Informatica - Area scientifica
Dipartimento di Scienze matematiche, informatiche e multimediali
Università di Udine

Progetto di Architetture Parallele

Rasera Giovanni (143395)

Anno accademico 2024/2025

Indice

1 Descrizione del problema affrontato

Scrivere un programma che, dato un nodo x in V (diverso da s), determini un sottoinsieme D di nodi tale che:

- 1) la sorgente s non appartiene a D e il nodo x non appartiene a D
- 2) ogni cammino diretto da s a x passa per almeno un nodo in D
- 3) D è minimale rispetto alla cardinalità

Un algoritmo che risolve un problema analogo è l'algoritmo MinCutMaxFlow con la differenza che permette di trovare gli archi che rappresentano il massimo flusso che può attraversare un grafo.

1.1 MinCutMaxFlow in un contesto reale

Scenario bellico: Gestione strategica delle vie di comunicazione
In un contesto bellico, un comandante deve impedire al nemico di trasportare rifornimenti dalla loro base principale (sorgente s) a un obiettivo strategico (t) attraverso una rete stradale.
La rete è rappresentata come un grafo, dove:

- I nodi rappresentano le intersezioni stradali.
- Gli archi rappresentano le strade che collegano queste intersezioni, con capacità che indicano la quantità massima di rifornimenti che possono attraversare ciascuna strada.

Obiettivo:
Individuare il set minimo di strade (arco taglio) che, se bloccate o distrutte, interromperebbero efficacemente tutti i rifornimenti dal punto s al punto t.

2 Metodologia adottata per la soluzione

2.1 Sfruttare l'algoritmo MinCut per risolvere il problema richiesto

L'algoritmo MinCut risolve il problema di trovare gli archi con il costo minimo. Il problema presentato cerca di tagliare i nodi, una soluzione è quella di procedere come suggerito da: Professor Andrea Formisano
Un nodo v in G=(V, E) diventa v' in G'=(V', E'), composto da (v'even -> v'odd).

- Tutti gli archi originali in(v) che arrivavano a v sono ora collegati a v'even con costo |V'| * 2.
- Tutti gli archi originali out(v) che partivano da v partono ora da v'odd con costo infinito.
- Il costo da v'odd a v'even è pari a 1.

Esempio:

- G : 0 -a-> 1 -b-> 2 -c-> 3
- G' : (0 -1-> 1) -16-> (2 -1-> 3) -16-> (4 -1-> 5) -16-> (6 -1-> 7)

Il risultato del MinCut nel grafo G' darà come risultato il taglio di archi di costo 1 e indicheranno i nodi da eliminare con un semplice calcolo nodo_da_eliminare = (v'even / 2).

2.1.1 Condizione di taglio del nodo

Nell'algoritmo che verrà presentato, per verificare quali archi tagliare in G' (di conseguenza i nodi in G) bisogna controllare che per ogni arco (u -> v) siano verificate contemporaneamente le condizioni:

- excess_flow del nodo u abbia un eccesso di (V'*2)-1
- forward_flow di (u -> v) 0

- backward_flow di (u -> v) 1

Tali nodi tagliati comporranno l'insieme D del problema proposto.
IMPORTATE: Le strutture dati excess_flow, forward_flow e backward_flow derivano da come è stato risolto il problema, e le condizioni di taglio dipendono dal tipo di algoritmo e di struttura dati che si decide di utilizzare per risolvere il problema.

2.1.2 Esempio 1

L'insieme D è il nodo 2 e l'output dell'algoritmo di taglio dovrebbe essere:
Nodes D are:
(2) : 4 -/-> 5
|D|: 1

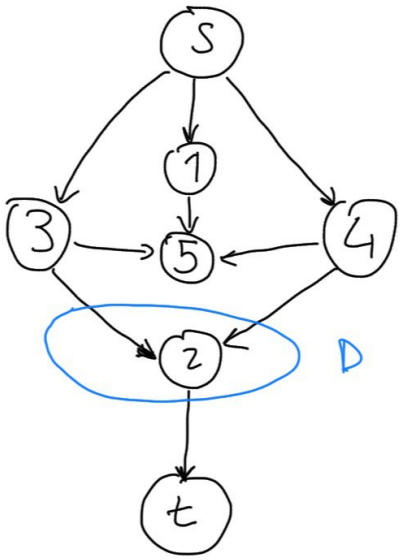

Figura 2.1: /graphs/esempio1.txt

2.1.3 Esempio 2

L'insieme D sono i nodi 1 2 3 4 5 l'output dell'algoritmo di taglio dovrebbe essere:
Nodes D are:
(1) : 2 -/-> 3
(2) : 4 -/-> 5
(3) : 6 -/-> 7
(4) : 8 -/-> 9
(5) : 10 -/-> 11
|D|: 5

