Progetto di High Performance Computing 2023/2024

Dario Bekic, matricola 0001021740

Introduzione

Il progetto consiste nella parallelizzazione di un programma che, per un numero finito di iterazioni, calcola la posizione di un insieme di cerchi C la cui posizione e proprietà iniziali sono casualmente generate.
La posizione del cerchio c_j alla conclusione dell'iterazione i è influenzata dalla posizione degli altri $\tilde{\mathrm{C}} = \mathrm{C} \setminus \{c_j\}$ cerchi conclusa l'iterazione $i - 1$.
Da qui in avanti si fa uso dei termini "ciclo esterno" e "ciclo interno", questi fanno riferimento ai due cicli annidati della funzione *compute_forces* del codice seriale. Quando si parla di "spostamento di un cerchio X" si intende il valore dei campi *dx* e *dy* di una variabile X di tipo *circle_t* che indicano dove il cerchio verrà spostato alla fine dell'iterazione.

Due osservazioni:

1. Non si può parallelizzare banalmente il ciclo che chiama la funzione *compute_forces*, perché siamo di fronte ad una *loop-carried dependency* di tipo Read-After-Write.

2. Per ogni coppia di cerchi c_i, c_j $\forall i, j \in \{1, ...n\}$ $i \neq j$, la computazione, all'iterazione i, della nuova posizione di c_i, non è influenzata dalla computazione della nuova posizione di c_j(e viceversa). Questo è il fatto principale che è stato utilizzato nelle implementazioni delle versioni parallele.

Versione OpenMP

Due versioni

Per OpenMP si sono sviluppate due versioni, che si distinguono essenzialmente per come viene suddiviso il carico di lavoro nella funzione *compute_forces*.

Parallelizzazione interna Nel file *omp-circles-2.c* si è scelto di parallelizzare soltanto il ciclo interno, facendo cooperare i thread per il calcolo dello spostamento del cerchio indicizzato dal ciclo esterno. Questa soluzione si comporta bene quando il numero di cerchi inizia a crescere, poiché la singola iterazione del ciclo interno diventa computazionalmente pesante e riusciamo a fare cooperare tutti i thread a disposizione. Uno svantaggio di questa soluzione è l'overhead dovuto alla creazione e disfacimento dell'area parallela ad ogni iterazione del ciclo esterno(oppure, se si decidesse di creare prima il pool, dovuto alla sincronizzazione) che è proporzionale al numero di cerchi.

Parallelizzazione esterna Nel file *omp-circles.c* si è adottato un approccio work-inefficient ma che evita gli svantaggi della precedente soluzione.
Ad ogni thread vengono assegnati, grazie alla clausola **for**, $\lfloor \frac{n}{p} \rfloor \pm 1$ cerchi dove n è il numero di cerchi totali e p è il numero di thread.
Ogni thread esegue il ciclo interno per ogni cerchio che gli è stato assegnato e, per evitare conflitti, modifica solamente gli spostamenti dei cerchi che gli sono stati assegnati.
La soluzione è work-inefficient: una singola iterazione del ciclo interno del codice seriale si ripete 2 volte per ogni coppia di cerchi.
Sebbene la soluzione sia work-inefficient, la quantità di lavoro ripetuto è costante in relazione all'input e la computazione dello spostamento tra due cerchi non è particolarmente pesante, quindi una sua ripetizione non influisce sulla complessità generale dell'algoritmo.

Analisi delle prestazioni

Per l'analisi delle prestazioni sono stati adoperati i due script *strong-scaling.sh* e *weak-scaling.sh*.
Tutti i test sono stati effettuati sulla macchina di laboratorio, isi-raptor03.csr.unibo.it, che dispone di 12 core fisici.
Tutti i risultati temporali sono stati ottenuti come media del risultato di 5 esecuzioni consecutive dello stesso test.

Per calcolare lo speed-up e la strong scaling efficiency si è deciso un input di 10000 cerchi con 10 iterazioni.
La figura 1 mostra lo speed-up della soluzione comparato allo speed-up ideale.
La figura 2 mostra la strong scaling efficiency.
Sia nel calcolo della strong scaling efficiency, che in quello dello speed-up, notiamo che quando portiamo a 10, 11 e 12 il numero di thread/processi utilizzati in parallelo abbiamo un decremento considerevole dello speed-up/SSE; si attribuisce la causa di questo alla condivisione delle risorse computazionali della macchina che porta 1/2 processori ad essere quasi sempre utilizzati, almeno in parte.

