Alma Mater Studiorum
Università di Bologna
Campus di Cesena

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

STEREO MATCHING SU GPU

Elaborato in:
High Performance Computing

Relatore:
Prof.
MORENO MARZOLLA

Presentata da:
CHRISTIAN RICCI

Sessione III
Anno Accademico 2021-2022

Indice

Elenco delle figure

Introduzione

La *Stereo Vision* è un popolare argomento di ricerca nel campo della Visione Artificiale; esso consiste nell'usare due immagini di una stessa scena, prodotte da due fotocamere diverse, per estrarre informazioni in 3D. L'idea di base della *Stereo Vision* è la simulazione della visione binoculare umana: le due fotocamere sono disposte in orizzontale per fungere da "occhi" che guardano la scena in 3D. Confrontando le due immagini ottenute, si possono ottenere informazioni riguardo alle posizioni degli oggetti della scena.

In questa relazione presenteremo un algoritmo di Stereo Vision: si tratta di un algoritmo parallelo che ha come obiettivo di tracciare le linee di livello di un area geografica. L'algoritmo in origine era stato implementato per la *Connection Machine CM-2*, un supercomputer sviluppato negli anni 80, ed era espresso in ***Lisp**, un linguaggio derivato dal Lisp e ideato per la macchina stessa.

Questa relazione tratta anche la traduzione e l'implementazione dell'algoritmo in **CUDA**, ovvero un'architettura hardware per l'elaborazione parallela sviluppata da NVIDIA, che consente di eseguire codice parallelo su GPU. Si darà inoltre uno sguardo alle difficoltà che sono state riscontrate nella traduzione da *Lisp a CUDA.

Capitolo 1

Programmazione Parallela

La programmazione parallela è un paradigma di programmazione che sfrutta l'esecuzione parallela di processori per ottenere prestazioni migliori e risultati nel minor tempo possibile. I calcoli vengono eseguiti in parallelo, ovvero vengono eseguiti simultaneamente su più processori. Questo paradigma si colloca all'interno dell'High Performance Computing (HPC), una materia che studia le tecniche e soluzioni hardware e software che facilitano l'esecuzione parallela dei calcoli.

1.1 Evoluzione dei Processori

Perché ricorrere all'esecuzione parallela per ottenere prestazioni migliori? Questa domanda trova risposta nello studio dell'evoluzione dei processori: per anni si è tenuto conto della cosiddetta "Legge di Moore", che ipotizza che "il numero di transistor contenuti all'interno di un chip raddoppia ogni due anni". Questa legge fu enunciata da Gordon E. Moore nel 1965 e spiega che effettivamente le prestazioni di un chip si raddoppiavano ogni due anni. La sua osservazione si rivelò corretta per due decenni, ma intorno agli anni 2000 questo andamento iniziò a calare. I motivi principali di questo calo sono:

- La dimensione dei transistor. Il miglior modo per aumentare il numero di transistor è di renderli sempre più piccoli, ma esiste un limite fisico della grandezza minima. I transistor di oggi vanno dai 20 nm ai 3 nm;
- Il calore dissipato dal processore. Un processore più veloce corrisponde ad una più grande quantità di calore dissipato, che porta a problemi di instabilità nel processore.

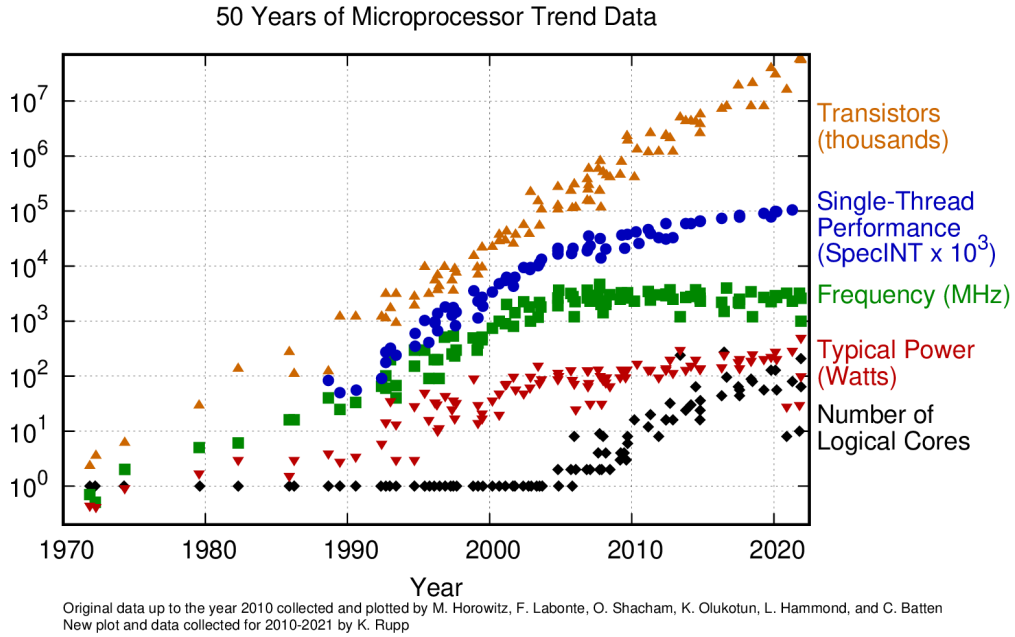

Figura 1.1: Evoluzione delle caratteristiche dei processori

Fonte: `https://github.com/karlrupp/microprocessor-trend-data`

Al posto di aumentare il numero di transistor, negli ultimi due decenni sono state ideate nuove architetture che sfruttano il parallelismo: queste sono le architetture parallele, anche chiamate multicore o multiprocessore. Queste architetture sono sviluppate aggiungendo più processori fisici dentro ad un chip. Le prestazioni che si possono ottenere con le architetture parallele derivano quindi dalla suddivisione del lavoro che è distribuito in più unità operazionali.

Nella figura 1.1 viene mostrata l'evoluzione dei processori nel corso degli ultimi decenni. In particolare, si può notare come la frequenza media dei processori si sia stabilizzata, mentre più o meno dalla metà degli anni 2000 sia cresciuto il numero di core logici, a seguito di ricerche sullo sviluppo delle architetture parallele.

1.2 GPU e CUDA

Esistono diverse tipologie di architetture parallele; in questa relazione ci interesseremo delle architetture basate su GPU. Queste architetture vengono chiamate GPGPU (General Purpose Graphics Processing Units), ovvero schede grafiche programmabili in maniera generica.

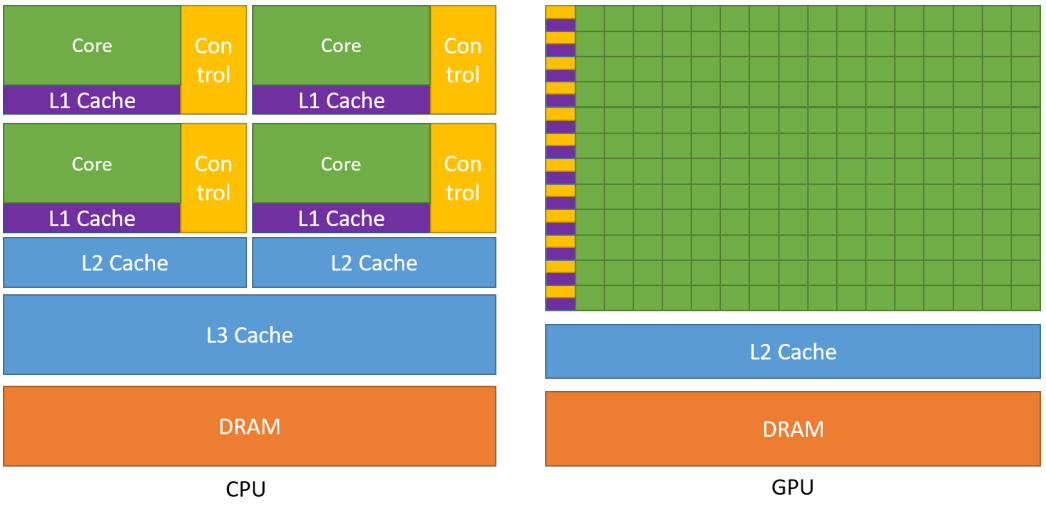

Figura 1.2: Confronto tra l'architettura di una CPU e quella di una GPU. Da notare l'elevato numero di core della GPU.

Fonte: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

Il nome di queste architetture potrebbe dare confusione dato che, in generale, la funzione di una scheda video è strettamente legata alla grafica. Il motivo per cui sono interessanti è per la presenza di un grande numero di core all'interno della scheda, molto più grande del numero di core presenti in

una normale CPU (si tratta di due ordini di grandezza di differenza); questo implica che possono essere molto utili per programmi paralleli con dati di grandi dimensioni. La figura 1.2 mostra le due architetture a confronto.

Le prime ricerche riguardanti la programmazione generale su GPU furono svolte nel 2001 e nel 2003; diventarono più popolari nel 2007 con l'avvento di CUDA.

CUDA (Compute Unified Device Architecture) è una piattaforma per il calcolo parallelo sviluppata da NVIDIA. Si tratta di una piattaforma che consente l'utilizzo delle schede grafiche NVIDIA per eseguire programmi paralleli, sfruttando appieno le loro potenzialità. La prima versione di CUDA è stata pubblicata nel 2007.

I concetti di base di CUDA sono molto semplici: CUDA prevede una separazione tra l'*host*, che corrisponde la CPU, e il *device*, che corrisponde ad una o più schede grafiche. Queste sono considerate unità indipendenti, che cooperano in modo asincrono: in particolare, l'*host* ha la possibilità di invocare funzioni specifiche che vengono eseguite sul *device*. Queste funzioni vengono dette *kernel*; appena vengono chiamate, queste vengono eseguite in modo asincrono e restituiscono subito il controllo all'*host*.

