ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea Triennale in Ingegneria e Scienze Informatiche

Implementazione CUDA dell'algoritmo di Bellman-Ford

Tesi di laurea in
High-Performance Computing

Relatore
Prof. Moreno Marzolla

Candidato
Filippo Barbari

Seconda Sessione di Laurea
Anno Accademico 2020-2021

Indice

Elenco delle figure

Elenco delle tabelle

Elenco degli algoritmi

Introduzione

L'obiettivo di questa tesi consiste nello sviluppo e analisi di implementazioni efficienti dell'algoritmo di Bellman-Ford su GPU. Allo scopo di svolgere un'analisi e una ricerca quanto più complete possibili, in questa tesi misuriamo le prestazioni di alcune implementazioni utilizzando vari tipi di test su tre macchine diverse. L'algoritmo di Bellman-Ford è stato uno dei primi ad essere proposto come soluzione del problema SSSP ed è probabilmente l'algoritmo che meglio si presta alla parallelizzazione massiva offerta da CUDA.

Questa tesi è strutturata come segue:

- il capitolo 1 è un'introduzione al calcolo parallelo: vengono presentati alcuni cenni storici e poi una spiegazione del funzionamento di una GPU CUDA;
- nel capitolo 2 vengono illustrati i concetti base riguardanti grafi e cammini minimi;
- il capitolo 3 contiene una descrizione dell'algoritmo di Bellman-Ford;
- nel capitolo 4, basandosi sull'analisi effettuata, si progettano tre implementazioni differenti;
- il capitolo 5 illustra brevemente le metriche di valutazione utilizzate;
- il capitolo 6 contiene i risultati dei test effettuati.

Capitolo 1

Calcolo parallelo

1.1 La Legge di Moore

Negli anni '60, Gordon Moore, futuro fondatore e direttore esecutivo di Intel, ipotizzò che il numero di componenti elettronici (transistor) contenuti all'interno di un chip sarebbe raddoppiato ogni 18 mesi circa[6]. Tale previsione si rivelò corretta per tutti gli anni '70 e i tempi si allungarono a partire dagli anni '80, per poi giungere all'epilogo di questa crescita intorno agli anni 2000. Nella figura 1.1 si può notare l'incremento del numero di transistor per millimetro quadrato, in funzione del tempo, su scala logaritmica. Per questo motivo, la potenza di calcolo dei processori a singolo core è cresciuta sempre più lentamente fino ad arrestarsi e si è dovuti ricorrere alle architetture parallele. Raddoppiare il numero di core all'interno di un chip può, in certi casi, raddoppiare le prestazioni e ridurre allo stesso tempo il consumo di energia ma richiede uno sforzo maggiore da parte del programmatore per lo sviluppo di codice parallelo. A causa di questa difficoltà e anche della scarsità di programmatori qualificati, al giorno d'oggi il software parallelo è molto meno presente dell'hardware parallelo.

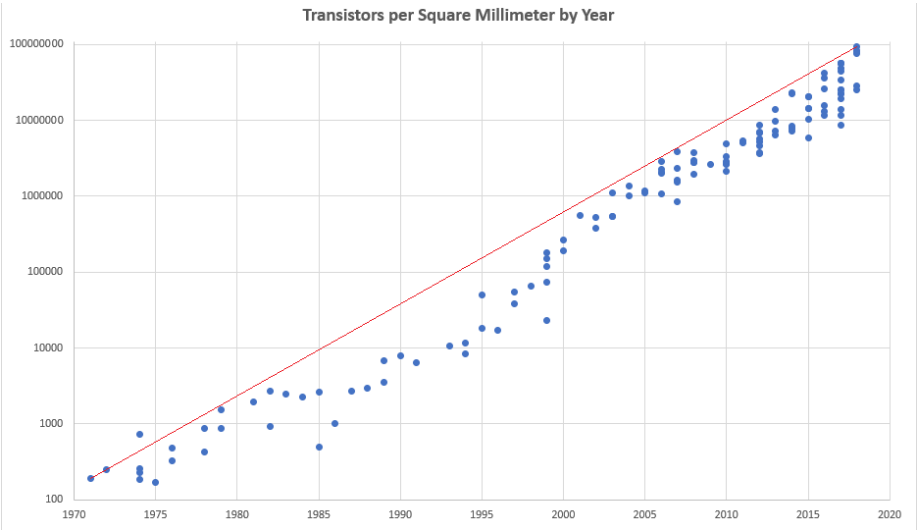

Figura 1.1: Rappresentazione della legge di Moore: l'aumento del numero di transistor in un chip su scala logaritmica rispetto al passare degli anni. Fonte: `https://medium.com/predict/moores-law-is-alive-and-well-eaa49a450188`

Intorno al 2007, sono apparse sul mercato le prime General-Purpose Graphic Processing Unit (GPGPU), ovvero le prime schede grafiche programmabili in maniera generica. Ciò non significa che, prima di quella data, non fossero programmabili ma si intende dire che erano utilizzabili solamente per applicazioni strettamente legate alla grafica. Le GPGPU hanno aperto una nuova era del calcolo parallelo, permettendo ai programmatori di sfruttare l'elevatissimo numero di core per computazioni estremamente parallelizzabili. La differenza principale, a livello hardware, tra una CPU multi-core e una GPGPU consiste infatti nel numero di core molto maggiore nella seconda, come mostra la figura 1.2. Questa abbondanza di risorse di calcolo ha cambiato anche il modo di strutturare e progettare i programmi paralleli.

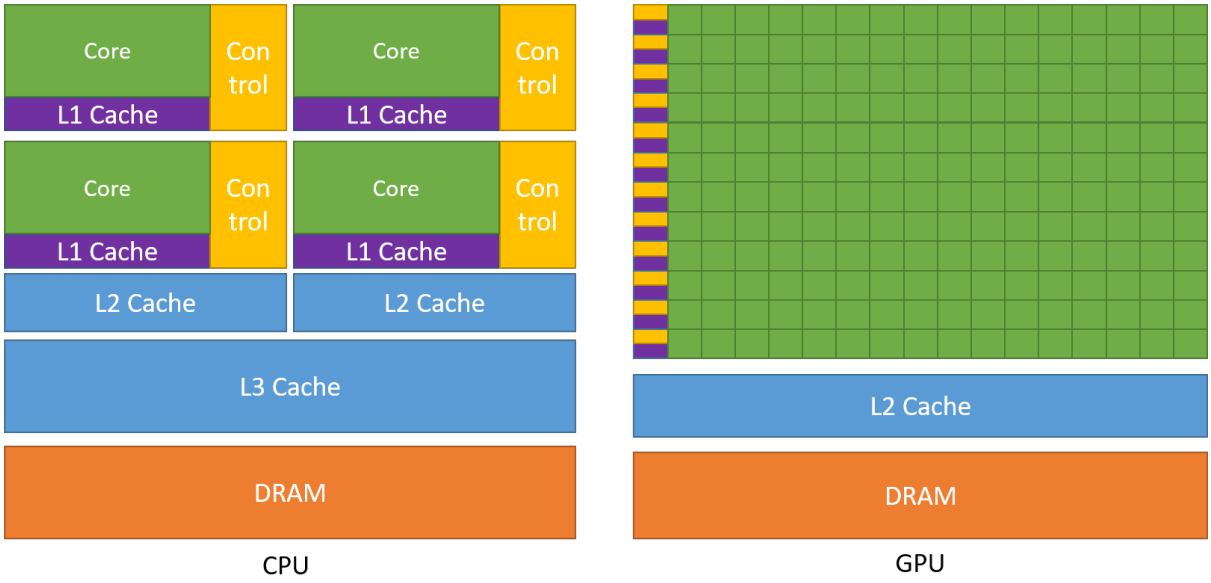

Figura 1.2: A sinistra l'architettura di una CPU multi-core, a destra l'architettura di una GPGPU. Fonte: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

1.2 GPU e CUDA

CUDA (acronimo di Compute Unified Device Architecture) è un'architettura hardware realizzata da NVIDIA, la cui prima versione è stata pubblicata nel 2007. Tramite CUDA è possibile scrivere programmi che eseguono parte del loro codice su una scheda grafica NVIDIA. CUDA si basa su alcuni semplici concetti: la CPU su cui viene lanciato il programma è chiamata *host* e può interagire con una o più schede grafiche NVIDIA, chiamate *device*. L'host può invocare alcune funzioni specifiche che vengono eseguite dal device. L'esecuzione di queste funzioni è asincrona, ovvero il controllo viene restituito subito all'host, senza attendere il termine dell'esecuzione. Tuttavia, esiste la possibilità di sincronizzare esplicitamente l'host con un device. Le funzioni più importanti sono i *kernel*, le uniche che possono essere scritte dal programmatore, che possono effettuare qualunque tipo di calcolo. Anche i kernel sono eseguiti in maniera asincrona ma, a differenza delle altre funzioni, richiedono due parametri obbligatori, ovvero il numero di blocchi e il numero di thread per ogni blocco da utilizzare per l'esecuzione.

CUDA raggruppa logicamente tutti i thread, ovvero le unità logiche di esecuzione, in blocchi che a loro volta sono raggruppati in una **griglia**. Ogni device può avere solo una griglia in esecuzione e ogni griglia può eseguire un singolo kernel. É possibile identificare i blocchi e i thread mediante un indice a 1, 2 o 3 dimensioni in base all'occorrenza. Nella figura 1.3 è mostrato l'esempio di indicizzazione per una griglia bidimensionale formata da blocchi bidimensionali di thread.

