Progetto High Performance Compiuting 2022/2023

Filippo Di Pietro, Matr: 00009761606

March 2023

1 Introduzione

Come metodi di parallelizzazione ho deciso di usare oltre che OMP, come da specifiche, CUDA. Inoltre ho provato a percorrre una delle varie strade con le istruzioni SIMD.

Per la versione OpenMP vi è la principale problematica che vengono creati 4 thread pool per step, aggiungendo quindi un overhead, poi vi è da scegliere un partizionamento delle particelle per ogni thread, infine ho combinato la versione simd e OpenMP con prestazioni migliori in una unica. Per la versione CUDA ho trasformato le 4 funzioni, che operano sulle particelle, in kernel, poi ho provato a passare da AoS a SoA, per vedere se vi è un aumento di prestazioni, infine ho provato ad usare la shared memory. Per fare le misurazioni di OpenMP ho usato il mio processore di casa: 3900X (di AMD, con 12 core + SMT), per CUDA ho usato la mia GPU di casa: 2060 SUPER, tutti valori usati per fare i conti, sono la media di 12 misurazioni.

2 Versione SIMD

Ho provato una strada per usare le istruzioni simd, nelle funzioni *compute_forces*, *compute_density_pressure* ho dichiarato un tipo vettore (del gcc) a 16 poi 32 byte, per i valori float che ho bisogno di accumulare (densità, viscosità e pressione), poi ho impostato il codice in maniera tale da far si che i primi 4 / 8 valori da accumulare (dipende dalla grandezza del vettore) li salvo nel vettore gcc, appena riempio il vettore eseguo la somma di tutti gli elementi, ripeto fino alla fine del confronto tra particelle, una volta finito gestisco le rimanenze sommando i valori rimanenti.

Infine ho testato i vettori a 16 e 32, ma non vi è alcuna differenza di prestazioni, quindi ho scelto di mantenere quelli da 16 per rendere il codice facile da leggere.

2.1 AoS a SoA

Infine per la soluzione (2) ho anche optato di passare da AoS a SoA per far si che si usi al meglio la cache, per l'array contenente la posizione, o la densità di tutte le particelle.

500	1000	2000	4000	6000
0.527 - 0.300	2.121 - 1.207	8.428 - 4.784	33.331 - 19.155	74.843 - 42.785

Table 1: In questa tabella sono riportati (i valori medi di 12 misurazioni), della versione seriale e della versione simd (2.1). La prima riga indica il numero di particelle usate, mentre la seconda i tempi della versione seriale a sinistra (del -) a destra invece i tempi della versione SIMD + SoA

Con le seguenti misurazioni posso affermare di aver ottenuto uno speed up medio di 1.74 un aumento quindi del 74 %, rispetto alla versione seriale.

3 Versione OMP

3.1 Partizione statica

Il modo più rapido, pulito e semplice per parallelizzare il simulatore è quello di mettere le direttive *pragma omp for* sopra al loop più esterno del calcolo della pressione, forze e integrazione poi nel loop di calcolo della velocità

media, si aggiunge la clausola *reduction* per ottenere in maniera efficente la velocità media delle particelle. Però con questa soluzione vengono creati 4 thread-pool per ogni step, aggiungendo quindi un overhead ad ogni step, infatti vi sono delle istanze particolari (piccole) in cui non vale la pena usare la versione OpenMP poichè l'overhead rallenta la soluzione.

3.1.1 Da 4 thread-pool ad 1

Inoltre vi è una maniera più efficace della (3.1) per parallelizzare i cicli mantenendo il partizionamento statico. Si può ottenere più velocità evitando di distruggere e creare 4 threadpool per ogni singolo step, creandone uno unico ad ogni step, poi dopo il calcolo della pressione e delle forze, si mette una sincronizzazione a barriera, dopo la funzione integrate non è necessario, poichè per il calcolo della velocità media non vi sono più look up dependecies per effettuare i calcoli, quindi ogni thread può calcolare la velocità media delle sue particelle subio dopo aver finito di eseguire la *integrate*. Con questa tecnica si guadagna qualcosa in termini di prestazioni rispetto alla banale (3.1), senza perdere troppa leggibilità del codice.