1.2.1 Architettura di CUDA

In CUDA, l'unità minima di elaborazione è il **thread**. CUDA raggruppa logicamente i thread in più **blocchi**, e a loro volta i blocchi sono raggruppati in una **griglia** [1]. La figura 1.3 mostra sinteticamente questa gerarchia.

Un blocco è considerato un unità indipendente di lavoro: esso può avere una, due o tre dimensioni, il che significa che può essere indicizzato in più dimensioni in base all'occorrenza; ogni blocco può essere eseguito in qualunque ordine all'interno di una griglia. Il numero di thread in un blocco è limitato a va fino a 1024 thread per blocco.

Una griglia rappresenta un lavoro affidato alla GPU: anch'esso può essere tridimensionale e non ha limiti sul numero di blocchi.

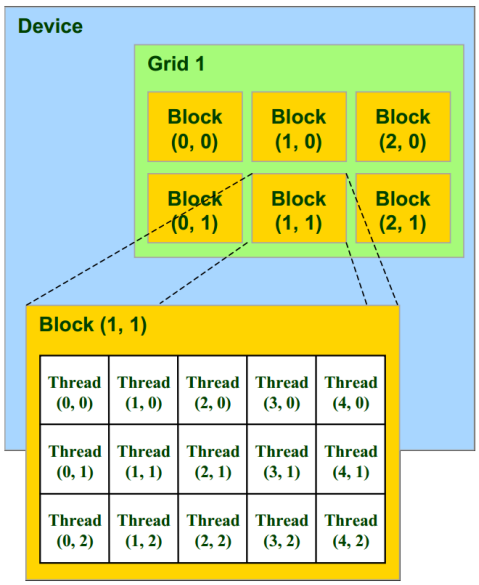

Figura 1.3: Gerarchia dei thread in CUDA

Fonte: `https://ipython-books.github.io/58-writing-massively-parallel-code-for-nvidia-graphics-cards-gpus-with-cuda/`

Lo **scheduling dei thread** in CUDA è completamente automatico: non è possibile scegliere quali e quanti core (che in CUDA vengono chiamati *Streaming Multiprocessor*) debbano venire utilizzati.

1.2.2 Struttura e gestione della memoria

Quando si scrive un *kernel*, non ci si deve mai preoccupare di come vengono gestiti i thread. Sarà l'hardware stesso a preoccuparsi, durante l'esecuzione, di effettuare lo scheduling migliore sfruttando al massimo l'hardware: in particolare, in CUDA esiste il concetto di *warp*, che rappresenta l'unità minima di schedulazione e contiene un certo numero di thread (massimo 32). L'hardware raggruppa assieme in un *warp* i thread che devono eseguire la stessa porzione di codice; i thread eseguono le istruzioni seguendo il paradigma SIMD.

Nel caso ci sia una biforcazione nel flusso di controllo (causato, ad esempio, da costrutti `if-else`), la condizione viene calcolata per ogni thread,

dopodichè vengono eseguiti i thread per cui è stata verificata, mentre gli altri rimangono fermi. L'hardware comunque può trasferire i thread che sono rimasti fermi in altri warp. La figura 1.4 mostra un esempio visuale del problema.

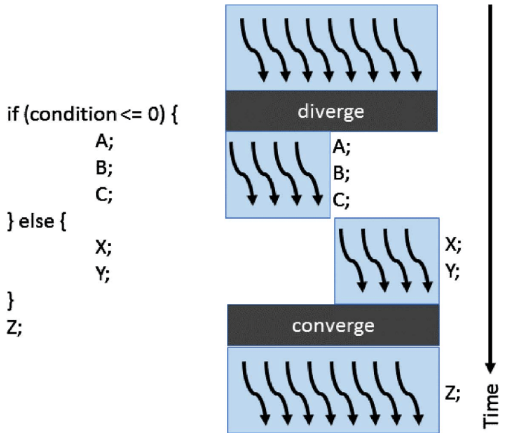

Figura 1.4: Esecuzione di un thread in caso di biforcazione

Fonte: https://www.researchgate.net/figure/Warp-divergence-due-to-if-else-statements_fig15_317608134

La memoria in CUDA è suddivisa in più tipi:

- **Memoria globale**: è la memoria più grande (alcuni GB). A questa memoria possono accedere ogni thread di ogni blocco, ma presenta una latenza di accesso più alta;
- **Memoria delle constanti**: una memoria persistente anch'essa accessibili da ogni thread, ma ci i thread possono accedere in sola lettura. Lhost può comunque accedere in scrittura. È più piccola, di soli 64 KB;
- **Memoria texture**: un'altra memoria di sola lettura e accessibile da tutti, ma è ottimizzata per le texture;
- **Memoria condivisa**: è una memoria molto veloce che viene condivisa tra tutti i thread di un solo blocco. Per blocco ci sono 48 KB di memoria condivisa;

- **Memoria locale**: una memoria dedicata per ogni thread, accessibile solo da esso. Non persiste tra l'esecuzione di due kernel;

La figura 1.5 mostra una rappresentazione visuale della memoria.

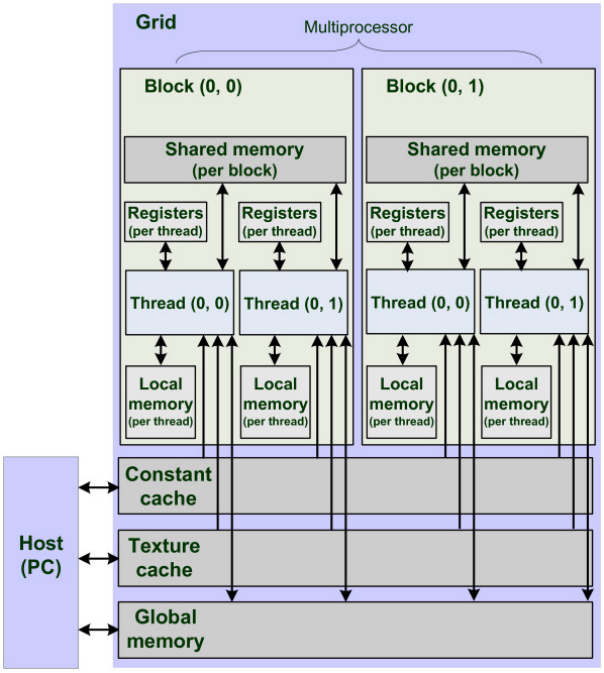

Figura 1.5: La struttura della memoria in CUDA.

Fonte: `https://www.researchgate.net/figure/Memory-model-CUDA-memory-hierarchy_fig2_51529494`

In CUDA va prestata particolare attenzione all'accesso della memoria. Infatti, quando un thread esegue un accesso alla memoria anche molto piccolo (4 byte), la GPU legge almeno 32 byte consecutivi. Una caratteristica molto importante è il **raggruppamento degli accessi**: se più thread accedono a posizioni di memoria consecutive, la GPU raggruppa questi accessi in una sola lettura.

Questo in pratica significa che in CUDA il pattern di memorizzazione classico, ovvero l'Array-of-Structures, non è adeguato e si dovrebbe preferire la Structure-of-Arrays. Il primo consiste nel conversare i dati di un elemento (ad esempio un punto con coordinate x, y e z) in posizioni adiacenti di memoria: di conseguenza, un array di punti consisterà in un array di coordinate (x,

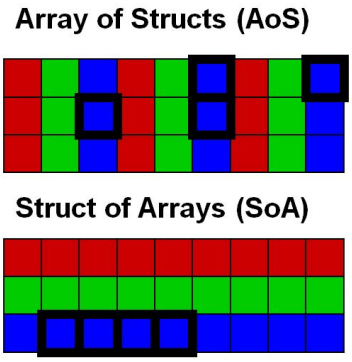

Figura 1.6: AoS e SoA a confronto

Fonte: `https://asc.ziti.uni-heidelberg.de/en/node/18`

y, z) che si ripete per ogni punto. Il secondo consiste nell'avere un array per ogni dato dell'elemento: si avrà un array per ogni coordinata X, coordinata Y e coordinata Z. La figura 1.6 mostra un confronto visuale dei due pattern.

Immaginiamo di avere più thread che devono leggere ognuno le coordinate X di ogni punto. Se i dati sono memorizzati mediante AoS, i valori in sè sarebbero più lontani tra loro, con la conseguenza di avere maggiori letture e maggior utilizzo del bus nel programma. Al contrario, con SoA si avrebbe tutti elementi adiacenti e si avrebbe bisogno di meno letture.

1.2.3 Il linguaggio CUDA

Per scrivere un kernel, CUDA prevede un apposito linguaggio: il linguaggio che si usa è un estensione del C e del C++ con l'aggiunta di parole chiave, funzioni di libreria ed altre caratteristiche per facilitare la scrittura del codice per GPU.

In particolare, CUDA aggiunge le seguenti parole chiave per le funzioni:

- `__host__`: usata per funzioni scritte per la CPU (presente di default se non ci sono altre parole chiave di CUDA);
- `__device__`: usata per le funzioni scritte per la GPU e chiamate dalla GPU;

- `__global__`: usata per le funzioni scritte per la GPU, ma chiamate dalla CPU. In essenza sono il punto d'entrata tra codice dell'*host* e codice del *device*;

`__host__` e `__device__` possono essere combinati assieme. CUDA aggiunge anche le seguenti parole chiave per le variabili:

- `__constant__`: memorizza la variabile in memoria costante;
- `__device__`: memorizza la variabile nella memoria globale della GPU. Essa è comunque accessibile dall'host tramite opportune funzioni;
- `__shared__`: memorizza la variabile nella memoria shared. In questo caso, il suo accesso è limitato al singolo blocco;

L'estensione più importante in CUDA riguarda il modo in cui si chiamano le funzioni dichiarate `__global__`: queste richiedono due parametri obbligatori, ovvero il numero di blocchi e il numero di thread per blocco da utilizzare per l'esecuzione.

Capitolo 2

Stereo Vision

La visione artificiale è il campo dell'informatica che si occupa di analizzare ed interpretare le immagini per ottenere un modello approssimato del mondo reale. In sostanza, si occupa di capire ed automatizzare compiti che il sistema di visione umano riesce a fare.

