Report Algoritmo A-Star

Confronto tra algoritmo sequenziale e parallelo

Sommario

Obiettivo Principale

Il progetto ha come obiettivo principale l'implementazione e il confronto delle prestazioni dell'algoritmo A* (A-Star) per il calcolo del percorso minimo in un grafo bidimensionale, sia sul lato host (CPU) che sul lato device (GPU) utilizzando CUDA. L'intento è di:

1. **Confrontare le Prestazioni**: Valutare il miglioramento delle prestazioni ottenuto attraverso la parallelizzazione dell'algoritmo A* sulla GPU rispetto all'implementazione sequenziale sulla CPU.
2. **Analizzare le Differenze Implementative**: Esplorare le sfide e le soluzioni nell'adattare l'algoritmo A* per l'esecuzione parallela sulla GPU, inclusa la gestione delle strutture dati e l'ottimizzazione delle operazioni per l'ambiente CUDA.
3. **Dimostrare la Correttezza dell'Algoritmo Parallelo:** Assicurarsi che l'implementazione su GPU fornisca risultati corretti e consistenti con l'algoritmo sequenziale, garantendo che i percorsi minimi trovati siano equivalenti.
4. **Valutare lo Speedup e la Scalabilità**: Misurare lo speedup ottenuto con l'implementazione parallela su diversi tipi di grafo e dimensioni, analizzando come il tempo di esecuzione varia al variare della dimensione del problema.
5. **Applicare Diversi Tipi di Euristiche**: Implementare e confrontare diverse funzioni euristiche all'interno dell'algoritmo A*, valutando l'impatto sulle prestazioni sia in ambiente sequenziale che parallelo.

L'algoritmo A* è ampiamente utilizzato in problemi di pathfinding e navigazione, come nella robotica, nei videogiochi e nei sistemi di navigazione GPS. Tuttavia, su grafi di grandi dimensioni o in applicazioni che richiedono calcoli in tempo reale, l'esecuzione sequenziale dell'algoritmo può risultare inefficiente.

La parallelizzazione dell'algoritmo A* su GPU offre la possibilità di sfruttare la potenza di calcolo parallela, riducendo significativamente i tempi di esecuzione e permettendo di affrontare problemi più complessi o di ottenere risposte in tempi più rapidi.

Rappresentazione del Grafo sul Lato Host (CPU)

Nodo:

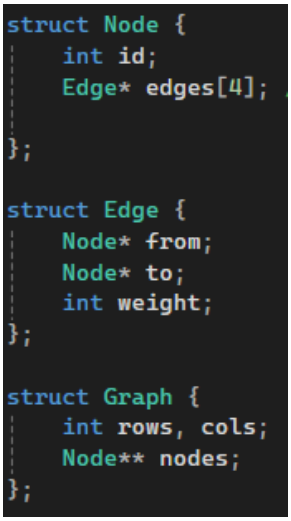
- **id**: Identificatore unico del nodo, calcolato come id = row * cols + col.
- **edges[4]**: Array di puntatori agli archi uscenti dal nodo, corrispondenti alle possibili direzioni (sopra, sotto, sinistra, destra). Se non esiste un arco in una certa direzione, il puntatore è NULL.

Arco:

- **from e to**: Puntatori ai nodi di partenza e di arrivo dell'arco.
- **weight**: Peso associato all'arco, generato casualmente tra 1 e 10.

Grafo:

- **rows e cols**: Dimensioni della griglia (numero di righe e colonne).
- **nodes**: Puntatore a un array bidimensionale di nodi, organizzato come nodes[rows][cols].



Creazione del grafo:

Tramite la funzione ***createGraph*** viene allocata memoria per una griglia bidimensionale di nodi pari a ***rows * cols***. Ogni nodo viene poi inizializzato con un ***id*** univoco. Successivamente tramite la funzione ***addEdges*** si itera su ogni nodo della griglia e per ogni una delle quattro direzioni si decide se aggiungere un arco verso un nodo adiacente. Viene utilizzata una funzione probabilistica per determinare la presenza o meno degli archi. Quando un nuovo arco viene aggiunto la funzione ***addEdge*** crea un nuovo arco assegnandogli un peso che varia in maniera casuale da 1 a 10.