3.2 Partizione Dinamica

Come seconda strada ho scelto di prendere la soluzione (3.1) e partizionare le particelle per thread in maniera dinamica, facendo alcune prove sperimentali ho ottenuto un chunksize ottimale di 15 particelle, e il codice rimane molto leggibile.

La soluzione non ha cali drastici di prestazioni quando subentra la tecnologia SMT di AMD, quindi la preferisco alla soluzione (3.1.1)

3.3 Parallelizzazione con istruzioni SIMD

Per ottenere una soluzione che sia più veloce, ho deciso di unire la versione migliore di OpenMP (3.2) e Simd (2.1).

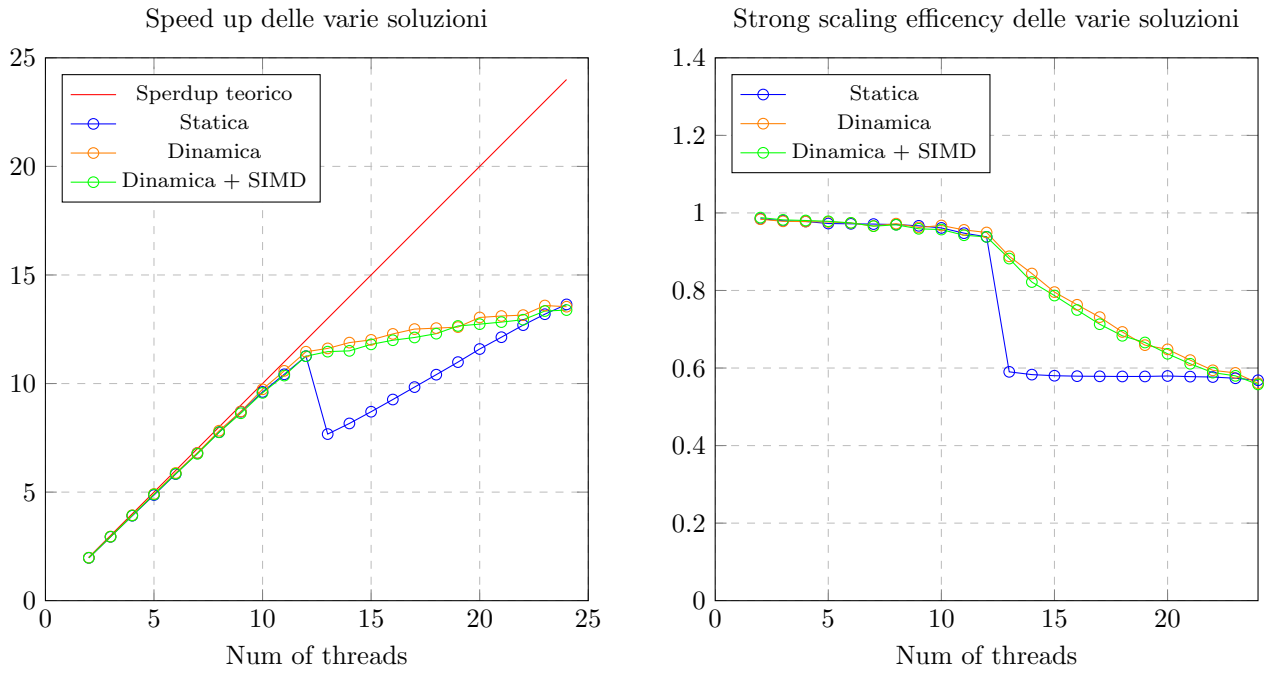

Figure 1: In questa figura si mostra il programma avviato con 6000 particelle e 100 passi sul mio pc di casa, le soluzioni scalano bene (weak scaling piatta, speedup vicino a quello teorico) con tutti i core fisici, poi quando esauriscono, subentra la tecnologia SMT che inizia a distribuire il carico su un numero di thread maggiore rispetto ai cores fisici. Nella verisone a partizionamento statico (3.1.1) perde di efficenza subito, mentre a partizionamento dinamico riesce a distribuire il carico in maniera più efficente (3.2). Alla fine 24 thread, la versione statica, raggiunge la stessa efficenza e speedup della versione a partizionamento dinamico. Con queste misurazioni ho deciso quindi di prendere la versione OpenMP a partizionamento dinamico, perché non ha questo calo drastico delle prestazioni come la verione statica e l'ho unito con la versione SIMD come scritto nella sezione (3.3)

