UNIVERSITÀ DEGLI STUDI DI UDINE

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

PROGETTO DI ARCHITETTURE PARALLELE

Ricerca di SCC in un grafo con vincoli legati ad un dato sottoinsieme

AUTORI

Lunardi Riccardo, D'Abrosca Gianluca

Anno accademico 2021-2022

Contatti dell'Istituto
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
https://www.dmif.uniud.it/

Indice

Elenco delle figure

1 Problema

In questo capitolo si affronterà il testo del problema e lo si interpreterà per capirlo a fondo.

1.1 Testo del problema

Dato un grafo diretto $G(V, E)$ con insieme di nodi V e di archi E, richiamo alcune definizioni preliminari:

- Una SCC (strongly connected component) di G è un qualsiasi insieme S (massimale e di almeno due nodi) tale che per ogni coppia di nodi x, y in S esiste un cammino diretto da x a y.
- Dato un insieme di nodi C, indico con $prec(C)$ l'insieme di nodi così definito:

$$prec(C) = \{n | n \in \{V - C\}, \text{ tale che } \forall m \in C \ \exists \text{ un arco } (n, m)\}$$

Ossia:
L'insieme di tutti i nodi esterni all'insieme C che però sono adiacenti ad un qualsiasi nodo di C.
Il problema da risolvere in CUDA è il seguente:
Dato un grafo $G(V, E)$ e un insieme di nodi U (un qualsiasi sottoinsieme di V), determinare se esiste almeno una SCC S del grafo tale che S è sottoinsieme di U e nessun elemento di $prec(C)$ appartiene a U.

1.2 Studio del problema

1.2.1 Input

Gli input sono abbastanza chiari, e sono:

- Un grafo $G(V, E)$, dove V è l'insieme dei nodi e E è l'insieme degli archi
- Un insieme U, contenente nodi facenti parte dell'insieme V

1.2.2 Output

Invece l'output non è altrettanto chiaro, secondo il testo bisogna capire se esiste almeno una SCC S del grafo, tale che S sia sottoinsieme di U e nessun elemento di $prec(C)$ appartiene a U, nonostante ciò C non era mai stato dichiarato nel testo, quindi assumiamo che sia un errore di battitura e consideriamo $prec(S)$.
Come output, verrà ritornato 1 o 0 dalla funzione principale, indicando se esiste o no almeno una SCC valida, ovvero che rispetti i vincoli imposti.

1.2.3 Rielaborazione di punti critici

- Trovare una SCC S nel grafo G tale che sia sottoinsieme di U.
  Di conseguenza ogni SCC per essere considerata come output, dovrà avere tutti i nodi che fanno parte solamente di U per la definizione di Strongly Connected Component.
- Nessun elemento di $prec(S)$ appartiene a U.
  Rielaborando la formula si ha:

  $$prec(S) = \{n | n \in \{V - S\}, \text{ tale che } \forall m \in S \ \exists \text{ un arco } (n, m)\}$$

  Ovvero, $prec(S)$ conterrà tutti i nodi che non fanno parte della SCC e da cui parte un arco che punta ad un nodo della SCC stessa.
  Quindi, la SCC non potrà ricevere archi da nodi di U che non facciano parte della SCC stessa per essere considerata come output.

Combinando questi due punti si ha che una SCC per essere valida come output deve avere solamente nodi appartenenti ad U e non può ricevere archi entranti da nodi appartenenti ad U che non facciano parte della SCC.

1.3 Alcuni esempi

Poniamo il caso in cui:

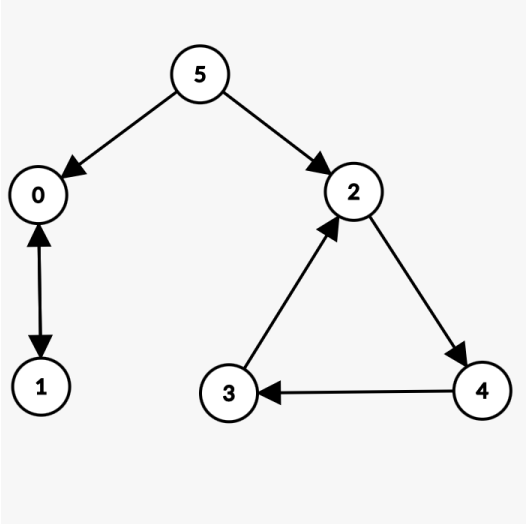
- $G(V, E)$, con nodi $V = 0, 1, 2, 3, 4, 5$
- $U = 0, 1, 2, 3, 4$

Con l'aiuto della figura 1.1 si riproducono 4 esempi, dove cambiano solo pochi archi:

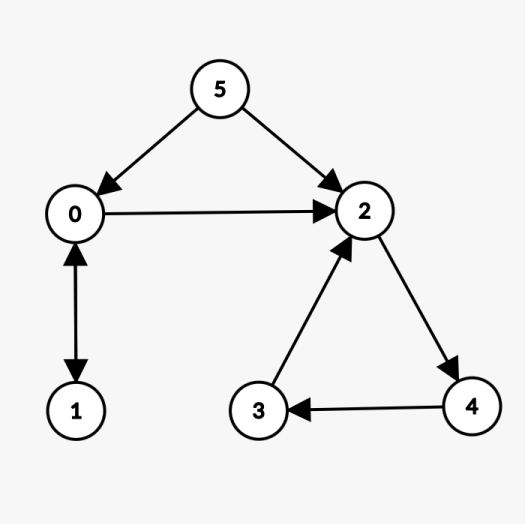
- Esempio 1.1a.
  In questo caso sono presenti 2 SCC:
  - $S_1 = 0, 1$. Perchè tra loro formano una componente strettamente connessa e ricevono archi solamente da 5 che non fa parte di U
  - $S_2 = 2, 3, 4$. Perchè tra loro formano una componente strettamente connessa e ricevono archi solamente da 5 che non fa parte di U
- Esempio 1.1b.
  In questo caso è presente 1 SCC:
  - $S_1 = 0, 1$. Perchè tra loro formano una componente strettamente connessa e ricevono archi solamente da 5 che non fa parte di U