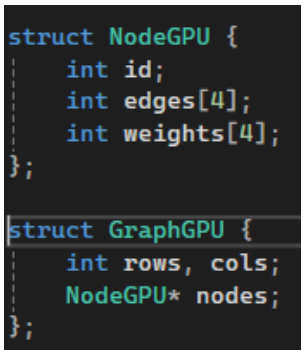
Rappresentazione del Grafo sul Lato Device (GPU)

Nodo e arco:

- **id**: Identificatore unico del nodo, come nella rappresentazione host.
- **edges[4]**: Array di interi che rappresenta gli ID dei nodi adiacenti nelle quattro direzioni. Se non esiste un arco in una certa direzione, il valore è -1.
- **weights[4]**: Array di interi che contiene i pesi degli archi corrispondenti in edges

Grafo:

- **rows e cols**: Dimensioni della griglia, come nella rappresentazione host.
- **nodes**: Puntatore a un array lineare di NodeGPU, di dimensione rows * cols.



Creazione del grafo:

La funzione ***copyGraphToGPU*** si occupa di copiare e trasformare la rappresentazione del grafo host a quella device. Tale operazione viene effettuata, allocando memoria sul device e convertendo i nodi dalla struttura host a quella device.

Struttura dati AS_Data per algoritmo A-Star (GPU):

La struttura AS Data ricopre un ruolo fondamentale per il calcolo del percorso minimo tramite algoritmo A-Star in ambiente cuda. Tale struttura ha il compito di tener traccia delle informazioni legate ai nodi da visitare e visitati. Inoltre, permette di salvare le informazioni legate al ***gCost*** e ***fCost*** per il calcolo del percorso minimo.

OpenList

- **heap:** array contenente i nodi di una determinata openList ordinati in base al proprio fCost
- **hashingTable:** array contenente la lista dei nodi attualmente presenti nel heap. Usata per verificare la presenza di un nodo nel heap in maniera veloce tramite una funzione di hashing.
- **fCost:** array dei costi di ogni Nodo
- **size:** numero di elementi nella openList

ClosedList:

- **idNode:** id del nodo già visitato.

AS_Data

- **gCost:** array di tutti i gCost dei nodi.
- **fCost:** array di tutti gli fCost dei nodi.
- **Predecessor**: array dei nodi per calcolare il percorso minimo.
- **openList_queues:** array di OpenList, una per ogni thread.
- **ClosedList:** array dei nodi già visitati.

```
struct OpenList {
    int* heap;
    int* hashingTable;
    int* fCost;
    int size;
};

struct ClosedList {
    int idNode;
};

struct AS_Data {
    int* gCost;
    int* fCost;
    int* predecessor;
    OpenList* openList_queues;
    ClosedList* closedList;
};
```

Implementazione e funzionamento algoritmo A-Star

Funzionamento Algoritmo lato Host (CPU):

L'algoritmo inizializza i costi associati a ciascun nodo. Il costo dal nodo iniziale al nodo stesso, denotato come ***g(n)***, è impostato a zero, mentre per tutti gli altri nodi è inizialmente considerato infinito. La funzione di valutazione ***f(n),*** che combina il costo accumulato ***g(n)*** e una stima del costo rimanente ***h(n)*** (funzione euristica), è calcolata per il nodo iniziale.

Per gestire i nodi da esplorare, viene utilizzata una open list implementata come una coda di priorità basata su un min-heap. Questa struttura permette di estrarre rapidamente il nodo con il costo ***f(n)***minimo. All'inizio, il nodo iniziale viene inserito nella open list.