3.4 Weak Scaling Efficency

3.4.1 Al variare delle particelle

Per calcolare le waek scaling efficency al variare del numero di particelle, sappiamo che il tempo è proporzionale al quadrato rispetto al numero di particelle per una costante: $T_{pa} = C_{pa}N_{pa}^2$ poiché ha complessità computazionale $\Theta(N_{pa}^2)$, suddividendo poi il carico per P processori si ha che il carico di lavoro per processore è $T_P = \frac{C_{pa}N_{pa}^2}{P}$ spostando i fattori posso esprimere il numero di particelle in funzione del numero di processori $N_{pa}(P) = \sqrt{P} \cdot \sqrt{C_{pa2}}$ (dove $C_{pa2} = T_P/C_{pa}$) scegliendo un valore della costante C_{pa2}, in questo caso 1500, posso calcolare il numero di particelle per far si che il lavoro di ogni thread rimanga costante all'aumetare del numero di thread.

3.4.2 Al variare dei numero di passi

Per calcolare le waek scaling efficency al variare del numero di passi, sappiamo che il tempo è direttamente proporzionale al numero di passi per una costante: $T_s = C_s \cdot S$ poiché ha complessità computazionale $\Theta(S)$, suddividendo poi il carico per P processori si ha che il carico di lavoro per processore è $T_P = \frac{C_s \cdot S}{P}$ spostando i fattori posso esprimere il numero di particelle in funzione del numero di processori $S(P) = P \cdot C_{s2}$, scegliendo un valore della costante C_{s2} (dove $C_{s2} = T_P/C_s$) in questo caso 100, posso calcolare il numero di passi per far si che il lavoro di ogni thread rimanga costante all'aumetare del numero di thread.

Figure 2: Come si può notare per il grafico in blu fino a 12 thread l'efficenza rimane costante vicino al valore 1, quindi scala bene all'aumento del numero di thread, oltre i 12 thread entra in gioco la tecnologia SMT, e l'efficenza crolla (come abbiamo già visto nella figura 1). Mentre per il grafico rosso, tra 10 e 14 thread vi è della volatilità non indifferente, oltre che a diminure già drasticamente rispetto alla versione per particelle, poichè il carico di lavoro non rimane costante all'umentare dei passi, potrebbe essere dovuto dal fatto che all'interno delle funzioni *compute_forces*, *compute_density_pressure* trovino più vicini rispetto a prima, quindi vengono effettuate più somme. Poi si ha un peggioramento drastico come era normale aspettarsi, sempre come detto prima per il grafico blu.

4 Versione CUDA

4.1 Parallelizzazione Banale

Per parallelizzare in CUDA, ho trasformato le 4 funzioni, *pressione e densità*, *forze*, *integrazione*, *velocità media* in 4 kernel, che vengono chiamati ad ogni step dall'host. Poi ho adattato i kernel in modo tale da far si che ogni thread CUDA possa eseguirli, e ho aggiunto una reduction nella funzione che calcola la velocità media.

4.2 Passaggio da AoS a SoA

Per velocizzare la soluzione CUDA 4.1, ho deciso di passare da una Array of Struct ad una Struct of Array, in modo tale da sfuttare la cache della GPU.

