ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica - Scienza e Ingegneria - DISI

Corso di Laurea in Ingegneria e Scienze Informatiche

IMPLEMENTAZIONE CUDA DI UN ALGORITMO DI PROIEZIONE TOMOGRAFICA

Tesi di Laurea in Ingegneria e Scienze Informatiche

Relatore:
Prof. Moreno Marzolla

Presentata da:
Enrico Marchionni

Correlatore:
Prof. Elena Loli Piccolomini

Sessione marzo 2025

Anno Accademico 2023/2024

Sommario

La tomografia computerizzata (TC) è una tecnica diagnostica che sfrutta le radiazioni ionizzanti (o raggi X) per ottenere immagini dettagliate di un oggetto in esame. Questa tecnica si basa sulla raccolta, da parte di un rilevatore, di diverse viste, chiamate proiezioni. Tali proiezioni sono il risultato dell'attenuazione dei raggi X, che, diretti da una sorgente verso un rilevatore, attraversano l'oggetto di studio. Ruotando la sorgente e il rilevatore lungo una traiettoria prestabilita, vengono raccolte molteplici proiezioni. Le informazioni ottenute vengono successivamente trasferite a un calcolatore, che ha il compito di convertirle in immagini, fondamentali per la ricostruzione di aree 2D o del volume 3D dell'oggetto scansionato. La TC consente di analizzare l'oggetto esaminato, rivelando proprietà di densità della sua struttura interna precedentemente ignote, senza dover ricorrere a tecniche invasive.

Dal punto di vista matematico, la ricostruzione dell'oggetto a partire dalle proiezioni rappresenta un problema inverso, in cui si determinano le cause a partire dagli effetti osservati. In questa tesi viene invece affrontato il problema diretto, partendo dal corpo studiato, si devono determinare le proiezioni che una TC 3D potrebbe generare.

Gli obiettivi di questa tesi progettuale sono:

- Sviluppare un programma per GPU NVIDIA ottimizzato dell'algoritmo di proiezione proposto da Siddon [Sid85], utilizzando il linguaggio C con CUDA, partendo da una sua implementazione multicore per CPU scritta in C con OpenMP [Col24].
- Analizzare le prestazioni della versione per GPU realizzata e confrontarle con quelle della versione per CPU già nota.

Di seguito si riassumono i principali contenuti dei capitoli che compongono questa tesi.

Nell'introduzione viene presentata una panoramica sull'evoluzione della tecnica e sui suoi principi fondamentali.

Successivamente, vengono approfonditi i fondamenti matematici e geometrici alla base del problema diretto.

Nel capitolo dedicato alla descrizione del progetto di tesi, si analizzano le versioni OpenMP di partenza, il processo di parallelizzazione adottato per lo sviluppo della versione CUDA e la verifica della correttezza dei risultati.

Segue la valutazione delle prestazioni, in cui la versione CUDA viene confrontata con l'implementazione OpenMP iniziale.

Infine, sulla base dei risultati ottenuti, vengono tratte le opportune conclusioni.

Indice

Capitolo 1

Introduzione

Introdotta nel 1972, la tomografia computerizzata a raggi X (TC) è una tecnica di imaging, che consente di esplorare le strutture interne di un corpo in modo non invasivo, ampiamente utilizzata per individuare eventuali anomalie nei volumi corporei esaminati. In ambito medico è diventata uno strumento essenziale per lo studio di molteplici condizioni cliniche.

L'acronimo TAC (tomografia "assiale" computerizzata) indica che l'esame veniva condotto lungo un solo asse, con sezioni perpendicolari alla lunghezza del corpo. Oggi esistono macchinari più moderni e la tomografia computerizzata non non è più limitata alla modalità assiale, le immagini sono acquisite con una tecnica a spirale che consente di ottenere immagini tridimensionali. Il termine TAC è ormai considerato improprio e obsoleto.

La TC viene spesso paragonata alla radiografia tradizionale. Entrambe sono tecniche di imaging basate sui raggi X, ma hanno differenze fondamentali in termini di acquisizione delle immagini, qualità e applicazioni. La radiografia è una tecnica rapida ed economica che fornisce immagini bidimensionali con sovrapposizione delle strutture e, di conseguenza, minore dettaglio. Essa espone l'oggetto a una bassa dose di radiazioni ed è ideale per lo studio di strutture ad alto contrasto. La TC, invece, utilizza varie scansioni da diverse angolazioni per generare immagini dettagliate in sezioni assiali (2D) o volumi (3D), permettendo una visualizzazione migliore dell'oggetto, ma a un costo più elevato e con una maggiore esposizione alle radiazioni rispetto alla radiografia.