Durante l'esecuzione, l'algoritmo entra in un ciclo in cui estrae il nodo con il valore ***f(n)*** più basso dalla open list. Se questo nodo è il nodo obiettivo, l'algoritmo termina e il percorso minimo viene ricostruito risalendo l'array dei predecessori. Altrimenti, il nodo viene aggiunto alla closed list per evitare di riesaminarlo.

Per ciascun nodo adiacente al nodo corrente, l'algoritmo calcola un costo provvisorio ***tentativeGScore*** sommando il costo ***g(n)*** del nodo corrente al costo dell'arco che conduce al nodo adiacente. Se questo costo provvisorio è inferiore al costo ***g(n)*** precedentemente registrato per il nodo adiacente, vengono aggiornati il costo ***g(n)***, il valore ***f(n)*** e il predecessore del nodo adiacente. Se il nodo adiacente non è già presente nella open list, viene inserito in essa.

Questo processo continua iterativamente, espandendo i nodi in ordine di priorità basata sul valore f***(n)***, fino a quando il nodo obiettivo non viene raggiunto o la open list diventa vuota, indicando che non esiste un percorso possibile.

Funzionamento Algoritmo lato Device (GPU):

L'implementazione dell'algoritmo A* sul lato device presenta sfide significative a causa della natura parallela delle GPU Per superare questa sfida, l'algoritmo è stato riprogettato per sfruttare il parallelismo offerto dalla GPU, adattando le strutture dati e le modalità di accesso alla memoria.

Invece di utilizzare una singola open list, come avviene sul lato host, l'algoritmo su GPU impiega multiple open list, una per ciascun thread. Questo approccio riduce i conflitti di accesso e la necessità di sincronizzazione tra i thread, consentendo una gestione più efficiente delle operazioni in parallelo.

All'inizio dell'algoritmo, vengono inizializzati gli array globali per i costi ***g(n)*** e ***f(n)***, assegnando a tutti i nodi un costo infinito, tranne che per il nodo iniziale, il cui costo ***g(n)*** è zero. Il nodo iniziale viene inserito nella open list del primo thread.

L'algoritmo esegue un kernel principale, in cui ogni thread opera sulla propria open list. Ogni thread estrae il nodo con il valore ***f(n)*** minimo dalla sua open list. Se il nodo estratto corrisponde al nodo obiettivo, il thread imposta un flag globale per indicare che il percorso è stato trovato e l'algoritmo termina.

Per ciascun nodo adiacente al nodo corrente, il thread verifica se il nodo è già presente nella closed list. Se non lo è, calcola un costo provvisorio ***tentativeGCost*** e utilizza operazioni atomiche, come ***atomicMin***, per aggiornare in modo sicuro il costo ***g(n)*** globale del nodo adiacente. Se il costo viene aggiornato, vengono anche aggiornati il valore ***f(n)*** e il predecessore del nodo adiacente. Il nodo adiacente viene quindi inserito nella open list del thread, utilizzando una **tabella di hashing e** l'operazione atomica ***atomicCAS*** per evitare duplicazioni.

Le operazioni atomiche sono essenziali per garantire la coerenza dei dati quando più thread potrebbero tentare di aggiornare gli stessi valori simultaneamente. Questo meccanismo evita race condition e garantisce che il costo minimo venga sempre registrato per ciascun nodo.

L'uso di multiple open list e tabelle di hashing consente di distribuire il carico di lavoro tra i thread e di ridurre la necessità di sincronizzazione.

Una volta che il nodo obiettivo è stato raggiunto, il percorso minimo può essere ricostruito risalendo l'array dei predecessori, come avviene nella versione host.

Implementazione Algoritmo A-Star (GPU)

Utilizzo di code parallele:

Un aspetto essenziale dell'implementazione su GPU dell'algoritmo A* è la scelta del numero di code parallele (***nQueues***), che viene determinato dividendo il numero totale di nodi per un fattore ***k***. Il fattore ***k*** rappresenta il numero di elementi che ogni coda può contenere. Selezionare il valore ottimale di ***k*** è necessario per bilanciare il carico di lavoro tra i thread della GPU e massimizzare le prestazioni.