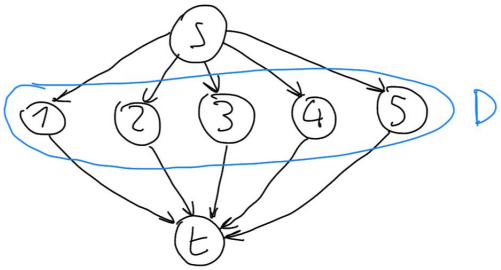

Figura 2.2: /graphs/esempio2.txt

2.1.4 Esempio 3

L'insieme D sono i nodi 1 2 l'output dell'algoritmo di taglio dovrebbe essere:
Nodes D are:
(1) : 2 -/-> 3
(2) : 4 -/-> 5
|D|: 2

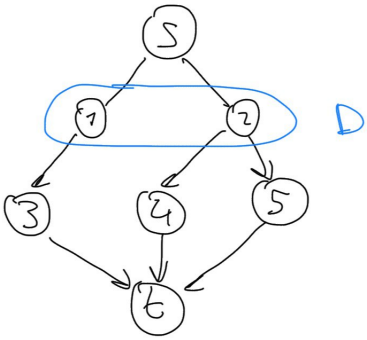

Figura 2.3: /graphs/esempio3.txt

2.1.5 Esempio Reale

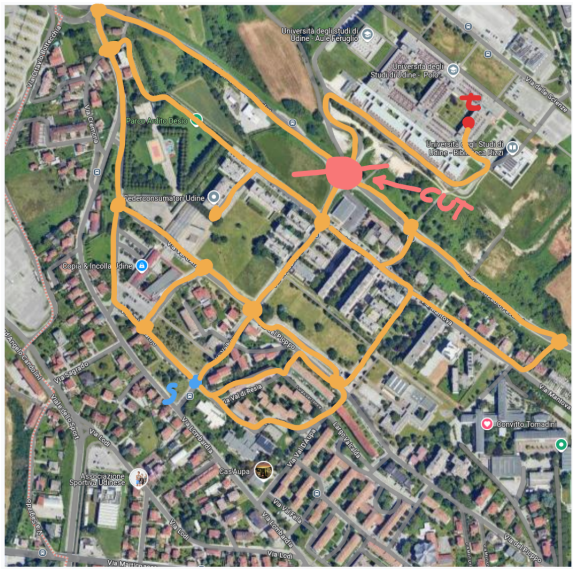

Figura 2.4: Esempio di difesa dell'aula A036

In questo caso il minimo taglio di un nodo che si può fare per separare (s) e (t) è il taglio del nodo rosso in figura.

2.2 MinCutMaxFlow

Esistono diversi algoritmi di MinCutMaxFlow, quelli che vengono presi in considerazione sono i seguenti:

1. Ford-Fulkerson CPU
2. Goldberg-Tarjan CPU
   (a) Goldberg-Tarjan Parallelo GPU

2.2.1 Ford-Fulkerson

L'algoritmo Ford-Fulkerson, uno dei metodi classici per risolvere problemi di flusso massimo, si basa sull'individuazione iterativa di cammini aumentanti nel grafo residuo. Di seguito, riportiamo lo pseudocodice:

Ford-Fulkerson

```
auto minCutMaxFlow(graph, rGraph, source, to) {
    // Inizializzazione delle strutture dati
    // ...

    // Continua finché ci sono cammini aumentanti nel grafo residuo
    while (bfs(rGraph, parent, source, to)) {
        path_flow = INFINITY;

        // Individua il cammino minimo dal nodo di arrivo al nodo di
   partenza
        for (v = to; v != source; v = parent[v]) {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }

        // Aggiorna il flusso nel grafo residuo
        for (v = to; v != source; v = parent[v]) {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }
    }

    // Esegue una DFS per identificare i nodi collegati alla sorgente nel
   grafo residuo
    dfs(rGraph, visited, source);

    // Calcolo dei nodi da tagliare
}
```

Il metodo utilizza la BFS (*Breadth-First Search*) per identificare i cammini aumentanti e la DFS (*Depth-First Search*) per determinare i nodi accessibili dalla sorgente nel grafo residuo, utile per calcolare il taglio minimo.
Ford-Fulkerson è un approccio versatile ma presenta sfide significative nella parallelizzazione. Questo perché sia BFS che DFS, essendo algoritmi di tipo sequenziale, sono difficili da implementare in modo efficiente su piattaforme parallele.