Segue una breve introduzione alla TC; per ulteriori approfondimenti, si rimanda a [FPN11] e [MP21].

1.1 Cenni storici

Gli studi sull'imaging medico ebbero inizio dopo la scoperta dei raggi X da parte di Wilhelm Röntge, nel 1895. Non appena gli scienziati compresero la capacità dei raggi X di attraversare gli oggetti, le immagini 2D (radiografie) furono ampiamente utilizzate per studiare i volumi interni del corpo umano. Tuttavia, queste immagini forniscono solo una rappresentazione bidimensionale del volume acquisito. La teoria matematica di Johann Radon e gli studi del fisico tedesco Grossmann, uniti al desiderio di superare i limiti della radiografia a raggi X, portarono allo sviluppo della tomografia come nuovo strumento di analisi degli oggetti. In particolare, gli studi del matematico austriaco Johann Radon del 1917 dimostrarono teoricamente la possibilità

di ricostruire un oggetto tridimensionale a partire da un numero infinito di proiezioni bidimensionali dello stesso. L'adattamento di queste teorie all'uso di un numero finito di proiezioni ha reso concreta l'applicazione della TC.

Con l'avvento dei computer, intorno al 1970, la TC ha rivoluzionato l'imaging diagnostico, permettendo di ricostruire l'anatomia tridimensionale di un oggetto. La transizione dell'imaging medico da semplice curiosità sperimentale a vera e propria applicazione clinica è stata resa possibile, in gran parte, dal lavoro dell'ingegnere inglese Godfrey Hounsfield.

Nell'ottobre del 1971, con l'aiuto di un prototipo di tomografo, nel reparto radiologico dell'Atkinson Morley Hospital di Londra, venne prodotta la prima immagine di un encefalo, che mostrava chiaramente una lesione tumorale nel lobo frontale. Questo tomografo permetteva movimenti di traslazione e rotazione del complesso tubo radiogeno-detettore e produceva un'immagine su una matrice di 80×80 con una risoluzione spaziale di 0,5 cm, richiedendo per l'acquisizione e la ricostruzione di ciascuna fetta, rispettivamente, 4 e 7 minuti.

Nel 1972, 70 pazienti furono sottoposti a esame tomografico e i risultati furono presentati al congresso annuale del British Institute of Radiology di Londra. Nello stesso anno, essi vennero pubblicati in un articolo del London Times, suscitando grande entusiasmo per questa nuova tecnica diagnostica.

