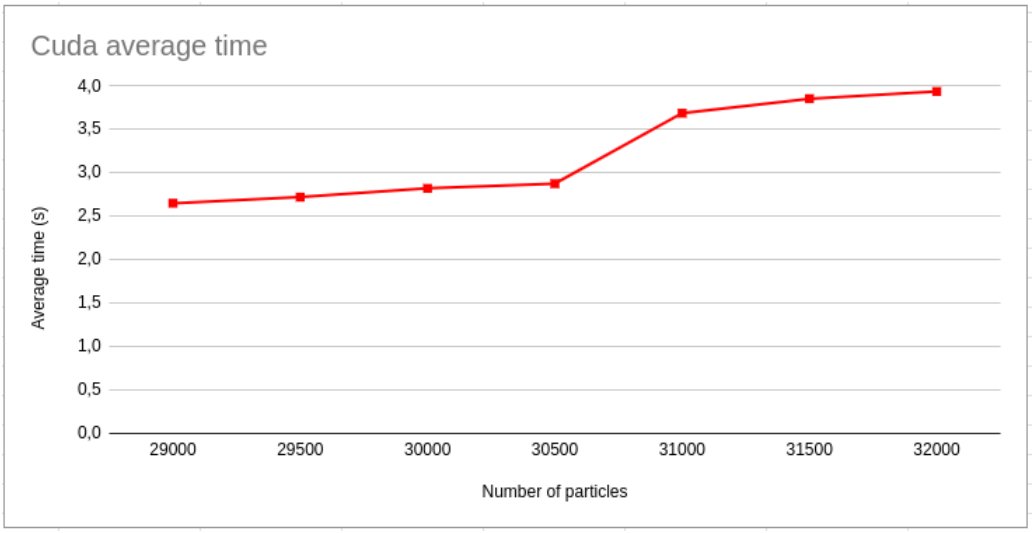

Figura 9: Grafico tempo medio per dimensioni del problema da 29000 a 32000

Conclusioni

In entrambe le versioni le prestazioni rispetto al programma seriale sono notevolmente migliorate. L'analisi ha permesso poi di capire dove è possibile focalizzarsi per ottimizzare ancora di più il codice e diminuire il tempo di esecuzione. OpenMP ci permette di parallelizzare codice seriale molto più facilmente attraverso le direttive pragma del preprocessore. CUDA richiede modifiche sostanziali al programma, ma permette di risolvere istanze del problema in tempi molto minori. Sarebbe particolarmente interessante approfondire la scelta della dimensione dei blocchi e della griglia in CUDA e come questa influenzi le prestazioni dei programmi.

Riferimenti bibliografici

- GPU Architecture notes
- Demystifying the Placement Policies of the NVIDIA GPU Thread Block Scheduler for Concurrent Kernels

[1] Inizialmente ipotizzato e poi confermato dall'articolo "Demystifying the Placement Policies of the NVIDIA GPU Thread Block Scheduler for Concurrent Kernels" citato nella bibliografia.