2.2.2 Goldberg-Tarjan

L'algoritmo Goldberg-Tarjan rappresenta un approccio diverso. Lo pseudocodice è il seguente:

Goldberg-Tarjan

```
auto minCutMaxFlow(source, sink) {
    // Inizializza il preflusso dalla sorgente
    preflow(source);

    // Finché ci sono nodi attivi, esegui push e relabel
    while (any_active()) {
        push(active_node);
        relabel(active_node);
    }
}
```

L'idea principale è identificare un nodo attivo (ossia un nodo con flusso in eccesso) e cercare di spingere tale flusso verso nodi adiacenti, fino a quando non è più possibile effettuare ulteriori movimenti. Se un nodo attivo non può spingere il flusso, la sua altezza viene aumentata (*relabel*) per consentire nuovi movimenti.
Le funzioni chiave, `push` e `relabel`, sono implementate come segue:

Push e Relabel

```
auto push(x) {
    if (active(x)) {
        for (y = neighbor(x)) {
            if (height(y) == height(x) - 1) {
                flow = min(capacity(x, y), excess_flow(x));

                // Aggiorna il flusso
                excess_flow(x) -= flow;
                excess_flow(y) += flow;
                capacity(x, y) -= flow;
                capacity(y, x) += flow;
            }
        }
    }
}
```

```
auto relabel(x) {
    if (active(x)) {
        my_height = V; // Altezza massima iniziale

        // Calcola la nuova altezza minima
        for (y = neighbor(x)) {
            if (capacity(x, y) > 0) {
                my_height = min(my_height, height(y) + 1);
            }
        }
        height(x) = my_height;
    }
}
```

Un nodo x è considerato attivo se ha un flusso in eccesso ($excess\_flow(x) > 0$) e la sua altezza è inferiore a un valore massimo ($HEIGHT\_MAX$). Le operazioni possibili per un nodo attivo sono:

- Spingere il flusso verso un vicino y se $capacity(x, y) > 0$ e $height(y) = height(x) - 1$.
- Eseguire un *relabel*, aumentando la propria altezza se tutti i vicini hanno un'altezza maggiore o uguale alla propria.

L'algoritmo Goldberg-Tarjan è particolarmente adatto alla parallelizzazione, poiché ogni nodo può essere gestito da un thread separato. Gli aggiornamenti alle strutture dati (*excess_flow* e *capacity*) avvengono in modo atomico, garantendo coerenza nelle operazioni concorrenti.

2.3 Goldberg-Tarjan Parallelo

L'implementazione dell'algoritmo descritta deriva dal paper [CK24]. Questa implementazione si basa su una serie di strutture dati di supporto fondamentali per la risoluzione del problema, descritte di seguito:

- **excessTotal**: rappresenta la quantità totale di flusso in eccesso. Questa variabile viene inizializzata come la somma delle capacità di tutti gli archi in uscita dal nodo sorgente (s).
- **excesses**: è un vettore di dimensione $|V|$ che rappresenta il flusso in eccesso associato a ciascun nodo. Viene aggiornato in corrispondenza di una coppia di nodi collegati da un arco selezionato come arco di costo minimo. L'accesso a questa struttura avviene in modo atomico per garantire la correttezza in un ambiente parallelo.
- **flow_index**: è un vettore di dimensione $|E|$ che associa a ciascun arco *forward* l'arco *backward* corrispondente. Per comprendere meglio la sua inizializzazione, si può fare riferimento al codice disponibile al seguente link: `parallel/loader.hpp#L258-L280`.
- **heights**: è un vettore di dimensione $|V|$ che rappresenta l'altezza di ciascun nodo. I nodi con altezza maggiore possono spingere il flusso verso nodi con altezza minore. La struttura viene aggiornata incrementando di 1 l'altezza dei nodi dai quali non è possibile spingere flusso.
- **forward_flows** e **backward_flows**: sono vettori di dimensione $|E|$ che rappresentano rispettivamente la quantità di flusso spinta attraverso un arco in avanti o all'indietro.