  In questo caso S_2 non è una SCC valida perchè riceve un arco da 0: un nodo esterno alla SCC ma interno a U.
- Esempio 1.1c.
  In questo caso è presente 1 SCC:
  - $S_2 = 2, 3, 4$. Perchè tra loro formano una componente strettamente connessa e ricevono archi solamente da 5 che non fa parte di U

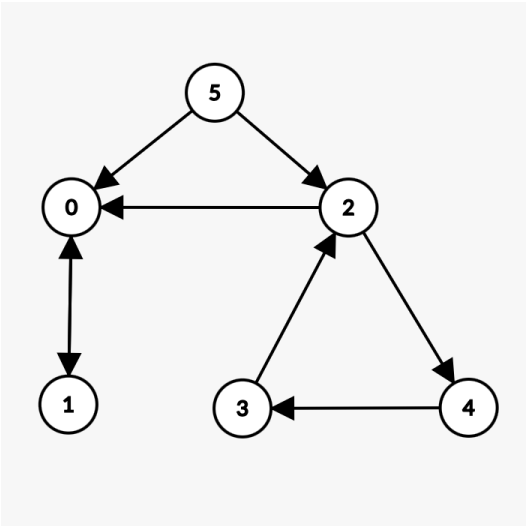
  In questo caso S_1 non è una SCC valida perchè riceve un arco da 2: un nodo esterno alla SCC ma interno a U.
- Esempio 1.1d.
  In questo caso non è presente nessuna SCC perchè:
  - S_1 non è una SCC valida perchè riceve un arco da 3: un nodo esterno alla SCC ma interno a U.
  - S_2 non è una SCC valida perchè riceve un arco da 0: un nodo esterno alla SCC ma interno a U.



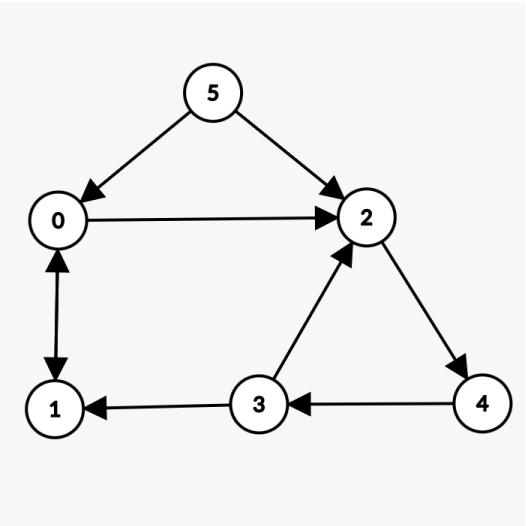
(a) Grafo 1



(b) Grafo 2



(c) Grafo 3



(d) Grafo 4

Figura 1.1: 4 Casi presi in esempio

2

Rappresentazione dei dati e Algoritmo

L'algoritmo preso in considerazione è l'algoritmo Forward-Backward che viene affrontato nel paper [2]. Si vedrà come rappresentare i dati al meglio e poi uno studio dell'algoritmo con le strutture dati che abbiamo visto. Con questo capitolo si andranno ad affrontare alcune scelte che influenzeranno lo sviluppo del codice.

2.1 Struttura degli input

I file rappresentati i grafi hanno una struttura rigida, per rendere più semplice la lettura dei grafi da input.
La struttura di un file per l'input è composta da 3 parti così ordinate:

- Header. Composto da una sola riga indicata con il carattere %. Oltre il carattere speciale troviamo, in ordine, $numero_{archi}$ e $numero_{nodi}$.
- Archi. Composto da $numero_{archi}$ righe.
  Ogni riga è composta da 2 numeri. Il primo rappresenta il nodo da cui parte l'arco e il secondo il nodo in cui termina l'arco.
- U. Composto da un numero arbitrario di righe. Ogni riga è composta da 1 numero, questo indica che il nodo corrispondente a quel numero fa parte dell'insieme U

Oltre a questa struttura, i file in input hanno anche un ulteriore vincolo nella parte degli archi. Tutte le righe devono essere ordinate in modo crescente in base al primo numero. Oltre a questo primo ordinamento, ce n'è un secondo in base al secondo numero, sempre in ordine crescente. Nella figura 2.1 di seguito si osserva un file d'esempio rappresentante il grafo in figura 1.1a.

```
1  % 7 6
2  0 1
3  1 0
4  2 4
5  3 2
6  4 3
7  5 0
8  5 2
9  0
10 1
11 2
12 3
13 4
14 5
```

Figura 2.1: File di input d'esempio rappresentante il grafo in figura 1.1a

2.2 Rappresentazione dei dati

In seguito si osserverà come vengono rappresentate le informazioni in memoria:

- $numero_{archi}$ e $numero_{nodi}$. Vengono rappresentati semplicemente con due interi unsigned. Quindi il numero massimo permesso di archi e di nodi è dell'ordine di 4 miliardi.
- $G(V, E)$. Essendo un argomento delicato merita una sezione a parte.
- *pivots*. I pivots delle SCC sono dei nodi, scelti arbitrariamente, che vengono utilizzati per rappresentare le SCC. In questo vettore, tutti i nodi che hanno lo stesso pivot rappresentano una SCC. Quindi un pivot corrisponde univocamente ad una SCC e viceversa.
- *status*. Ulteriori informazioni utili all'interno del codice. Anche queste informazioni meritano una sezione a loro dedicata.