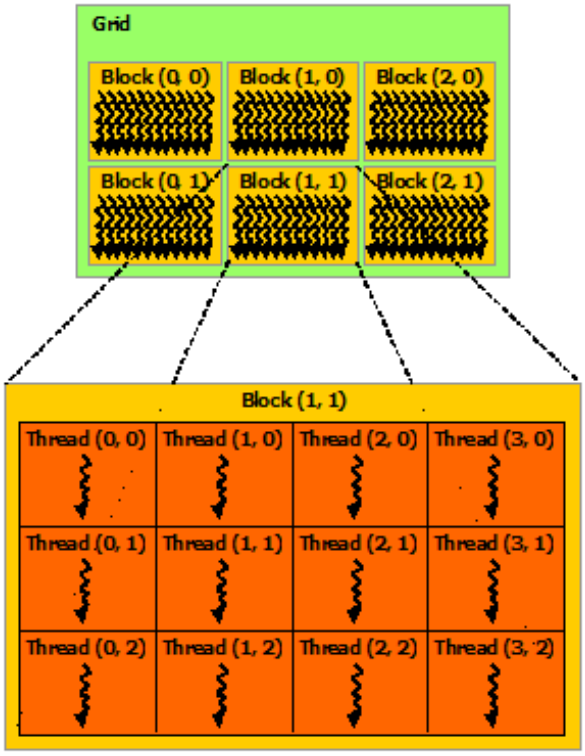

Figura 1.3: Suddivisione della griglia di una GPU CUDA in blocchi di thread. Fonte: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

La memoria di un device CUDA, come si nota dalla figura 1.4, è suddivisa in vari livelli:

- la **memoria globale** è la più voluminosa all'interno di un device (alcuni GB) ed è accessibile da tutti i thread di tutti i blocchi
- la **memoria texture** è ottimizzata per contenere dati di texture, ha la stessa accessibilità di quella globale anche se più piccola (alcuni MB)
- la **memoria delle costanti** è ottimizzata per il broadcast, è molto più piccola di quella globale (in genere 64KB) ed ha la stessa accessibilità
- la **memoria condivisa** è più piccola (alcuni MB) ma molto più veloce di quella globale però è accessibile solamente da thread all'interno dello stesso blocco
- la **memoria locale** è quella "privata" di ogni thread e per questo motivo è accessibile solamente da esso

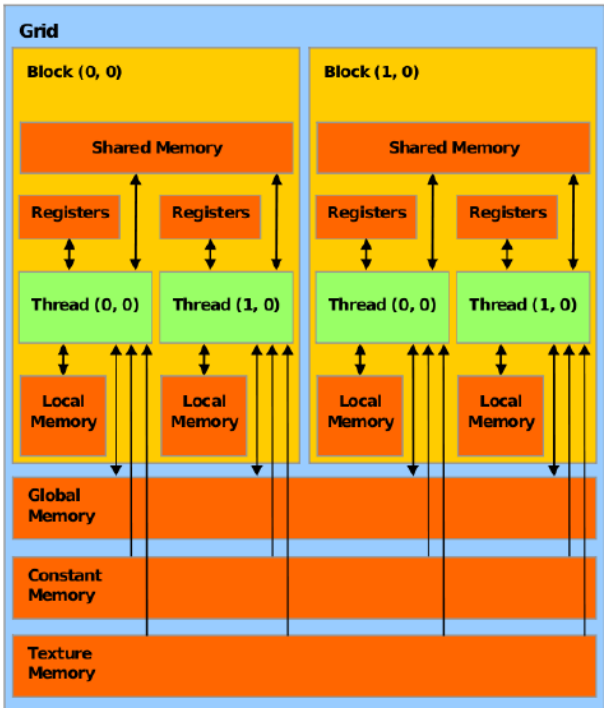

Figura 1.4: Struttura e gerarchia delle varie memorie all'interno di una GPU CUDA. Fonte: `https://www.researchgate.net/figure/CUDA-memory-model-from-7_fig1_4373162`

Lo **scheduling dei thread** in una scheda CUDA funziona in modo particolare. Innanzitutto, va precisato che, nonostante il numero di core (che in una scheda NVIDIA vengono chiamati Streaming Multiprocessors) vari tra una scheda e l'altra, non è possibile scegliere quali e quanti utilizzare per l'esecuzione di un kernel. Quando viene invocato un kernel, infatti, i thread richiesti vengono creati in memoria ma poi è l'hardware che sceglie automaticamente, durante l'esecuzione, la schedulazione migliore di questi thread sui SM a disposizione. La figura 1.5 mostra l'esempio di un programma eseguito su due GPU con numero diverso di SM.

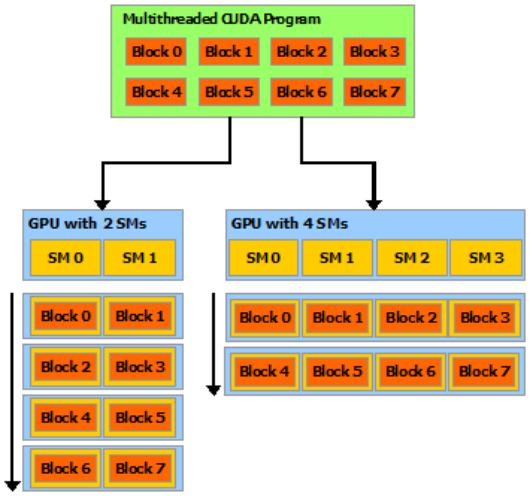

Figura 1.5: Schedulazione automatica dei thread in una GPU CUDA. Fonte: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

A causa di questa funzionalità, si utilizza il concetto di *thread warp*: un gruppo di thread (genericamente non più di 32) appartenenti allo stesso blocco, che devono eseguire la stessa porzione di codice, vengono schedulati tutti insieme sullo stesso SM ed eseguiti contemporaneamente. In caso di una biforcazione del flusso di controllo (ad esempio a causa di un costrutto `if-else` nel codice), viene calcolata la condizione per tutti i thread, dopodiché vengono eseguiti contemporaneamente solo i thread che hanno verificato la condizione (ovvero il ramo `if`), poi tutti gli altri (il ramo `else`). Nella figura 1.6 è mostrato l'esempio di una porzione di

codice in cui i primi 16 thread eseguono il blocco di codice A mentre gli altri eseguono il blocco B.

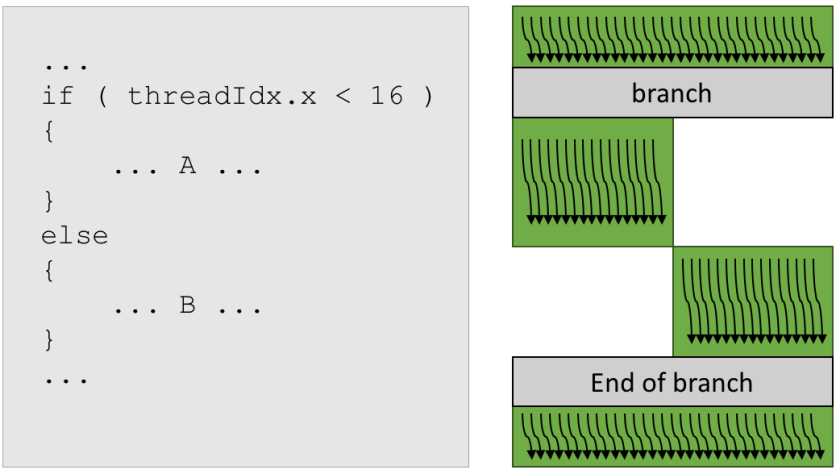

Figura 1.6: Esecuzione di un thread warp in caso di biforcazione.
Fonte: `https://subscription.packtpub.com/book/programming/9781788996242/3/ch03lvl1sec23/minimizing-the-cuda-warp-divergence-effect`

Va posta particolare attenzione anche al modo con cui i thread di una GPU CUDA accedono alla memoria. Infatti, la caratteristica più importante a riguardo è il **raggruppamento delle letture** in memoria: se un certo numero di thread, appartenenti allo stesso warp, deve accedere in locazioni di memoria adiacenti, viene effettuata una singola operazione di lettura che trasferisce tutti i dati richiesti dai thread nelle rispettive memorie locali. Può sembrare una funzionalità irrilevante, ma in realtà può determinare differenze significative di prestazioni in programmi che non memorizzano i dati in maniera corretta. In particolare, esistono due pattern generali di memorizzazione: Array-of-Structures (AoS) e Structure-of-Arrays (SoA). Il primo, il più utilizzato, consiste nel memorizzare le caratteristiche di uno stesso elemento (come le coordinate di un punto) in locazioni adiacenti di memoria. Il secondo, invece, memorizza ogni dato in un array separato. Ipotizziamo che tutti i thread appartenenti ad un warp debbano leggere un valore dal vettore x (con riferimento alla figura 1.7) in posizioni adiacenti. Se i dati in questione fossero memorizzati mediante SoA, tutti i valori richiesti si troverebbero all'interno dello stesso array adiacenti tra loro a partire da una certa posizione iniziale. La vicinanza in memoria di questi valori determina quindi un ridotto numero di letture effettive e un maggiore utilizzo del bus dati. Al contrario, se i dati fossero memorizzati tramite AoS, i valori sarebbero più lontani fra loro e quindi richiederebbero un maggior numero di letture in memoria.

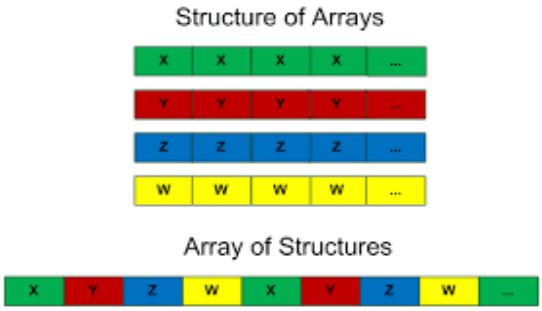

Figura 1.7: Differenza tra AoS e SoA.
Fonte: `https://arxiv.org/ftp/arxiv/papers/1907/1907.05560.pdf`

Capitolo 2

Teoria

2.1 Grafi

Un grafo è una struttura matematica in cui un insieme di elementi, chiamati nodi, sono collegati tra loro mediante archi. Un grafo è genericamente indicato come una coppia ordinata di insiemi (V, E) in cui V rappresenta l'insieme dei nodi ed E l'insieme degli archi. Ogni elemento di E consiste in una coppia ordinata di elementi di V, da cui segue che $E \subseteq V \times V$. Ad esempio, per il grafo 2.1 un nodo può essere A e un arco può essere (B, C). Una prima classificazione dei grafi consiste nel peso degli archi: un grafo si dice non pesato se i suoi archi hanno lo stesso peso (o, per semplicità, non hanno peso), altrimenti si definisce pesato. Un'ulteriore classificazione dei grafi consiste nell'orientamento degli archi: se almeno un arco ha una direzione si dice che il grafo è orientato (o diretto), altrimenti si dice non orientato (o indiretto).