L'obiettivo dell'algoritmo analizzato è risolvere il problema del taglio minimo sugli archi, ossia determinare il valore del taglio minimo di un grafo. Per estendere l'algoritmo al problema del taglio dei nodi, sono state apportate delle modifiche applicando la trasformazione discussa nel Capitolo 2.1. La verifica dei nodi di interesse avviene utilizzando il controllo presentato anch'esso nel Capitolo 2.1.
Di seguito è riportato l'algoritmo in forma di pseudocodice:

Algorithm 1: The lock-free push-relabel algorithm

Input: $G(V, E)$: the directed graph, $G_f(V, E_f)$: the residual graph, $c_f(v, u)$: the residual flow on (u, v), $e(v)$: the excess flow of the vertex v, $h(v)$: the height of the vertex v, $Excess\_total$: the sum of excess flow

Output: $e(t)$: the maximum flow value

Data: Initialize $c_f(v, u), e, h$, $Excess\_total \leftarrow 0$

```
/* Step 0: Preflow                                          */
1  foreach (s, v) ∈ E do
2      c_f(s, v) ← 0
3      c_f(v, s) ← c(s, v)
4      e(v) ← c(s, v)
5      Excess_total ← Excess_total + c(s, v)
6  while e(s) + e(t) < Excess_total do
       /* Step 1: Push-relabel kernel (GPU)                */
7      cycle = |V|
8      while cycle > 0 do
9          foreach u ∈ V and e(u) > 0 and h(u) < |V| do
10             h' ← ∞ , v' = -1
11             foreach (u, v) ∈ E_f do
12                 if h(v) < h' then
13                     h' ← h(v) , v' = v      // the value of the min node
14             if h(u) > h' then
15                 d ← MIN(e(u), c_f(u, v'))
16                 AtomicSub(c_f(u, v'), d)
17                 AtomicSub(e(u), d)
18                 AtomicAdd(c_f(v', u), d)
19                 AtomicAdd(e(v'), d)
20             else
21                 h(u) ← h' + 1
22         cycle ← cycle − 1
       /* Step 2: Heuristic Optimization (CPU)             */
23     GlobalRelabel()
```

Figura 2.5: Push-Relabel Algorithm

La Figura 2.3 contiene alcune correzioni rispetto all'implementazione originale descritta nel paper. Queste modifiche sono state apportate dopo aver individuato errori nell'algoritmo originale, che sono stati segnalati agli autori del lavoro. Un riferimento agli errori individuati è disponibile al seguente link: `GitHub Issue #4`.

L'algoritmo presentato assegna un thread a ciascun vertice u e verifica se tale vertice è attivo. Se u è attivo, il thread cerca un nodo adiacente v con altezza minima tra i vicini. Quando $h(u) > h(v)$, il thread spinge flusso da u a v; in caso contrario, aumenta di 1 l'altezza di u, portandola a $h(v) + 1$.

Al termine delle operazioni di tutti i thread, viene eseguita una procedura chiamata **GlobalRelabel**, che riduce al minimo le altezze dei nodi e decrementa il valore di `excessTotal`, consentendo all'algoritmo di convergere verso la soluzione finale.

Gpu Call

```
// Configure the GPU
int device = -1;
```

```
cudaGetDevice(&device);
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, device);
dim3 num_blocks(deviceProp.multiProcessorCount * numBlocksPerSM);
dim3 block_size(numThreadsPerBlock);
size_t sharedMemSize = 3 * block_size.x * sizeof(int);

....
// Algo
    // Update GPU values
    ( cudaMemcpy ... , cudaMemcpyHostToDevice) ) ;

    // Gpu Call
    // calling the kernel
    cudaLaunchCooperativeKernel(push_relabel_kernel, ...args);
    cudaDeviceSynchronize();

    // Get results
    ( cudaMemcpy ... , cudaMemcpyDeviceToHost ) ) ;
```

