# Relazione progetto
# High Performance Computing

Giacomo Amadio mat. 971304

05/06/2023

# Indice



## 0.1 Introduzione

In questo documento viene presa in esame l'implementazione seriale dell'algoritmo SPH (Smoothed Particle Hydrodynamics), utilizzato per descrivere il comportamento di corpi deformabili, poi successivamente esteso per simulare i fluidi.

L'implementazione considerata richiede tempo $\Theta(N^2)$ ad ogni iterazione. L'esecuzione, anche con input limitati, richiede una considerevole quantità di tempo.

Lo scopo dunque è quello di realizzare due versioni parallelizzate del programma, una a memoria condivisa utilizzando `OpenMP` e l'altra utilizzando `CUDA` per ridurne i tempi di escuzione e poi confrontare i risultati.

# Capitolo 1

# Implementazione

## 1.1 OpenMP

Per quanto riguarda le funzioni di calcolo delle forze e densità-pressione è stata applicata la stessa logica, in quanto strutturalmente simili. Entrambe costituite da due cicli annidati, in quello interno vengono fatte operazioni di accumulo su variabili condivise al verificarsi delle condizioni di vicinanza fra particelle. Queste operazioni, oltre a creare sbilanciamento del carico, richiederebbero una gestione ulteriore per evitare race conditions tra i thread, causando di fatto un overhead significativo. Si noti inoltre che la parallelizzazione del ciclo interno pur preservando il pool di thread richiederebbe l'esecuzione della direttiva `omp for` N volte creando ulteriore overhead. Per questi motivi si è optato per la parallelizzazione del ciclo esterno usando uno scheduling statico a grana grossa, poiché meno affetto da sbilanciamento del carico.

La funzione per il calcolo di posizione-velocità non avendo loop carried dependencies e non avendo problemi di sbilanciamento del carico è stata parallelizzata facendo uso di scheduling statico a grana grossa.

Infine, la funzione per il calcolo della velocità media, che è costituita da un ciclo che accumula i valori su una variabile risultato, può essere parallelizzata facendo una riduzione.

### Valutazione delle prestazioni

I dati presi in esame sono stati raccolti su una macchina con processore Intel Xeon x86 12 core 1.70 Ghz e sono riportati in Tabella 1.1.

Valutando dunque gli *Speedup* dell'implementazione parallela, riportati in Figura 1.1, si può notare come per input di dimensioni ridotte questi siano limitati dall'overhead di gestione dei thread, per poi tendere ad allinearsi

con il loro andamento ideale all'aumentare delle dimensioni del problema. Si noti inoltre che sfruttando tutte le risorse a disposizione sulla macchina non si ottiene lo *Speedup* ideale, fenomeno probabilmente causato dall'esecuzione in background di altri processi. Questo comportamento non si ripete invece facendo un uso parziale delle risorse.

In Tabella 1.2 sono riportati i dati sull'efficienza, questi danno conferma del fatto che la parallelizzazione risulta meno efficace con problemi di dimensioni ridotte.

| T(p) / N | $5{\cdot}10^2$ | $25{\cdot}10^2$ | $50\cdot10^2$ | $75{\cdot}10^2$ | $100{\cdot}10^2$ | $150{\cdot}10^2$ | $200{\cdot}10^2$ |
|---|---|---|---|---|---|---|---|
| T(1) | 0.466 s | 9.25 s | 36.7 s | 82.5 s | 146.7 s | 330 s | 585 s |
| T(4) | 0.155 s | 2.53 s | 9.56 s | 21.2 s | 37.4 s | 83.5 s | 147.9 s |
| T(8) | 0.095 s | 1.29 s | 4.80 s | 10.6 s | 18.7 s | 41.7 s | 74.0 s |
| T(12) | 0.075 s | 0.95 s | 3.55 s | 7.90 s | 13.7 s | 30.7 s | 53.9 s |

Tabella 1.1: Confronto dei tempi di esecuzione della versione OpenMP fissato il numero di core p al variare della dimensione dell'input N.