Se il grafo non è orientato, E è un insieme di coppie non ordinate di nodi in cui (u, v) e (v, u) rappresentano lo stesso arco. Inoltre, è possibile considerare anche i cosiddetti "cappi" o "auto-anelli", ovvero archi della forma (u, u) che collegano un nodo con sé stesso. Se un grafo non possiede auto-anelli, allora si definisce semplice. In un grafo orientato semplice il numero massimo possibile di archi è pari a $|V| \cdot (|V|–1)$. Se il grafo non è orientato, il numero massimo di archi è $|V| \cdot (|V|–1)/2$. Se un grafo possiede esattamente il numero massimo di archi si dice completo. Una metrica molto utilizzata è la densità di un grafo, che si calcola come il rapporto tra il numero di archi e il numero massimo possibile di archi. Se un grafo ha "pochi" archi, dove per pochi si intende che il numero di archi è comparabile o inferiore al numero di nodi, si dice sparso, altrimenti denso. Gli archi diretti (u, v) si dicono uscenti per il nodo u ed entranti per il nodo v. Se un nodo non ha archi né entranti né uscenti, si dice isolato (nel grafo in figura 2.1, il nodo D è isolato).

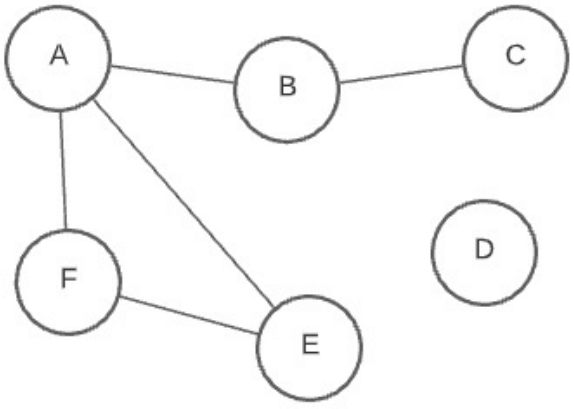

Figura 2.1: Un esempio di grafo non orientato e non pesato.

Un grafo si dice planare se è possibile rappesentarlo su di un piano senza che nessuno

dei suoi archi si intersechi con altri archi. Un grafo planare è genericamente sparso a causa del limitato numero di archi che un nodo può avere.

Un grafo può essere rappresentato in molti modi oltre a quello grafico. Il più conosciuto è la matrice di adiacenza che consiste in una matrice di dimensioni $|V| \times |V|$ in cui ogni elemento alla riga i e alla colonna j corrisponde al peso dell'arco diretto (i, j). Una matrice di adiacenza richiede spazio $O(|V|^2)$ in memoria. Un altro metodo molto utilizzato per rappresentare grafi è la lista di adiacenza in cui si memorizza una lista di nodi e ogni nodo memorizza la lista dei propri archi. Una lista di adiacenza richiede spazio $O(|V| + |E|)$. Infine, è possibile memorizzare un grafo come una lista di archi in cui ognuno contiene un riferimento ai nodi sorgente e destinazione e il proprio peso. Questo metodo richiede spazio $O(|E|)$ in memoria ma, essendo basato solamente sugli archi, non considera i nodi isolati.

2.2 Cammini minimi e problema SSSP

In un grafo, un qualunque percorso che congiunge due nodi è detto cammino ed è rappresentabile come una lista ordinata di nodi. É importante notare la possibilità che uno stesso nodo sia presente più volte all'interno dello stesso cammino: quando ciò accade si parla di cicli (che approfondiremo di seguito). Il primo e l'ultimo nodo di questa lista ordinata sono chiamati rispettivamente nodo sorgente e nodo destinazione. In un grafo non pesato, il costo (o peso) di un cammino è pari al numero di nodi nella lista mentre in uno pesato è pari alla somma dei pesi degli archi coinvolti. Un cammino è detto minimo se non esiste un altro cammino nello stesso grafo, con uguali nodi sorgente e destinazione, con costo strettamente minore. Ad esempio, nel grafo della figura 2.1, un cammino dal nodo B al nodo E può essere (B,A,F,E) mentre (B,A,E) è un cammino minimo. Va precisato che nello stesso grafo possono esistere cammini minimi diversi che collegano la stessa coppia di nodi.

Un ciclo è un cammino che comincia e termina nello stesso nodo. Il costo (o peso) di un ciclo è pari alla somma dei pesi degli archi tra i nodi da cui è composto (nei grafi non pesati, invece, è pari al numero di nodi). Nel grafo della figura 2.2, ad esempio, (B,C,D,B) è un ciclo di costo 5 mentre (A,D,F,E,C,A) è un ciclo di costo 23. É possibile che in un grafo pesato siano presenti archi di costo negativo: sebbene siano poche le situazioni reali modellabili in questo modo, è molto importante determinare la presenza di cicli di costo negativo. Se in un grafo è presente un ciclo di costo negativo, allora ogni cammino minimo (in cui è possibile raggiungere tale ciclo dal nodo sorgente) avrà costo $-\infty$.

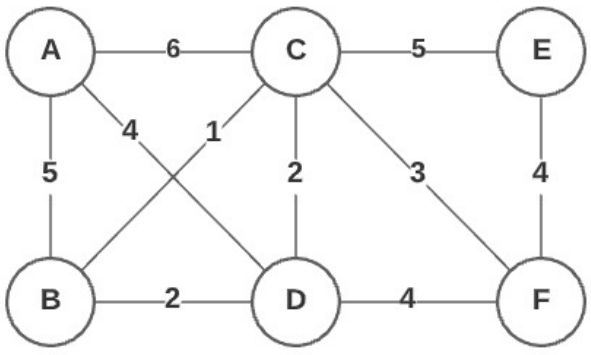

Figura 2.2: Un esempio di grafo pesato non orientato.

Un cammino minimo rispetta la cosiddetta proprietà della sottostruttura ottima, ovvero ogni sottocammino di un cammino minimo è a sua volta un cammino minimo. Più formalmente, se abbiamo un cammino minimo C dal nodo u al nodo v in cui è incluso anche il nodo t, allora sappiamo che tutti i nodi in C da u a t costituiscono

un cammino minimo e anche tutti i nodi in C tra t e v. Prendiamo come esempio il grafo della figura 2.2. L'unico cammino minimo dal nodo A al nodo F è (A,D,F), di costo 8. É facile notare come il cammino (A,D) sia l'unico cammino minimo tra i nodi A e D e lo stesso vale per il cammino (D,F).

Dei tanti problemi che si possono applicare ad un grafo, uno dei più studiati è il problema dei cammini minimi da singola sorgente (abbreviato SSSP). Dato un grafo $G = (V, E)$ e un nodo sorgente s, il problema consiste nel trovare tutti i cammini minimi in G che partono da s e terminano in ogni altro nodo raggiungibile. Viene chiamato così per differenziarlo dalla sua versione generalizzata, l'All-Pairs Shortest Path (APSP) che invece consiste nel trovare il cammino minimo per ogni coppia di vertici. Il risultato del SSSP consiste in un array di distanze dal nodo sorgente e in un array di predecessori tramite cui è possibile costruire l'albero dei cammini minimi.

Capitolo 3

L'algoritmo di Bellman-Ford

3.1 Descrizione dell'algoritmo

Proposto per primo dal matematico Alfonso Shimbel nel 1955 [1], questo algoritmo è chiamato Bellman-Ford perché Richard Bellman, nel 1958 [3], e Lester Ford Jr., nel 1956 [2], pubblicarono due articoli in cui presentavano lo stesso algoritmo. Ford lo propose come soluzione al problema logistico di trasportare un certo numero di beni da un punto A ad un punto B. Invece, Bellman lo propose come soluzione al problema più realistico di trovare il percorso più veloce attraverso una rete di città collegate tra loro. Edward F. Moore pubblicò poi una variante [4] che riduce drasticamente il numero di rilassamenti da effettuare ad ogni iterazione. A causa di questa pubblicazione del 1959, questo algoritmo viene anche chiamato di Bellman-Ford-Moore. Nel 1970 Yen [5] ha pubblicato una complicata ottimizzazione che riduce il numero totale di iterazioni da $|V|-1$ a $|V|/2$. Nel 2012 Bannister e Eppstein [7] hanno migliorato la variante di Yen riducendo il numero di iterazioni a $|V|/3$.

La versione che riportiamo di seguito e che verrà implementata è quella originale proposta prima da Lester Ford Jr. e poi da Richard Bellman.

Bellman-Ford 1: L'algoritmo di Bellman-Ford

```
   Input: G, w, s
   Result: FALSE se il grafo G contiene cicli di costo negativo, TRUE altrimenti
 1 Initialize-Single-Source(G,s);
 2 for i=1 to |G.V| - 1 do
 3 |   foreach edge (u,v) ∈ G.E do
 4 |   |   Relax(u,v,w);
 5 |   end
 6 end
 7 foreach edge (u,v) ∈ G.E do
 8 |   if v.d > u.d + w(u,v) then
 9 |   |   return FALSE;
10 |   end
11 end
12 return TRUE;
```

Come è possibile notare dallo pseudocodice, il corpo effettivo dell'algoritmo è contenuto nelle righe 2-6. Difatti, `Initialize-Single-Source` (algoritmo 2) è una procedura

molto semplice che imposta prima la distanza di ogni nodo dalla sorgente a $+\infty$, poi pone a 0 la distanza del nodo sorgente. Le istruzioni alle righe 7-11 consistono, invece, in un ciclo per verificare o meno la presenza di cicli di costo negativo nel grafo di input.

