Programmazione su Architetture Parallele

Calcolo delle Componenti Fortemente Connesse

Andrea Marcon - 152098

Anno Accademico 2024 - 2025

Indice

1 Introduzione

In questo progetto verranno comparate le implementazioni di due algoritmi per la ricerca delle Componenti Fortemente Connesse all'interno di grafi sparsi: una versione seriale basata sull'algoritmo di Tarjan e una parallelizzata su GPU utilizzando CUDA.
Con Componente Fortemente Connessa (Strong Connected Component - SCC) di un grafo orientato G, si intende un sottografo massimale G' in cui esiste un cammino orientato tra ogni coppia di nodi.
Il calcolo delle SCC è un'operazione cruciale in diverse applicazioni reali, tra cui l'analisi delle dipendenze nei software, l'ottimizzazione dei sistemi di trasporto e lo studio delle reti sociali. In questi contesti, la computazione efficiente delle SCC è essenziale, richiedendo lo sviluppo di algoritmi scalabili e performanti. L'architettura CUDA, con la sua elevata capacità di parallelizzazione e gestione di grandi volumi di dati, rappresenta quindi una soluzione ottimale per accelerare la computazione delle SCC.

2 Soluzione

2.1 Rappresentazione dei dati

I grafi vengono letti direttamente da file di testo e rappresentati in formato CSR (Compressed Sparse Row).
Gli archi orientati sono rappresentati nel file di testo come coppie di nodi, numeri interi indicizzati a partire da 1.
Il grafo letto dal file di testo viene poi convertito in formato CSR, una struttura dati utilizzata per rappresentare grafi sparsi in modo compatto e che ottimizza l'accesso a nodi adiacenti. La struttura dati CSR è stata rappresentata nel seguente modo:

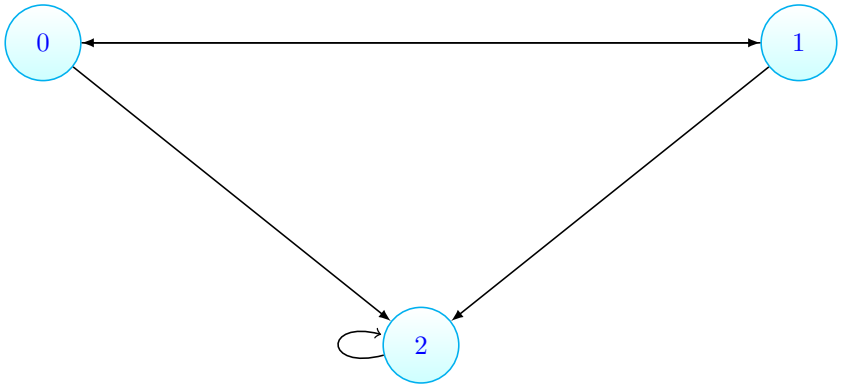
```
typedef struct {
    int *row_ptr;
    int *col_idx;
    int num_nodes;
    int num_edges;
} CSRGraph;
```

dove:

- **row_ptr**: è l'array che memorizza gli offset degli archi per ciascun nodo. L'elemento *row_ptr[i]*, indica l'indice del vettore *col_idx* dal quale iniziano gli archi in uscita dal nodo *i*.
- **col_idx**: è l'array che contiene i nodi di destinazione per ogni nodo. Gli archi uscenti dal nodo *i* sono memorizzati consecutivamente nell'array *col_idx* a partire dall'indice specificato da *row_ptr[i]*,

- **num_nodes**: è un intero che indica il numero di nodi nel grafo,
- **num_edges**: è un intero che indica il numero di archi nel grafo.

Il formato CSR utilizza solamente due array per rappresentare un grafo, richiedendo uno spazio di memoria $O(V + E)$, dove V e E rappresentano rispettivamente il numero di vertici e archi. Questa rappresentazione è particolarmente vantaggiosa per grafi sparsi, poiché evita il costo di memoria $O(V^2)$ della matrice di adiacenza, risultando molto più efficiente quando $E \ll V^2$. Inoltre la rappresentazione CSR consente una navigazione efficiente del grafo, riducendo l'overhead dovuto da strutture dati non contigue e permettendo l'accesso coalescente tra i thread ai vicini di un nodo.
Per esempio il seguente grafo:



può essere rappresentato con il seguente formato CSR:

```
Nodes array:
0 2 4
Edges array:
1 2 0 2 2
```