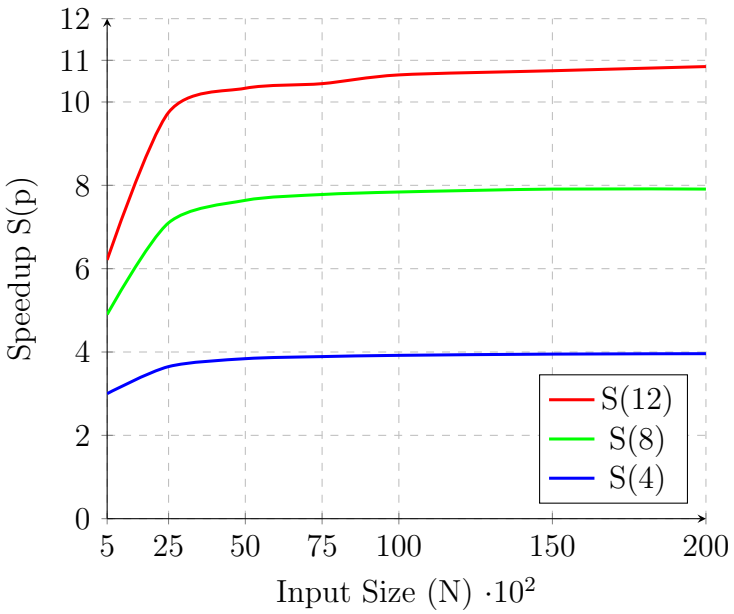


Figura 1.1: Confronto dell'andamento degli Speedup all'aumentare delle dimensioni dell'input N, fissato il numero di core p.

| Eff(p) / N | $5{\cdot}10^2$ | $25{\cdot}10^2$ | $50\cdot10^2$ | $75{\cdot}10^2$ | $100{\cdot}10^2$ | $150{\cdot}10^2$ | $200{\cdot}10^2$ |
|---|---|---|---|---|---|---|---|
| E(4) | 75% | 94.2% | 96% | 97.2% | 98% | 98.7% | 99% |
| W(4) | 96% | 96.8% | 98.2% | – | – | – | – |
| E(8) | 61% | 92.5% | 95.5% | 97.2% | 98% | 98.8% | 98.9% |
| W(8) | 98.9% | 97.6% | 98.6% | – | – | – | – |
| E(12) | 51% | 81% | 86% | 87% | 88.7% | 89.5% | 90.4% |
| W(12) | 98.9% | 89% | 91% | – | – | – | – |

Tabella 1.2: Confronto dell'efficienza della versione OpenMP rispetto a quella seriale, fissato il numero di core p al variare della dimensione dell'input N.

## 1.2 CUDA

L'implementazione CUDA del modello `SPH`, per rendere più efficaci gli accessi in memoria della GPU, utilizza una struttura dati modificata. La versione OpenMP e quella seriale fanno uso di *Array Of Structures*, mentre questa usa una *Structure Of Arrays*.

Per le funzioni di calcolo delle forze e densità-pressione, in quanto strutturalmente simili, come per OpenMP, è stata applicata la stessa logica. Le iterazioni del ciclo esterno sono state partizionate fra i blocchi e dato che le particelle prese in esame verranno lette N volte, prima di iniziare il ciclo, i thread concorrono alla copia in *shared memory* della partizione. Le iterazioni del ciclo interno sono invece suddivise fra i thread, che calcolano in maniera concorrente i risultati parziali, per poi salvarli in *shared memory*. Questi vengono poi sommati tramite riduzione e il thread di indice zero copia i risultati in memoria globale.

L'implementazione della funzione per il calcolo di posizione-velocità non avendo riletture di stessi elementi non fa uso di *shared memory* e data l'assenza di *loop-carried dependecies* è stata assegnata ogni iterazione ad un thread diverso.

Infine la funzione per il calcolo della velocità media è stata realizzata assegnando ad ogni thread il calcolo di un risultato parziale, successivamente salvato in shared memory per calcolare la somma, facendo una riduzione con l'ausilio di tutti i thread del blocco.

### Valutazione delle prestazioni

I dati sono stati raccolti su una macchina con processore Intel Xeon 12 core 1.70 Ghz e scheda grafica Nvidia GTX 1070 e sono riportati in Tabella 1.3.

Confrontando dunque il *Throughput(Melements/s)* dell'implementazione CUDA con quello di OpenMP, riportati in Figura 1.2, si può notare come per input di dimensioni ridotte la versione CUDA abbia *Throughput* minore a causa dell'overhead di gestione di memoria e alla partizione a grana grossa utilizzata, per poi crescere fino ad allinearsi con il suo andamento ideale all'aumentare delle dimensioni del problema.