La Stereo Vision è una sottobranca della visione artificiale che si occupa di ottenere informazioni spaziali utilizzando due immagini (da qui viene il termine *stereo*, che significa *due*).

Gli uomini posseggono due occhi frontali che vedono il mondo esterno da due posizioni differenti: queste due posizioni vengono combinate, usando le piccole differenze presenti tra le due per recuperare le informazioni in 3D sull'ambiente esterno. L'idea di base della Stereo Vision è la simulazione di questo sistema: per far ciò si utilizzano due fotocamere, disposte orizzontalmente l'una all'altra, in modo che fungano come da occhi umani che guardano la scena di cui si è interessati. Le due immagini che si ottengono dalle fotocamere sono chiamate *Stereo Pair*. La figura 2.1 ne mostra un esempio.

Un problema importante nella Stereo Vision è come inferire la profondità degli oggetti della scena usando le due immagini prese dalle fotocamere [2]. È qui che entrano in gioco gli algoritmi di *Stereo Matching*: durante la fase di analisi delle due immagini, si cerca una corrispondenza (*match*) tra un

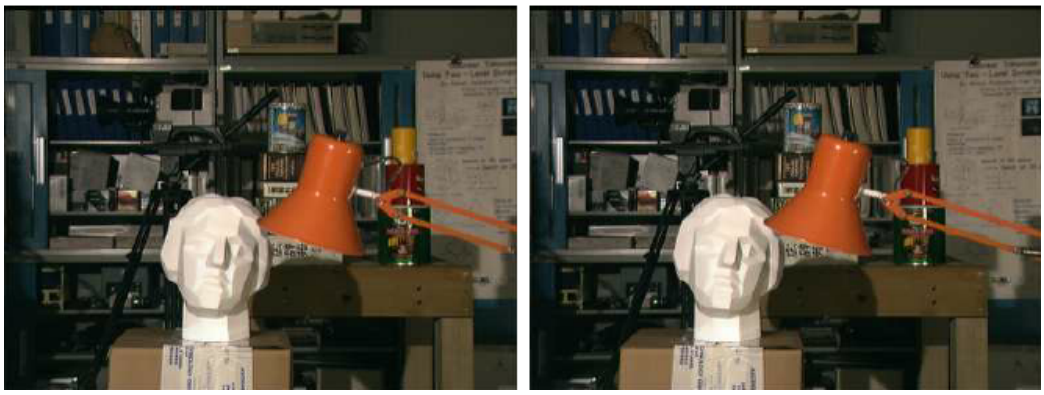

Figura 2.1: Uno *stereo pair*. La seconda immagine è leggermente spostata orizzontalmente rispetto alla prima.

Fonte: `https://www.researchgate.net/figure/Tsukuba-stereo-pair_fig1_278622997`

oggetto di un immagine e la posizione dell'oggetto nell'altra immagine: la differenza della sua posizione nelle due immagini determina la posizione finale dell'oggetto. Le informazioni che si ottengono sulla profondità degli oggetti vengono chiamate nel loro complesso *mappa di disparità*. Una mappa di disparità si può vedere nella figura 2.2.

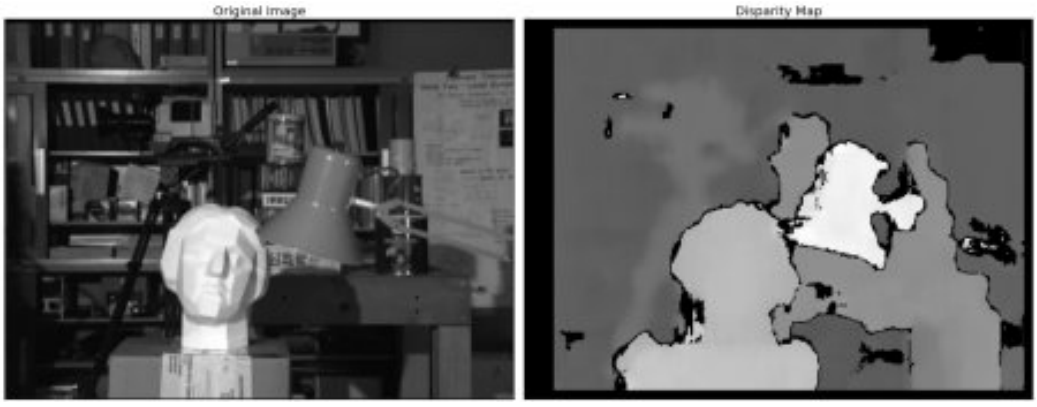

Figura 2.2: Esempio di mappa di disparità. Le regioni più chiare indicano oggetti più vicini, mentre quelle più scure indicano oggetti più lontani.

Fonte: `https://docs.opencv.org/4.x/dd/d53/tutorial_py_depthmap.html`

2.1 Le fasi di analisi delle immagini

Per estrarre le informazioni 3D, la maggior parte degli algoritmi di Stereo Vision seguono i seguenti passi:

1. Nel caso le immagini siano distorte o abbiano qualche difetto, bisognerà effettuare una **calibrazione della fotocamera**.

2. Le immagini vengono riproiettate in un piano comune per permettere il confronto diretto. Questo passo viene detto **rettificazione delle immagini** e in generale si effettua soltanto se i piani delle due fotocamere non giacciono sulla stessa linea (questa linea viene chiamata *baseline*.

3. Si fa il **matching** tra le due immagini e si crea la mappa di disparità. Questo di solito avviene pixel per pixel: dato il pixel p_1 della prima immagine, si cerca un secondo pixel p_2 nella seconda immagine che corrisponda al primo. Come calcolare se due pixel corrispondono dipende dall'algoritmo.

4. Opzionalmente, si effettua una **ricostruzione** in 3D della scena. Questo viene fatto determinando le coordinate 3D di ogni punto a partire dalle coordinate 2D delle coppie di punti corrispondenti nelle immagini.

2.2 Tecniche di *matching*

L'approccio classico per lo stereo matching è: per ogni pixel $p = (x, y)$ nell'immagine di partenza, si trova la linea corrispondente nell'immagine di destinazione, poi si trova il pixel p' più somigliante nella linea. Nei casi più semplici, la linea corrispondente è la coordinata y del punto p.

Il calcolo della **somiglianza** differisce in ogni algoritmo, ma in generale viene calcolata utilizzando la regione intorno al punto. Si definiscono due *finestre* di grandezza $N \times N$: $W(x, y)$, centrata nel punto p, e W', che viene fatta scorrere in orizzontale lungo la linea. Ad ogni scorrimento corrisponde un nuovo punto p'. Per ogni punto p' trovato, si calcola una metrica

particolare derivata dall'intorno. Tre metriche popolari sono la *Correlazione Normalizzata*, la *Sum of Squared Differences* e la *Sum of Absolute Differences* [3].

Un approccio alternativo consiste nell'*edge matching* (*corrispondenza dei contorni*). In questo caso per le due immagini si trovano i contorni, poi si fa il *match* dei contorni in modo simile al metodo delle finestre: in questo caso si calcola la somiglianza tra contorni. La figura 2.3 mostra un immagine di cui sono stati prelevati i contorni.

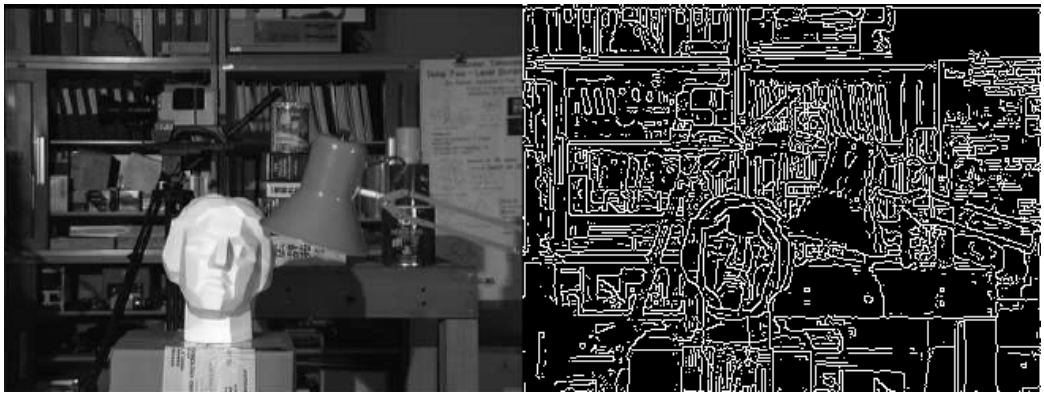

Figura 2.3: I contorni di un immagine.

Capitolo 3

La *Connection Machine* e *Lisp

L'algoritmo di stereo matching che presentiamo fu descritto e implementato per la **Connection Machine CM-2** nel linguaggio ***Lisp**. In questo capitolo diamo una breve descrizione delle loro caratteristiche, in modo da poter confrontare l'algoritmo originale con la versione in CUDA.

3.1 La Connection Machine CM-2

Figura 3.1: Il design esterno della Connection Machine CM-2.

Fonte: `https://commons.wikimedia.org/wiki/File:Computer_Museum_of_America_(51).jpg`

La *Connection Machine CM-2* fa parte di una serie di supercomputer sviluppati dall'azienda Thinking Machines nei primo anni 80. La macchina era un supercomputer parallelo basata sul parallelismo a livello di dati (*data level parallelism*), che consiste nell'eseguire un'operazione su più dati in parallelo, in contrasto con il parallelismo a livello di istruzioni, che consiste nell'eseguire in parallelo le istruzioni che possono essere eseguite indipendentemente [4]. In pratica, ciò significa che la Connection Machine operava come una gigantesca macchina SIMD (Single Instruction Multiple Data). La figura 3.1 ne mostra una foto dall'esterno.