2.2 Algoritmo

I tipici algoritmi seriali per il rilevamento delle SCC, come l'algoritmo di Tarjan o quello di Kosaraju, si basano sulla ricerca depht-first, un processo intrinsecamente sequenziale a causa delle dipendenze sui dati elaborati in precedenza. È necessario quindi un altro tipo approccio per parallelizzare la ricerca delle SCC sull'architettura CUDA.
L'implementazione parallela proposta sfrutta una strategia di propagazione delle label, che si articola in più fasi eseguite iterativamente dalla GPU. Più in particolare viene sfruttato il principio secondo il quale, per ogni nodo appartenente a una SCC, l'insieme dei nodi raggiungibili tramite una propagazione in avanti e una all'indietro, trasponendo il grafo, coincidono. L'intersezione dei nodi raggiunti in entrambe le direzioni di conseguenza coincide esattamente alla SCC.
L'algoritmo implementato si suddivide nei seguenti step eseguiti in parallelo:

- **Forward propagation**:
  Ogni thread della GPU processa un singolo nodo del grafo. Se il nodo è attivo, ovvero che non è stato assegnato definitivamente a una SCC, il thread itera sui suoi vicini e, per ogni vicino attivo, tenta di aggiornare la propria etichetta *fwd_label* con il valore minimo trovato. L'aggiornamento è realizzato attraverso l'operazione atomica *atomicMin*, che garantisce l'integrità del valore in presenza di accessi concorrenti tra i thread. Se il valore della SCC di un nodo viene modificato, il cambiamento viene segnalato aggiornando la variabile condivisa *d_changed* mediante *atomicExch*, permettendo di continuare finché non si raggiunge la convergenza, ovvero che nessun valore in *fwd_label* viene aggiornato tra un'iterazione e l'altra.

- **Backward propagation**:
  Il processo è simile a quello di forward propagation, ma viene eseguito utilizzando il grafo trasposto, in modo da invertire la direzione degli archi e verificare la coerenza delle label tra nodi appartenenti alla stessa SCC. L'operazione di trasposizione viene effettuata lato host.
  Durante la propagazione, viene effettuato un controllo aggiuntivo: per ciascun nodo attivo, si verifica che la forward label del nodo corrente coincida con quella del vicino, in modo da propagare le label solamente all'interno di pontenziali SCC. Se la condizione è soddisfatta, si utilizza nuovamente l'operazione atomica *atomicMin* per aggiornare la label *bwd_label* e, in caso di variazione, il cambiamento viene segnalato tramite *atomicExch* sulla variabile condivisa *d_changed*.

- **Identificazione delle SCC**:
  Una volta che entrambe le propagazioni convergono, il kernel *intersetionKernel* esegue l'intersezione tra le forward e backward label di ogni nodo: se queste coincidono, il nodo viene riconosciuto come parte di una componente fortemente connessa, andando a segnare il nodo come "completato" e disattivandolo attraverso il flag *active[i]*, in modo da non essere considerato nelle successive iterazioni.

- **Reset delle etichette per i nodi attivi**:
  Prima di iniziare un nuovo ciclo di propagazione, le etichette dei nodi ancora attivi vengono reimpostate al loro indice originale, utilizzando il kernel *resetLabelsKernel*. Questa operazione viene utilizzata per rimuovere le SCC già identificate e per concentrarsi solamente sui nodi attivi rimanenti.

Questo processo ciclico, composto da forward propagation, backward propagation, intersezione e reset delle etichette, continua finché non restano più nodi attivi, ovvero quando tutti i nodi sono stati assegnati a una determinata SCC.

3 Risultati

3.1 Setup sperimentale

I test sono stati condotti su una macchina con le seguenti caratteristiche:

- **CPU**: Intel Xeon W-3323 (12 core, 3.50GHz)
- **CPU**: NVIDIA GeForce RTX 2080 Ti (11 GB GDDR6)
- **CUDA version**: 12.7
- **RAM**: 64 GB
- **OS**: Ubuntu 24.04.2 LTS

3.2 Tempi di esecuzione

Per valutare l'efficacia della parallelizzazione nel rilevamento delle componenti fortemente connesse, sono stati messi a confronto i tempi di esecuzione dell'algoritmo parallelo proposto con quelli di una classica implementazione sequenziale dell'algoritmo di Tarjan, al variare del numero di nodi del grafo di input. Più nello specifico, sono stati generati casualmente una serie di grafi, con un numero crescente di nodi e un numero di archi apari a venti volte il numero di nodi. Per ogni grafo sono stati eseguite dieci volte entrambe le implementazioni, registrando il tempo medio di esecuzione.
Nell'immagine 1 viene mostrato il grafico dei tempi di esecuzione dell'algoritmo sequenziale (linea blue) e parallelo (linea rossa) al variare delle dimensioni in nodi del grafo di input. In figura 2 viene mostrato lo stesso grafico, ma con un range di dimensioni dell'input più ristretto, evidenziando il comportamento degli algoritmi su grafi di piccole dimensioni.