4.3 Uso della shared

Per sfruttare la shared memory, ho pensato di percorrere la seguente strada: ottengo il numero di byte di shared memory per blocco (la metto in una costante, poiché non si può allocare a runtime), 49152 byte, e la occupo tutta per immagazzinare le posizioni delle particelle del ciclo più innestato (quello che controlla tutte le posizioni di tutte le particelle, nelle funzioni di calcolo di densità e forze), poiché sono le variabili che vengono più rilette di tutte. Una volta riempita la shared memory, controllo tutte le posizioni che vi ho salvato dentro, vengono eseguite le operazioni di accumulo (su rho, viscosità e pressione, come le altre versioni) per le particelle vicine, una volta esaurite le particelle in shared memory, la riempio con le prossime 6144 particelle (dato da $\frac{49152}{sizeof(float)\cdot 2}$ diviso 2 perché immagazzino sia la x che la y di ogni particella), ripetendo per tutto il numero di particelle, gestendo le rimanenze. Inoltre bisogna azzerare la shared memory prima di effettuare la reduction sul calcolo della velocità media, cosa che viene fatta all'interno della funzione stessa. Lo svantaggio di questa soluzione è che rende più difficile da leggere il codice.

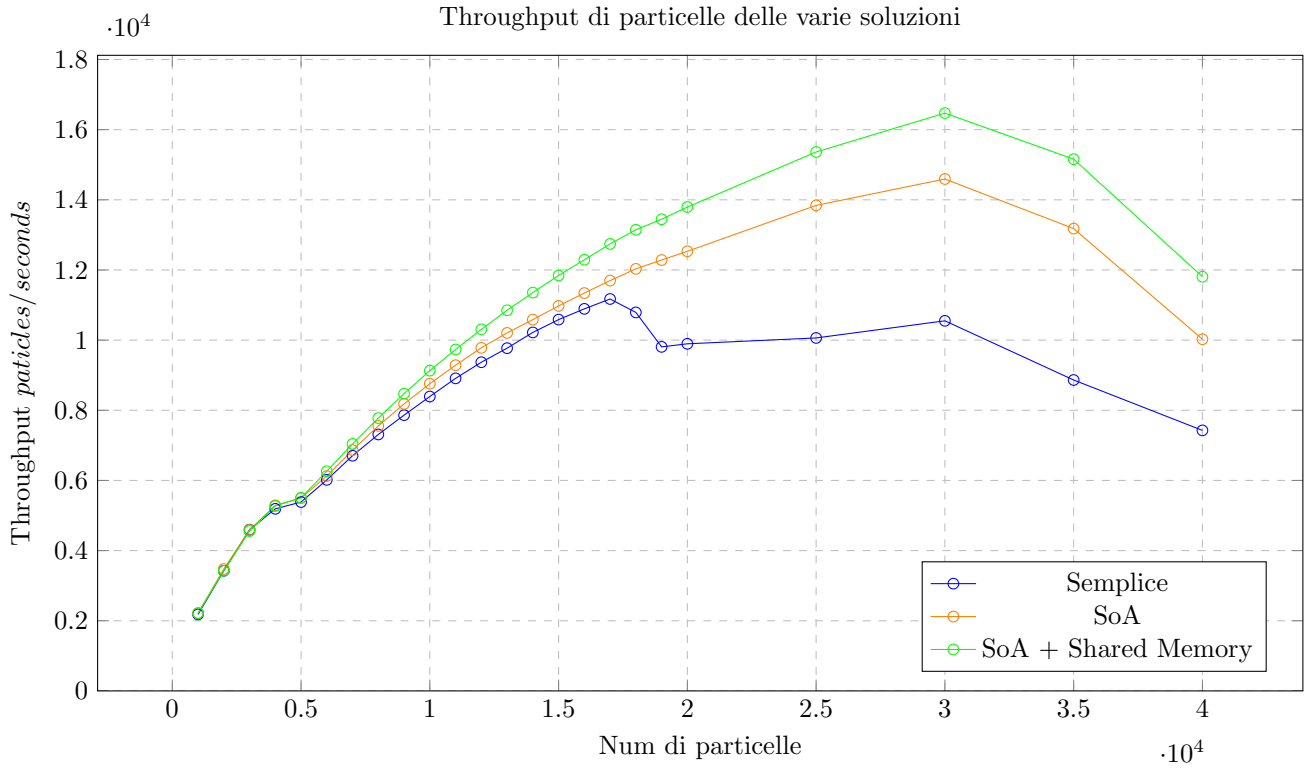

Figure 3: In questo grafico viene mostrato il Throughput delle varie soluzioni adoperate (con numero di passi costante a 100), si può notare come tutte scalino bene poi ad un certo punto il Throughput, cala drasticamente. La soluzione semplice dopo 17000 particelle inizia a scendere mentre per vedere il punto di discesa delle altre due soluzioni ho dovuto aumentare il limite di particelle a 40000 e si può notare che dopo le 30000 iniziano a calare entrabe le soluzioni rimanenti. Come possiamo notare la versione che usa la shared memory è più veloce.

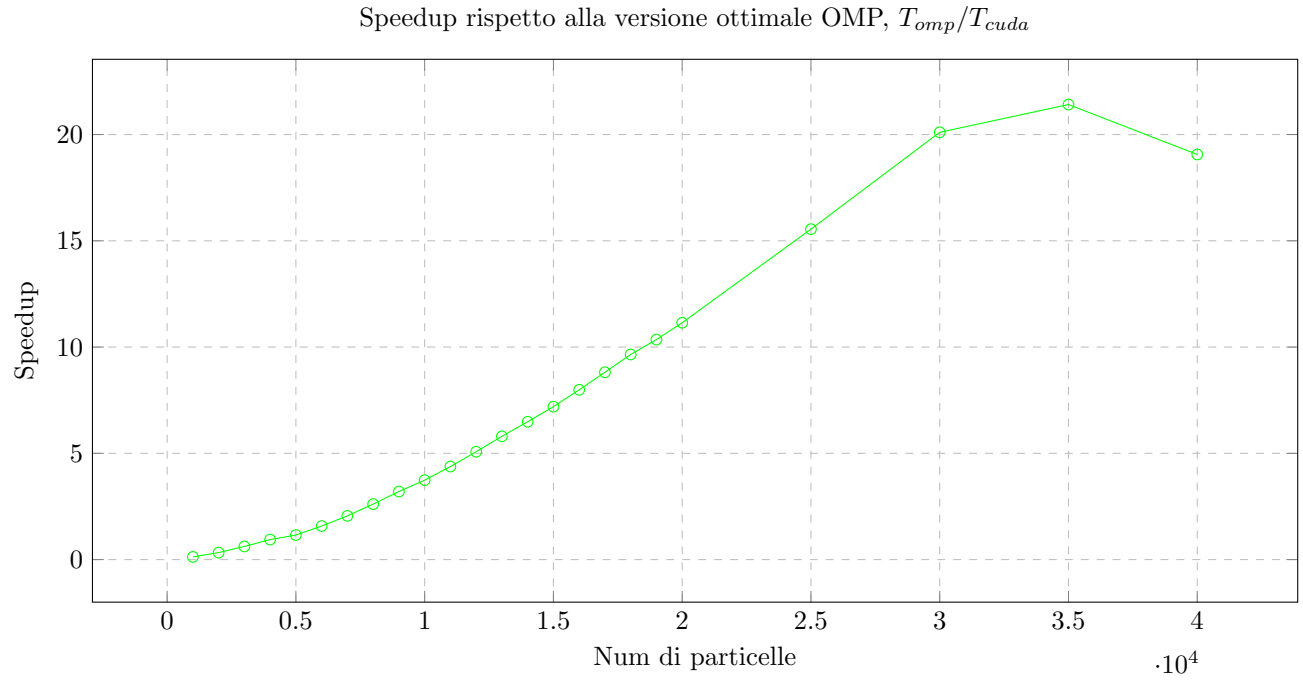

Figure 4: In questo grafico ho mostrato come varia lo speedup all'aumentare del numero di particelle (con numero di passi costante a 100), confrontando la versione OpenMP meglio ottimizzata (3.3), con la versione migliore di CUDA (4.3), anche qui abbiamo un picco, lo speedup reggiunge 21.4 per 35000 particelle, poi inizia a scendere.