La macchina, per eseguire i programma, utilizzava una parte di front-end, dove veniva memorizzato il programma e dove veniva eseguito il codice seriale. Quando un programma incontrava istruzioni parallele (istruzioni che operava sui dati in modo parallelo), le passava all'hardware della Connection Machine: questo era composto da 65535 processori, ognuno con 4096 bit di memoria.

Ogni processore operava su un solo elemento: se c'erano meno di 65535 elementi, allora i processori rimanenti venivano disabilitati; se ce n'erano di più, allora la macchina operava con processori virtuali. I processori potevano comunicare tra di loro attraverso un sistema di inter-connessione chiamato *router*: questo funzionava con meccanismi di comunicazione a scambio di messaggi ed era implementato in software.

3.2 *Lisp

Lisp è un linguaggio di programmazione ideato all'inizio degli anni 60 da John McCarthy. Si tratta di un linguaggio basato sui paradigmi imperativo e funzionale, con una sintassi molto unica. Il seguente codice presenta un esempio del linguaggio; si tratta di una funzione che trova se un elemento esiste in un array:

```
1 (defun contains-element (arr i element)
2   (if (= i (array-dimension arr 0))
3     0
4     (if (= (aref arr i) element)
5       1
6       (contains-element arr (+ i 1) element))))
```

Listing 3.1: Esempio di codice Lisp. Questa funzione implementa la ricerca nell'array in maniera ricorsiva, in concordanza con il paradigma funzionale.

Il Lisp ha molti dialetti e *Lisp è uno di questi (in particolare, è un dialetto del Common Lisp). *Lisp, come linguaggio pensato per essere usato nella Connection Machine, aggiunge nuove caratteristiche per la programmazione parallela: ad esempio, molti costrutti come `if` e `=` hanno una versione parallela che agiscono su tutti i dati in contemporanea. In particolare, introduce una struttura dati, la *pvar*, che rappresenta una collezione di valori memorizzati uno per ogni processore [5]. Il seguente codice mostra un esempio di codice *Lisp, che è una traduzione della funzione sopra:

```
1 (*defun contains-element (pvar element)
2   (if!! (=!! pvar element)
3         (!! 1)
4         (!! 0)))
```

Listing 3.2: Esempio di codice *Lisp. Questa versione della funzione implementa la ricerca in parallelo su tutti gli elementi dell'array.

Capitolo 4

Descrizione dell'algoritmo di *stereo matching*

L'algoritmo di *stereo matching* che andiamo a presentare ha come obbiettivo il disegnare le linee di livello di un'area geografica. Questo problema viene chiamato *contour mapping* e consiste nell'estrarre informazioni sull'altezza da immagini che memorizzano informazioni sulla luminosità dell'area. L'immagine risultante viene chiamata *contour map* (mappa dei contorni o dei livelli).

L'algoritmo prende come input due immagini aeree di un'area geografica: le due immagini devono essere uno *stereo pair* e devono essere siano in bianco e nero, ovvero all'inizio dell'algoritmo le due immagini devono rappresentare la luminosità che c'è in ogni punto dell'area geografica.

L'algoritmo è composto da tre fasi:

1. La ricerca dei bordi o contorni;
2. La ricerca di corrispondenze tra i contorni trovati;
3. Il disegno delle linee di livello;

Questo algoritmo si tratta di un algoritmo di *edge matching*: le operazioni più importanti sono quelle legate alla ricerca e alla corrispondenza dei bordi (*edges*).

Per avere più dettagli sull'implementazione originale dell'algoritmo, si può consultare [4].

4.1 Ricerca ed estrazione dei bordi

La ricerca dei contorni avviene indipendentemente su tutte e due le immagini. L'algoritmo definisce un contorno come una linea che giace sul confine tra aree di luminosità molto diversa. A questo proposito, l'algoritmo da una classificazione per ogni pixel basandosi sui suoi intorni: se da un lato del pixel la luminosità è molto differente che dall'altro, allora il pixel viene classificato come un pixel del contorno. Questa operazione è un esempio di *local neighborhood operation*, operazione locale sui vicini.

La seguente funzione in *Lisp mostra l'implementazione originale dell'operazione. Questa funzione si occupa solo di un pixel: prima calcola la media della luminosità nei lati destro e sinistro del pixel, poi calcola la differenza di luminosità e la confronta con una certa soglia: se questa risulta maggiore, allora il pixel viene considerato come parte di un bordo.

A riga 19, la soglia è moltiplicata con la media complessiva di luminosità: in questo modo, la soglia si adatta all'immagine, diventando minore o maggiore a seconda di quanto sia "luminosa" l'immagine in quel punto.

```
1  (*defun find-edges-between-left-and-right!! (brightness-pvar threshold)
2    (*let*
3      ((average-brightness-on-the-left
4         (/!! (+!! (pref-grid-relative!! brightness-pvar (!! -1) (!! -1))
5                   (pref-grid-relative!! brightness-pvar (!! -1) (!!  0))
6                   (pref-grid-relative!! brightness-pvar (!! -1) (!!  1)))
7              (!! 3.0)))
8       (average-brightness-on-the-right
9         (/!! (+!! (pref-grid-relative!! brightness-pvar (!!  1) (!! -1))
10                  (pref-grid-relative!! brightness-pvar (!!  1) (!!  0))
11                  (pref-grid-relative!! brightness-pvar (!!  1) (!!  1)))
12             (!! 3.0)))
13      (average-brightness-overall
```

```
14        (/!! (+!! average-brightness-on-the-left
15                   average-brightness-on-the-right)
16             (!! 2.0))))
17     (if!! (>!! (absolute-value!! (-!! average-brightness-on-the-left
18                                       average-brightness-on-the-right))
19                (*!! (!! threshold) average-brightness-overall))
20           (!! 1)
21           (!! 0))))
```

Questa e altre funzioni simili, che confronteranno anche gli altri lati (sopra, sotto e nelle diagonali), vengono combinate con la seguente funzione, per garantire il ritrovo di ogni contorno:

```
1  (*defun find-all-edges!! (brightness-pvar threshold)
2    (if!! (or!! (=! (!! 1) (find-edges-between-left-and-right!!
3                             brightness-pvar threshold))
4                (=! (!! 1) (find-edges-between-above-and-below!!
5                             brightness-pvar threshold))
6                (=! (!! 1) (find-edges-between-upper-left-and-lower-right!!
7                             brightness-pvar threshold))
8                (=! (!! 1) (find-edges-between-lower-left-and-upper-right!!
9                             brightness-pvar threshold)))
10         (!! 1)
11         (!! 0)))
```

Il risultato è una nuova immagine con solo valori 0 o 1, dove gli uni indicano i bordi.

4.2 Ricerca di corrispondenze tra i bordi

Il prossimo passo consiste nell'*edge matching* tra le immagini dei bordi ottenute. Questo passo viene fatto con un processo di scorrimento: la prima immagine resta ferma, mentre la seconda viene fatta scorrere sulla prima. Per ogni scorrimento (*shift*), tutti i pixel della prima immagine vengono confrontati con la seconda: se un pixel della prima è uguale al pixel alla

stessa posizione nella seconda, allora si ha una corrispondenza. Il risultato è un immagine binaria dove gli uni indicano corrispondenze.

La funzione seguente è l'implementazione originale. L'array alla riga 1 memorizza le immagini risultanti per ogni scorrimento. L'operazione alla riga 6 implementa l'operazione di scorrimento: la funzione `pref-grid-relative` esegue un operazione di indicizzazione relativa, ovvero prende un immagine in input, la sposta di `i` posizioni in orizzontale e ritorna una nuova immagine. Il risultato è che ad ogni ciclo di `dotimes` (che corrisponde ad un ciclo `for` da 0 a 30), l'immagine destra sarà spostata orizzontalmente e confrontata con l'immagine di sinistra.

```
1  (defvar *array-of-pvars-holding-matches-at-each-shift* (make-array 30))
2
3  (*defun fillup-pvars-wherever-edges-align (left-edges right-edges)
4    (dotimes (i 30)
5      (aset (if!! (=!! left-edges
6                       (pref-grid-relative!! right-edges (!! i) (!! 0))
7                       )      ; ^ this PREF-GRID-RELATIVE!! accomplishes
8                  (!! 1)      ; the "sliding" process.
9                  (!! 0))
10           *array-of-pvars-holding-matches-at-each-shift*
11           i)))
```

A questo punto, l'algoritmo usa le informazioni appena calcolate per misurare la qualità dell'allineamento: per ogni pixel, si calcola quale scorrimento è il migliore tra tutti. L'indice dello scorrimento migliore corrisponderà direttamente all'altezza di quel pixel.

Lo scorrimento migliore viene calcolato in più fasi: prima, per ogni scorrimento si assegna un punteggio, che indicherà la qualità di quello scorrimento per quel pixel. Il punteggio per ogni pixel viene calcolato con un'altra operazione locale sui vicini: vengono contati il numero di pixel nell'intorno che hanno avuto una corrispondenza in quello scorrimento. Nell'implementazione originale, questo intorno è un quadrato di lato variabile (di solito 21) e vengono sommati tutti i pixel posti a 1 in precedenza.