Osservando i due grafici è possibile affermare che:

- La versione sequenziale ha un incremento pressoché lineare dei tempi di esecuzione all'aumentare del numero di nodi, raggiungendo valori significativi anche per grafi di medie dimensioni.
- La versione parallela, sfruttando il parallelismo massivo offerto dalla GPU, mantiene tempi di esecuzione molto più bassi, anche per grafi di grandi dimensioni, dove l'overhead iniziale di comunicazione e sincronizzazione diventa trascurabile rispetto ai benefici del parallelismo.
- Per grafi di piccole dimensioni, l'algoritmo sequenziale risulta più vantaggioso a causa degli alti costi di gestione (trasferimento dati, inizializzazione dei kernel, ecc.) richiesti dall'esecuzione su GPU

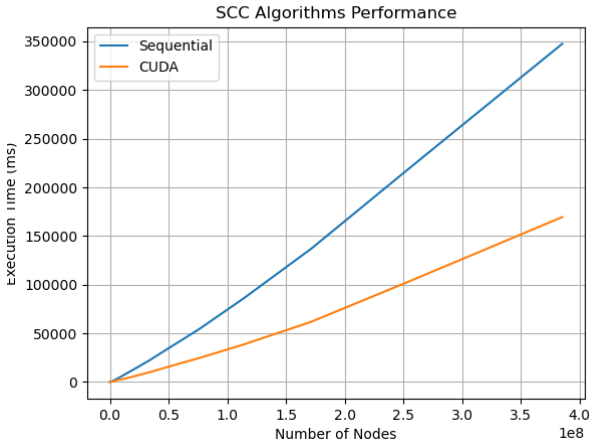


Figure 1: Prestazioni dell'algoritmo sequenziale (blu) e parallelo (rosso) in funzione della dimensione del grafo di input.

4 Compilazione

Per facilitare la compilazione del codice sorgente è stato fornito un makefile. Per compilare tutti i file necessari all'esecuzione utilizzare il comando *make*. Una volta compilato il codice sorgente, è possibile testare l'algoritmo proposto utilizzando il comando *make test*, che esegue sia la soluzione parallela sia quella sequenziale di riferimento. In questo caso, il grafo di input deve essere presente nella directory tramite il file di testo *graph.txt*, formattato come lista di archi. L'output mostrerà i risultati di entrambe le implementazioni in modo da permettere il confronto diretto.
Per riprodurre gli esperimenti descritti nella sezione 3, si può utilizzare il comando *make experiments*. I grafi di input vengono generati automaticamente in runtime e salvati nel file di testo *graph.txt*. I risultati degli esperimenti vengono salvati nel file *results.csv*.

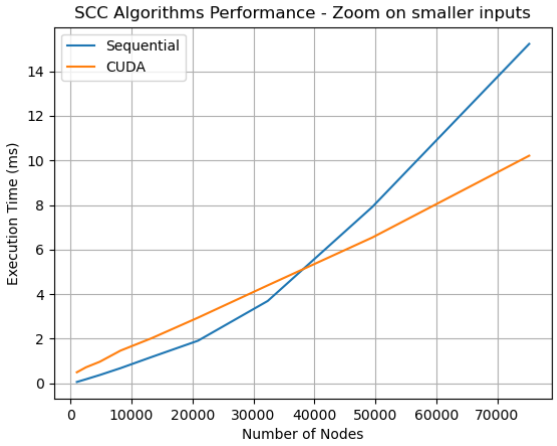


Figure 2: Prestazioni dell'algoritmo sequenziale (blu) e parallelo (rosso) con grafi di input di piccole dimensioni.

5 Conclusioni

In questa relazione è stata presentata l'implementazione di un algoritmo parallelo in CUDA per la ricerca delle componenti fortemente connesse in grafi sparsi. È stato posposto un approccio basato sulla propagazione di etichette tra i nodi del grafo, sia in avanti che all'indietro, dimostrando che la soluzione converge correttamente alle componenti fortemente connesse.
L'implementazione parallela è stato comparata con una classica versione sequenziale basata sull'algoritmo di Tarjan, mostrando come quest'ultima, pur essendo più performante per grafi di piccole dimensioni, subisca un incremento quasi lineare dei tempi di esecuzione al crescere del numero di nodi. Al contrario, la soluzione parallela riesce a mantenere tempi significativamente inferiori per input di grandi dimensioni, grazie al pieno sfruttamento del parallelismo hardware offerto dalla GPU.
I risultati sperimentali hanno dimostrato che, una volta superata la soglia di qualche decina di migliaia di nodi, l'overhead iniziale introdotto dall'algoritmo parallelo viene completamente compensato dall'alto throughput della GPU. L'andamento dei tempi di esecuzione suggerisce che il parallelismo hardware permette di scalare l'algoritmo molto più efficientemente rispetto all'approccio sequenziale, aspetto cruciale in scenari reali in cui i grafi presentano dimensioni rilevanti.