Initialize-Single-Source 2: La procedura di inizializzazione del grafo

```
   Input: G, s
1  foreach vertex v ∈ G.V do
2  |  v.d = +∞;
3  |  v.π = NIL;
4  end
5  s.d = 0;
```

Il corpo dell'algoritmo esegue un'operazione di rilassamento (o meglio, invoca la procedura Relax) su ogni arco del grafo per $|V|$–1 iterazioni. Questo limite nasce dal numero di rilassamenti effettuati nel caso pessimo. Consideriamo, ad esempio, il grafo della figura 3.1 in cui tutti i nodi sono collegati in linea: ognuno è collegato solo con il precedente ed il successivo tranne i nodi A ed I che hanno un solo arco. Se la sorgente è il nodo A, ad ogni iterazione verrà rilassato un singolo arco. Nel grafo in questione sono presenti esattamente $|V|$–1 archi e, quindi, l'algoritmo dovrà effettuare altrettante iterazioni.

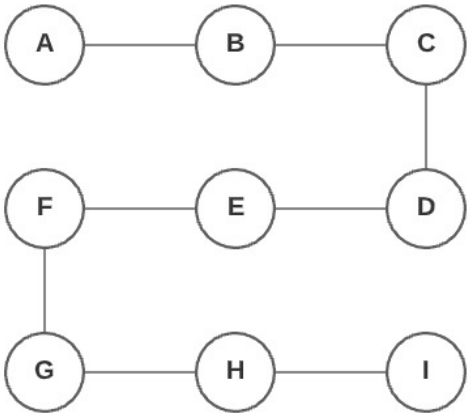

Figura 3.1: Un esempio di grafo lineare.

Come si nota, la procedura Relax (algoritmo 3) viene invocata un gran numero di volte e, su determinati tipi di grafi, non risulta spesso in un effettivo rilassamento. Per questo motivo, sono state proposte alcune ottimizzazioni per ridurre sia il numero complessivo di iterazioni che il numero di rilassamenti in ogni iterazione.

La procedura Relax richiede in input due nodi (u e v) e la funzione w che restituisce, dati due nodi, il peso dell'arco tra i due. Questa procedura non consiste in un effettivo rilassamento dell'arco (u, v) ma è un tentativo di rilassamento: infatti verifica se sia possibile una riduzione della distanza del nodo v dalla sorgente. In parole povere, la procedura Relax verifica se, aggiungendo u come ultimo nodo del cammino fino a v, il costo totale del cammino diminuisce.

Relax 3: La procedura di rilassamento di un arco

```
   Input: u, v, w
1  if v.d > u.d + w(u,v) then
2  |  v.d = u.d + w(u,v);
3  |  v.π = u;
4  end
```

L'algoritmo di Bellman-Ford riesce a rilevare i cicli di costo negativo verificando se sia possibile rilassare un arco alla fine delle iterazioni di rilassamento. Se in un grafo è presente un ciclo di costo negativo, almeno un arco di quel ciclo verrà rilassato ad ogni

iterazione a causa della presenza di almeno un arco di peso negativo. Come già spiegato nella sezione 2.2, quando è presente un ciclo di costo negativo è sempre possibile ridurre il costo di un cammino che passa per esso. Per questo motivo, se alla fine delle iterazioni di rilassamento, almeno un arco è ancora rilassabile, allora vuol dire che il grafo contiene un ciclo di costo negativo e l'arco rilassabile ne fa parte.

3.2 Complessità computazionale

La procedura `Initialize-Single-Source` esegue due operazioni di costo $O(1)$ in ciascuna delle $|V|$ iterazioni e infine una singola istruzione di costo $O(1)$. Dunque, la complessità temporale complessiva di questa procedura è $O(|V|)$.

La procedura Relax esegue un confronto e, se verificato, esegue due istruzioni di costo $O(1)$. Complessivamente, dunque, la procedura Relax ha complessità temporale $O(1)$.

Le istruzioni tra le righe 2-6 dell'algoritmo sono un doppio ciclo annidato in cui, per ognuna delle $|V|$–1 iterazioni, viene invocata la procedura Relax su ogni arco, dunque per $|E|$ volte. Quindi, questa porzione dell'algoritmo ha complessità temporale $O(|V| \cdot |E|)$. Il ciclo alle righe 7-11 ha complessità temporale $O(|E|)$ poiché esegue un confronto di costo costante per $|E|$ volte. Complessivamente, l'algoritmo ha costo computazionale $O(|V| \cdot |E|)$.

Riguardo alla memoria, l'algoritmo di Bellman-Ford richiede complessivamente $O(|V|)$ spazio poiché necessita di memorizzare un valore di distanza e un riferimento al predecessore per ogni nodo.

3.3 Dove parallelizzare

Analizziamo ora le dipendenze tra dati contenute nell'algoritmo. Ognuna delle iterazioni del ciclo alle righe 2-6 è dipendente dal risultato dell'iterazione precedente. Ogni iterazione del ciclo tra le righe 3-5, invece, presenta una dipendenza particolare: ogni rilassamento di un arco (u, v) dipende dal risultato dell'ultimo rilassamento di un arco diretto verso v.

Alternativamente, questo algoritmo può essere modificato scambiando l'ordine dei due cicli annidati. Questa variazione lo rende errato solamente se seriale. Infatti, l'algoritmo seriale effettua un gran numero di tentativi di rilassamento. Una versione parallelizzata in questo modo, invece, effettuerebbbe su ogni arco un rilassamento. Dopodiché, eseguirebbe un numero massimo di $|V|$–1 tentativi di aggiornamento della distanza del nodo destinazione.

3.4 Precisazioni

Siccome, per lo scopo di questa tesi, non è necessario che l'algoritmo costruisca l'albero dei cammini minimi, non è stata implementata questa funzionalità. Con riferimento allo pseudocodice, questa modifica si traduce nella rimozione della riga 3 dalle procedure Relax (algoritmo 3) e di inizializzazione (algoritmo 2). Inoltre, i test utilizzati per le misurazioni non presentano né cicli nè archi di costo negativo. Per questo motivo, l'algoritmo implementato produce solamente un array di distanze di ogni nodo dalla sorgente.

Capitolo 4

Progettazione

Basandoci sull'analisi effettuata nel capitolo precedente, proponiamo di seguito tre implementazioni differenti dell'algoritmo di Bellman-Ford.

4.1 La versione `bf0`

La prima versione sfrutta ampiamente il parallelismo massivo offerto da CUDA, memorizzando il grafo di input come un array di archi e parallelizzando il ciclo alle righe 3-5 dell'algoritmo 1, utilizzando un thread per ogni arco del grafo. Ad ogni iterazione, ogni thread invoca la procedura Relax (algoritmo 3) su un singolo arco. Infine, tutti i thread si sincronizzano prima di eseguire l'iterazione successiva.

Il problema principale di questa versione consiste negli aggiornamenti concorrenti della distanza del nodo v. Come già accennato nella sezione 3.3, possiamo interpretare il lavoro svolto da ogni thread come $|V|$–1 tentativi di aggiornamento di un singolo valore. Per questo motivo, riteniamo possa essere comunque corretta una versione che non utilizza meccanismi di mutua esclusione. Tale versione sarà chiamata `bf0-none`. Ovviamente proponiamo anche la sua controparte `bf0-mutex` perché garantisce la correttezza del risultato al prezzo dell'impiego di operazioni atomiche. Nella tabella 4.1 sono riassunte le caratteristiche delle varie versioni implementate di questo algoritmo.

Nome versione	**Mutex**	**AoS/SoA**	**Memoria condivisa**
`bf0-none-aos-nosh`	no	AoS	no
`bf0-none-aos-sh`	no	AoS	si
`bf0-none-soa-nosh`	no	SoA	no
`bf0-none-soa-sh`	no	SoA	si
`bf0-mutex-aos-nosh`	si	AoS	no
`bf0-mutex-aos-sh`	si	AoS	si
`bf0-mutex-soa-nosh`	si	SoA	no
`bf0-mutex-soa-sh`	si	SoA	si

Tabella 4.1: Tabella riassuntiva delle caratteristiche delle versioni dell'algoritmo `bf0`

4.2 Rimuovere i mutex: `bf1`

Il problema degli aggiornamenti concorrenti può essere risolto senza impiego di mutex ma utilizzando un numero molto inferiore di thread. Se il grafo di input fosse memorizzato mediante una lista di adiacenza in cui ogni nodo ha un riferimento solo ai nodi raggiungibili tramite archi uscenti, allora sarebbe possibile parallelizzare i rilassamenti da effettuare su ogni nodo. Nel dettaglio, fissato un nodo u, utilizziamo un thread per effettuare un rilassamento su ogni arco uscente da u. Com'è evidente, questa versione non presenta il problema di aggiornamenti concorrenti, ma utilizza molti meno thread. Probabilmente, le prestazioni di questa versione potrebbero essere comparabili alle prestazioni dell'algoritmo `bf0` solamente su grafi piccoli e densi, in cui ogni nodo ha un elevato numero di archi uscenti. Nella tabella 4.2 sono riassunte le caratteristiche delle varie versioni implementate dell'algoritmo `bf1`.