Per calcolare la Weak scaling efficiency è stato scelta una base di 4096 operazioni ottenendo quindi: $n(p) = \sqrt{p} \cdot 4096$ come funzione del numero di cerchi da generare in funzione del numero di thread p per mantenere il carico di lavoro dei processori costante.

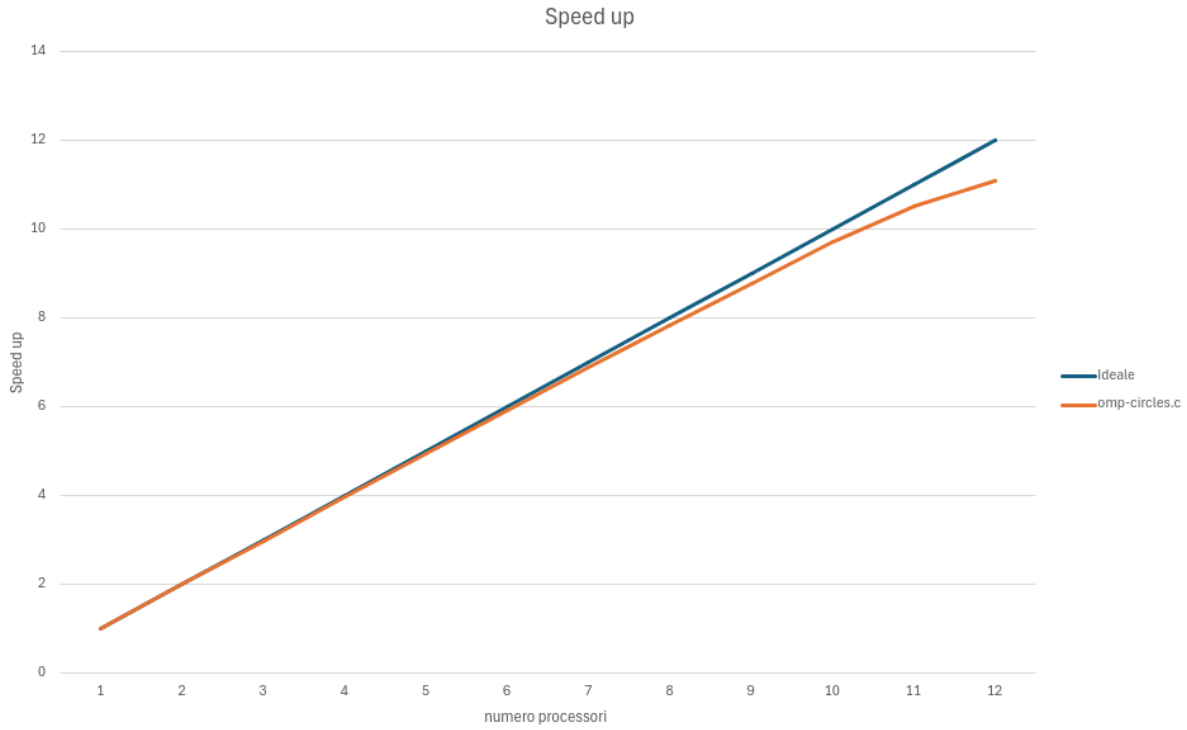

Figura 1: Speed-up con N. cerchi=10K, 10 Iterazioni

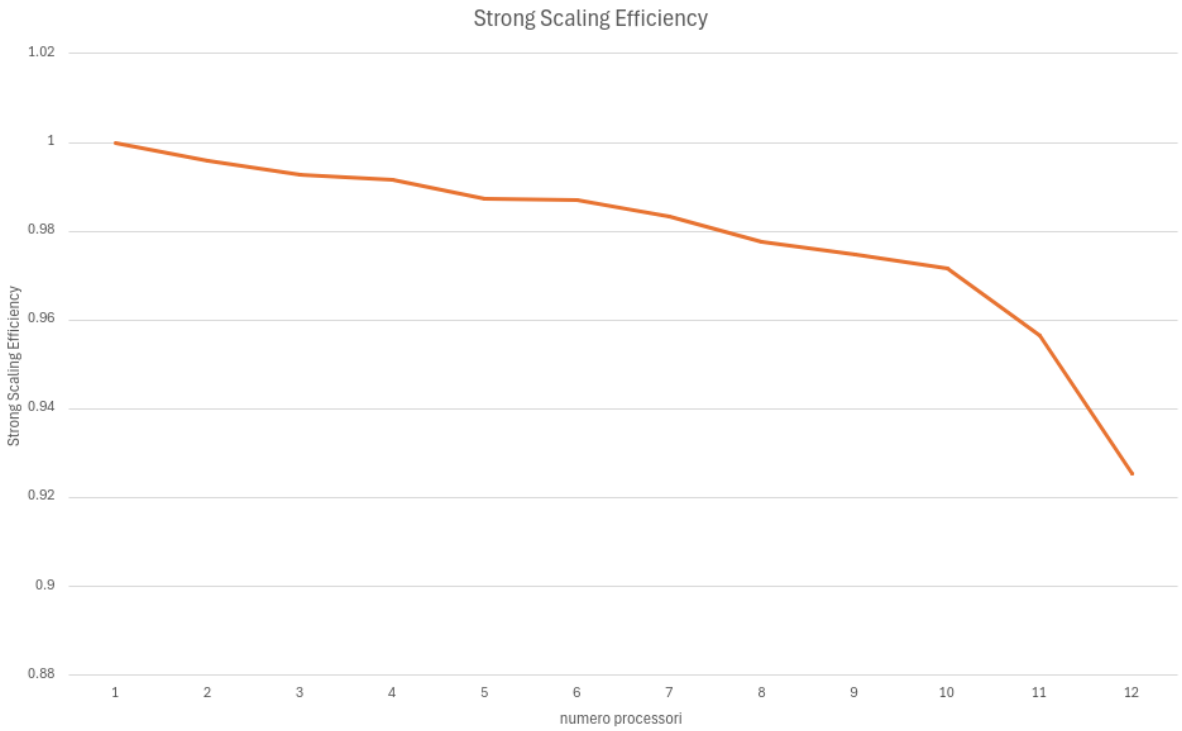

Figura 2: Strong Scaling Efficiency

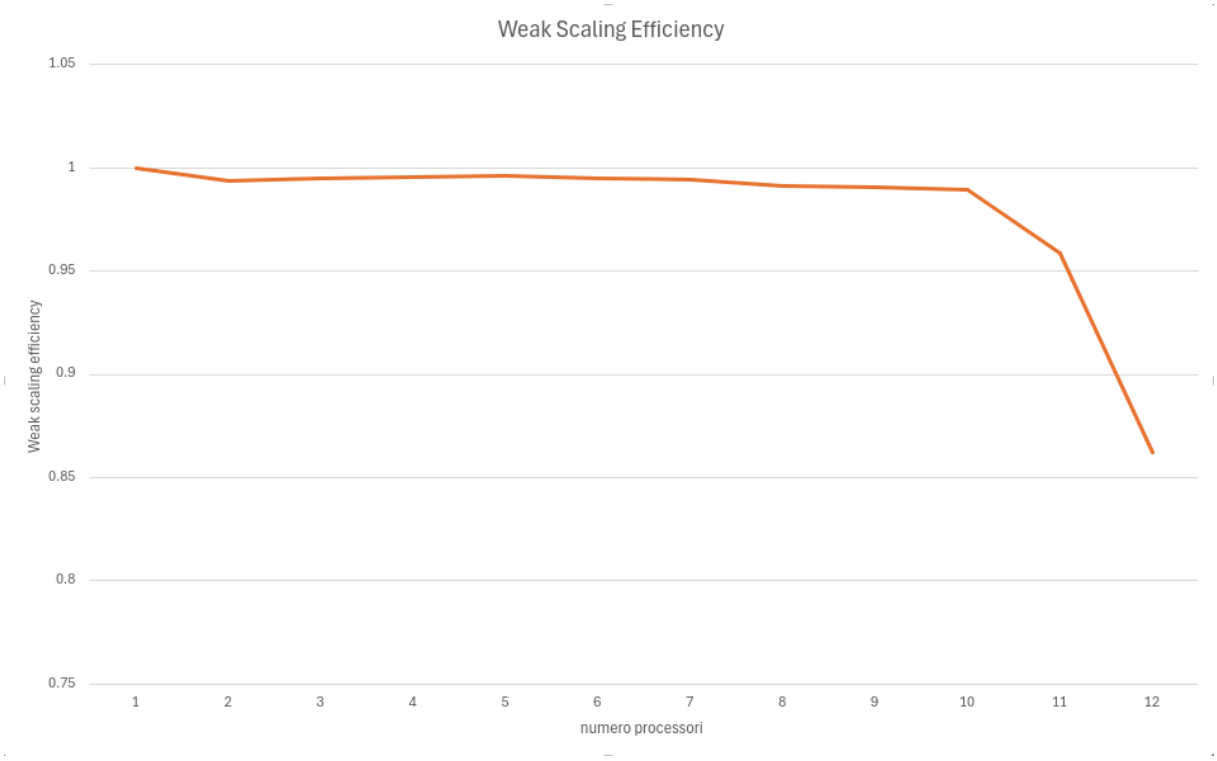

Figura 3: Weak scaling efficiency

Versione CUDA

Per la versione CUDA si è deciso di sfruttare il parallelismo massiccio affidando ad ogni CUDA thread il