La funzione seguente esegue l'operazione descritta in parallelo per ogni pixel di un immagine.

```
1  (*defun add-up-all-pixels-in-a-square (pvar width-of-square)
2    (let ((one-half-the-square-width (/ width-of-square 2)))
3      (*let ((total (!! 0)))
4        (dotimes (relative-x width-of-square)
5          (dotimes (relative-y width-of-square)
6            (*set total
7                  (+!! total
8                       (pred-grid-relative!!
9                         pvar
10                        (- relative-x one-half-the-square-width)
11                        (- relative-y one-half-the-square-width))))))
12       total)))
```

Per calcolare i punteggi per ogni scorrimento, basta usare la funzione precedente in un ciclo. La funzione seguente implementa questo ciclo, poi esclude dal calcolo della qualità tutti i pixel che non fanno parte dei bordi (per questi il punteggio è zero).

```
1  (defvar *array-of-pvars-holding-scores-at-each-shift* (make-array 30))
2
3  (*defun fillup-pvars-with-match-scores (width-of-square)
4    (dotimes (i 30)
5      (*let ((sum-of-all-nearby-pixels
6              (add-up-all-pixels-in-a-square
7                (aref *array-of-pvars-holding-matches-at-each-shift* i)
8                width-of-square)))
9        (*if (=!! (aref *array-of-pvars-holding-matches-at-each-shift* i)
10                 (!! 1)) ;;; record a score wherever there was a match-up
11             (*set sum-of-all-nearby-pixels
12                   *array-of-pvars-holding-scores-at-each-shift*
13                   i)))))
```

Infine, per trovare il migliore scorrimento per ogni pixel, basta usare la classica operazione di ricerca del massimo. La funzione seguente implemen-

ta prima un ciclo che trova i punteggi massimi (righe 6-10), poi trova a quali scorrimenti sono associati quei punteggi (righe 13-16). Il risultato della funzione è un immagine con gli indici degli scorrimenti migliori, che corrispondono all'altezza per ogni pixel.

```
1  (*defun find-the-shifts-of-the-highest-scoring-matches ()
2    (*let ((best-scores (!! 0))
3          (winning-shifts (!! 0)))
4      ;; the following DOTIMES loop makes sure that each pixel in the
5      ;; BEST-SCORES pvar contains the maximum score found at any shift.
6      (dotimes (i 30)
7        (*if (>!! (aref *array-of-pvars-holding-scores-at-each-shift* i)
8                  best-scores)
9             (*set best-scores
10                 (aref *array-of-pvars-holding-scores-at-each-shift* i))))
11     ;; the following DOTIMES loop records a "winning" shift at every
12     ;; pixel whose score is the best.
13     (dotimes (i 30)
14       (*if (=!! (aref *array-of-pvars-holding-scores-at-each-shift* i)
15                 best-scores)
16            (*set winning-shifts (!! (1+ i)))))
17     winning-shifts))
```

4.3 Disegno delle linee di livello

Il processo fatto finora ci dà un'immagine che rappresenta l'altezza dell'area ad ogni pixel. Le altezze memorizzate però valgono solo per i pixel che giacciono sui bordi: essenzialmente tra i bordi ci sono ancora dei buchi e serve un altro passo per trovare le informazioni mancanti per questi buchi. L'algoritmo trova queste informazioni usando un operazione di interpolazione: per ogni pixel la cui altezza è zero, calcola una media sugli intorni dei pixel.

Compiendo questa operazione una sola volta, solo i pixel giacenti vicino ai bordi si riempirebbero, dato che per un pixel ad altezza 0 a sua volta

circondato da pixel ad altezza 0 la media rimane 0. Per questo l'operazione viene ripetuta un certo numero di volte, riempiendo gradualmente i buchi ad ogni iterazione, finché tutti i pixel diventino coperti e nessuno abbia altezza 0.

```
1  (defun fill-in-web-holes (web-of-known-elevations times-to-repeat)
2    (dotimes (i times-to-repeat)
3      (*let ((not-fixed (zerop!! web-of-known-elevations)))
4        (*if not-fixed
5          (*set web-of-known-elevations
6                (/! (+!!
7                      (pref-grid-relative!!  web-of-known-elevations
8                        (!!  1) (!!  0))  ; neighbor to the right
9                      (pref-grid-relative!!  web-of-known-elevations
10                       (!!  0) (!!  1))  ; neighbor above
11                     (pref-grid-relative!!  web-of-known-elevations
12                       (!! -1) (!!  0))  ; neighbor to the left
13                     (pref-grid-relative!!  web-of-known-elevations
14                       (!!  0) (!! -1))) ; neighbot below
15                   (!! 4))))))
16   ;;; this is now a more or less smooth surface.
17   web-of-known-elevations)
```

L'ultimo passo consiste nel disegnare le linee di livello. L'implementazione originale prende in input il numero di linee da tracciare, divide le altezze in più intervalli e traccia una linea per ogni intervallo. La variabile `contour-line-interval` a riga 9 rappresenta quante "altezze" si deve saltare prima di tracciare una nuova linea.

```
1  (defun draw-contour-map (number-of-contour-lines
2                           pvar-of-smooth-continuous-elevations)
3    ;; The idea is to divide the whole range of elevations into
4    ;; a number of intervals, then to draw a contour line at every
5    ;; interval.
6    (let ((max-elevation (*max pvar-of-smooth-continuous-elevations))
7          (min-elevation (*min pvar-of-smooth-continuous-elevations))
8          (range-of-elevations (- max-elevation min-elevation))
```

```
 9          (contour-line-interval (/ range-of-elevations
10                                    number-of-contour-lines)))
11      ;; Now the variable CONTOUR-LINE-INTERVAL tells us how many
12      ;; elevations, or shifts, to skip between contour lines.
13      (if!! (zerop!!
14              (mod!! (-!! pvar-of-smooth-continuous-elevations
15                          (!! min-elevation))
16                            (!! contour-line-interval)))
17            (!! 1)        ;; this if!! draws all the elevation contours
18            (!! 0))))     ;; at once, returning a bit map suitable
19                          ;; for immediate display.
```

Capitolo 5

Progettazione e traduzione

Basandoci sulla descrizione dell'algoritmo nel capitolo precedente, ne presentiamo una traduzione in CUDA. Il codice nel suo complesso è reperibile al link [6].

5.1 Ricerca ed estrazione dei bordi

La prima funzione tradotta è stata quella che implementa la classificazione di un pixel come parte del bordo. Un numero pari a `width * height` di thread eseguono la funzione in parallelo: ogni thread è identificato da una coppia di coordinate `x` e `y`.

```
1  __device__ int find_edges_left_right(double *brightness,
2          int width, int height, int x, int y, double threshold)
3  {
4      double avg_left  = (brightness[idx(x-1, y-1, width, height)]
5                       +  brightness[idx(x-1, y  , width, height)]
6                       +  brightness[idx(x-1, y+1, width, height)]) / 3.0;
7      double avg_right = (brightness[idx(x+1, y-1, width, height)]
8                       +  brightness[idx(x+1, y  , width, height)]
9                       +  brightness[idx(x+1, y+1, width, height)]) / 3.0;
10     double overall   = (avg_left + avg_right) / 2.0;
11     return fabs(avg_left - avg_right) > CLAMP(threshold * overall,
12                                               0.0, 1.0);
```

```
13 }
```

Da questa funzione notiamo alcuni dettagli:

- La descrizione originale dell'algoritmo non dice niente sui tipi di dati: si sa solo che le immagini in input sono in bianco e nero, ovvero in scala di grigi, ma non dice nulla sul range di valori. La rappresentazione più comune per immagini in scala di grigi è avere un valore da 0 a 255 per ogni pixel; in questo programma, però, si è scelto un'altra rappresentazione: i valori sono in virgola mobile (`double`) e vanno da 0 a 1, dove 0 indica il nero e 1 indica il bianco. Si è scelto questo per via del fatto che l'operazione di moltiplicazione della soglia a riga 11 ha senso solo con questa rappresentazione (anche la soglia ha un valore da 0 a 1).

- Mentre in *Lisp le funzioni parallele agiscono su tutti gli elementi in automatico, in CUDA non è così: bisogna sempre fare un indicizzazione esplicita all'accesso degli array.

- Il codice in *Lisp non ha nessun controllo sui bordi della *pvar*: è il linguaggio stesso che fa questi controlli. CUDA non fa nessun controllo, per cui si è scelto che la funzione `idx`, che ha il compito di semplificare l'indicizzazione, faccia un *wrap-around* ogni volta che gli indici sono superiori ai bordi (`w` e `h`).

- Infine, la funzione `CLAMP` fa si che il risultato della moltiplicazione resti tra 0 e 1.

Come nell'implementazione originale, questa funzione viene implementata per tutti gli altri lati e le quattro funzioni risultanti vengono combinate:

```
1  __global__ void find_all_edges(double *brightness, int w, int h,
2          double threshold, u8 *edges)
3  {
4      DECLARE_INDEXES(w, h)
5      edges[IDX(x, y, w)] =
6                 find_edges_left_right(brightness, w, h, x, y, threshold)
7       ||        find_edges_top_bottom(brightness, w, h, x, y, threshold)
8       || find_edges_upleft_downright(brightness, w, h, x, y, threshold)
9       || find_edges_downleft_upright(brightness, w, h, x, y, threshold);
10 }
```

Il risultato viene messo in `edges`. Quest'ultimo è un immagine *binaria*, ovvero un immagine con solo 0 o 1, dove 0 questa volta indica il bianco e 1 il nero.

La macro `DECLARE_INDEXES` dichiara una coppia di coordinate `x` e `y` che identifica il thread che sta eseguendo, poi fa un controllo su quest'ultima per sapere se attivare il thread per questo kernel. Questo controllo serve nel caso vengano usati troppi thread (che può capitare se la larghezza o l'altezza delle immagini di input non sono multipli della dimensione del blocco predefinita). La macro viene usata in ogni kernel e viene espansa nel seguente codice:

```
1  int x = threadIdx.x + blockIdx.x * blockDim.x;
2  int y = threadIdx.y + blockIdx.y * blockDim.y;
3  if (x >= width || y >= height)
4      return;
```

La funzione `IDX`, utilizzata per indicizzare `edges`, è una versione di `idx` che non fa nessun *wrap-around*, dato che in questo caso è impossibile che `x` e `y` siano fuori dai bordi.