Impatto del Fattore k sulle Prestazioni

Quando k è Grande (Poche Code, Molti Nodi per Coda):

- **Vantaggi:**

  Ridotto Overhead di Sincronizzazione: Con un minor numero di code, c'è meno necessità di sincronizzazione tra i thread, poiché vengono accedute contemporaneamente meno strutture dati.

  Gestione Semplificata: Gestire un numero minore di code può semplificare il codice e ridurre l'overhead associato alla gestione delle code.

- **Svantaggi:**

  Sottoutilizzo del Parallelismo: Avere meno code significa che un numero minore di thread è attivamente impegnato nell'elaborazione dei nodi, il che può portare a un sottoutilizzo delle capacità di elaborazione parallela della GPU.

  Aumento del Carico Computazionale per Thread: Ogni coda contiene più nodi, aumentando il carico computazionale sui thread responsabili di quelle code.

Quando k è Piccolo (Molte Code, Pochi Nodi per Coda):

- **Vantaggi:**

  Massimizzazione del Parallelismo: Aumentando il numero di code, più thread possono partecipare al calcolo, migliorando l'utilizzo delle risorse della GPU.

  Riduzione del Carico per Thread: Con meno nodi per coda, ogni thread ha meno dati da elaborare, il che può accelerare le computazioni individuali.

- **Svantaggi:**

  Aumento dell'Overhead di Sincronizzazione: Più thread che accedono a risorse condivise possono portare a maggiore contesa e richiedere più sincronizzazione, introducendo latenza.

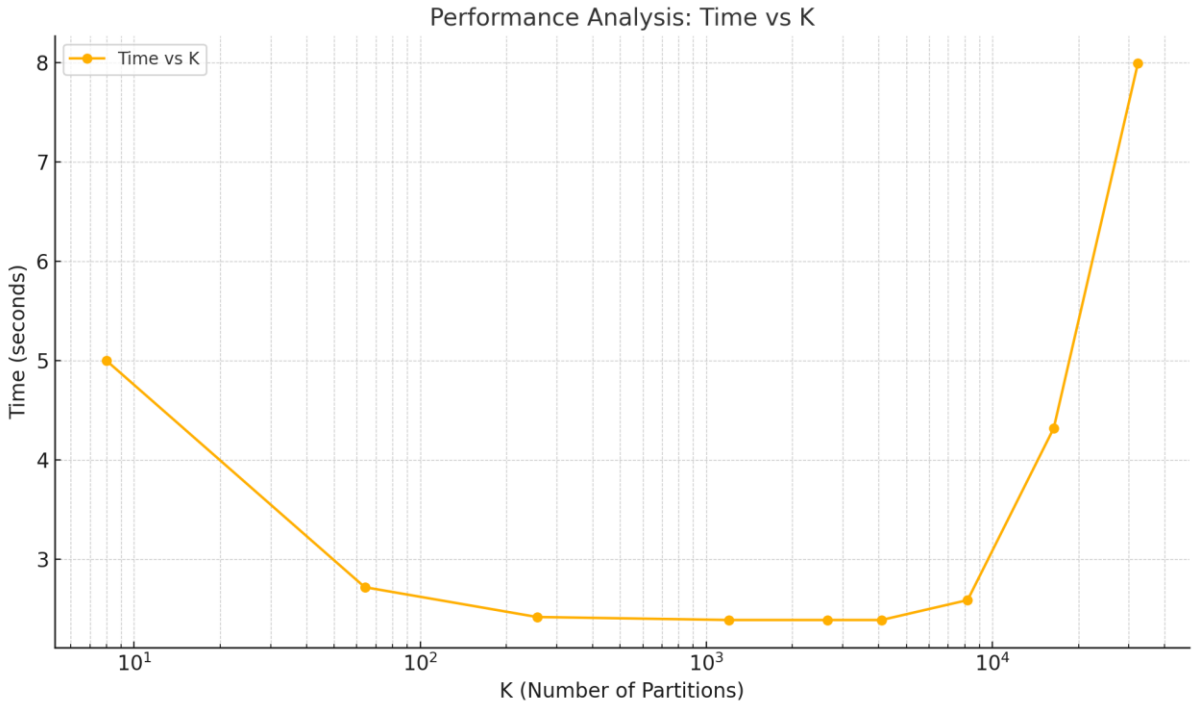
  Rallentamenti dati dalle Operazioni Atomiche: Un numero elevato di thread che eseguono operazioni atomiche su strutture dati condivise può rallentare le prestazioni a causa dell'aumentata contesa.