compito di calcolare lo spostamento di un singolo cerchio e di accumulare il numero di interazioni che ha rilevato nel vettore *intersections*. Ogni thread quindi compierà $O(n)$ operazioni, dove n è il numero di cerchi.

Un problema di questa soluzione è l'accesso alla global memory da parte di ogni thread per leggere gli stessi dati dal vettore *circles*.
Una possibile soluzione sarebbe l'uso di shared memory ma la dimensione di quest'ultima è limitata nella maggior parte delle architetture ($\leq$ 1 MB) e la dimensione di *circles* è stimabile a $n \times 16$ byte che per $n = 100.000$ è ≈ 1.5 MB.

Un'altra possibile soluzione è quella di mantenere il ciclo esterno nel codice seriale e richiamare un nuovo kernel che esegue un'iterazione del ciclo interno del codice seriale. Su questo approccio si sono sviluppate due versioni *cuda-circles2.cu* e *cuda-circles3.cu*.
Le due versioni differiscono nel modo in cui viene aggiornato lo spostamento del cerchio indicizzato dal ciclo esterno.
L'approccio adottato in *cuda-circles2.cu* consiste nell'utilizzo della primitiva atomica *atomicAdd*; un'alternativa è quella di eseguire sempre lo stesso numero di CUDA thread e rendere la soluzione work-inefficient rimuovendo però l'uso della primitiva e il costo di gestione che porta.