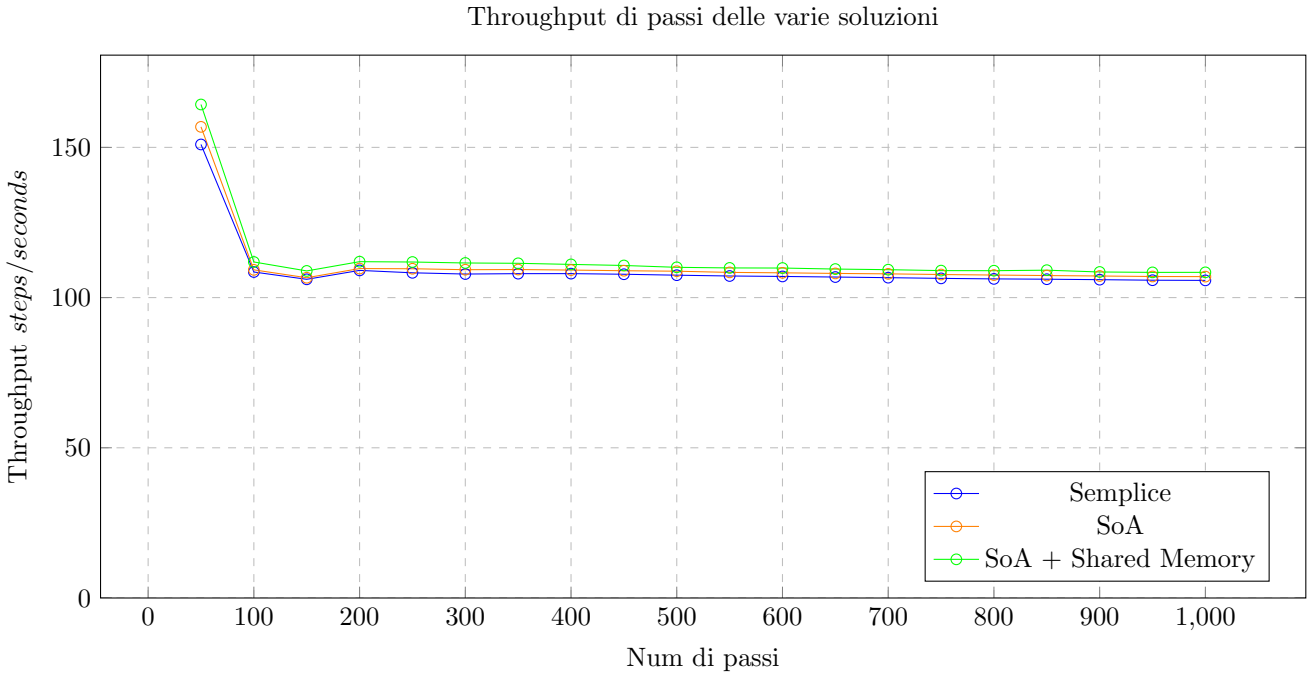

Figure 5: In questo grafico viene mostrato il Throughput delle varie soluzioni adoperate all'aumento del numero di passi, (e le particelle costanti a 5000) si può notare come tutte abbiano il Throughput circa uguale, è alto ai 50 passi poi cala subito drasticamente e continua a scendere leggermente avvicinandosi ad una costante.