Bilanciamento del Carico di Lavoro

La scelta del valore ottimale di k comporta il bilanciamento tra la massimizzazione del parallelismo e la minimizzazione dell'overhead di sincronizzazione. L'obiettivo è trovare un valore che distribuisca uniformemente il carico di lavoro tra i thread della GPU, mantenendo i costi di sincronizzazione gestibili.

Analisi benchmark per bilanciamento carico di lavoro su una griglia 5000x5000 con punto finale [4999,4999]

K	Grid size	Parallel queues	Time
8	5000x5000	3125000	5 secondi
64	5000x5000	390625	2.72 secondi
256	5000x5000	97656	2.42 secondi
1200	5000x5000	20833	2.39 secondi
2656	5000x5000	9412	2.39 secondi
4096	5000x5000	3051	2.39 secondi
8192	5000x5000	3051	2.59 secondi
16384	5000x5000	1525	4.32 secondi
32250	5000x5000	775	7.99 secondi



Accesso alle code parallele:

Dato che ogni thread è associato a una specifica coda parallela basata sul suo identificatore (indice del thread). Ciò consente di mappare i thread alle code in modo deterministico, evitando conflitti tra thread che accedono alla stessa coda.

L'indice della coda a cui un thread deve accedere è calcolato utilizzando l'indice globale del thread (idx) e il numero totale di code parallele (***nQueues***):

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
int hashIndexIdx = idx % nQueues;
int currentNodeIdx = popFromOpenList(&data->openList_queues[hashIndexIdx]);
```

Verifica nodi esistenti

Nell'implementazione dell'algoritmo A* su GPU, è fondamentale evitare che lo stesso nodo venga inserito più volte nella coda di priorità (open list). La verifica della presenza di un nodo nella coda viene effettuata utilizzando una tabella di hashing (***hashing table***) associata a ciascuna coda parallela.

L'indice nella hashing table viene calcolato utilizzando una funzione di hashing semplice, tramite il modulo dell'ID del nodo con la dimensione della tabella.

```
int modHash = (totalNodes / nQueues) ;
int hashNodeIndex = neighborNodeIdx % modHash;
```

Si utilizza poi l'operazione atomica atomicCAS (Compare And Swap) per verificare e inserire il nodo.

```
int hashNodeIndex = neighborNodeIdx % modHash;
if (atomicCAS(&data->openList_queues[neighborHashIndex].hashingTable[hashNodeIndex], -1, neighborNodeIdx) == -1) {
    pushToOpenList(&data->openList_queues[neighborHashIndex], neighborNodeIdx, newFcost);
```

Approccio dinamico nella scelta del numero di blocchi

Nell'implementazione dell'algoritmo A* su GPU, la scelta del numero di thread per blocco e del numero di blocchi per grid è un fattore cruciale che influenza significativamente le prestazioni dell'algoritmo. Questa scelta deve essere adattabile e dinamica, variando in base alla dimensione del grafo, ossia al numero totale di nodi. Un approccio dinamico garantisce un utilizzo efficiente delle risorse della GPU, indipendentemente dalle dimensioni del problema da risolvere