Nome versione	**AoS/SoA**	**Memoria condivisa**
`bf1-aos-nosh`	AoS	no
`bf1-aos-sh`	AoS	si
`bf1-soa-nosh`	SoA	no
`bf1-soa-sh`	SoA	si

Tabella 4.2: Tabella riassuntiva delle caratteristiche delle versioni dell'algoritmo `bf1`

4.3 Sfruttare più thread: `bf2`

Il difetto maggiore della versione `bf1` è il suo ridottissimo numero di thread. É possibile, però, sfruttare un numero maggiore di thread al prezzo di introdurre nuovamente aggiornamenti concorrenti nell'algoritmo. In questo caso il grafo dovrebbe essere memorizzato mediante una lista di adiacenza in cui ogni nodo memorizza il riferimento ai nodi che possono raggiungerlo, dunque gli archi entranti. Con questa modifica, il numero di thread utilizzabili aumenta considerevolmente perché è possibile effettuare tutti i rilassamenti degli archi entranti, su ogni nodo del grafo, contemporaneamente. Per questi motivi, l'algoritmo `bf2` utilizza un blocco di thread per ogni nodo del grafo. Nella tabella 4.3 sono riassunte le caratteristiche delle varie versioni implementate di questo algoritmo.

Nome versione	**AoS/SoA**	**Memoria condivisa**
`bf2-aos-nosh`	AoS	no
`bf2-aos-sh`	AoS	si
`bf2-soa-nosh`	SoA	no
`bf2-soa-sh`	SoA	si

Tabella 4.3: Tabella riassuntiva delle caratteristiche delle versioni dell'algoritmo `bf2`

4.4 Dettagli implementativi

Di seguito si elencano alcuni dettagli o limiti non trascurabili presenti nelle versioni implementate.

Come già spiegato nel capitolo 3, ogni algoritmo che risolve il problema SSSP necessita dell'indice del nodo sorgente. In tutte le versioni implementate, il nodo sorgente è il nodo di indice 0.

Inoltre, in tutte le versioni implementate il peso di un arco è memorizzato come un numero intero. L'unica eccezione è costituita dalle versioni `bf0-mutex` in cui il peso di un arco è implementato mediante un `float` a 32 bit. Ciò è dovuto alla necessità di implementare il meccanismo di mutua esclusione: rispetto ad un'esplicita acquisizione e rilascio di una variabile mutex si è preferito utilizzare la funzione CUDA `atomicCAS`. L'utilizzo di un numero intero per il peso di un arco si è tradotto però in un overflow durante l'esecuzione di un rilassamento. Per questo motivo, in tutte le versioni (esclusa `bf0-mutex`) è stato implementato un controllo che non produce overflow e che non ha più la stessa struttura della procedura Relax (algoritmo 3). L'algoritmo 4 rappresenta lo pseudocodice della procedura Relax effettivamente implementata.

Relax-No-Overflow 4: La procedura di rilassamento di un arco che non produce overflow

```
   Input: u, v, w
1  if v.d > u.d and v.d - u.d > w(u,v) then
2  |  v.d = u.d + w(u,v);
3  end
```

Tutti gli algoritmi che utilizzano la memoria condivisa ottimizzano gli accessi in lettura all'array dei dati di input (nel caso si usi AoS, altrimenti sui vari array di input per SoA). Nonostante l'array `distances` sia acceduto più volte, esso richiede anche un accesso in scrittura e tutti i suoi accessi in lettura non sono allineati. Come già spiegato nella sezione 1.2, questi fattori contribuiscono ad un peggioramento significativo delle prestazioni. Tale peggioramento è stato rilevato mediante misurazioni empiriche i cui risultati non sono riportati in questa tesi.

Gli algoritmi `bf0` utilizzano un thread per ogni arco del grafo. Nella maggior parte delle schede CUDA il numero massimo di thread per blocco è pari a 1024 e il numero massimo di blocchi utilizzabili è pari a 2^{31}–1 [10]. Per questo motivo, utilizzare come input un grafo con un numero di archi maggiore di $(2^{31}–1)\cdot 1024$ genererebbe un comportamento imprevisto.

Gli algoritmi `bf1` utilizzano un singolo blocco di 1024 thread che opera su ogni nodo del grafo, ma non presentano limitazioni in questo senso.

Gli algoritmi `bf2`, come spiegato nella sezione 4.3, utilizzano un blocco di thread per ogni nodo. Dunque, se venissero utilizzati su grafi con un numero di nodi maggiore di 2^{31}–1, potrebbero generare un comportamento imprevisto.

Gli algoritmi `bf1`, come già spiegato nella sezione 4.2, memorizzano il grafo come una lista di adiacenza basata sugli archi uscenti. Dunque, se fossero utilizzati su grafi in cui sono presenti nodi senza tali archi potrebbero generare un comportamento imprevisto. Lo stesso problema è presente per gli algoritmi `bf2` riguardo agli archi entranti. Gli algoritmi `bf0`, invece, produrranno un valore di $+\infty$ per ogni nodo irraggiungibile.

Capitolo 5

Metriche di valutazione

5.1 Speedup

Lo speedup è la principale metrica di misurazione di un programma parallelo e rappresenta l'accelerazione che tale programma ha ricevuto tramite la parallelizzazione. In particolare, lo speedup di un programma parallelo, eseguito con p core, si calcola come:

$$S(p) = \frac{T_s}{T(p)} \tag{5.1}$$

in cui T_s rappresenta il tempo di esecuzione del programma seriale (la "versione originale") e $T(p)$ il tempo di esecuzione del programma parallelo utilizzando p core.

Idealmente, si vorrebbe ottenere uno **speedup lineare** nel proprio programma, ovvero $S(p) = p$ indipendentemente dal numero di processori utilizzati. Anche se può sembrare controintuitivo, raddoppiando il numero di processori impiegati non si ottiene mai un dimezzamento perfetto del tempo di esecuzione; ciò è dovuto all'aumentare dei tempi per la sincronizzazione e la comunicazione, oltre all'*overhead* per l'allocazione della memoria necessaria. Per questo motivo, lo speedup $S(p)$ è sempre inferiore a p. Nonostante ciò, esistono alcuni rari casi in cui è possibile misurare empiricamente uno **speedup superlineare**, ovvero $S(p) > p$. In questi casi, la misurazione indica che raddoppiando il numero di processori il tempo di esecuzione si è più che dimezzato. Questo fenomeno, più unico che raro, è dovuto nella maggior parte dei casi alle elevate velocità e dimensioni delle cache integrate all'interno dei processori utilizzati. Ad esempio, se i dati di un programma fossero abbastanza piccoli da poter risiedere interamente nella cache, il programma stesso otterrebbe una grande accelerazione per i tempi di accesso alla memoria.

Come è possibile notare dalla formula 5.1, lo speedup di un programma parallelo si osserva misurando il tempo di esecuzione man mano che si aumenta il numero di processori utilizzati. Questo non è possibile con un programma CUDA perché il programmatore non ha potere decisionale sul numero di core da utilizzare né sulla schedulazione dei thread. L'unico aspetto della computazione che può controllare è la dimensione della *grid*, intesa come numero di blocchi e numero di thread per ciascun blocco. Per questo motivo, non ha senso misurare lo speedup di un'applicazione CUDA semplicemente aumentando il numero di thread. Tuttavia, in questa tesi misuriamo un singolo valore di accelerazione per ogni algoritmo per meglio confrontare tra loro le varie versioni.

5.2 Throughput

Il throughput (o produttività) è una metrica molto più rappresentativa delle prestazioni di un'applicazione CUDA. Esso consiste nel numero di "elementi" processati in un secondo, oppure di "unità di lavoro" eseguite nello stesso tempo. Cosa si intenda per unità di lavoro, però, dipende dal problema, dal tipo di dati in input e dall'implementazione di un certo algoritmo. Ad esempio, per un'applicazione che applica filtri di colore ad un'immagine il throughput può essere misurato come numero di pixel modificati al secondo. Un altro esempio può essere un'algoritmo che cerca la mossa migliore in una partita di scacchi per il quale la produttività si può misurare come il numero di stati della scacchiera analizzati ogni secondo. Nel nostro caso, il throughput è misurato come il numero di rilassamenti di un arco effettuati ogni secondo.

Ogni algoritmo ha un limite massimo di throughput che è possibile raggiungere solamente aumentando le dimensioni dei dati di input. Quel valore rappresenta l'effettiva produttività di un dato algoritmo. Per questo motivo, è molto importante misurare le prestazioni di un'applicazione parallela impiegando test voluminosi.

Capitolo 6

Valutazione delle prestazioni

6.1 Descrizione dei test

I primi cinque test sono rappresentazioni, in forma di grafo, delle mappe stradali della città di Roma e di quattro paesi degli USA. Questi file sono reperibili ai link [8] e [9]. Gli altri quattro test sono, invece, grafi sintetici casuali a densità crescente, realizzati tramite il programma `graphgen` (disponibile nel repository [1]). L'impiego di questi ultimi si è rivelato necessario, al fine di una ricerca più completa possibile, dal momento che i primi cinque test non sono altro che grafi planari e, in quanto tali, hanno forti limitazioni sul numero di archi.

Entrando più nello specifico, tutti i grafi elencati sono orientati, pesati e semplici. Nessuno di questi grafi presenta archi di costo negativo. Solamente i primi 5 grafi sono planari. L'ultimo grafo, `100`, è l'unico completo. In nessuno di questi grafi sono presenti nodi isolati. La tabella 6.1 contiene un breve riassunto delle caratteristiche di ogni test.

Nome test	**Numero nodi**	**Numero archi**	**Descrizione**
rome	3353	8859	Mappa metropolitana di Roma
DE	49109	119744	Mappa stradale del Delaware
VT	97975	212979	Mappa stradale del Vermont
ME	194505	425708	Mappa stradale del Maine
NV	261155	618175	Mappa stradale del Nevada
025	1000	249808	Grafo casuale con densità 0.25
050	1000	499574	Grafo casuale con densità 0.5
075	1000	749262	Grafo casuale con densità 0.75
100	1000	999000	Grafo casuale completo

Tabella 6.1: Tabella riassuntiva delle caratteristiche dei test utilizzati.