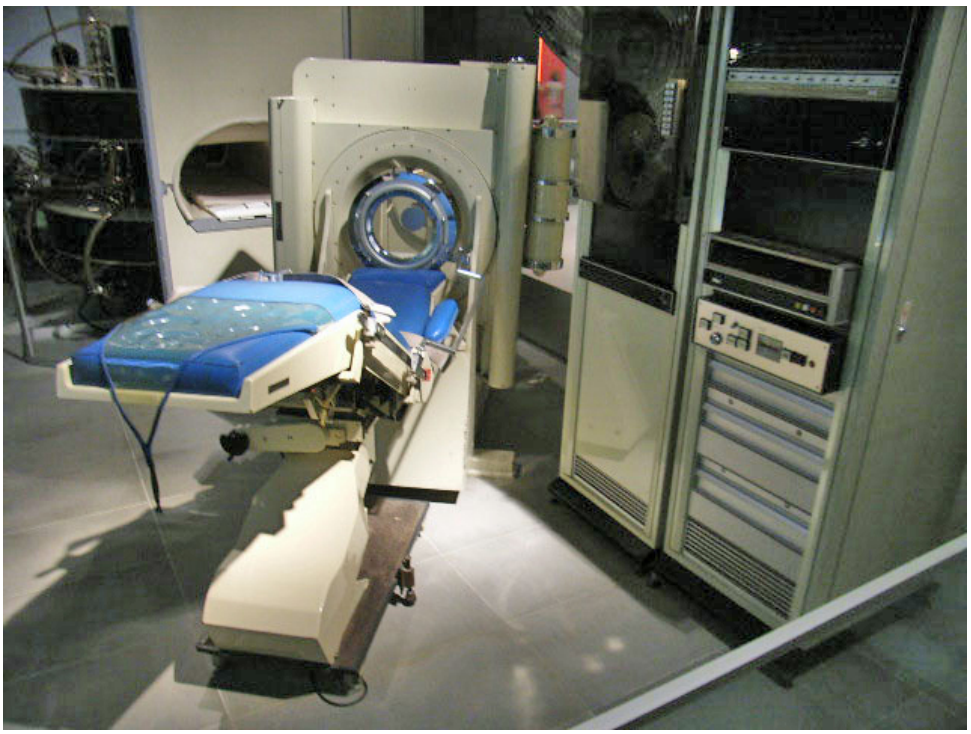


Figura 1.1: EMI Mark 1, il primo tomografo computerizzato commerciale. Sulla destra si vede il computer usato per processare i dati raccolti dal tomografo.

La Figura 1.1 mostra l'EMI Mark 1, il primo tomografo computerizzato commerciale prodotto dalla EMI nel 1973 per lo studio del cranio.

Oggi, grazie ai continui avanzamenti tecnologici e all'innovazione nelle metodologie, la TC è sempre più diffusa e impiegata in svariati ambiti, dalla diagnostica medica alla ricerca scientifica, fino alle applicazioni industriali e archeologiche. La maggiore precisione e velocità di

acquisizione dei dati nei tomografi insieme alle necessità di migliorare la qualità delle analisi e ottimizzare i tempi di risposta ha reso necessaria l'evoluzione dei software di elaborazione delle proiezioni ottenute, argomento di cui questa tesi si occupa.

1.2 Le tecnologie

L'evoluzione dei dispositivi tomografici ha seguito un percorso di costante innovazione tecnologica, migliorando progressivamente la qualità delle immagini diagnostiche e la velocità di acquisizione dei dati.

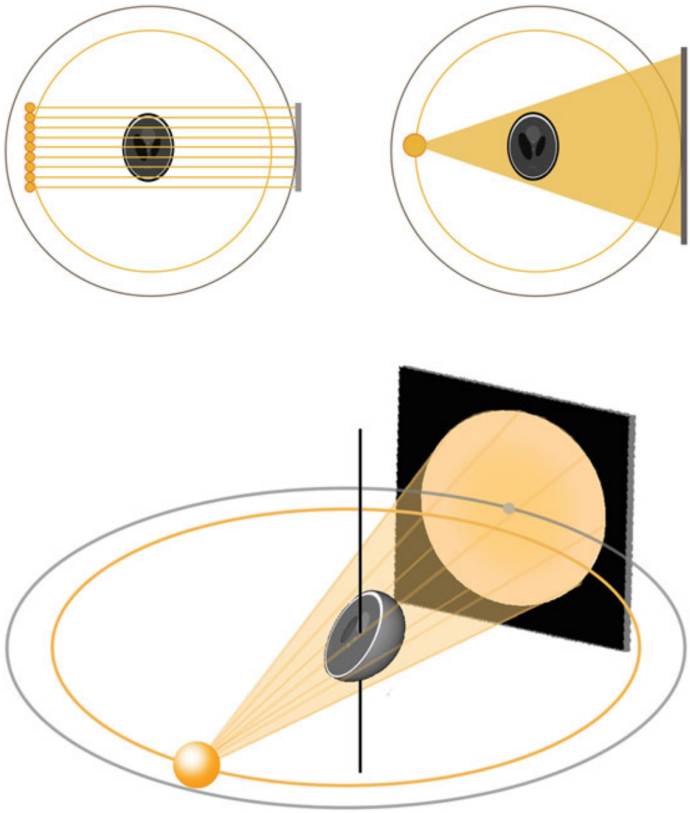


Figura 1.2: Schizzi di dispositivi tomografici, dalla tecnologia primordiale con scansioni parallele a raggi X (in alto a sinistra) alle soluzioni più moderne che utilizzano fasci a ventaglio per la TC 2D (in alto a destra) e fasci conici per la TC 3D (in basso). [MP21]

Si osservi la Figura 1.2. La prima generazione di scanner per la TC, sviluppata intorno al 1970, utilizzava un fascio di raggi X paralleli con movimenti alternati di traslazione e rotazione. Questo approccio, sebbene pionieristico, richiedeva tempi di scansione prolungati, rendendo possibile l'analisi solo di organi statici come l'encefalo. Successivamente, la seconda generazione di scanner introdusse fasci di raggi a ventaglio, migliorando l'efficienza della scansione e riducendo i tempi necessari per acquisire le immagini. Come ulteriore evoluzione, la terza generazione adottò un fascio di raggi a forma di cono per acquisire immagini tridimensionali in una singola rotazione attorno al paziente, offrendo dettagli ad alta risoluzione con un'esposizione inferiore alle radiazioni rispetto alle generazioni precedenti.