```
int threadPBlock = 256;
int nBlocks = (nQueues + threadPBlock - 1) / threadPBlock;

dim3 threadsPerBlock(threadPBlock);
dim3 numBlocks((nQueues + threadsPerBlock.x - 1) / threadsPerBlock.x);
```

L'algoritmo si adatta automaticamente a grafi di qualsiasi dimensione senza necessità di modificare manualmente i parametri di lancio dei kernel. Così facendo si garantisce che un numero adeguato di thread sia lanciato per sfruttare al massimo le capacità della GPU, evitando sia la saturazione che il sottoutilizzo

Test Case

Hardware:

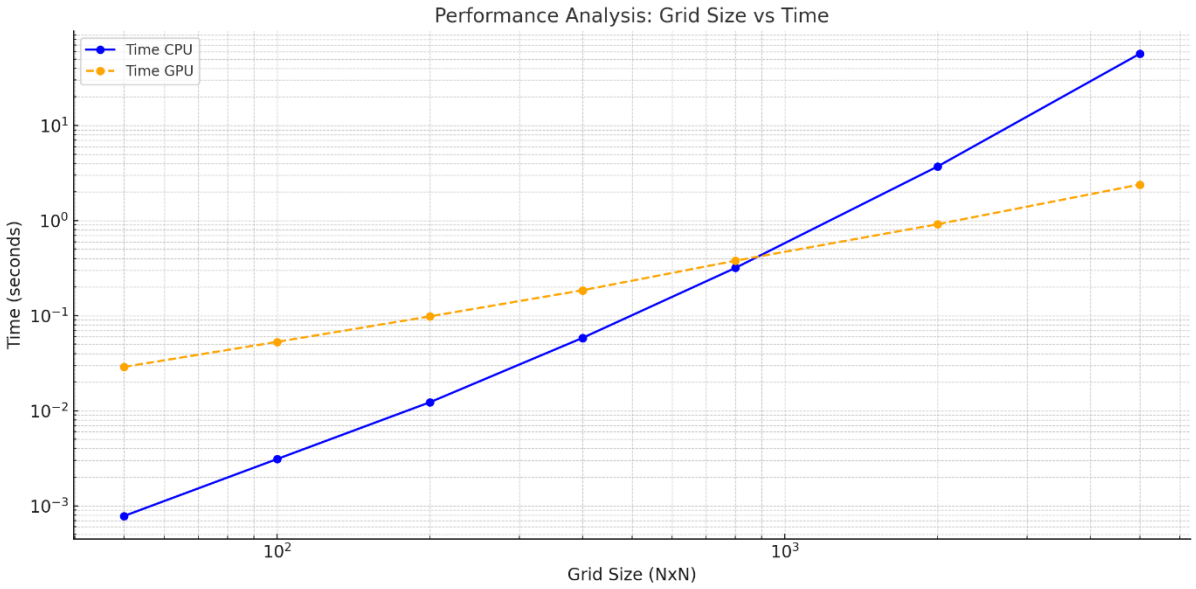
Graphics Card: Nvidia RTX 3070

- **Architettura GPU:** NVIDIA Ampere
- **Compute Capability: 8.6**
- **Numero di CUDA Cores: 5.888**
- **Numero di Streaming Multiprocessors (SMs): 46**
- **Tipo di Memoria:** 8 GB GDDR6

Test per numero di nodi crescenti

Seed : 1732652188 - Edge 100%

K	Start	Goal	Grid	Time **CPU**	**Time GPU**	**SpeedUp**
4	[0,0]	[49,49]	50x50	0.0007821	0.0289075	0.025
8	[0,0]	[99,99]	100x100	0.0030995	0.0529844	0.06
16	[0,0]	[199,199]	200x200	0.0122685	0.0981872	0.12
30	[0,0]	[399,399]	400x400	0.058428	0.184969	0.27
54	[0,0]	[799,799]	800x800	0.318093	0.378232	0.83
256	[0,0]	[1999,1999]	2000x2000	3.70401	0.915175	4.06