Nella fase iniziale, il codice *host* si occupa di configurare i parametri necessari per la chiamata al codice *device*. Vengono definiti il numero di blocchi, la dimensione di ciascun blocco e la grandezza della memoria condivisa (*Shared Memory*). L'accesso ai nodi è organizzato suddividendo il lavoro in un numero di blocchi (`BLS`) ognuno contenente un determinato numero di thread (`THS`). Questi due parametri possono essere impostati durante la fase di compilazione. Successivamente, il *kernel* viene avviato tramite la funzione `cudaLaunchCooperativeKernel`, che permette di lanciare un *kernel* su una griglia composta da tanti blocchi quanti sono gli *SMs* (*Streaming Multiprocessors*) disponibili sul *device*.
Ogni thread esegue il calcolo dell'indice del nodo u a cui accedere utilizzando il seguente approccio:

```
grid_group grid = this_grid();
int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
```

Se il nodo individuato è attivo, il thread cerca un arco uscente dal nodo u che si collega al nodo v con l'altezza minima e verso cui è possibile spingere del flusso. Ogni thread è incaricato di analizzare più di un nodo: in particolare, ciascun thread esamina un sottoinsieme di nodi pari a $|V|/(\texttt{blockDim.x} \cdot \texttt{gridDim.x})$. Questi nodi sono selezionati con un intervallo regolare, ossia uno ogni $\texttt{blockDim.x} \cdot \texttt{gridDim.x}$.
L'algoritmo ottimizza l'esecuzione sfruttando lo stesso thread all'interno della griglia per un totale di $|V|$ iterazioni. Questa strategia evita di restituire il controllo al codice *host* e di rilanciare il codice *device*, riducendo così l'*overhead*.
Quando si decide di spingere il flusso attraverso un arco, è necessario aggiornare in modo atomico alcune strutture dati di supporto, come `gpu_bflows`, `gpu_fflows`, `gpu_excess_flow` e `gpu_excess_flow`. Per garantire l'atomicità delle operazioni, si utilizzano funzioni specifiche come:

```
atomicAdd(&*int, int)
atomicSub(&*int, int)
```

Questi accorgimenti assicurano che le modifiche alle strutture dati siano sicure ed esenti da conflitti durante l'esecuzione parallela.
L'algoritmo richiede una particolare rappresentazione del grafo, nota come CSR (*Compressed Sparse Row*), per poter funzionare correttamente. Questa struttura consente di rappresentare un grafo $G(V, E)$ in maniera compatta ed efficiente, utilizzando tre elementi principali:

- **offsets**: si tratta di un array di dimensione $|V|$, in cui ogni elemento indica l'offset nel vettore delle destinazioni. In pratica, per ogni nodo del grafo, questo vettore specifica l'indice a partire dal quale si possono trovare i nodi a cui esso è collegato.
- **destinations**: questo vettore, di dimensione $|E|$, contiene i nodi di destinazione per tutti gli archi del grafo. Ogni valore rappresenta un nodo verso cui esiste un arco uscente.
- **capacities**: anch'esso di dimensione $|E|$, rappresenta la capacità associata a ciascun arco. Ogni elemento indica il peso o la capacità massima che può essere trasportata lungo un dato arco.

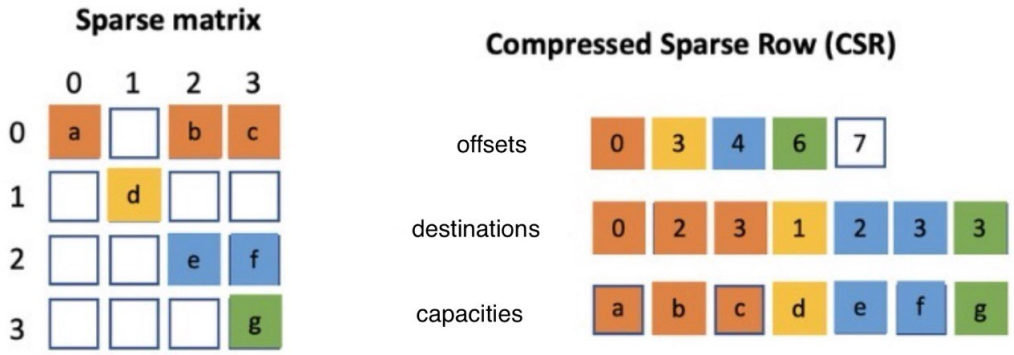

Figura 2.6: https://www.researchgate.net

3 Risultati sperimentali

I risultati sui tempi di esecuzione sono stati presi generando grafi di grandezze diverse con numero di nodi diverso. In particolare sono stati utilizzati i seguenti grafi:

1. graphs/basic1.txt –nodes 70 –edges 6000
2. graphs/basic2.txt –nodes 130 –edges 10000
3. graphs/basic3.txt –nodes 260 –edges 20000
4. graphs/basic4.txt –nodes 520 –edges 50000
5. graphs/basic5.txt –nodes 1030 –edges 100000
6. graphs/nanoone.txt –nodes 2050 –edges 200000
7. graphs/microone.txt –nodes 4100 –edges 400000
8. graphs/minione.txt –nodes 8200 –edges 800000
9. graphs/midone.txt –nodes 16400 –edges 1600000
10. graphs/bigone.txt –nodes 32800 –edges 3200000

I dati raccolti e i grafici sono disponibili al link : `Grafici Algoritmo`.

3.1 Comparazione tra algoritmi diversi

Nella tabella vengono riportati i tempi di esecuzione in microsecondi delle diverse implementazioni.

Tempi

Grafo	Nodi	Archi	GPU GodbergTarjan	CPU GoldbergTarjan	FordFulkerson	CPUgt/GPUgt	FoFu/GPUgt
basic1	70	6000	1386	931	9581	0,6	6,9
basic2	130	10000	2873	8617	37298	2,9	12,9
basic3	260	20000	6025	21724	183634	3,6	30,4
basic4	520	50000	6353	637	758920	0,1	119,4
basic5	1030	100000	31103	833646	2790481	26,8	89,7
nanoone	2050	200000	17285	5954	11523114	0,3	666,6
microone	4100	400000	135952	39740650	41343893	292,3	304,1
minione	8200	800000	57861	81863	179611458	1,4	3104,1
midone	16400	1600000	122384	304478	699148515	2,4	5712,7
bigone	32800	3200000	241725	1065764		4,4	

Figura 3.1: Tabella dei tempi di microsecondi

Dalla tabella è evidente come l'implementazione dell'algoritmo su GPU abbia determinato un notevole miglioramento nella velocità di esecuzione rispetto alla controparte su CPU. Questo miglioramento è particolarmente rilevante per grafi di grandi dimensioni.

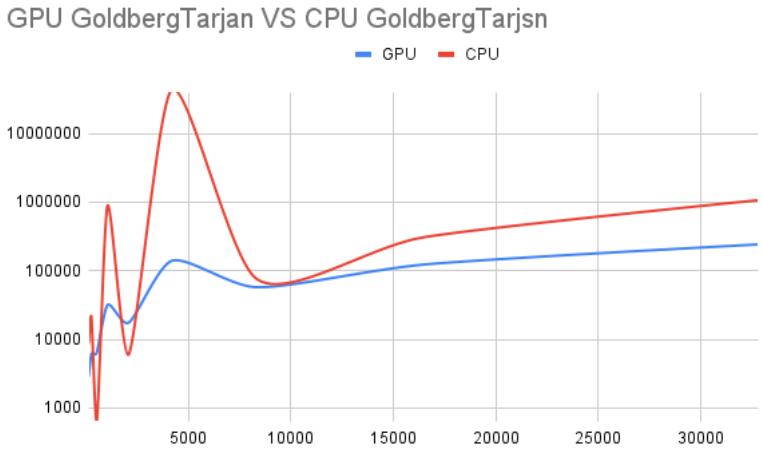

Figura 3.2: X: numero di nodi, Y: log(tempo di esecuzione)

Dal grafico si nota che l'implementazione dell'algoritmo su GPU porta a un miglioramento delle prestazioni fino a quattro volte rispetto alla CPU, in particolare per grafi con oltre 8000 nodi. La sezione con un numero minore di 8000 nodi ha un comportamento particolare che potrebbe essere derivante da fatto che il generatore di grafi potrebbe introdurre molti cicli in grafi con pochi nodi e tanti archi.
Nel grafico seguente vengono comparati l'algoritmo GPU GoldbergTarjan e Ford-Fulkerson:

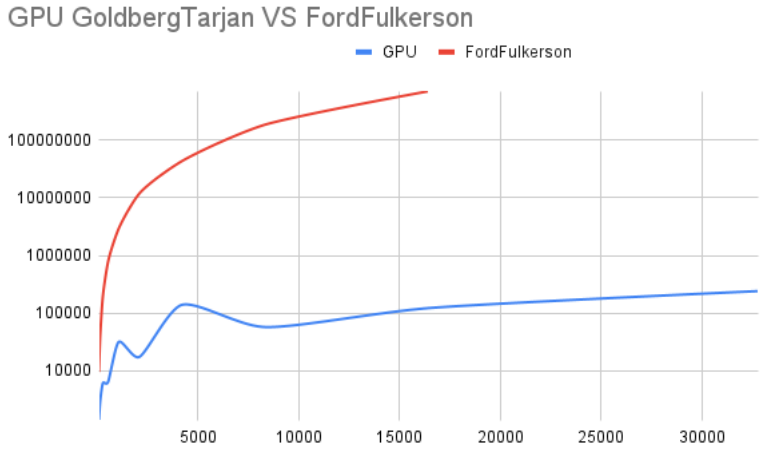

Figura 3.3: X: numero di nodi, Y: log(tempo di esecuzione)

Dal grafico si nota che la versione GPU è decisamente migliore. La versione Ford-Fulkerson utilizza una rappresentazione completa del grafo con matrice di adiacenza che comporta la scanzione completa di tutti gli archi, anche quelli di capacità 0. L'algoritmo GPU utilizza una rappresentazione CSR che gli permette di non dover

scansionare gli archi di capacità 0.
Di seguito viene riportato un grafico che confronta le prestazioni di due algoritmi eseguiti su CPU:

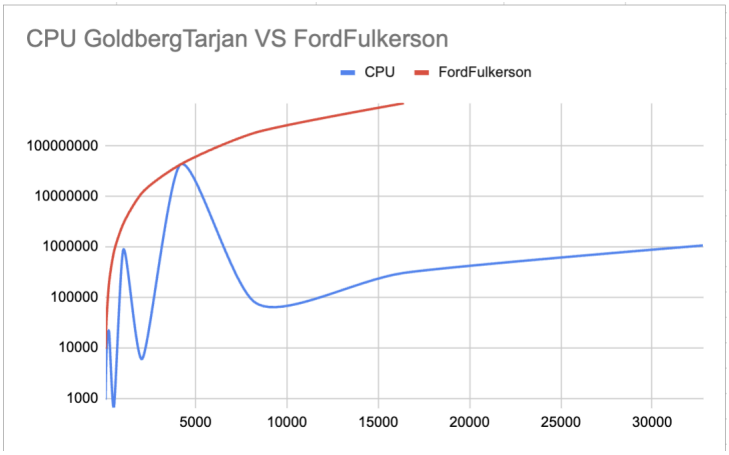

Figura 3.4: X: numero di nodi, Y: log(tempo di esecuzione)

L'algoritmo Goldberg-Tarjan CPU utilizzando una rappresentazione CSR (Compressed Sparse Row), mostra un miglioramento significativo in termini di prestazioni rispetto al Ford-Fulkerson. Questo risultato evidenzia l'efficacia di tale rappresentazione del grafo nel ridurre i tempi di esecuzione.

3.2 Confronto dei kernel con parametri BLS e THS

Nell'implementazione dell'algoritmo su GPU, è possibile configurare due parametri fondamentali:
BLS che indica il numero di blocchi assegnati a ciascun SM; THS che specifica il numero di thread per blocco. Di seguito, sono presentati i tempi di esecuzione ottenuti con diverse configurazioni di questi parametri:

Grafo	Nodi	BLS=1, THS=1024	BLS=2, THS=512	BLS=4, THS=256	BLS=8, THS=128	BLS=16, THS=64
basic1	70	1520	1386	1667	2719	4669
basic2	130	3007	2873	3177	5837	10624
basic3	260	5959	6025	6328	11654	22137
basic4	520	6506	6353	6867	12272	22404
basic5	1030	31600	31103	32828	63981	126934
nanoone	2050	16860	17285	17910	29478	38247
microone	4100	145528	135952	146370	240393	350766
minione	8200	53598	57861	57614	81312	139594
midone	16400	123635	122384	145042	219204	241164
bigone	32800	266265	241725	240535	391576	583796

Figura 3.5: Tabella dei tempi di esecuzione del kernel con diverse configurazioni di BLS e THS.