In Figura 1.3 abbiamo conferma del fatto che la versione CUDA risulta meno efficace con problemi di dimensioni ridotte per poi andare a stabilizzarsi per $N > 150 \cdot 10^2$.

| N | $5 \cdot 10^2$ | $25 \cdot 10^2$ | $50 \cdot 10^2$ | $75 \cdot 10^2$ | $100 \cdot 10^2$ | $150 \cdot 10^2$ | $200 \cdot 10^2$ |
|---|---|---|---|---|---|---|---|
| OpenMP | 0.075 s | 0.97 s | 3.55 s | 7.90 s | 13.7 s | 30.7 s | 53.9 s |
| CUDA | 0.23 s | 0.52 s | 0.57 s | 0.63 s | 0.69 s | 0.8 s | 1.8 s |

Tabella 1.3: Confronto dei tempi di esecuzione della versione CUDA e OpenMP usando tutti i cores della macchina, al variare della dimensione dell'input N.

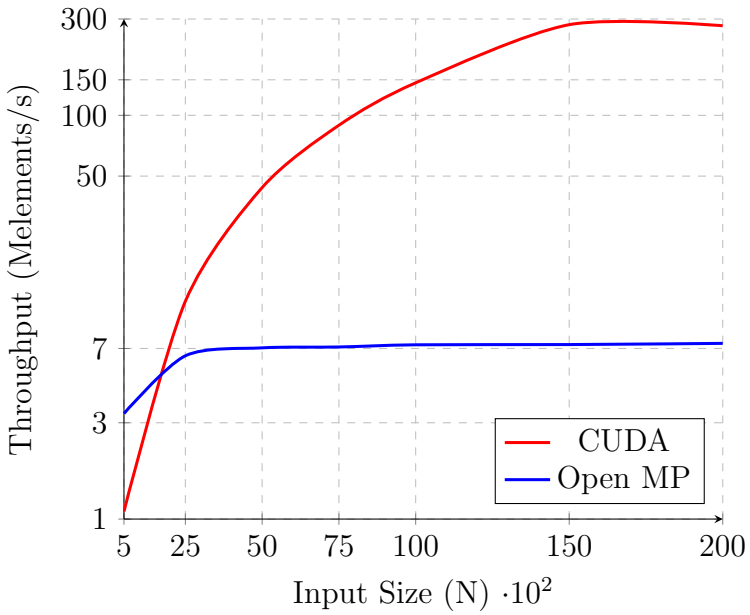


Figura 1.2: Confronto dell'andamento del Throughput all'aumentare delle dimensioni dell'input N, della verisione CUDA e OpenMP.

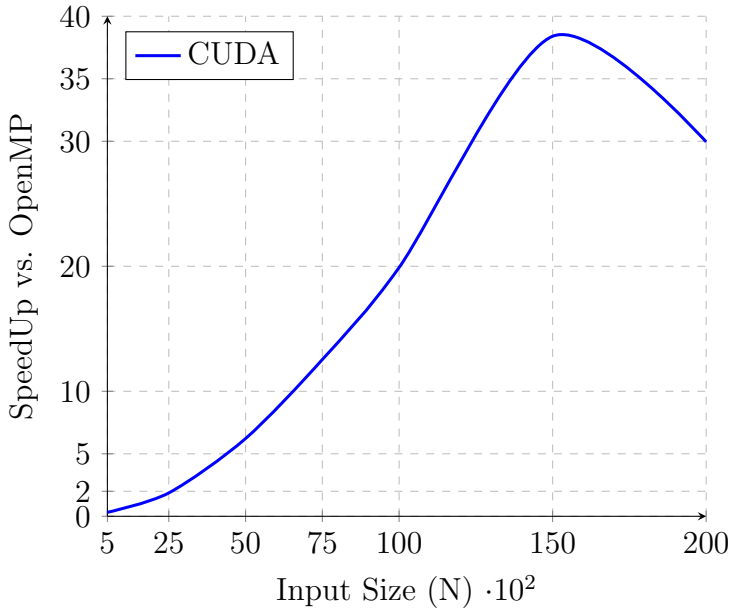


Figura 1.3: Andamento dello Speedup della versione CUDA del programma all'aumentare delle dimensioni dell'input N.

## 1.3 Conclusioni

Analizzando i risultati si giunge alla conclusione che le implementazioni prodotte raggiungono un buon livello di efficienza, risulta inoltre decisamente più efficace la versione CUDA rispetto a quella OpenMP. Anche se entrambe migliorabili è evidente il problema di efficienza della versione CUDA con input di grandezze ridotte, dovuto probabilmente alla scelta di un partizionamento fra blocchi a grana grossa.

Un possibile sviluppo futuro vertirebbe sulla ricerca di un partizionamento più efficace per migliorare le prestazioni sul dominio in esame, e l'adattamento dell'implementazione CUDA per funzionare anche in versione grafica.