2.2.1 Grafo

Per quanto riguarda il grafo $G(V, E)$ si doveva scegliere tra 2 rappresentazioni:

- Matrice di adiacenza. Una matrice avente $numero_{nodi}$ colonne e $numero_{nodi}$ righe che occupa quindi uno spazio pari a $\Theta(|V|^2)$
- Liste di adiacenza. Due liste che rappresentano rispettivamente i nodi e gli archi, che occupano quindi uno spazio pari a $\Theta(|V| + |E|)$

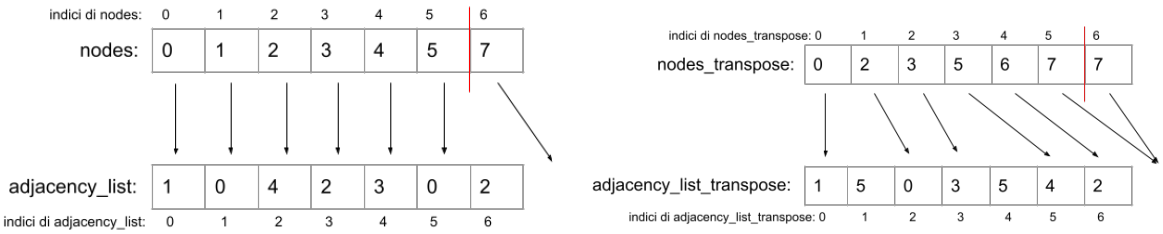
In entrambi gli articoli consultati [1][2] si consiglia di scegliere la seconda. In particoare, nel paper dell'NUDT[2], viene specificato che con degli accorgimenti è possibile accedere agli archi uscenti o entranti di un dato nodo in tempo costante.
Verranno quindi utilizzate le liste di adiacenza. Per poterle implementare, sono state previsti 4 vettori:

- *adjacency_list*. Composto da $|E|$ elementi. Questo vettore contiene tutti i nodi v per tutti gli archi (u, v) ordinati in primis per u e solo dopo per v.
- *nodes*. Composto da $|V| + 1$ elementi, dove:
  - $nodes[i]$ contiene l'indice di *adjacency_list* da cui partono gli archi uscenti da i.
  - $nodes[i + 1]$ contiene il primo indice di *adjacency_list* in cui non ci sono più gli archi uscenti da i.

  Quindi il numero $nodes[i + 1] - nodes[i]$ corrisponde a quanti archi escono da i.
- *adjacency_list_transpose*. Composto da $|E|$ elementi. Questo vettore contiene tutti i nodi u per tutti gli archi (u, v) ordinati in primis per v e solo dopo per u.
- *nodes_transpose*. Composto da $|V| + 1$ elementi, dove:
  - $nodes\_transpose[i]$ contiene l'indice di *adjacency_list_transpose* da cui partono gli archi entranti in i.
  - $nodes\_transpose[i+1]$ contiene il primo indice di *adjacency_list_transpose* in cui non ci sono più gli archi entranti in i.

  Quindi il numero $nodes\_transpose[i + 1] - nodes\_transpose[i]$ corrisponde a quanti archi entrano in i.

nodes e *nodes_transpose* contengono un elemento in più rispetto al numero di nodi per riuscire ad usare la stessa logica implementativa anche per l'ultimo nodo. Altrimenti, per trattare l'ultimo elemento dei 2 vettori si sarebbe dovuta usare una logica diversa.



(a) *nodes* e *adjacency_list* (b) *nodes_transpose* e *adjacency_list_transpose*

Figura 2.2: Rappresentazione del grafo 1.1a

2.2.2 status

Per ogni nodo si sono dovute mantenere in memoria 7 informazioni booleane, che insieme chiameremo *status*, le quali servivano per tutto l'arco dell'esecuzione. Queste informazioni indicano se il nodo:

- É stato visitato in avanti
- É stato visitato all'indietro
- É stato eliminato
- É stato espanso in avanti
- É stato espanso all'indietro
- Fa parte di U
- Fa parte di una SCC

Normalmente, per memorizzare tutte queste informazioni si farebbe uso di 7 vettori booleani, solo che 1 booleano occupa 1 Byte nonostante l'informazione occupi solamente 1 bit lasciando gli altri 7 inutilizzabili.
Verrà osservato in seguito come queste informazioni verranno trattate per cercare di massimizzare l'efficienza.

2.3 Algoritmo

Si è deciso di dividere l'algoritmo in 3 sotto-problemi:

- Elimare i nodi non U. Si prevede un primo trim perché non si vogliono considerare i nodi non U.
- Individuare le SCC. Questo risulta essere il sotto-problema principale da risolvere.
  Per questa task si sono esaminati alcuni paper che affrontassero il problema della ricerca delle componenti fortemente connesse con algoritmi parallelizzabili.
- Eliminare le SCC riceventi archi da nodi U non facenti parte delle stesse.

Per quanto visto nell'analisi dei tempi nel paper [1], tra gli algoritmi parallelizzabili di ricerca delle SCC, si poteva scegliere tra vari algoritmi del tipo divide-and-conquer, come il Coloring e il Forward-Backward. Si è quindi deciso di optare per quest'ultimo, in quanto risulta essere più performante se nella rete ci sono meno SCC. Questo risulta essere ottimale per il problema presentato, in quanto nelle reti considerate si presenta questo fenomeno più spesso, dovuto all'eliminazione dei nodi non appartenenti a U.
L'algoritmo Forward-Backward può essere così riassunto:

- Si sceglie randomicamente un pivot iniziale
- Si cercano tutti i nodi raggiungibili in avanti dal nodo pivot
- Si cercano tutti i nodi raggiungibili all'indietro dal nodo pivot
- Si divide il grafo creato in 4 sotto-grafi
  - Visitati sia in avanti che all'indietro
  - Visitati solo in avanti
  - Visitati solo all'indietro
  - Non visitati
- Si ricalcolano i pivot e si ripete da capo il processo per gli ultimi 3 sotto-grafi

Tutti i nodi che hanno stesso pivot e sono visitati sia in avanti che all'indietro compongono una SCC.