6.2 Specifiche hardware

Al fine di svolgere una ricerca quanto più completa possibile, si è scelto di effettuare le misurazioni su tre macchine dotate di schede grafiche NVIDIA diverse tra loro. Nella

[1] `https://github.com/Ledmington/bellman-ford-cuda`

tabella 6.2 sono riportate le specifiche hardware e software delle macchine utilizzate. Le macchine saranno identificate tramite il nome della propria scheda grafica.

	940MX	**GTX1070**	**RTX2080**
Sistema Operativo	Windows	Ubuntu	Windows
Versione SO	10.0.19042.1165	16.04.7	10.0.19043.1165
Compilatore C	CL 19.29.30133	GCC 5.4.0	CL 19.29.30040
Nome CPU	Intel Core i5-7200U	Intel Xeon E5-2603 v4	Intel Core i7-8700
Frequenza CPU	2,5 - 2,7 GHz	1,2 - 1,7 GHz	3,2 GHz
Cores fisici	2	6	6
Hyper-Threading	si	no	si
L1 cache	128 KB	32 KB	384 KB
L2 cache	512 KB	256 KB	1,5 MB
L3 cache	3 MB	15 MB	12 MB
RAM totale	8 GB	64 GB	32 GB
Frequenza RAM	2133 MHz	2400 MHz	2133 MHz
Tipologia RAM	DDR4	DDR4	DDR4
Nome GPU	GeForce 940MX	GeForce GTX1070	GeForce RTX2080
CUDA Driver	11.4	10.1	11.4
CUDA Runtime	11.1	8.0	11.1
CUDA capability	5.0	6.1	7.5
Architettura GPU	Maxwell	Pascal	Turing
Memoria globale	2 GB	8 GB	8 GB
Tipologia Memoria	GDDR5	GDDR5	GDDR6
Frequenza Memoria	2505 MHz	4004 MHz	7000 MHz
Ampiezza Bus	64 bit	256 bit	256 bit
Numero SM	3	15	46
CUDA core totali	384	1920	2944
Frequenza core	1189 MHz	1797 MHz	1860 MHz

Tabella 6.2: Specifiche hardware e software delle macchine utilizzate per le misurazioni.

6.3 Prestazioni dell'algoritmo seriale

Nella tabella 6.3 sono riportati i tempi di esecuzione e i valori di throughput misurati sull'algoritmo `bf-serial` sulla CPU della macchina 940MX. Il tempo di esecuzione è espresso in secondi. Il throughput è espresso in milioni di operazioni di rilassamento effettuate al secondo (10^6 relax/s).

6.4 Precisazioni preliminari

Tutti gli algoritmi producono il risultato corretto su ogni grafo utilizzato, tranne l'algoritmo `bf2-soa-sh` che produce valori errati con il grafo `rome`. Per questo motivo, tale

Nome test	Wall Clock Time	Throughput
rome	0,38	78
DE	81,45	75
VT	217,06	95,8
ME	874,61	94
NV	1743,02	92
025	2,39	104,2
050	4,55	109,4
075	6,8	109,8
100	8,12	122,4

Tabella 6.3: Prestazioni dell'algoritmo seriale su CPU.

algoritmo è considerato errato e non ne saranno riportate le misurazioni delle prestazioni.

Inoltre, per gli algoritmi `bf1` e `bf2` non sono riportate le misurazioni per alcuni test come `NV`, `ME` e in alcuni casi `VT`. Non si è ritenuto utile riportarle in quanto i tempi di esecuzione, superiori ad un'ora, non sono comparabili con quelli raggiunti dagli algoritmi `bf0`. Allo stesso modo, per gli stessi algoritmi e gli stessi test non sono riportati neanche i valori di throughput.

6.5 Speedup

Di seguito sono riportati i valori di speedup di ogni versione, misurati sui test a disposizione e su ogni macchina, rispetto ai valori riportati nella tabella 6.3.

Innanzitutto, è possibile notare l'enorme differenza di speedup in generale tra la scheda 940MX e le altre. Evidente in particolar modo nelle figure 6.1 e 6.2, ciò è dovuto soprattutto alla superiore potenza computazionale determinata dalle risorse hardware. Come specificato nella tabella 6.2, non solo la frequenza di clock dei core CUDA è maggiore ma il loro numero è moltiplicato nelle schede GTX1070 e RTX2080.

Nonostante i valori di speedup per gli algoritmi `bf0` siano molto simili tra di loro, si nota un modesto miglioramento da una qualunque versione `nosh` alla corrispettiva `sh`. I valori maggiori si ottengono su versioni che utilizzano AoS anche se sono molto più variabili rispetto ai valori misurati sugli algoritmi che utilizzano SoA. I valori misurati sul grafo `rome` sono estremamente ridotti (speedup compreso tra 1 e 2) a causa delle sue dimensioni.

Riguardo agli algoritmi `bf1`, si nota subito dai grafici della figura 6.3 che la scala dello speedup ottenuto è molto inferiore rispetto a quanto ottenuto con gli algoritmi `bf0`. Infatti, solamente sui grafi sintetici si ottiene uno speedup significativo, mentre sui grafi "reali" il valore misurato è molto vicino allo 0. Ciò era prevedibile a causa del ridottissimo numero di thread impiegati che, come già accennato nella sezione 4.2, si sarebbe rivelato utile solamente su grafi piccoli e densi. Anche qui si osserva, in generale, un lieve miglioramento delle prestazioni grazie all'impiego della memoria condivisa e del pattern SoA. I valori misurati sui grafi `rome` e `DE` sono prossimi allo 0.

Infine, con gli algoritmi `bf2` si notano valori di speedup decisamente più alti rispetto agli algoritmi `bf1`, ma solamente sui grafi sintetici. Nella figura 6.4, si può notare che sugli altri grafi sono stati misurati valori intorno ad 1 e quindi non significativi. Questa grande

differenza rispecchia perfettamente la struttura degli algoritmi `bf2`: utilizzando solamente 1024 thread per rilassare gli archi entranti di ogni nodo l'incremento di prestazioni diventa significativo solamente su grafi con elevata densità. Va notato, però, che l'unica versione che impiega SoA riporta valori di speedup molto più elevati rispetto alle due versioni `bf2-aos`.

6.6 Throughput

Di seguito sono riportati i valori di throughput di ogni versione, misurati sui test a disposizione e su ogni scheda, indicati in milioni di operazioni di rilassamento effettuate ogni secondo (10^6 relax/s).

Come già spiegato nella sezione precedente, nelle figure 6.5 e 6.6 si nota l'enorme differenza tra la scheda 940MX e le altre. É comunque interessante notare come i valori della scheda GTX1070, pur sempre elevati, risultino meno variabili di quelli misurati sulla scheda RTX2080.

I valori misurati sul grafo `rome`, come per lo speedup, risultano di almeno due ordini di grandezza minori rispetto ai valori misurati sugli altri test.

In figura 6.7 osserviamo come l'impiego di un singolo blocco, per l'esecuzione dei kernel degli algoritmi `bf1`, abbia determinato una riduzione di prestazioni pari a circa un fattore 10 rispetto agli algoritmi `bf0`. I valori misurati sui grafi `rome` e `DE` sono prossimi a 0 a causa dell'elevato tempo di esecuzione impiegato.

Come già osservato per lo speedup, in figura 6.8 osserviamo un aumento delle prestazioni, rispetto agli algoritmi `bf1`, di almeno un fattore 4. Nonostante sia un incremento importante, il throughput delle versioni `bf2` risulta molto inferiore rispetto alle prestazioni degli algoritmi `bf0`.