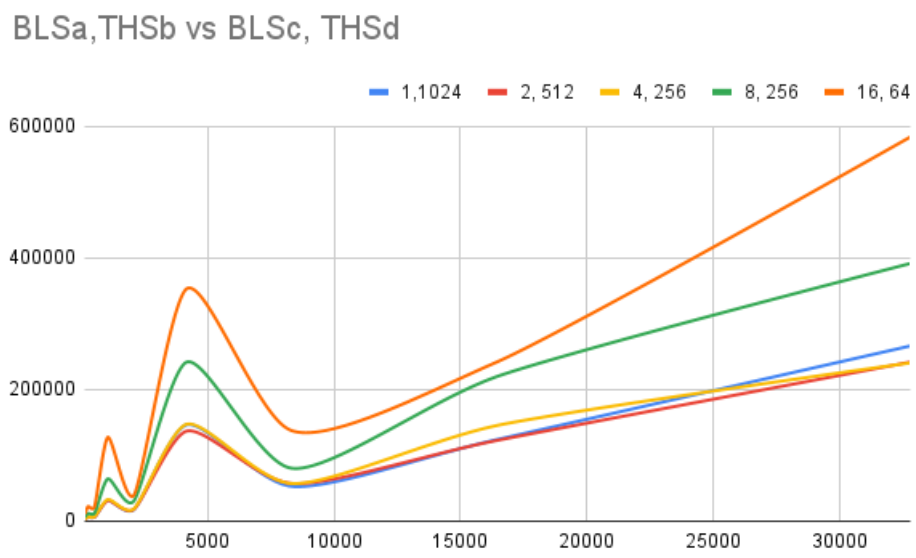

Figura 3.6: Grafico dei tempi kernel con parametri diversi

Dal grafico emerge che la configurazione ottimale è quella in cui BLS=1 e THS=1024.

3.2.1 Effetto del riutilizzo dei thread

Un'altra ottimizzazione significativa consiste nel riutilizzare i thread nella grid il più possibile. Questa strategia, implementata nel codice come impostazione di default, permette di ridurre il tempo complessivo di esecuzione, come mostrato nel seguente grafico:

Figura 3.7: X: numero di nodi, Y: log(tempo di esecuzione)

L'ottimizzazione del riutilizzo dei thread porta a un miglioramento evidente delle performance. Tuttavia, ulteriori margini di miglioramento possono essere esplorati utilizzando tecniche più avanzate di suddivisione dei dati e work balancing. Alcune di

queste tecniche sono descritte nel dettaglio nel paper [CK24], dove vengono proposte implementazioni avanzate che potrebbero ulteriormente ottimizzare l'algoritmo.

4 Build del codice

I tre algoritmi presi in considerazione sono nelle cartelle:

- sequential: Versione FordFulkerson
- paperSerialCode: Versione GoldbergTarjan CPU
- parallel: Versione GoldbergTarjan GPU

Scrivendo sul terminale:
make test
varranno eseguiti i test su diversi grafi.
Nella versione CUDA è possibile settare dei parametri BLS e THS
make custom_params BLS=1, THS=1024
L'output degli eseguibili sarà il tempo di esecuzione dell'algoritmo.

Bibliografia

[CK24] Po-Chieh Lin Chou-Ying Hsieh e Sy-Yen Kuo. «Engineering A Workload-balanced Push-Relabel Algorithm for Massive Graphs on GPUs». In: *arXiv* (2024).

[] URL: https://web.stanford.edu/class/archive/cs/cs161/cs161.1172/CS161Lecture16.pdf.

[] URL: https://www.tutorialspoint.com/data_structures_algorithms/dsa_kargers_minimum_cut_algorithm.htm.

[] URL: https://www.baeldung.com/cs/minimum-cut-graphs.

[] URL: https://it.wikipedia.org/wiki/Algoritmo_di_Ford-Fulkerson.

[] URL: https://www.nvidia.com/content/GTC/documents/1060_GTC09.pdf.

[] URL: https://en.wikipedia.org/wiki/Push%E2%80%93relabel_maximum_flow_algorithm.

[] URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4563095.

[] URL: https://arxiv.org/pdf/2404.00270.

[] URL: https://github.com/NTUDDSNLab/WBPR/tree/master/maxflow-cuda.

[] URL: https://www.adrian-haarbach.de/idp-graph-algorithms/implementation/maxflow-push-relabel/index_en.html.

[] URL: https://www.geeksforgeeks.org/push-relabel-algorithm-set-2-implementation/.