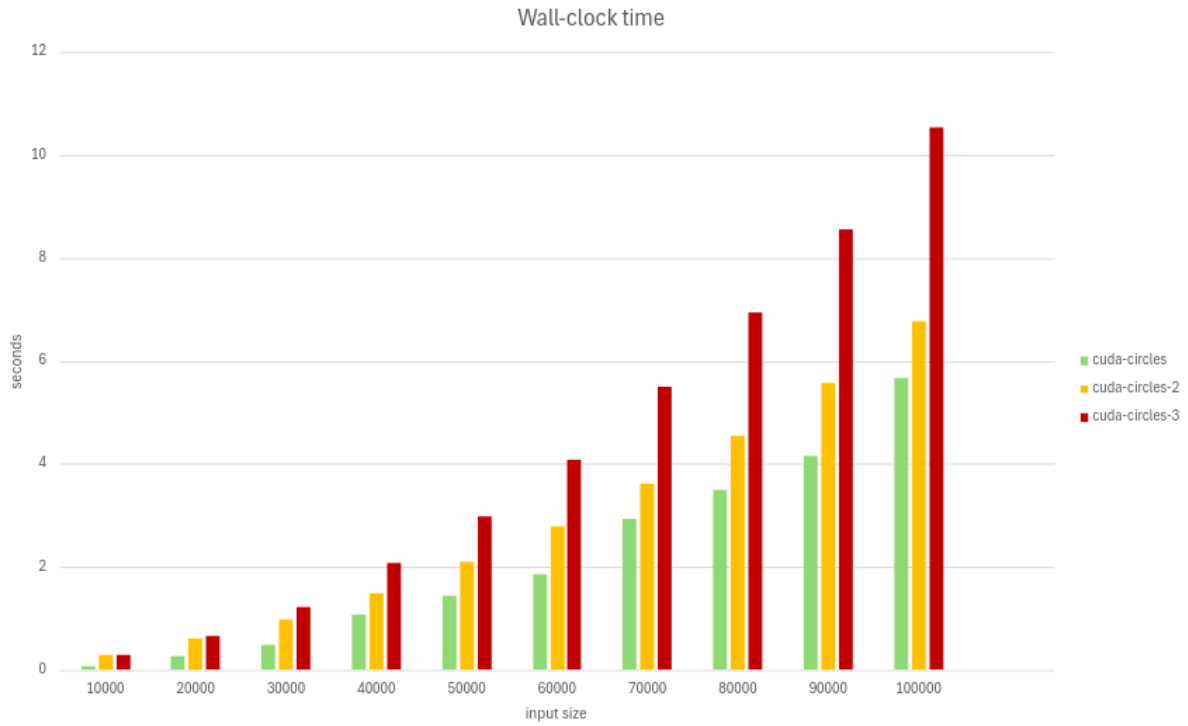

Figura 4: Comparazione Wall-clock time soluzioni CUDA

Analisi delle prestazioni

cuda-circles-2.cu performa meglio di *cuda-circles-3.cu* mentre si comporta similmente a *cuda-circles.cu*: figura 4 mostra una comparazione del wall-clock time delle soluzioni su un numero crescente di cerchi iniziali.
Comparando *cuda-circles-2.cu* e *cuda-circles-3.cu* notiamo che l'uso della primitiva *atomicAdd* non sia particolarmente pesante rispetto all'uso di molti più thread per chiamata del kernel.
cuda-circles.cu si comporta meglio rispetto perché non incorriamo nell'overhead dovuto alle n chiamate chiamate del kernel nella singola iterazione del ciclo esterno che troviamo nelle altre due soluzioni.
Figura 5 mostra lo speed-up che la soluzione *cuda-circles.cu* esibisce rispetto ad *omp-circles.c* e, come ci aspettavamo, lo speed-up risulta superlineare per il diverso tipo di parallelismo e hardware che le soluzioni adottano. Figura 6 mostra la comparazione tra *omp-circles.c* e *cuda-circles.cu* in termini di milioni di operazioni al secondo(mops/sec).

Conclusioni

Come ci si aspettava il maggiore parallelismo offerto dall'architettura CUDA offre prestazioni superiori rispetto alla soluzione sviluppata utilizzando la libreria OpenMP.
Un interessante indagine futura sarebbe quella di approfondire quale sia la distribuzione ottimale delle computazioni per CUDA thread: nella soluzione *cuda-circles.cu* si è creata una corrispondenza $thread \leftrightarrow circle$, un'altra possibile distribuzione sarebbe $thread \leftrightarrow (c_i, c_j)$ dove $(c_i, c_j) \in \mathrm{C} \times \mathrm{C}$ nella quale ogni CUDA thread modifica lo sposta-