1200	[0,0]	[4999,4999]	5000x5000	56.6827	2.391204	23.68

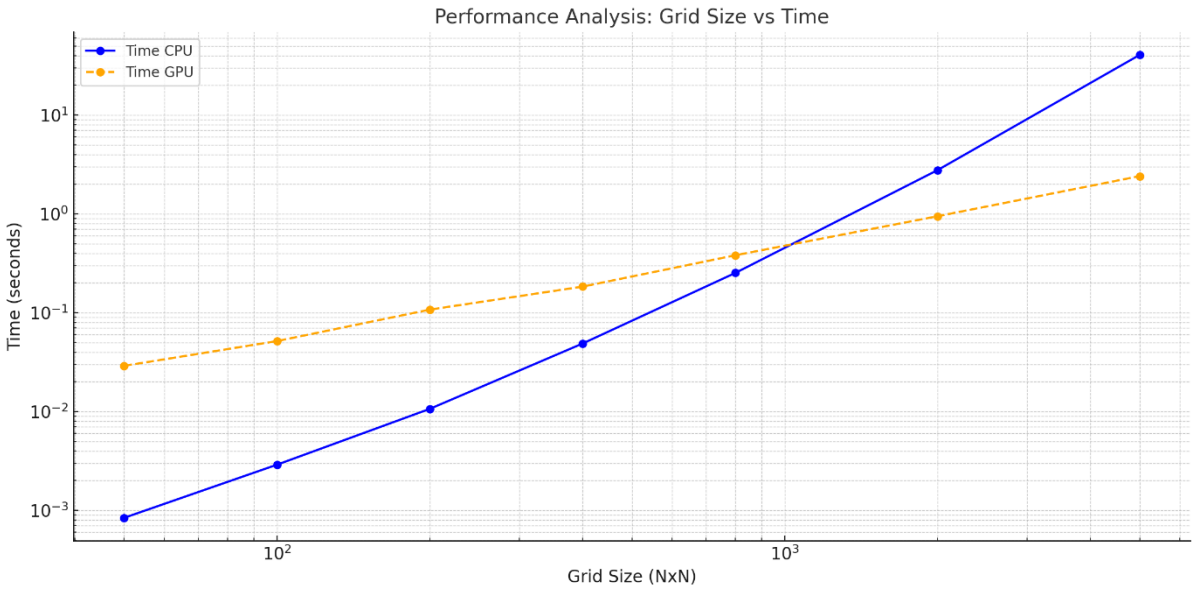


Osservazione:

L'algoritmo A-Star parallelo rispetto al sequenziale tende ad essere più veloce all'aumentare del numero di nodi coinvolti nel calcolo del percorso minimo. Si nota, quindi , il decremento significativo nei tempi di esecuzione dell'algoritmo parallelo rispetto al sequenziale, che nel caso di una griglia pari a 5000x5000 risulta essere 23.68 volte più veloce di quello sequenziale.

Seed : 1732652188 - Edge 70%

K	Start	Goal	Grid	Time **CPU**	**Time GPU**	**SpeedUp**
4	[0,0]	[49,49]	50x50	0.0008414	0.0290196	0.027
8	[0,0]	[99,99]	100x100	0.0029005	0.0516374	0.056
16	[0,0]	[199,199]	200x200	0.010653	0.107484	0.1
30	[0,0]	[399,399]	400x400	0.0488351	0.183712	0.26
54	[0,0]	[799,799]	800x800	0.253115	0.381493	0.66
256	[0,0]	[1999,1999]	2000x2000	2.77191	0.946775	2.94
1200	[0,0]	[4999,4999]	5000x5000	40.5561	2.4074	16.89



Osservazione:

Si osserva che al decrementare del numero di connessioni e quindi anche al numero di nodi che probabilmente vengono visitati l'algoritmo sequenziale guadagna dei punti nel calcolo del percorso minimo. Si nota, inoltre, che se pur c'è un leggero miglioramento dell'algoritmo sequenziale rispetto al parallelo, quest'ultimo tende ad essere sempre più veloce con uno speedup nel caso di una griglia 5000x5000 pari a 16.89 volte quello sequenziale.