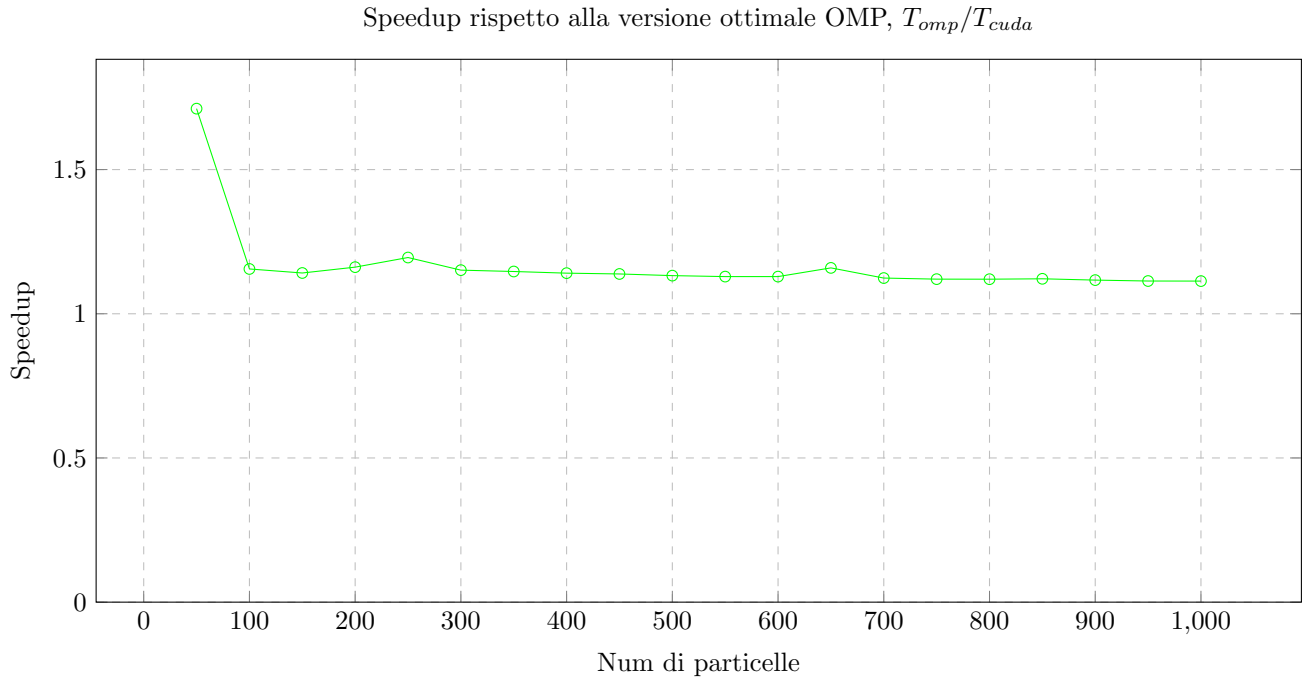

Figure 6: In questo grafico ho mostrato come varia lo speedup all'aumentare del numero di passi, (e le particelle costanti a 5000) confrontando la verione OpenMP meglio ottimizzata (3.3), con la verione migliore di CUDA (4.3), anche qui abbiamo un picco, a 50 passi inizia a scendere e continua a scendere leggermente avvicinandosi ad una costante. Come in figura 5

5 Conclusione

In conclusione ho deciso di consegnare le soluzioni più veloci sia per OpenMP che per CUDA; quindi la soluzione più veloce OpenMP è quella discussa nella sezione (3.3), mentre per CUDA è quella discussa nella sezione (4.3). Come possiamo notare la verione CUDA è decisamente più veloce di quella OpenMP all'aumento delle particelle, come si può notare in figura 4). Mentre nel caso in cui teniamo costante il numero di particelle e incrementiamo il numero di passi tra CUDA e OpenMP non vi sono grandi incrementi dello speedup, come visto in figura 6.

5.1 Sviluppi futuri

Mancano ancora altre migliorie che si possono fare in futuro, ad esempio nella versione CUDA si può usare l'addizione atomica al posto di una reducation per calcolare la velocità media, inoltre si potrebbe ottimizzare il codice per far si che possa avviarsi su più GPU dello stesso host, complicando ulteriormente il codice, poiché ad ogni fine delle computazioni di densità e forze, bisogna passare tali valori a tutte le schede video presenti.

Infine si potrebbe ottimizzare la ricerca delle particelle vicine tramite strutture dati spaziali come i Quad-Three che porterebbe a costo logaritmico la ricerca delle particelle vicine, anche se per far uso di questo albero si complica di molto il codice, maggiormente su CUDA.