5.2 Ricerca di corrispondenze tra i bordi

La creazione di strutture dati in CUDA non è trasparente come in *Lisp: per molte strutture si richiede l'allocazione esplicita. Le seguenti funzioni mo-

strano come viene gestito l'array `matches`, che ha il compito di memorizzare, per ogni scorrimento, il risultato dell'*edge matching*.

```
1  __device__ u8 *matches[NUM_SHIFTS];
2
3  void allocate_matches(int width, int height)
4  {
5      u8 *tmp[NUM_SHIFTS];
6      for (int i = 0; i < NUM_SHIFTS; i++)
7          tmp[i] = ALLOCATE_GPU(u8, width * height);
8      cudaMemcpyToSymbol(matches, tmp, sizeof(tmp));
9  }
10
11 void free_matches()
12 {
13     u8 *tmp[NUM_SHIFTS];
14     cudaMemcpyFromSymbol(tmp, matches, sizeof(tmp));
15     for (int i = 0; i < NUM_SHIFTS; i++)
16         cudaFree(tmp[i]);
17 }
```

La funzione `ALLOCATE_GPU` è un semplice *wrapper* della funzione `cudaMalloc` che ne semplifica il funzionamento. Dato che l'*host* non ha accesso diretto alla memoria sul *device*, si usa la funzione `cudaMemcpyToSymbol` per aggiornare `matches`.

La funzione seguente è una traduzione diretta della funzione originale `fillup-pvars-wherever-edges-align`. Da notare che gli elementi di `matches` sono tutte immagini binarie.

```
1  __global__ void fillup_matches(u8 *left_edges, u8 *right_edges,
2          int width, int height)
3  {
4      DECLARE_INDEXES(width, height)
5      int index = IDX(x, y, width);
6      for (int i = 0; i < NUM_SHIFTS; i++) {
7          int shift = idx(x+i, y, width, height);
8          // ^ the +i accomplishes the sliding process
```

```
9           matches[i][index] = left_edges[index] == right_edges[shift];
10      }
11  }
```

La seguente porzione di codice mostra la traduzione della funzione `add-up-all-pixels-in-a-square`. Il doppio ciclo `for` è leggermente differente dalla versione originale, dato che inizia da `-half` e arriva a `+half`.

```
1   __global__ void addup_pixels_in_square(int i, int width, int height,
2           int square_width, i32 *total)
3   {
4       DECLARE_INDEXES(width, height)
5       u8 *pixels = matches[i];
6       int cur = IDX(x, y, width);
7       int half = square_width / 2;
8       for (int sy = -half; sy <= half; sy++) {
9           for (int sx = -half; sx <= half; sx++) {
10              int rel = idx(x + sx, y + sy, width, height);
11              total[cur] += (i32) pixels[rel];
12          }
13      }
14  }
```

Per calcolare i punteggi ci serviamo di una struttura dati simile a `matches`: le funzioni per allocare e liberare la memoria sono equivalenti.

Le seguenti funzioni corrispondono alla funzione originale `fillup-pvars-with-match-scores`, ma sono divise in due per via della `cudaMemset` fatta prima di riempire l'array `sum`: quest'ultimo viene usato in questo modo per evitare di fare più allocazioni.

```
1   __device__ i32 *scores[NUM_SHIFTS];
2
3   /* ... functions for managing scores omitted ... */
4
5   __global__ void record_score(int i, i32 *sum, int width, int height)
6   {
7       DECLARE_INDEXES(width, height)
```

```
 8      int index = IDX(x, y, width);
 9      // record a score whenever there was a match-up
10      if (matches[i][index] == 1)
11          scores[i][index] = sum[index];
12  }
13
14  void fillup_scores(int width, int height, int square_width, i32 *sum)
15  {
16      DECLARE_BLOCKS(width, height)
17      for (int i = 0; i < NUM_SHIFTS; i++) {
18          cudaMemset(sum, 0, sizeof(sum[0]) * width * height);
19          addup_pixels_in_square<<<NUM_BLOCKS, BLOCK_DIM_2D>>>(
20              i, width, height, square_width, sum
21          );
22          record_score<<<NUM_BLOCKS, BLOCK_DIM_2D>>>(
23              i, sum, width, height
24          );
25      }
26  }
```

Qui si può anche notare il metodo di chiamata di una funzione kernel: la macro `DECLARE_BLOCKS` si espande in:

```
1  const int BLOCKS_WIDTH = ceil_div(width, BLOCK_DIM_SIDE);
2  const int BLOCKS_HEIGHT = ceil_div(height, BLOCK_DIM_SIDE);
3  const dim3 NUM_BLOCKS = dim3(BLOCKS_WIDTH, BLOCKS_HEIGHT);
```

Da come si è già potuto notare, il programma utilizza i blocchi e i thread come se se fossero una griglia 2D. La funzione `ceil_div` serve per calcolare l'esatto numero di blocchi che servono a partire dalle dimensioni delle immagini.

Infine, la funzione seguente esegue la ricerca del punteggio massimo per ogni pixel. La funzione `MAX` a riga 11 calcola il massimo tra due valori.

```
1  // this function computes the web of known shifts. recall that
2  // the shift at each pixel corresponds directly to the elevation.
3  __global__ void find_highest_scoring_shifts(i32 *best_scores,
```

```
4         i32 *winning_shifts, int width, int height)
5 {
6     DECLARE_INDEXES(width, height)
7     int index = IDX(x, y, width);
8     // the following for loop makes sure that each pixel in the
9     // 'best_scores' variable contains the maximum score found at any
      shift.
10    for (int i = 0; i < NUM_SHIFTS; i++)
11        best_scores[index] = MAX(scores[i][index], best_scores[index])
12    // the following for loop records a "winning"
13    // shift at every pixel whose score is the best.
14    for (int i = 0; i < NUM_SHIFTS; i++)
15        if (scores[i][index] == best_scores[index])
16            winning_shifts[index] = i+1;
17 }
```

5.3 Disegno delle linee di livello

La seguente funzione implementa l'operazione di riempimento dei buchi. Questa particolare funzione usa una tecnica di *double buffer*, dato che non si può aggiornare un pixel di `web` mentre si leggono altri pixel. Il codice originale non richiede tale operazione per via della semantica di *Lisp: la funzione originale legge prima tutti i pixel, poi aggiorna l'immagine. Facendo la stessa coda in CUDA richiederebbe una sincronizzazione prima dell'aggiornamento.

```
1  __global__ void fill_web_holes_step(i32 *web, i32 *tmp, int width)
2  {
3      DECLARE_INDEXES(width, height)
4      if (tmp[IDX(x, y, width)] == 0)
5          web[IDX(x, y, width)] =
6              (tmp[IDX(x+1, y,   width)]  // neighbor to the right
7             + tmp[IDX(x,   y+1, width)]  // neighbor above
8             + tmp[IDX(x-1, y,   width)]  // neighbor to the left
9             + tmp[IDX(x,   y-1, width)]) // neighbor below
10            / 4;
```

```
11 }
12
13 i32 *fill_web_holes(i32 *web, i32 *tmp, int width, int height, int times)
14 {
15     DECLARE_BLOCKS(width, height)
16     for (int i = 0; i < times; i++) {
17         fill_holes_step<<<NUM_BLOCKS, BLOCK_DIM_2D>>>(web, tmp, width);
18         SWAP(web, tmp, i32 *);
19     }
20     // this is now a more or less smooth surface.
21     return web;
22 }
```

Infine, le funzioni seguenti implementano la creazione della *contour map*: di particolare nota sono le due funzioni `image_max` e `imagine_min`, che sono *built-in* in *Lisp ma non in CUDA, per cui sono state implementate a mano usando un operazione di riduzione.

```
1  __global__ void draw_contour_map_kernel(i32 *web, int width,
2          int num_lines, i32 max_elevation, i32 min_elevation, u8 *out)
3  {
4      // the idea is to divide the whole range of elevations into a number
5      // of intervals, then to draw a contour line at every interval.
6      DECLARE_INDEXES(width, height)
7      int i = IDX(x, y, width);
8      // the variable 'interval' tells us how many
9      // elevations, or shifts, to skip between contour lines.
10     i32 range    = max_elevation - min_elevation,
11         interval = range / num_lines;
12     // this line draws all the elevation contours at once.
13     out[i] = ((web[i] - min_elevation) % interval) == 0;
14 }
15
16 void draw_contour_map(i32 *web, int width, int height,
17         int num_lines, u8 *out)
18 {
19     DECLARE_BLOCKS(width, height)
```

```
20      i32 immax = image_max(web, width, height),
21          immin = image_min(web, width, height);
22      draw_contour_map_kernel<<<num_blocks, BLOCK_DIM_2D>>>(
23          web, width, num_lines, immax, immin, out
24      );
25  }
```

5.4 Ottimizzazioni

Un ottimizzazione molto ovvia che si può fare al programma è l'utilizzo di una *ghost area*: questo consiste, nella fase di inizializzazione, nell'aggiungere un'area inutilizzata attorno all'immagine, cosicché, quando l'indicizzazione di un immagine esce fuori dal bordo, questa restituirà un valore dentro l'area inutilizzata.

La *ghost area* potrebbe quindi sostituire il *wrap-around* presente nell'indicizzazione di molte immagini. In particolare, si può aggiungere una *ghost area* sia per le immagini in input, sia per le immagini dei bordi, sia per le immagini contenute in `matches`, dato che queste sono le uniche ad essere indicizzate usando la funzione `idx`.

Un'altra ottimizzazione sarebbe provare a rimuovere il *double buffering* nella funzione di riempimento dei buchi e utilizzare sincronizzazioni esplicite. Si dubita però che questo porti a prestazioni migliorate. Questa tecnica però potrebbe portare ad un leggero miglioramento nell'uso della memoria.

5.5 Altri dettagli

Dal punto di vista della *Stereo Vision*, i risultati di questo algoritmo non sembrano molto buoni: le immagini che si ottengono sono molto rumorose. Non si capisce bene se questo sia colpa dell'algoritmo o dell'implementazione in CUDA; dato che lo scopo, però, è quello la traduzione dell'algoritmo, si è deciso di non indagare su questo problema.

Capitolo 6

Valutazione delle prestazioni

6.1 Descrizione dei test

Per testare le prestazioni dell'algoritmo, sono state create quattro versioni: una versione seriale, una versione parallela (da cui proviene il codice presentato) e due versioni seriale e parallela che usano la *ghost area*.

