Report GPU

Davide Polidori

Dicembre 2024

1 Lo Shortest Path problem

Il problema dello *Shortest Path* consiste nel trovare il percorso più breve tra due nodi in un grafo pesato, in cui ciascun arco ha un costo associato.

1.1 Definizione del problema

Dato un grafo $G = (V, E)$, dove V è l'insieme dei nodi ed E è l'insieme degli archi tra i nodi, ogni arco (u, v) è pesato con peso $w(u, v)$. Il problema del cammino minimo consiste nel determinare il percorso con il costo totale minimo tra un nodo sorgente $s \in V$ e un nodo destinazione $d \in V$.

L'algoritmo proposto in questo progetto ha lo scopo di risolvere il problema del cammino minimo tra tutte le coppie di nodi del grafo. Si assume che il grafo sia non orientato, privo di parti isolate e che tutti i pesi sugli archi siano positivi.

2 L'algoritmo di Dijkstra

L'algoritmo di Dijkstra, introdotto da Edsger W. Dijkstra nel 1956, è uno degli algoritmi più noti per la risoluzione del problema dello Shortest Path su grafi pesati.

2.1 Funzionamento dell'algoritmo

L'algoritmo parte da un nodo sorgente s e ad ogni passo seleziona il nodo con la distanza minima non ancora visitato, aggiorna il costo per raggiungere i suoi vicini avendo aggiunto il nuovo nodo e ripete il processo fino a quando tutti i nodi sono stati esplorati. Proporrò tre implementazioni dell'algoritmo : una sequenziale e due parallele di cui una con l'utilizzo di SMEM.

2.2 Versione sequenziale

La versione sequenziale di Dijkstra viene eseguita su CPU. Poiché vogliamo risolvere il problema All Pairs Shortest Paths l'algoritmo viene eseguito per

ogni nodo del grafo. Di seguito, sono elencati gli elementi principali utilizzati durante l'esecuzione dell'algoritmo.

- **Matrice di Adiacenza**: rappresenta il grafo pesato, dove ogni elemento (i, j) della matrice indica il peso dell'arco che collega il nodo i al nodo j. Se non esiste un arco diretto tra due nodi, il valore corrispondente è impostato a *infinito*.
- **Insieme** V_t: questo insieme contiene i nodi per cui è già stato trovato il percorso minimo dalla sorgente. V_t viene utilizzato per determinare quando terminare l'algoritmo.
- **Array** *distanze*: contiene le distanze minime dal nodo sorgente agli altri nodi del grafo. L'array viene aggiornato man mano che l'algoritmo procede.

2.2.1 Struttura dell'algoritmo

L'algoritmo viene eseguito per ogni nodo del grafo come nodo sorgente, calcolando le distanze minime da quel nodo a tutti gli altri nodi. Il funzionamento è quello descritto:

- **Inizializzazione:**
  - Per ciascun nodo sorgente, viene creato l'array di distanze chiamato `distanza`, che contiene i valori delle distanze iniziali tra il nodo sorgente e tutti gli altri nodi, presi dalla matrice di adiacenza.
  - Se non esiste un collegamento diretto tra due nodi, la distanza è impostata a infinito.
  - L'array booleano `V_t` tiene traccia dei nodi che sono già stati visitati, iniziando con il nodo sorgente impostato come visitato.
- **Ciclo principale:**
  - L'algoritmo continua finché non ha visitato tutti i nodi.
  - Ogni iterazione cerca il nodo non ancora visitato che ha la distanza minima dalla sorgente.
  - Una volta trovato, questo nodo viene aggiunto all'insieme dei nodi visitati.
- **Aggiornamento delle distanze:**
  - Dopo aver selezionato un nodo, l'algoritmo controlla se esistono percorsi più brevi verso altri nodi passando per il nodo appena selezionato.
  - Se sì, l'algoritmo aggiorna le distanze nel relativo array `distanza`.
- **Conclusione:**
  - Alla fine di ogni esecuzione dell'algoritmo per un nodo sorgente, le distanze minime vengono salvate.

2.2.2 Complessità della versione sequenziale

L'algoritmo sequenziale esegue n volte l'algoritmo di base, che ha una complessità di $O(n^2)$:

- **Tempo per sorgente:** $O(n^2)$
- **Numero di esecuzioni:** n
- **Tempo totale:** $O(n^3)$

2.3 Versione Source Partitioned

La versione Source Partitioned è la prima versione parallela che utilizza la GPU, l'idea alla base è molto semplice : per ogni nodo sorgente abbiamo un'iterazione di Dijkstra, assegnamo quindi ad ognuno di essi un thread che si occupa di calcolare i risultati per quella iterazione. Questi ultimi vengono poi inseriti nella riga appropriata della matrice nella memoria globale. In questa versione la SMEM non viene utilizzata perché le variabili necessarie per il calcolo sono tutte presenti localmente all'interno del thread. Per via di quest'ultimo punto le capacità della GPU non vengono sfruttate appieno, per questo viene implemetata la versione Source Parallel.

2.3.1 Complessità della versione Source Partitioned

- **Tempo per sorgente:** $O(n^2)$
- **Numero di esecuzioni:** 1
- **Tempo totale:** $O(n^2)$

2.4 Versione Source Parallel

Questa variante dell'algoritmo è concepita per massimizzare l'efficienza dell'esecuzione parallela sulla GPU. Viene assegnato un blocco a ciascun nodo del grafo e un thread dedicato per ogni nodo all'interno di quel blocco. In pratica ad ogni blocco è asseganta dell'intero algoritmo di Dijkstra per un nodo sorgente. In questa versione viene utilizzata la SMEM.