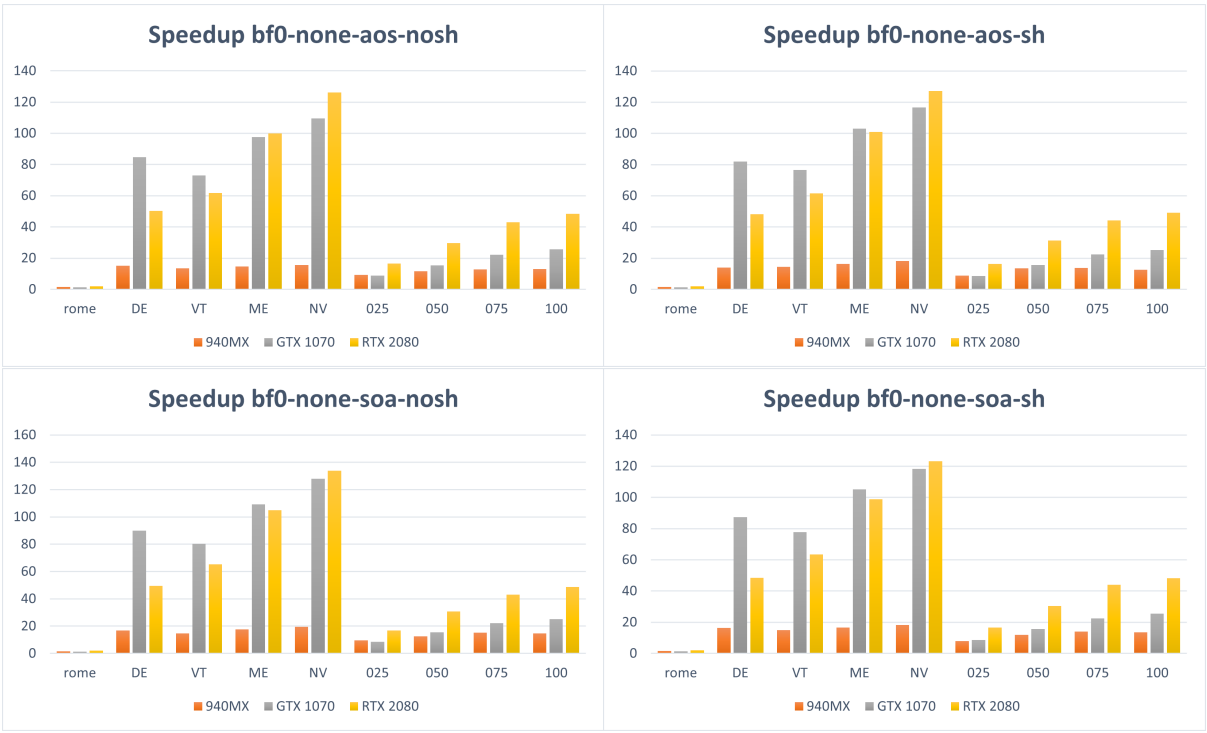

Figura 6.1: Speedup algoritmi `bf0-none`

Figura 6.2: Speedup algoritmi `bf0-mutex`

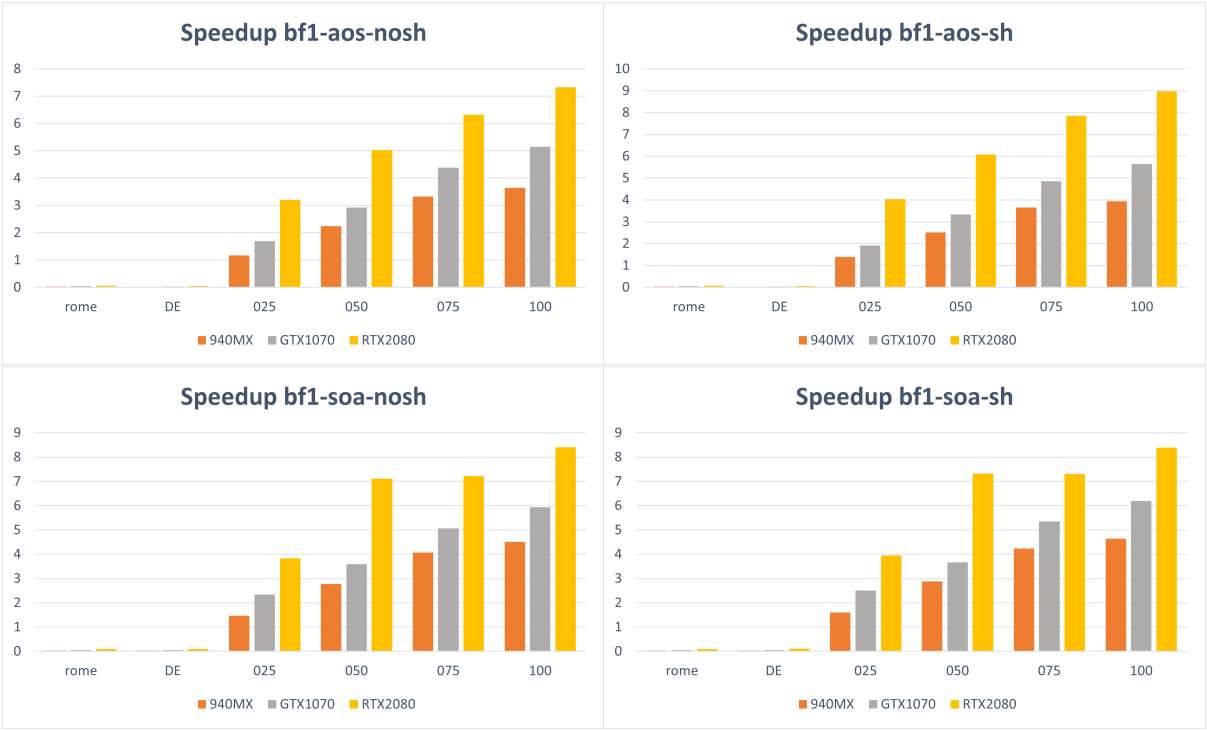

Figura 6.3: Speedup algoritmi `bf1`

Figura 6.4: Speedup algoritmi `bf2`

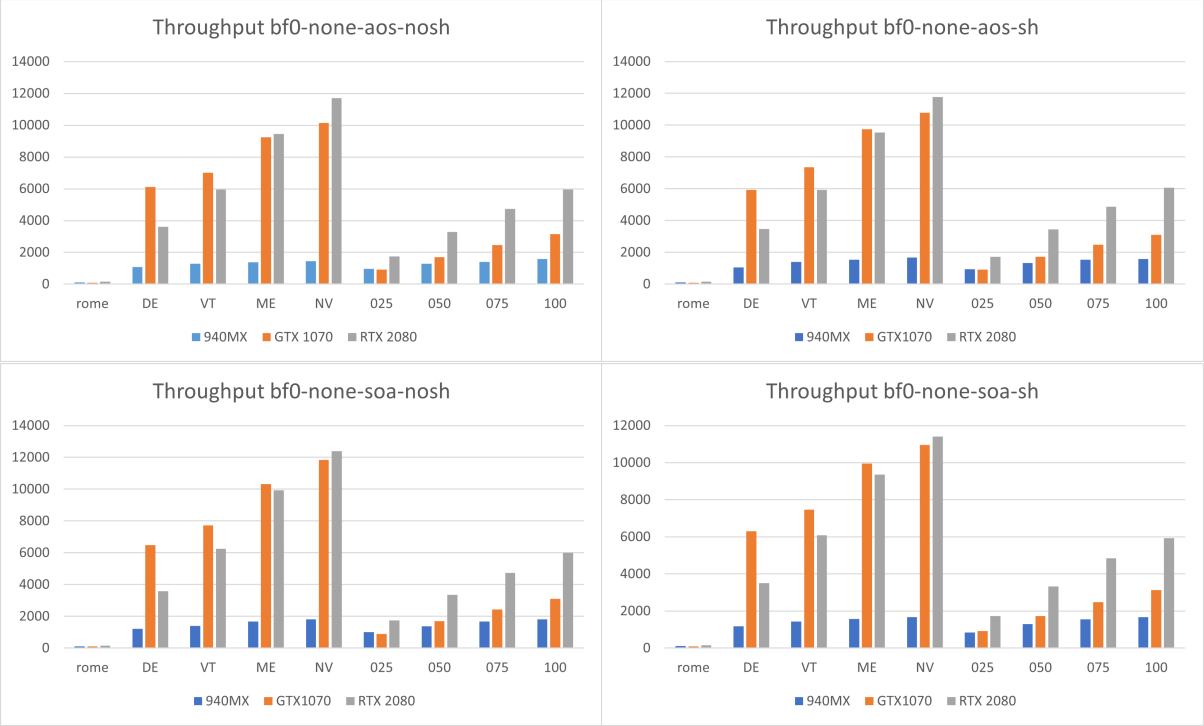

Figura 6.5: Throughput algoritmi `bf0-none`

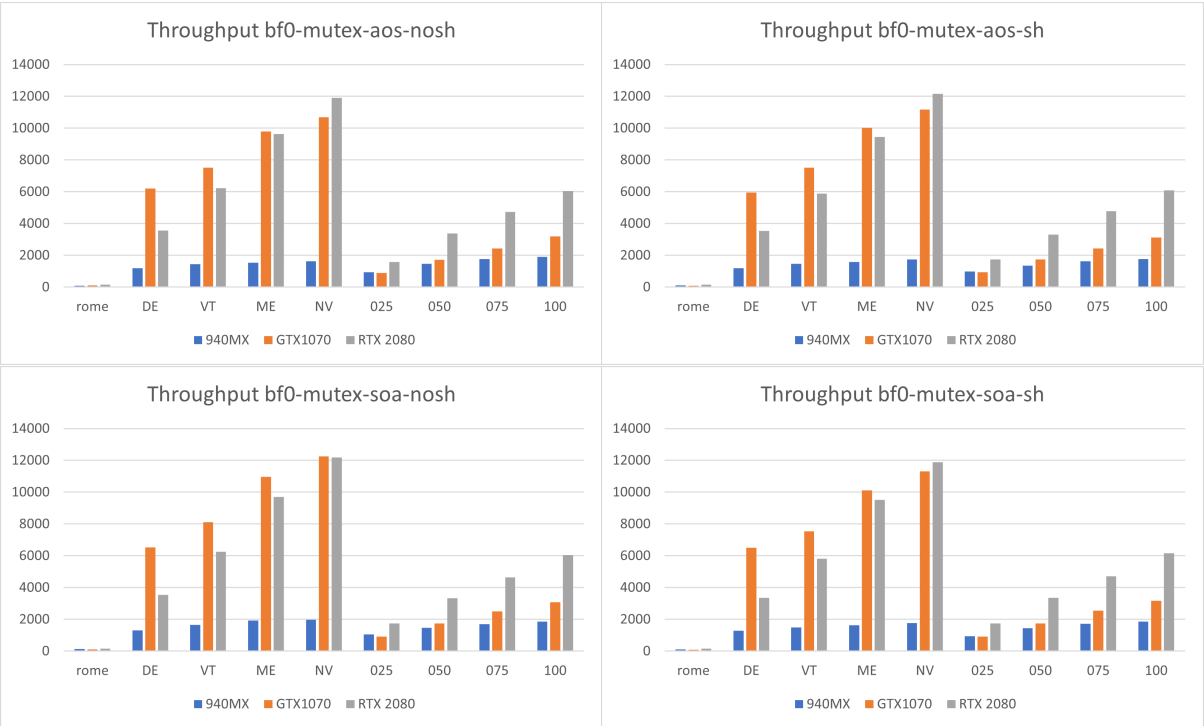

Figura 6.6: Throughput algoritmi `bf0-mutex`

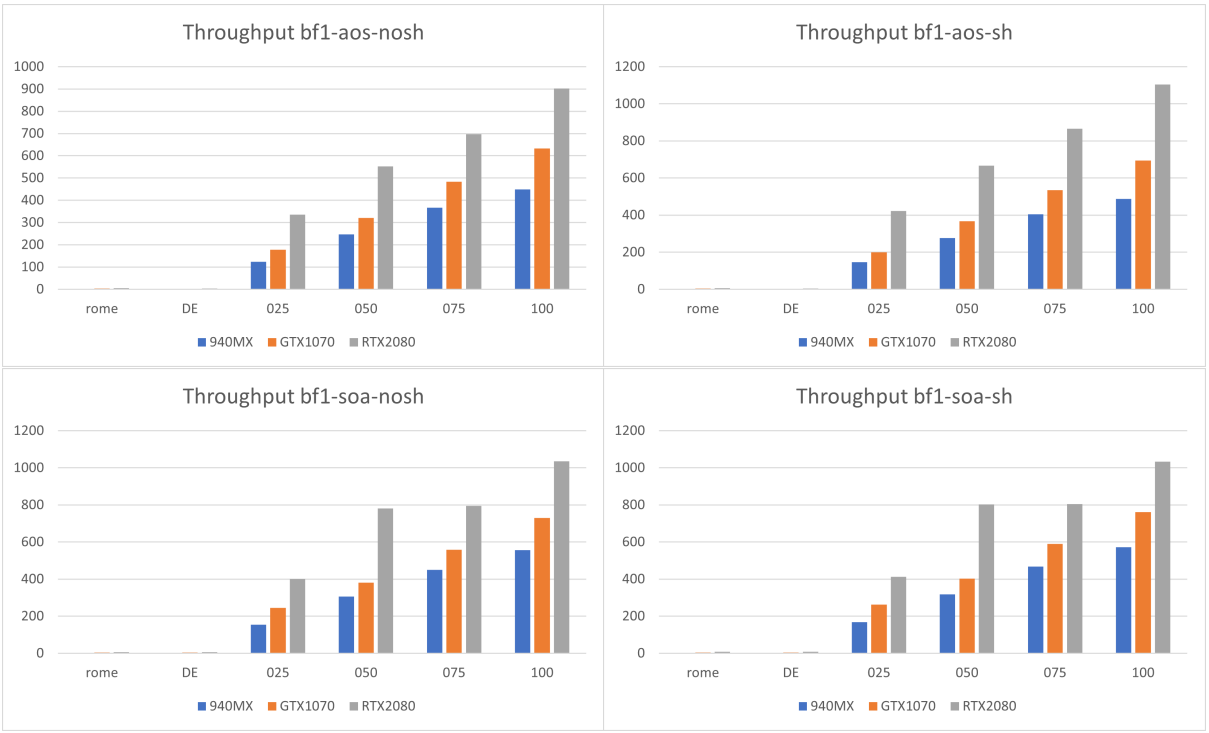

Figura 6.7: Throughput algoritmi `bf1`

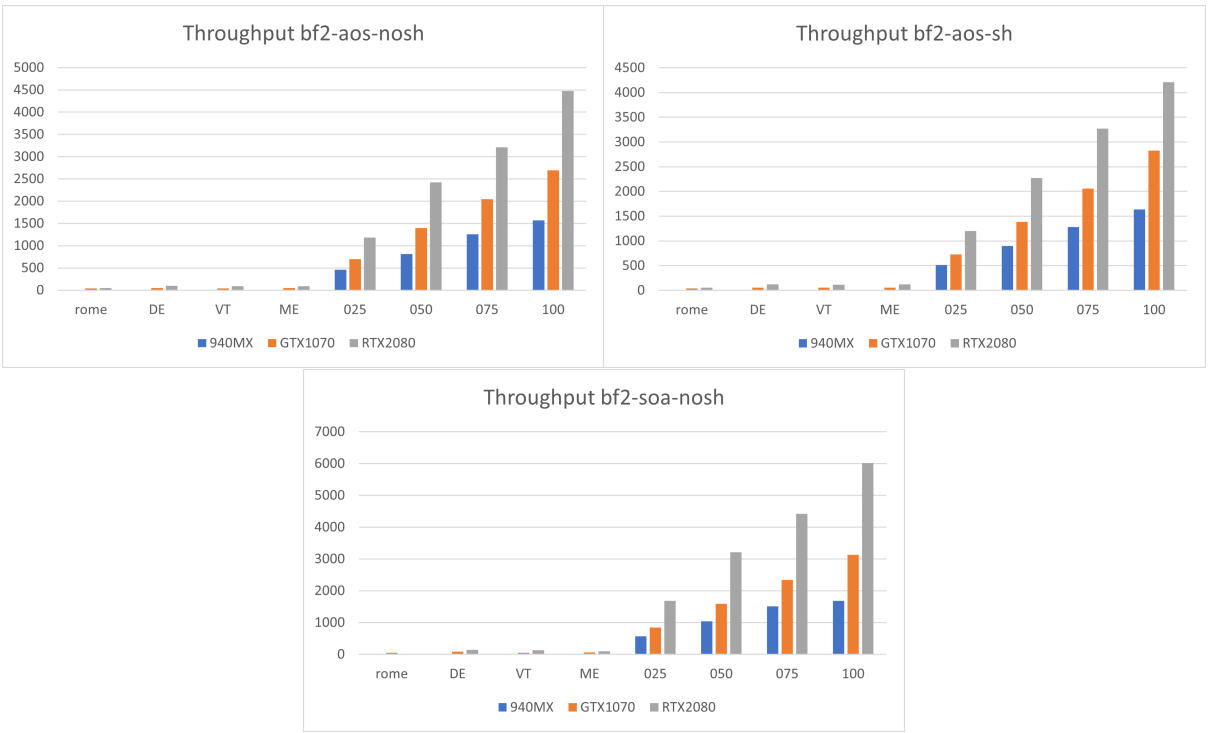

Figura 6.8: Throughput algoritmi `bf2`

Conclusioni e sviluppi futuri

In questa tesi sono state analizzate varie implementazioni dell'algoritmo di Bellman-Ford in CUDA, con un'ampia esplorazione delle ottimizzazioni possibili e molti test effettuati. L'algoritmo più veloce risulta essere `bf0-none-soa-nosh`, nonostante la differenza sia molto ridotta rispetto agli altri algoritmi della stessa famiglia. Tale algoritmo, ad esempio, riesce a processare in poco più di 13 secondi la mappa stradale del Nevada, uno stato grande quasi quanto l'Italia, sulla scheda RTX 2080. Se consideriamo che l'algoritmo implementato è privo di ottimizzazioni, questo dato fa riflettere sulle potenzialità che Bellman-Ford presenta riguardo all'impiego su GPGPU. Un interessante sviluppo futuro di questo progetto potrebbe riguardare, dunque, l'implementazione delle varie ottimizzazioni proposte fino ad oggi. La struttura dell'algoritmo di Bellman-Ford, inoltre, suggerisce una semplice divisione del lavoro per un sistema multi-GPU. Attuando anche questa ottimizzazione, l'algoritmo potrebbe essere utilizzato per processare il grafo contenente la struttura di Internet in tempi non proibitivi.

Bibliografia

[1] Alfonso Shimbel. «Structure in communication nets». In: *Proceedings of the Symposium on Information Networks* (1954). Brooklyn, New York: Polytechnic Press of the Polytechnic Institute of Brooklyn, 1955, pp. 199–203.

[2] Lester Randolph Ford. *Network Flow Theory*. Tech. rep. AD 422842. Cameron Station, Alexandria, Virginia (USA): Defense Documentatin Center for Scientific e Technical Information, 1956.