Per quanto riguarda i test, si è scelto di usare alcune coppie di immagini di varie grandezze. Le coppie di immagini sono tutte uguali, avendo le dimensioni come unica differenza; si è provato anche ad usare immagini diverse, ma non sono state riscontrate differenze reali nei tempi di esecuzione.

6.2 Metriche usate

Lo *Speedup* è la metrica più utilizzata nella misurazione delle prestazione di un programma parallelo e rappresenta l'accelerazione che ha avuto il programma tramite la parallelizzazione. In particolare, lo *Speedup* si misura con la formula:

$$S(P) = \frac{T_s}{T(P)}$$

Dove P è il numero di processori usati per il programma parallelo, T_s è il tempo di esecuzione del programma seriale e e $T(P)$ il tempo di esecuzione del programma parallelo utilizzando P processori.

La misurazione dello *speedup* però non risulta ideale quando si misura un programma CUDA: questa deriva dal fatto che in CUDA non si può decidere a priori quanti thread usare, dato che non si ha controllo diretto dell'hardware.

Un'altra metrica che si usa per programmi CUDA è il *throughput* (produttività), ovvero il numero di elementi processati per unità di tempo. Cosa significa il "numero di elementi" dipende da programma a programma. Nel nostro caso, il *throughput* è misurato come numero di pixel processati per secondo.

Solitamente il *throughput* aumenta con l'aumentare dell'input fino a raggiungere un certo limite, dovuto dal fatto che si sfrutta sempre meglio il parallelismo offerto dalla GPU.

6.3 Dati raccolti

Di seguito si possono vedere i dati raccolti sia per lo *Speedup* che per il *throughput* in forma di grafici.

La figura 6.1 un grafico dei tempi per i programmi seriali. Si noti in particolare il tempo per le immagini più grandi: 2 minuti e mezzo per la versione normale e mezzo minuto per quella con *ghost area*. Alla luce di ciò si può dire che la *ghost area* risulta come una buona ottimizzazione.

In figura 6.2 si trovano i tempi per i programmi paralleli, che risultano tutti minori di un secondo. Un cosa che non si nota dai grafici è che i tempi di esecuzione per la versione con *ghost area* risultano più piccoli di un ordine di grandezza rispetto alla versione normale.

In figura 6.3 troviamo lo *speedup* tra le versioni seriali e parallele, rispettivamente senza e con *ghost area*. Da questi grafici osserviamo uno *speedup*

di almeno un fattore di 150 per le versioni normali e di 30 per le versioni con *ghost area*.

Infine, la figura 6.4 mostra il *throughput* delle due versioni parallele. Il *throughput* qui viene misurato in pixel per secondo, contando anche il numero di volte in cui viene processata un intera immagine. Si noti come il grafico mostri un limite maggiore di 8-9 miliardi di pixel per secondo per il *throughput*.

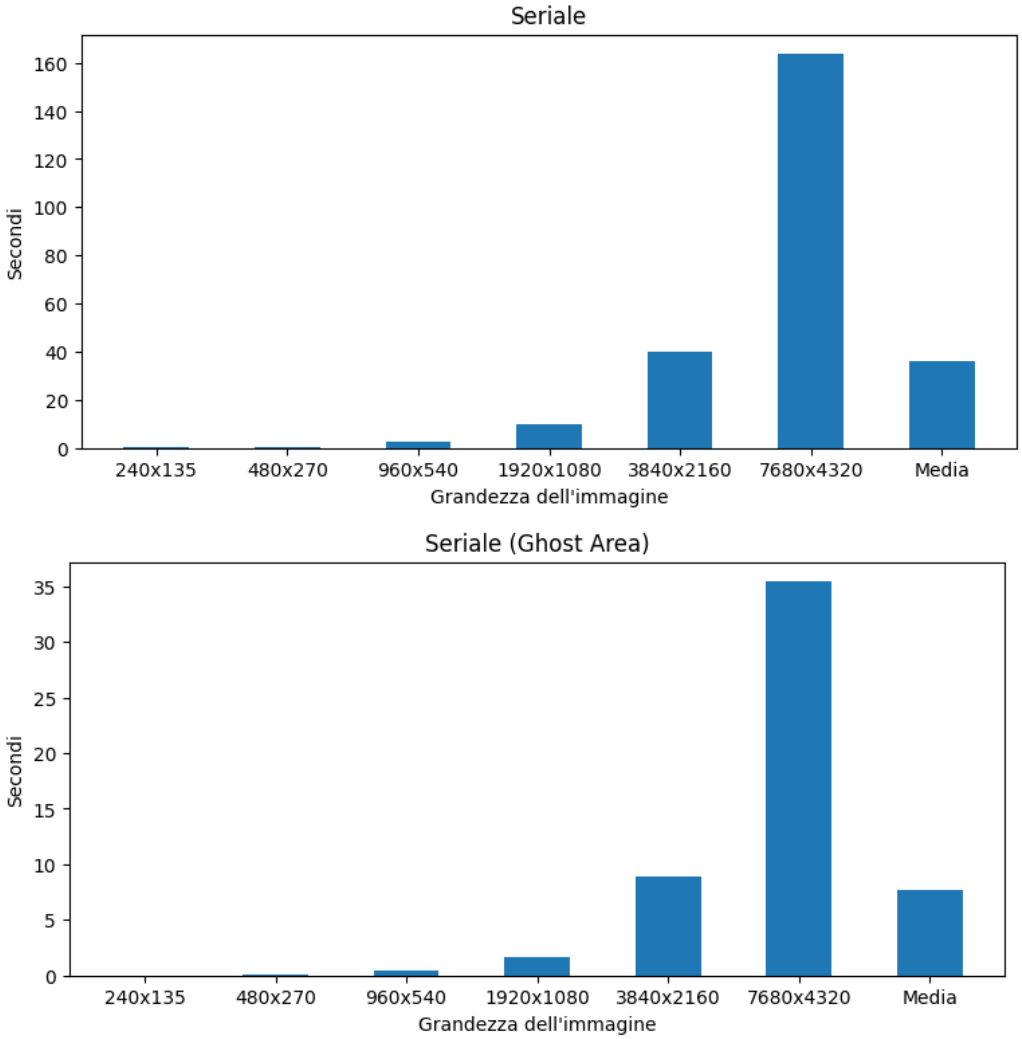

Figura 6.1: I tempi di esecuzione per i programmi seriali.

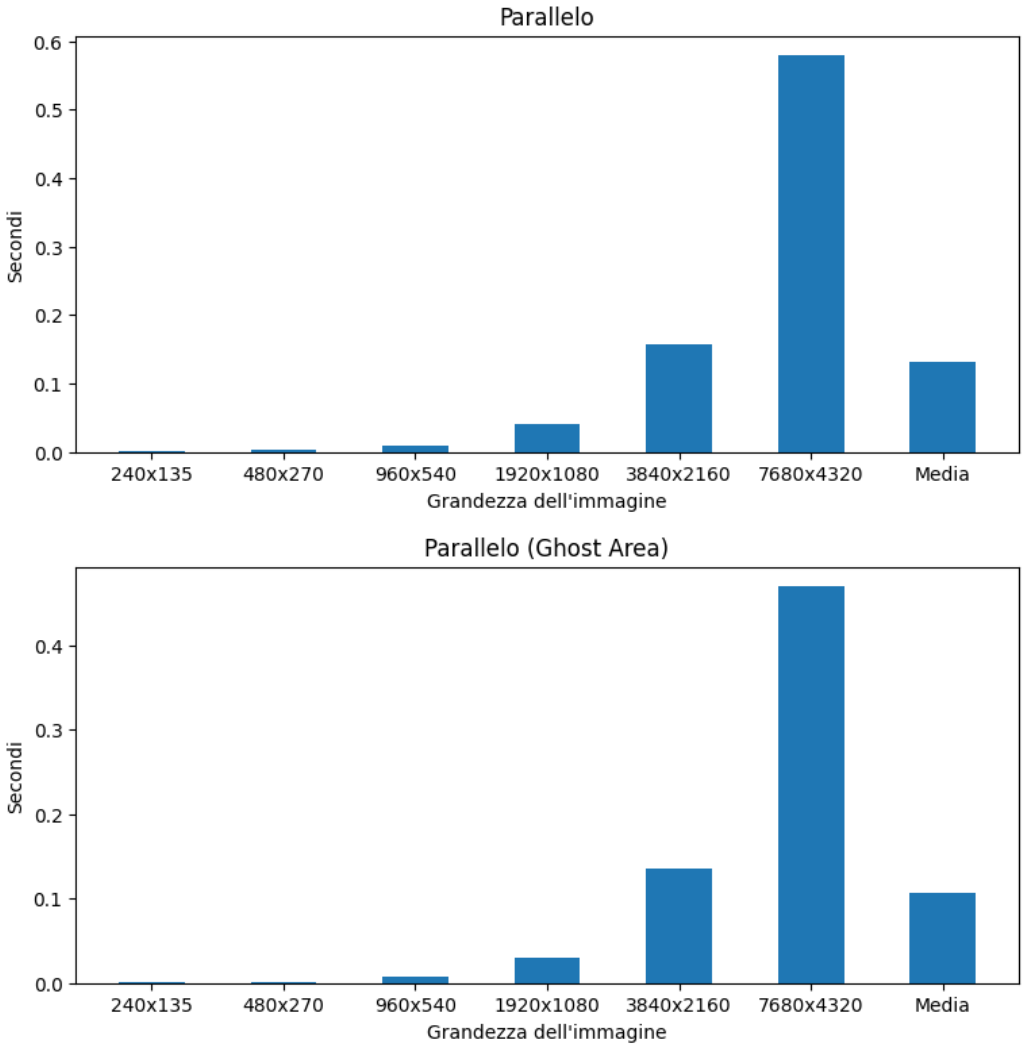

Figura 6.2: I tempi di esecuzione per i programmi paralleli.