I thread lavorano insieme per generare l'array finale contenente i risultati per il blocco.

L'algoritmo si sviluppa nei seguenti passaggi principali:

- **Inizializzazione:** Ogni blocco di thread rappresenta un nodo sorgente, e ogni thread all'interno del blocco rappresenta un nodo del grafo. Le distanze tra il nodo sorgente e gli altri nodi sono caricate nella SMEM per accelerare i calcoli successivi.

- **Ciclo principale:** I thread cooperano per trovare il nodo non ancora visitato con la distanza minima. Questa operazione è realizzata tramite una *riduzione parallela*, in cui i thread confrontano le distanze tra i nodi e trovano quello meno distante dal sorgente.
- **Aggiornamento delle distanze:** Dopo aver trovato il nodo con la distanza minima esso viene aggiunto al vettore dei nodi visitati e i thread aggiornano le distanze per i nodi non ancora visitati. Se viene trovato un percorso più breve attraverso il nodo appena identificato, la distanza viene aggiornata in parallelo.
- **Terminazione:** L'algoritmo per un blocco termina quando tutti i nodi sono visitati. Le distanze minime calcolate vengono salvate nella memoria globale per essere utilizzate o visualizzate.

2.4.1 Complessità versione Source Parallel

La versione migliorata aumenta la concorrenza utilizzando n processori per ogni istanza, quindi n^2 processori. Questo significa che ogni istanza viene eseguita da n processori, ma bisogna tenere conto del ritardo di comunicazione:

- **Tempo per sorgente:** $O\left(\frac{n^2}{p} + n\log(p)\right)$
- **Numero di processori:** n^2
- **Tempo totale:** $O\left(\frac{n^3}{n^2}\right) + O\left(n\log(p)\right) = O(n) + O\left(n\log(p)\right)$

3 Benchmarks

Ho eseguito dei test per l'algoritmo con diverse grandezze del grafo e comparato gli speedup. Questi test sono eseguiti su una CPU AMD Ryzen 5 3500u da 2.1 GHz e per la GPU la Tesla T4 fornita da Colab.

Numero di nodi	Source Partitioned Version	Source Parallel Version
128	0.84	11.23
256	0.96	12.56
512	1.93	24.93
1024	2.49	14.87

4 Profiling

Ho eseguito questo profiling con 1024 nodi

4.1 Profiling per SourcePartitioning

Table 1: Launch Statistics

Metric Name	Metric Unit	Metric Value
Block Size		1,024
Function Cache Configuration		CachePreferNone
Grid Size		1
Registers Per Thread	register/thread	30
Shared Memory Configuration Size	Kbyte	32.77
Driver Shared Memory Per Block	byte/block	0
Dynamic Shared Memory Per Block	byte/block	0
Static Shared Memory Per Block	byte/block	0
Threads	thread	1,024
Waves Per SM		0.03

Table 2: Occupancy

Metric Name	Metric Unit	Metric Value
Block Limit SM	block	16
Block Limit Registers	block	2
Block Limit Shared Mem	block	16
Block Limit Warps	block	1
Theoretical Active Warps per SM	warp	32
Theoretical Occupancy	%	100
Achieved Occupancy	%	98.73
Achieved Active Warps Per SM	warp	31.60

4.2 Profiling per SourceParallel

Table 3: Throughput

Metric Name	Metric Unit	Metric Value
DRAM Frequency	cycle/nsecond	5.00
SM Frequency	cycle/usecond	584.98
Elapsed Cycles	cycle	5,163,718,089
Memory Throughput	%	2.11
DRAM Throughput	%	0.23
Duration	second	8.83
L1/TEX Cache Throughput	%	84.60
L2 Cache Throughput	%	1.51
SM Active Cycles	cycle	129,086,362
Compute (SM) Throughput	%	0.19

Table 4: Launch Statistics

Metric Name	Metric Unit	Metric Value
Block Size		1,024
Function Cache Configuration		CachePreferNone
Grid Size		1,024
Registers Per Thread	register/thread	22
Shared Memory Configuration Size	Kbyte	32.77
Driver Shared Memory Per Block	byte/block	0
Dynamic Shared Memory Per Block	Kbyte/block	13.31
Static Shared Memory Per Block	byte/block	0
Threads	thread	1,048,576
Waves Per SM		25.60

Table 5: Occupancy

Metric Name	Metric Unit	Metric Value
Block Limit SM	block	16
Block Limit Registers	block	2
Block Limit Shared Mem	block	2
Block Limit Warps	block	1
Theoretical Active Warps per SM	warp	32
Theoretical Occupancy	%	100
Achieved Occupancy	%	100.00
Achieved Active Warps Per SM	warp	32.00

Table 6: Throughput

Metric Name	**Metric Unit**	**Metric Value**
DRAM Frequency	cycle/nsecond	5.00
SM Frequency	cycle/usecond	584.95
Elapsed Cycles	cycle	973,252,887
Memory Throughput	%	5.58
DRAM Throughput	%	0.01
Duration	second	1.66
L1/TEX Cache Throughput	%	11.11
L2 Cache Throughput	%	0.17
SM Active Cycles	cycle	957,296,636.25
Compute (SM) Throughput	%	6.70