3

Pseudo-codice seriale

Ecco un primo pseudo-codice dell'algoritmo 3.1 successivamente spiegato nelle relative sottosezioni.

```
1   int main() {
2       graph = read_graph();
3       status = read_u_and_eliminate_not_u();
4       stop = false;
5
6       trimming(graph, status, pivots);
7       pivot = choose_random_pivot(graph, status);
8       for(∀v ∈ V) {
9           pivots[v] = pivot;
10      }
11
12      while(!stop){
13          forward_reach(graph, status, pivots);
14          backward_reach(graph, status, pivots);
15
16          trimming(graph, status, pivots);
17
18          update(graph, status, stop);
19      }
20
21      trim_u(graph, status, pivots);
22      print(there_is_scc(graph, status));
23  }
```

Figura 3.1: Algoritmo in pseudo-codice

3.1 trimming

Questa funzione serve per eliminare logicamente i nodi dal grafo per non tenerli in considerazione durante le reach in avanti o in indietro (pseudo-codice in figura 3.2).

```
1   void trimming(graph, status, pivots) {
2       for(∀u ∈ V, non eliminato in graph) {
3           if(∄v ∈ V, (v, u) ∈ E e pivots[u] == pivots[v]) {
4               set_eliminated(status[u]);
5           }
6           if(∄v ∈ V, (u, v) ∈ E e pivots[u] == pivots[v]) {
7               set_eliminated(status[u]);
8           }
9       }
10  }
```

Figura 3.2: trimming in pseudo-codice

I nodi vengono eliminati logicamente quando:

- Non fanno parte di U, in quanto non ci possono essere nodi non appartenenti a U nelle SCC
- Non hanno almeno un arco uscente verso un nodo non eliminato e uno entrante da un nodo non eliminato.

Per cancellare un nodo, il vettore status dedica al Byte di ogni nodo 1 bit, per indicare se è stato cancellato o meno.
Una volta terminato il primo trimming si sceglie il primo nodo non eliminato disponibile all'interno del grafo, e lo si elegge a pivot per ogni nodo non eliminato.

3.2 reach

La distinzione tra forward e backward reach si spiega in poche parole:

- forward_reach. Partendo dai pivot, si vogliono osservare tutti i nodi raggiungibili con una visita in avanti. In questo caso, si fa riferimento unicamente ai vettori *nodes* e *adjacency_list* dato che rappresentato il grafo normalmente.
- backward_reach. Partendo dai pivot, si vogliono osservare tutti i nodi raggiungibili con una visita all'indietro. In questo caso, si fa riferimento unicamente ai vettori *nodes_transpose* e *adjacency_list_transpose* dato che rappresentato il grafo trasposto.

Studiamo lo pseudocodice del forward_reach in figura 3.3.

```
1   void forward_reach(graph, status, pivots) {
2       for(∀u ∈ V, visitato ma non espanso in avanti,
3           non eliminato in graph) {
4           set_forward_expanded(u);
5           for(∀v ∈ V, (u, v) ∈ E: con pivots[u] == pivots[v],
6               non visitato in avanti, non eliminato in graph) {
7               set_forward_visited(v);
8           }
9       }
10  }
```

Figura 3.3: forward_reach in pseudo-codice

Partendo dai nodi inizialmente considerati visitati in avanti, vengono visitati tutti i nodi per cui esiste un cammino in avanti i cui nodi hanno tutti stesso pivot.

3.3 update

Questa funzione serve a suddividere il grafo nei 4 sottografi visti nel capitolo precedente:

- Nodi visitati sia in avanti che all'indietro. Questi nodi fanno parte di una SCC.
- Nodi visitati solo in avanti. Questo sottografo dovrà ripetere la ruotine tenendo conto dei suoi soli nodi.
- Nodi visitati solo all'indietro. Come sopra.
- I rimanenti nodi. Anche questi dovranno ripetere la ruotine come un altro sottografo.

Per capire con quale modalità vengano separati i sottografi, si osservi lo pseucodice in figura 3.4.

```
1  void update(graph, status, pivots) {
2      write_id_for_pivots = malloc(4 * numero_{nodi})
3      colors = malloc(numero_{nodi})
4
5      for(∀v ∈ V) {
6          if(!get_is_eliminated(status[v])) {
7              switch(v) {
8                  visitato sia in avanti che all indietro:
9                      colors[v] = 4 * pivots[v];
10                     break;
11
12                 visitato solo in avanti:
13                     colors[v] = 4 * pivots[v] + 1;
14                     break;
15
16                 visitato solo all indietro:
17                     colors[v] = 4 * pivots[v] + 2;
18                     break;
19
20                 altrimenti:
21                     colors[v] = 4 * pivots[v] + 3;
22                     break;
23             }
24             write_id_for_pivots[colors[v]] = v;
25         }
26     }
27
28     for(∀v ∈ V) {
29         if(!get_is_eliminated(status[v])) {
30             pivots[v] = v;
31         } else {
32             pivots[v] = write_id_for_pivots[colors[v]];
33         }
34     }
35 }
```

Figura 3.4: update in pseudo-codice

I nodi di uno dei 4 sotto-grafi competono tra loro per essere eletti come pivot del loro relativo sotto-grafo. Si è scelta questa soluzione per il codice seriale, perché nonostante non sia la più efficiente è già

cuda-ready.
Ogni nodo salverà il proprio $colors[v]$ in una variabile locale e, dopo una sincronizzazione, tutti i thread entreranno in gara per essere eletti come pivot grazie a $write\_id\_for\_pivots[colors[v]]$.