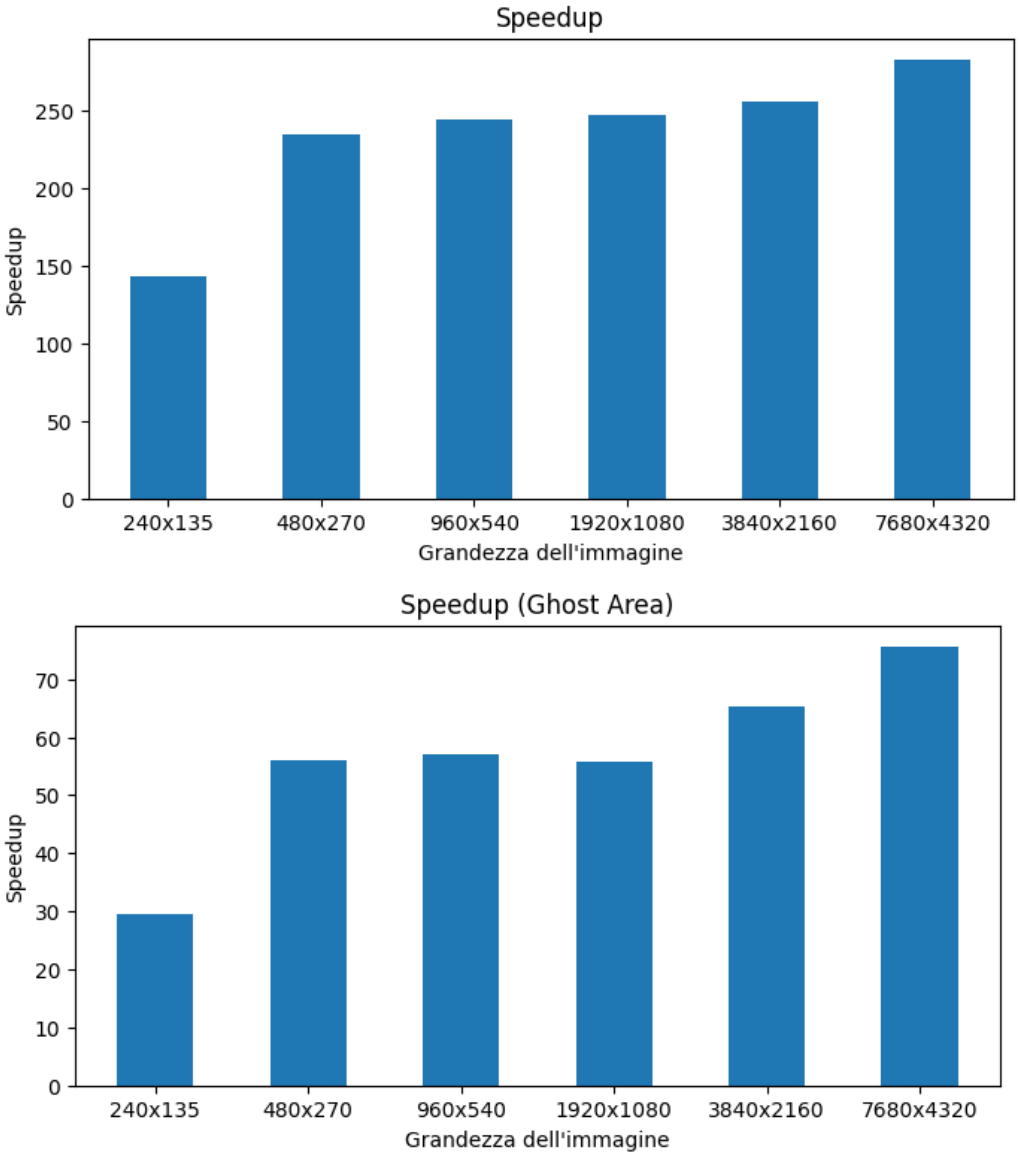

Figura 6.3: Gli speedup tra i programmi seriali e paralleli. Gli speedup sono tra i due programmi senza *ghost area* e i due programmi con la *ghost area*.

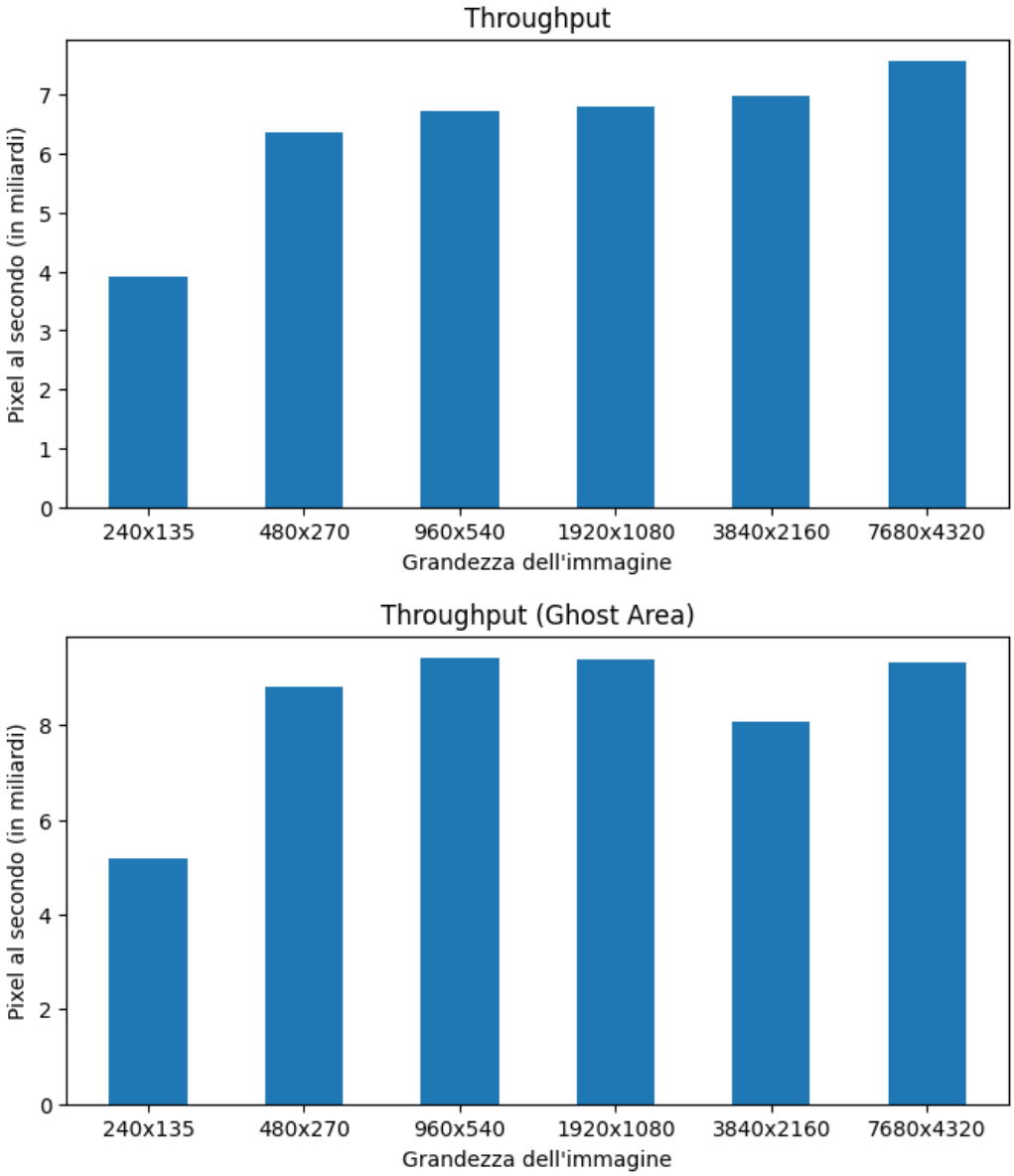

Figura 6.4: Il throughput nei due programmi paralleli.

Conclusioni

In questa tesi abbiamo analizzato e tradotto un vecchio algoritmo di *stereo matching* per un hardware più moderno. In particolare, abbiamo analizzato alcune delle difficoltà che si possono riscontrare in questi lavori di traduzione e abbiamo analizzato quali sono le scelte migliori da fare e le ottimizzazione che si possono applicare nell'ambito CUDA.

L'algoritmo si presta molto bene per la parallelizzazione su GPU; nonostante ciò, mentre i risultati di prestazione sono soddisfacenti, i risultati dell'algoritmo lo sono di meno, con immagini "rumorose" come output. In sviluppi futuri sarebbe interessante sviluppare una versione migliore dell'algoritmo e vedere come si presta alla parallelizzazione su GPU.

Un altro sviluppo interessante sarebbero ottimizzazioni più elaborate: ad esempio, si potrebbe cercare di "compattare" le immagini binarie create temporaneamente durante l'esecuzione dell'algoritmo, usando un solo bit per pixel invece che un byte.

Bibliografia

[1] Cuda c++ programming guide. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[2] J.K. Aggarwal Yang Liu. *Handbook of Image and Video Processing*, chapter 3. Academic Press, second edition, 2005.

[3] Hamid Fsian, Vahid Mohammadi, Pierre Gouton, and Saeid Minaei. Comparison of stereo matching algorithms for the development of disparity map, 2022. URL https://arxiv.org/abs/2210.15926.

[4] Introduction to data level parallelism. Technical report, Thinking Machines Corporation, Aprile 1986. URL https://archive.org/details/06Kahle002142.

[5] *starLisp reference*. Thinking Machines Corporation, Settembre 1988. URL https://archive.org/details/bitsavers_thinkingMafereneceVersion5.0Sep1988_4463275/mode/2up.

[6] URL https://github.com/chrg127/stereomatching.

Ringraziamenti

Ringrazio la mia famiglia per il sostegno e il supporto che mi hanno dato. Ringrazio anche i miei amici e i miei compagni di università, che mi hanno accompagnato e mi hanno aiutato durante questo percorso. Ringrazio infine il prof. Moreno Marzolla per avermi aiutato con questa tesi di laurea e per essere stato sempre gentile e professionale con me.