Test per le diverse euristiche

Seed : 1732652188 - Edge 70%

Heuristic	Grid	Time
Manhattan	5000x5000	2.44
Euclidean	5000x5000	2.40
Chebyshev	5000x5000	2.39
Octile	5000x5000	2.44
Hamming	5000x5000	2.41

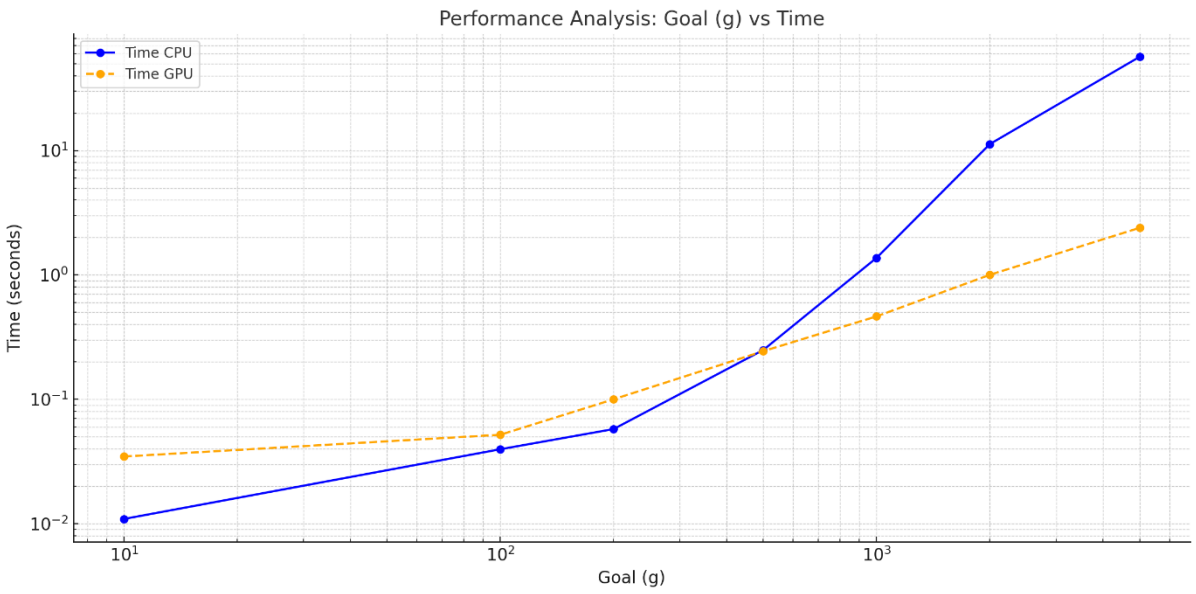
Osservazione:

Non si nota nessuna differenza sostanziale nell'utilizzo delle varie funzioni euristiche in ambito parallelo.

Test per distanze crescenti

Seed : 1732652188 - Edge 100%

Grid	Start	End	Time CPU	Time GPU	SpeedUp
5000x5000	0,0	10,10	0.0109216	0.0346913	0.29
5000x5000	0,0	100,100	0.0396831	0.0518886	0.76
5000x5000	0,0	200,200	0.0576574	0.10007	0.5
5000x5000	0,0	500,500	0.249433	0.243945	1
5000x5000	0,0	1000,1000	1.37319	0.464296	2.97
5000x5000	0,0	2000,2000	11.277	1.00284	11.2
5000x5000	0,0	4999,4999	56.6827	2.391204	23.68



Osservazione:

Si nota che all'aumentare della distanza dal nodo iniziale al nodo di arrivo, il tempo di esecuzione dell'algoritmo parallelo rispetto a quello sequenziale tende a decrementare. Pertanto, se ne deduce che il numero di nodi da analizzare ricopre un ruolo fondamentale nel calcolo parallelo rispetto al sequenziale.