3.4 trim_u

In questa sottosezione si osserva come una volta trovate tutte le SCC, queste vengano eliminate se ricevono archi da nodi U non appartenenti alle SCC stessa (fig. 3.5).

```
1   void trim_u(graph, status, pivots) {
2       for(∀u ∈ U) {
3           for(∀v ∈ V, (u, v) ∈ E) {
4               if(pivots[v] != pivots[u]) {
5                   set_not_is_scc(status[pivots[v]]);
6               }
7           }
8       }
9       for(∀v ∈ V) {
10          if(get_is_scc(status[pivots[v]])) {
11              set_is_scc(status[pivots[v]]);
12          } else {
13              set_not_is_scc(status[pivots[v]]);
14          }
15      }
16  }
```

Figura 3.5: trim_u in pseudo-codice

Il primo ciclo for serve ad individuare tutte le SCC che ricevono archi da nodi U non appartenenti ad esse e a settarne i pivot come nodi non facenti parte di una SCC, grazie al bit dedicato dentro a status. Invece, il secondo ciclo for serve per propagare la decisione presa sui pivot al primo for e propagarla a tutti gli altri nodi della relativa SCC.

4

Ottimizzazione in parallelo

In questo capitolo si esamineranno tutte le tecniche e le strategie adottate per il parallelismo in CUDA. Ogni versione nasce con l'idea di essere più efficiente della precedente, ma solo al termine del progetto si raccoglieranno i tempi e si analizzerà l'algoritmo più efficiente.

4.1 Naive senza array status

Questa è una primissima implementazione dell'algoritmo in parallelo. Essendo il codice seriale già di fatto pronto alla serializzazione e le strutture dati già pensate per essere usate in ambito parallelo, sono state introdotte solo poche novità:

- Sono state spostate le funzioni parallelizzabili sul device. In particolare si parla delle funzioni con suffisso *_kernel* e quelle relative eliminazione delle SCC che ricevono archi da nodi U
- Dei punti di sincronizzazione tra host e device. Prima di eseguire alcune funzioni vincolate dai risultati di altre funzioni spostate sulla GPU, devono essere inseriti dei punti di sincronizzazione. Così facendo si è sicuri che tutte le operazioni su GPU terminino prima dell'inizio della funzione successiva
- L'allocazione e lo spostamento della memoria device. Per poter operare sul grafo con le funzioni che vengono eseguite sulla GPU, serve prima spostare la sua struttura dati in memoria device. Per farlo, bisogna prima di tutto allocare la memoria necessaria e successivamente copiare i dati da host a device. Una volta terminata la funzione, invece, è necessario riportare le informazioni modificate dal device all'host. Quando questo passaggio è stato completato, la memoria device può essere liberata. Quest'ultima operazione è necessaria perchè C++/CUDA non ha un sistema di garbage collection efficiente come quelli di Java o Python ed è responsabilità del programmatore deallocare la memoria una volta usata.

4.2 Naive

Da questa versione è stato introdotto l'array status, che raccoglie tutte le 7 informazioni viste precedentemente nella sezione 2.2.2 occupando un solo Byte per nodo, quando precedentemente queste informazioni venivano raccolte in 7 vettori booleani diversi e avrebbero occupato uno spazio pari a 7 Byte per nodo.
Dato che l'informazione booleana occupa un solo bit, ma l'unità minima allocabile in memoria risulta essere di un Byte, si avevano così 7 bit inutilizzati, e avendo 7 vettori si avevano 7 bit usati e 49 bit inutilizzati.
Oltre a questo spreco bisogna tenere in considerazione anche che l'allocazione e lo swap dei vettori tra le memorie occupa tempo proporzionato alla quantità di memoria da spostare.
Si è quindi pensato come questi bit potrebbero essere sfruttati al meglio e all'interno di questo Byte: 7 bit vengono usati per codificare le 7 informazioni booleane e solo 1 bit rimane inutilizzato. Per leggere e modificare lo status di questi bit si usano le operazioni su bit.
Per esempio, considerando l'n-esimo bit, di un dato Byte:

- Lettura. Si ritorna la AND tra la n-esima potenza di 2 e il Byte corrispondente al nodo interessato. Se il bit è impostato a 1, allora la funzione ritorna True.
- Set True. Si assegna al Byte corrispondente al nodo interessato la OR tra la n-esima potenza di 2 e il Byte stesso. Così facendo si setta il bit a 1 indipendentemente dal valore precedente.
- Set False. Si assegna al Byte corrispondente al nodo interessato la AND tra il complementare, a 8 bit, con complemento a 1 della n-esima potenza di 2 e il Byte stesso. Così facendo si setta il bit a 0 indipendentemente dal valore precedente.

Si è pensato che essendo le operazioni sui bit effettuate su hardware, questo vettore portasse netti miglioramenti in termini di efficienza. Sarà poi l'analisi dei tempi a rivelare se questo è vero o meno.

4.3 Stream

Questa versione aggiunge l'implementazione delle stream. Queste consentono un'esecuzione:

- Asincrona. Usando diversi stream, è possibile sovrapporre l'esecuzione di diverse operazioni, permettendo di nascondere la latenza di un'operazione eseguendone un'altra nel frattempo. Ciò può contribuire a migliorare le prestazioni complessive delle applicazioni.
- Concorrente. Usando diversi stream, è possibile eseguire diverse operazioni in modo concorrente sulla GPU, sfruttando al massimo le risorse hardware disponibili.

Con questa implementazione si sono rese asincrone l'allocazione di memoria e lo spostamento della stessa da oppure a device. Nella teoria si sa che l'allocazione e lo spostamento dei dati può essere una delle operazioni più lente da fare in CUDA, si vuole vedere se questo possa essere il nostro caso. Il risultato lo si avrà dopo l'analisi dei tempi.