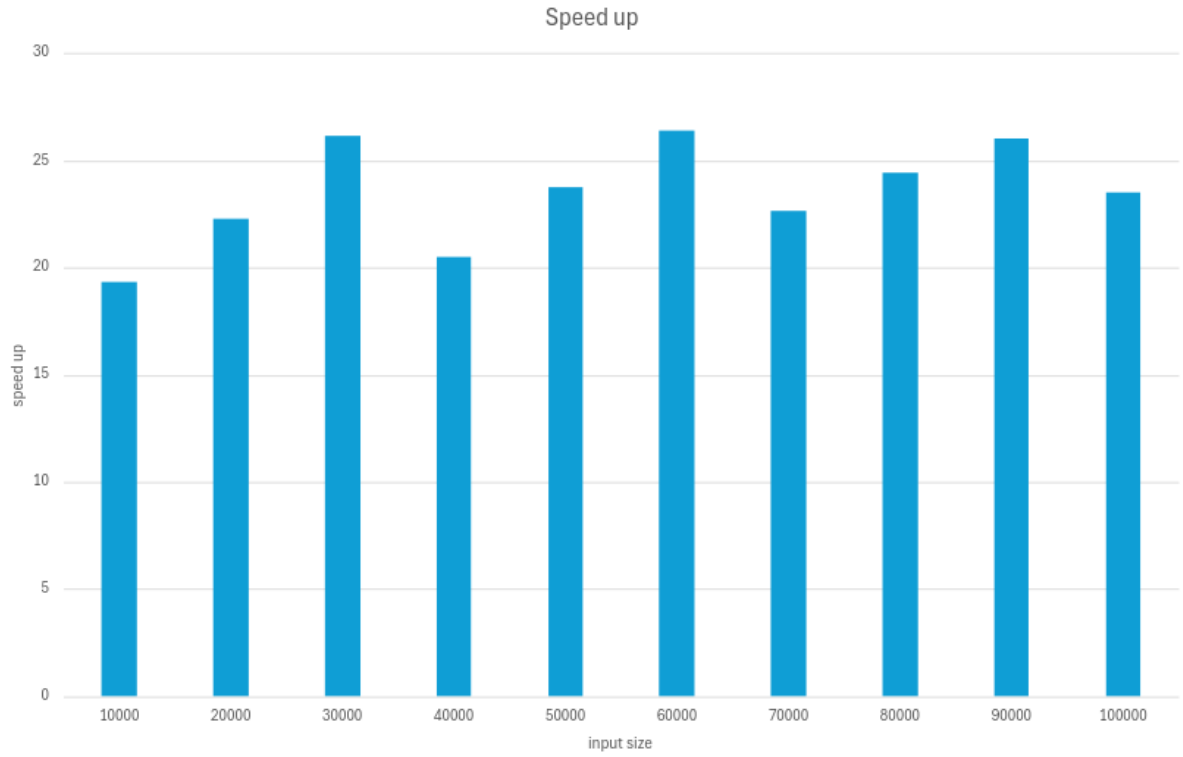

Figura 5: Speed up di cuda-circles.cu rispetto a omp-circles.c

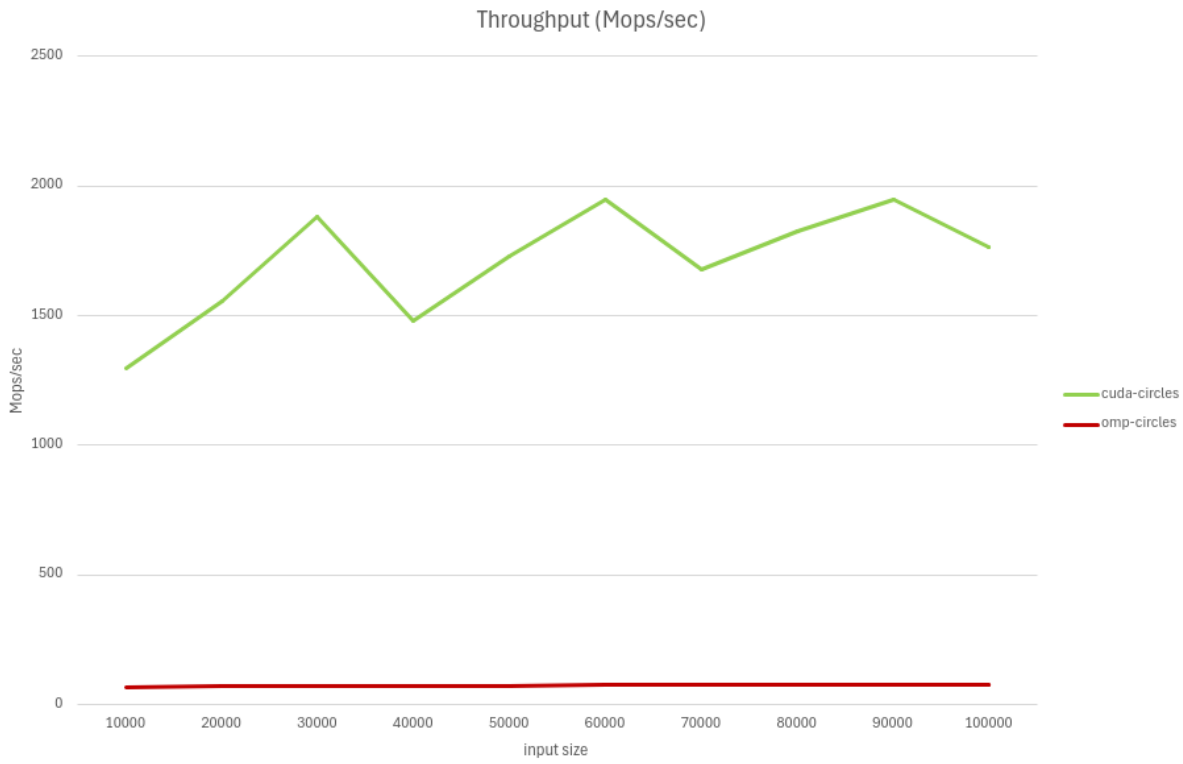

Figura 6: Throughput

mento di una sola coppia di cerchi.