In questa tesi verrà simulata una tecnologia di proiezione per la TC 3D, simile a quella a fasci conici, con l'obiettivo di generare scansioni 2D del corpo analizzato.

1.3 Il processo

Indipendentemente dalla tecnologia usata, la scansione nella TC consiste nell'acquisizione di numerose proiezioni dello stesso oggetto in esame, ottenute da varie angolazioni diverse, seguendo una specifica traiettoria. Durante questo processo, un fascio di raggi X attraversa l'oggetto, e i dati raccolti dai rivelatori vengono elaborati da un software per ricostruire un'immagine dettagliata tridimensionale della sua struttura interna.

Dal punto di vista matematico, nella TC 3D si parla di:

- Problema diretto: partendo dall'oggetto 3D si calcolano le proiezioni 2D ottenendo il sinogramma, cioè l'insieme di tutte le proiezioni.
- Problema inverso: partendo dalle proiezioni 2D viene ricostruito l'oggetto 3D.

Entrambi questi problemi possono essere affrontati mediante diversi algoritmi disponibili in letteratura, ciascuno con specifici vantaggi e limitazioni in termini di accuratezza, costo computazionale e robustezza ai disturbi.

Nei capitoli successivi, si approfondirà il problema diretto e la sua risoluzione mediante l'algoritmo di proiezione proposto da Siddon [Sid85], utilizzato per simulare il passaggio dei raggi X attraverso l'oggetto per ottenere il sinogramma.

Capitolo 2

Fondamenti matematici e geometrici

Durante una scansione tomografica sono acquisite numerose proiezioni attorno all'oggetto di studio. Nel caso più semplice, che verrà analizzato in questa tesi, la sorgente e il rilevatore si muovono lungo una traiettoria circolare. Intuitivamente più sono le proiezioni acquisite, più la ricostruzione dell'oggetto sarà accurata. D'altra parte, specialmente in ambito medico, ridurre l'esposizione del corpo alle radiazioni è vantaggioso per il paziente. Di conseguenza, il numero di proiezioni generate varia sensibilmente a seconda dell'applicazione.

Fisicamente, i dati di proiezione rappresentano l'assorbimento dei fotoni che compongono i raggi X. Una proiezione può essere rappresentata come una matrice, in cui ogni valore indica il livello di attenuazione subito dai raggi X dopo aver attraversato il corpo oggetto di studio. Considerando la matrice proiezione, i valori rappresentano i vari livelli di attenuazione: un raggio che attraversa una regione più densa del corpo subisce una maggiore attenuazione rispetto a un raggio che attraversa una regione meno densa.

Nelle sezioni successive verrà illustrato il modello matematico alla base del calcolo, evidenziandone il legame con l'algoritmo risolutivo. Infine, saranno esaminate le considerazioni necessarie per risolvere il problema dal punto di vista geometrico.

2.1 Il modello matematico

Di seguito verranno discusse le nozioni matematiche fondamentali per comprendere, dal punto di vista algebrico, il calcolo alla base della creazione delle proiezioni, vedi [MP21].

2.1.1 La legge di Lambert-Beer

Il modello fisico che descrive l'assorbimento dei fotoni in termini di coefficienti di attenuazione è descritto dalla legge di Lambert-Beer.

Il meccanismo fisico che porta all'attenuazione dell'intensità di un raggio è solitamente descritto da un singolo coefficiente di attenuazione $\mu = \mu(w) \geq 0$ che dipende dalla posizione nel segmento che approssima il suo percorso, indicata da w. Questi coefficienti determinano i vari $m(w)$, cioè le intensità del raggio nella posizione indicata da w.