[3] Richard Bellman. «On a routing problem». In: *Quarterly of Applied Mathematics* 16.1 (apr. 1958), pp. 87–90. DOI: 10.1090/qam/102435. URL: https://doi.org/10.1090/qam/102435.

[4] Edward Forrest Moore. «The shortest path through a maze». In: *Proceedings of an International Symposium on the Theory of Switching. Part II* (2–5 apr. 1957). A cura di H. Aiken. Cambridge, Massachusetts: Harvard University Press, 1959, pp. 285–292.

[5] Jin Y. Yen. «An algorithm for finding shortest routes from all source nodes to a given destination in general networks». In: *Quarterly of Applied Mathematics* 27.4 (gen. 1970), pp. 526–530. DOI: 10.1090/qam/253822. URL: https://doi.org/10.1090/qam/253822.

[6] Gordon E Moore et al. «Progress in digital integrated electronics». In: *Electron devices meeting*. Vol. 21. Washington, DC. 1975, pp. 11–13.

[7] Michael J. Bannister e David Eppstein. «Randomized Speedup of the Bellman–Ford Algorithm». In: *2012 Proceedings of the Ninth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*. Society for Industrial e Applied Mathematics, gen. 2012. DOI: 10.1137/1.9781611973020.6. URL: https://doi.org/10.1137/1.9781611973020.6.

[8] URL: https://www.moreno.marzolla.name/teaching/LabASD/handouts/bellman-ford.html.

[9] URL: http://users.diag.uniroma1.it/challenge9/download.shtml.

[10] *Cuda C++ Programming Guide*. NVIDIA Corporation. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications.

Ringraziamenti

Voglio ringraziare la mia famiglia per il sostegno e il supporto che mi hanno dato. Un affettuoso ringraziamento va alla mia fidanzata Doralice che mi ha sempre aiutato a superare le situazioni più difficili. Inoltre voglio ringraziare i miei amici di sempre per avermi dato la forza di migliorare e andare avanti. Un grazie va anche agli amici e colleghi dell'università e a tutti coloro che hanno contribuito alla stesura di questa tesi con consigli, osservazioni e critiche. Un ringraziamento speciale va all'amico Lorenzo Di Ghionno per avermi concesso di utilizzare la sua scheda RTX 2080 per la realizzazione di questa tesi. Ringrazio tutti i professori del corso di laurea che hanno contribuito alla mia formazione ed aiutato ad arrivare fin qui. Infine, desidero ringraziare il professore Moreno Marzolla per avermi sempre aiutato con simpatia e professionalità.