4.4 Pinned

Questa versione presenta l'utilizzo della memoria page-locked o pinned. Qui viene pinnata la struttura dati legata al grafo per rendere più efficienti le operazioni di copia della memoria. Si è fatto attenzione a non allocare la memoria pinned come write-combined, in quanto questa rende le operazione di lettura più lente, ma queste sono le uniche operazioni che faremo sulla memoria relativa al grafo.
Così facendo si sancisce un limite massimo per il $numero_{nodi}$ e il $numero_{archi}$. Infatti la memoria paged-locked è limitata dalla memoria RAM, non è possibile allocare più memoria page-locked di quanto sia disponibile fisicamente.
Il limite attuale è di 4 miliardi di nodi e 4 miliardi di archi. Purtroppo, creare 2 vettori per i nodi e 2 per gli archi, i quali occupano 4 Byte per nodo vorrebbe dire:

$$4\text{vettori} \cdot 4\text{miliardi di nodi} \cdot 4\text{Byte} = 64\text{GB in RAM}$$

Attualmente non tutti i computer hanno 64 GB di memoria RAM. Vengono ipotizzati come tetto massimo 6 GB di RAM. Quindi con un po' di formule inverse si ottiene che:

$$\frac{6 \text{ GB di RAM}}{2 \cdot 4 \text{ Byte per int}} = 805306368 \text{ numero di nodi} + \text{ numero di archi}$$

Il risultato viene anche diviso per 2, in quanto si vuole tenere in considerazione dei vettori trasposti, quindi in totale il vettore *nodes* e il vettore $adjacency_{list}$ potranno occupare al massimo soltanto 3 GB di RAM. Quindi, si consiglia un tetto massimo teorico di 805 milioni come somma tra numero_nodi e numero_archi del grafo.
I vantaggi della memoria pinned sono:

- Aumento delle performance, questo perché la GPU riesce ad accedere senza overhead e a bypassare la memoria virtuale del sistema per trasferire dati dalla RAM
- Può essere usata insieme alle stream per eseguire trasferimenti di memoria concerrentemente all'esecuzione dei kernel. Questo dettaglio risulta degno di nota nonostante non siamo riusciti a trovarne un'applicazione pratica per la nostra soluzione.

Viene usata anche la memoria mapped per il booleano *stop*. Nella teoria la memoria mapped dovrebbe essere usata quando non conviene spostare grandi porzioni di memoria, per poi usarla poche volte nel codice. In questo caso, si riesce a ridurre il numero di righe necessarie per spostare un solo booleano, che rendeva il codice meno leggibile.
Inoltre, da questa versione sono stati anche fatti altri cambiamenti che si sono ripercossi o meno sulle versioni precedenti:

- I vettori rappresentanti il grafo sono passato da integer ad unsigned. Questo ha permesso di raddoppiare il numero massimo utilizzabile di nodi o archi. Il limite è ora di 4 miliardi di archi o 4 miliardi di nodi. Dato che si ha bisogno di soli numeri positivi, il bit per il segno rimane inutilizzato, e il tipo unsigned sfrutta quel bit per avere il doppio dei numeri
- Tutte le operazioni sui bit sono state accorpate. Per esempio una or con il valore e 1 e una or con il valore e 2 sono state accorpate in unica or con il valore e 3

4.5 OpenMP

OpenMP è una libreria che consente di sfruttare la parallelizzazione di alto livello su processori multicore e su cluster di computer. Può essere utilizzata per aggiungere parallelizzazione ai programmi seriali esistenti senza modificare significativamente il codice sorgente.
Utilizzare OpenMP e CUDA insieme può essere utile in situazioni in cui si desidera utilizzare OpenMP per gestire la parallelizzazione sui processori multicore e CUDA per gestire la parallelizzazione sulla GPU. In questo caso, si sfrutterà OpenMP per semplificare il codice di parallelizzazione.
In questa prima versione in OpenMP sono state semplicemente parallelizzate le chiamate per la copia della memoria, giusto per vedere se cambiassero i tempi di esecuzione.

4.6 OpenMP con back e forward assieme

Dato che l'esecuzione della forward-reach e della backward-reach sono due operazioni che potrebbero essere eseguite in modo concorrente, in questa versione vengono parallelizzate. Nel caso migliore, ovvero quando il numero di forward e backward è lo stesso, il tempo di esecuzione sarà dimezzato. Nel caso peggiore, se non c'è nessuna forward da eseguire, ma solo backward (o viceversa), la complessità rimarrà la stessa.
Le due funzioni vengono eseguite con due vettori status differenti e i kernel vengono lanciati in due stream diversi.A fine esecuzione, vengono sincronizzate le streams delle due operazioni e si esegue una OR sui due vettori per avere un risultato unico.

4.7 OpenMP con back e forward assieme, ma un solo status

Questa versione risulta uguale alla precedente, ma si usa un solo vettore status. Ciò è possibile perchè ogni kernel della funzione di reach modifica il vettore status solo per la sua parte.
O meglio, il forward-reach-kernel non modifica il vettore status per la parte backward e viceversa.

5

Analisi dei risultati

Questo capitolo mostra un'analisi dei tempi per mostrare se le versioni implementate riportano o meno dei miglioramenti.

5.1 La raccolta dei risultati

Il terminale a nostra disposizione possiede:

- Intel Core i5-4590 CPU @ 3.30GHz
- NVIDIA GeForce GTX 1060 6GB

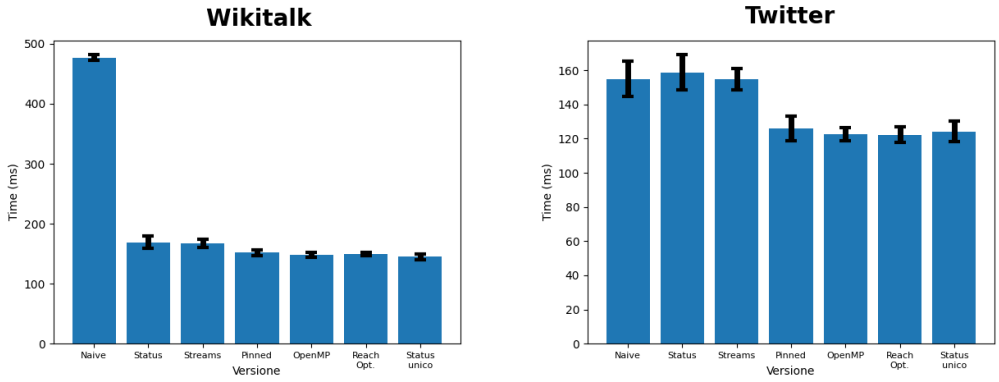
Per il benchmark dei risultati si sono creati numerosi script in Python per generare reti, ma non si riusciva a creare facilmente dei grafi adatti alle nostre esigenze, ossia con un numero di componenti strettamente connesse elevato.
Fortunatamente l'università di Stanford ha messo a disposizione di tutti alcuni grafi di grandi dimensione [3]. I grafi sono stati poi modificati e adattati per poterli gestire con il nostro metodo di lettura dei file. Per confermare o meno alcune ipotesi si sono presi gli stessi grafi e si sono create delle copie prendendo come nodi U il complementare dei nodi U dai grafi da cui vengono creati.