La **legge di Lambert-Beer** per il calcolo della proiezione del coefficiente di attenuazione lungo un segmento di lunghezza W è:

$$P_W\mu = -\ln\left(\frac{m}{m_0}\right) = \int_0^W \mu(w)\,dw \tag{2.1}$$

dove:

- $P_W\mu$ è un valore che indica l'attenuazione di un raggio; la notazione indica la *proiezione integrale* di μ lungo un segmento di lunghezza W.
- W è la lunghezza totale del percorso attraverso il materiale.
- $\mu(w)$ è una funzione continua che descrive il coefficiente di attenuazione alla posizione w lungo il raggio.
- m è l'intensità finale del raggio, dopo aver attraversato il materiale.
- m_0 è l'intensità iniziale del raggio.

2.1.2 La trasformata di Radon

La **trasformata di Radon**, in un modello continuo, è data dall'insieme delle proiezioni acquisite lungo l'intera traiettoria circolare. La rappresentazione grafica di tutti i dati misurati nel caso bidimensionale è chiamata sinogramma.

2.1.3 Il caso discreto

Nel caso discreto il corpo è diviso in volumi più piccoli, detti **voxel**, che sono elementi di dimensione molto piccola in cui l'oggetto è approssimato. Per ciascun voxel il coefficiente di attenuazione è un valore approssimato costante.

L'Equazione 2.1 nel caso discreto, per calcolare una singola proiezione, diventa:

$$g_i = \sum_{j=0}^{N} M_{i,j} f_j \quad \forall i \in 1, \ldots, N_p \tag{2.2}$$

dove:

- g_i è l'attenuazione subita dall'i-esimo raggio.
- i è l'i-esimo raggio, degli N_p considerati per quella proiezione.
- j è il j-esimo voxel, degli N totali che approssimano il corpo.
- $M_{i,j}$ è una matrice di $N_p \times N$ elementi che per ogni raggio i definisce la lunghezza della porzione sottesa al volume di ciascun voxel j.
- f_j è il valore del coefficiente di attenuazione assunto nel volume interno al j-esimo voxel.

L'indice j, nel sistema cartesiano, è determinato dalla combinazione delle tre coordinate cartesiane, in modo da identificare un unico indice.

Quindi dal punto di vista matematico, per il calcolo delle proiezioni, viene considerata un'approssimazione dell'oggetto analizzato, vedi Figura 2.1.

Ritornando all'Equazione 2.2, si può notare che il numero totale di voxel N, nel caso di uno spazio cartesiano, è dato da $N_x \times N_y \times N_z$.

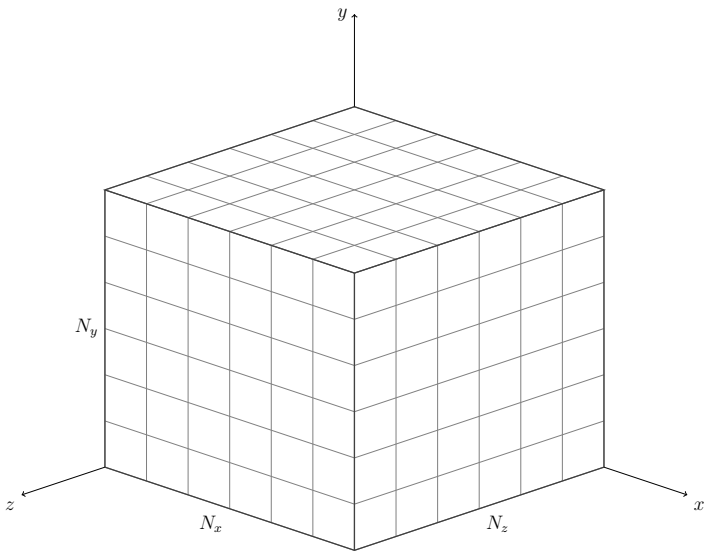


Figura 2.1: Esempio di oggetto cubico suddiviso in $N = N_x \times N_y \times N_z$ voxel cubici. Nel caso generale i voxel sono dei parallelepipedi retti, nel caso più semplice possono essere dei cubi con una dimensione $d = d_x = d_y = d_z$ fissata in base alle necessità.

Nel caso discreto della trasformata di Radon, il numero di proiezioni dipende dal numero di posizioni considerate in cui si effettuano le scansioni, indicato di seguito con N_θ. In ciascuna posizione, la sorgente e il rilevatore sono posti in modo da poter analizzare l'oggetto da diverse angolazioni.

2.2 Algoritmo risolutivo

Come già accennato in precedenza, alla base del progetto di tesi sviluppato si applica l'algoritmo di proiezione proposto da Siddon [Sid85].