5.2 L'analisi dei risultati

Si analizzano in seguito le versioni una per volta per provare a comprendere quando esse convengono oppure no. Si vogliono inoltre confermare o meno le ipotesi fatte in fase d'implementazione.

5.2.1 Da Naive a Status

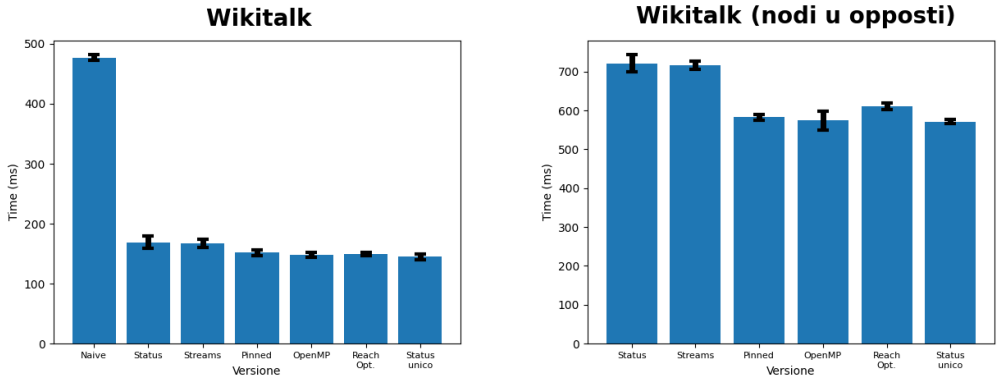
Questa versione dovrebbe ridurre il carico di memoria da spostare da oppure verso la memoria device. D'altro canto può rendere più difficile la modifica e l'accesso alle informazioni contenute nel Byte.
In figura 5.1 si analizzano due casi. Si può vedere come nel caso Wiki-Talk 5.1a questa versione convenga



(a) Wiki-Talk (b) Twitter

Figura 5.1: Differenze tra naive e status

rispetto al caso Twitter 5.1b.
L'idea è che il primo grafo presenta un basso numero di nodi U, eliminando una buona percentuale di nodi, quindi i costi dell'esecuzione sono dati principalmente dallo spostamento dei dati tra le memorie, che con questo miglioramento vengono tagliati.
Questa ipotesi è confermata dalla figura 5.2.



(a) Wiki-Talk (b) Wiki-Talk-not-U

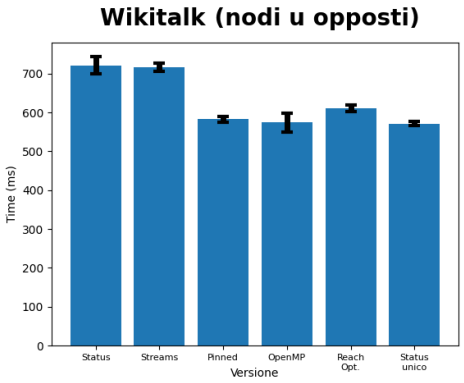
Figura 5.2: Differenze Wiki-Talk e Wiki-Talk-not-U in versione status

5.2.2 Da Status a Stream

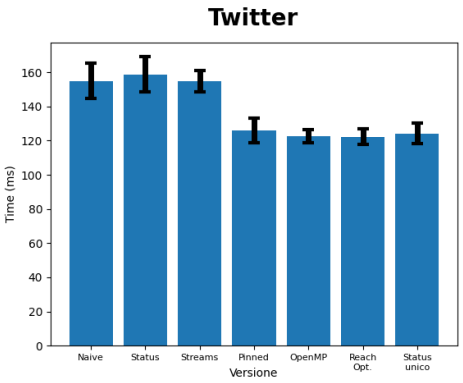
Come si può vedere sia in figura 5.1 che in figura 5.2 non c'è una grande differenza tra le 2 versioni. È anche vero che le operazioni che vengono svolte su diversi stream sono solamente le operazioni che hanno a che fare con la memoria e non si è parallelizzato null'altro (ad esempio esecuzione di kernel).

5.2.3 Da Stream a Pinned

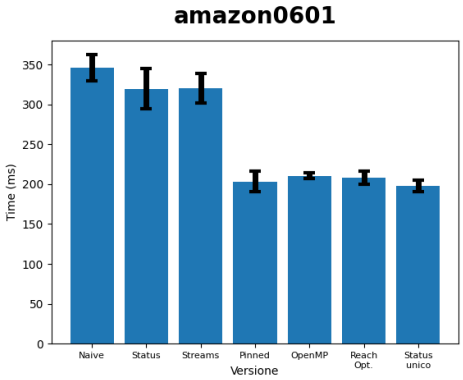
La versione Pinned è stato un punto di svolta per l'applicazione. Per i grafi not-U, che sono per la maggior parte molto più impegnativi in termini di risorse per il nostro codice, spesso non è stato possibile eseguire in tempi soddisfacenti le versioni precedenti, infatti per queste reti i barplot con i tempi partiranno da questa versione.
In figura 5.3 si può notare che nella maggior parte dei grafi si ha un netto miglioramento per quanto riguarda i tempi d'esecuzione.



(a) Wiki-Talk-not-U



(b) Wiki-Talk-not-U

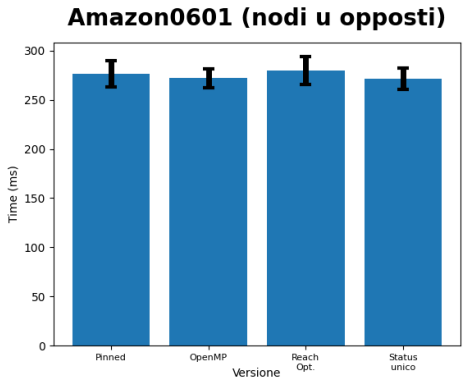


(c) Wiki-Talk

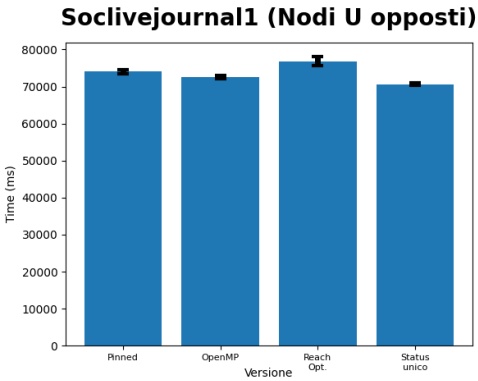
Figura 5.3: Differenze tra Stream e Pinned

5.2.4 Le ultime 3 versioni con OpenMP

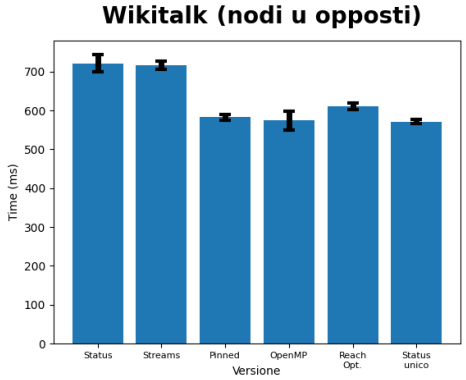
Nelle ultime 3 versioni non si è notato alcun netto miglioramento, le ipotesi fatte su queste versioni non hanno avuto un riscontro con i tempi d'esecuzione del codice. Era un risultato che ci potevamo aspettare per la prima versione di OpenMP, in quanto questa parallelizza solo le chiamate alla memoria che erano già asincrone e su diversi stream.
Una cosa che si può notare è che in quasi tutte le esecuzioni sembra essere il fatto che conviene usare la versione con il vettore status unico rispetto alla precedente, probabilmente per per evitare il bitwise-OR sui 2 vettori status e la copiatura di status in bw_status. Si conferma quest'ultima ipotesi con anche i rimanenti grafi not-U in figura 5.4.



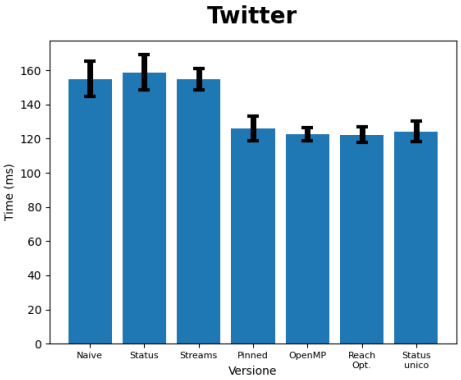
(a) Wiki-Talk-not-U



(b) Wiki-Talk-not-U



(c) Wiki-Talk-not-U



(d) Wiki-Talk-not-U

Figura 5.4: Differenze tra le ultime 3 versioni con OpenMP

6

Conclusioni

L'algoritmo nella sua versione parallela mostra un notevole incremento di prestazioni.
Infatti, per il codice seriale non è stato possibile sfruttare i grafi tenuti in considerazione per la fase di benchmark: l'esecuzione risulta essere talmente lenta da rendere la raccolta dei tempi un problema.
Come si evince dall'Analisi nel capitolo precedente, il vero incremento di prestazioni lo si ha con la versione in cui si implementa la memoria Page-locked o pinned, seguita dell'implementazione del vettore status. Contrariamente a quanto pensato, le altre versioni mostrano si un incremento di prestazioni, ma nulla di troppo rilevante da tenere troppo in considerazione.
In conclusione, per quanto riguarda questa applicazione si pensa che la versione parallela di questo algoritmo sia nettamente più efficiente e veloce della sua versione seriale, ma non ancora perfetta e potrebbe avere degli sviluppi futuri.

6.1 Possibili sviluppi futuri

Un possibile miglioramento dell'algoritmo potrebbe essere quello di controllare, ogni qualvolta viene identificata una SCC, se quest'ultima è valida oppure no, interrompendo l'esecuzione del programma. Questo potrebbe portare a diversi vantaggi e svantaggi in base al tipo di grafo in input.
Un miglioramento riguardo l'implementazione potrebbe essere quello di utilizzare la libreria "thrust" di CUDA per C++. Questa libreria offre strutture dati e algoritmi ottimizzati per l'uso in CUDA, che probabilmente aumenterebbe le prestazioni e la leggibilità del codice. In questo progetto non è stata utilizzata per riuscire a poter prendere confidenza con le API base di CUDA.

Bibliografia

[1] Jiri Barnat, Petr Bauch, Lubos Brim, e Milan Ceška. Computing strongly connected components in parallel on cuda. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pp. 544–555, 2011.

[2] Xuhao Chen, Cheng Chen, Jie Shen, Jianbin Fang, Tao Tang, Canqun Yang, e Zhiying Wang. Orchestrating parallel detection of strongly connected components on gpus. *Parallel Computing*, 78:101–114, 2018.

[3] Jure Leskovec e Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, giugno 2014.