Considerando quanto già visto nel modello matematico, nel nostro caso l'algoritmo di Siddon si occupa di calcolare la matrice M, mentre il vettore f deve essere già noto in partenza. Questi dati devono essere determinati per poter procedere con i passi successivi.

L'algoritmo svolge una computazione equivalente a quella descritta nell'Equazione 2.2:

$$g = \sum_i \sum_j \sum_k l_{i,j,k} \rho_{i,j,k} \tag{2.3}$$

dove gli indici i, j e k individuano un singolo voxel nello spazio tridimensionale. In particolare, per una specifica configurazione di (i, j, k), l rappresenta la lunghezza della porzione del segmento che approssima un raggio nello spazio ed è sottesa in quel voxel (equivalente di M) mentre ρ rappresenta la densità del voxel (equivalente di f).

2.2.1 La matrice M

La matrice M ci permette di identificare, per ogni raggio, la lunghezza del segmento di raggio che attraversa ciascun voxel.

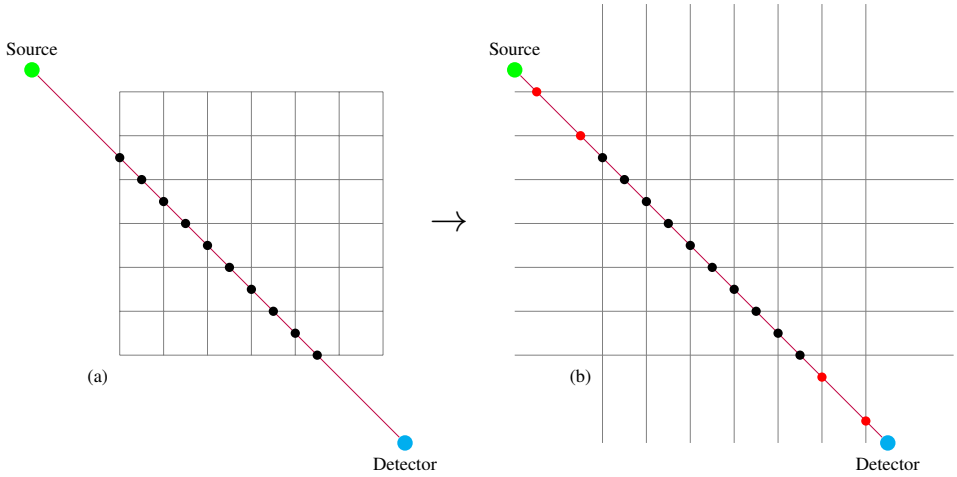


Figura 2.2: Esempio bidimensionale di una griglia di pixel, mostrato in figura *(a)*, che è rappresentata da due insiemi di rette parallele, come illustrato in figura *(b)*. Nell'algoritmo, invece, si considerano voxel come piani paralleli in un contesto tridimensionale. Di conseguenza, le intersezioni tra il raggio e i piani esterni alla griglia, indicate dai punti rossi, non devono essere prese in considerazione.

I punti chiave dell'algoritmo, nella determinazione della matrice M, possono essere così riassunti:

- L'algoritmo considera, invece dei singoli voxel indipendenti, tre insiemi di piani ortogonali ed equidistanti tra loro che vanno a costruire la griglia di discretizzazione del corpo nello spazio, vedi Figura 2.2.
- Si calcolano i punti di intersezione tra i raggi e i piani della griglia che delimitano i voxel.
- La lunghezza dei segmenti sottesi ai voxel viene determinata come differenza tra punti di intersezione consecutivi, ottenendo così la matrice M.

2.2.2 Il vettore f

Il vettore f riassume ciò che sappiamo dell'oggetto da scansionare. In particolare esso rappresenta il coefficiente di attenuazione assunto nella regione interna al volume occupato da ciascun voxel.

Nel progetto di tesi sviluppato, i coefficienti di attenuazione adottati sono: 1 per una regione "molto densa", approssimando un unico tipo di materiale, e 0 per una regione "poco densa", approssimando l'aria.

I tre oggetti considerati sono:

- **Cubo**: stabilita la lunghezza del lato del cubo che si intende rappresentare, gli elementi di f all'interno della regione cubica sono posti a 1, altrimenti a 0.
- **Cubo con cavità sferica**: stabilita la lunghezza del lato del cubo che si intende rappresentare, gli elementi di f all'interno della regione cubica, a eccezione di quelli la cui distanza da un punto stabilito all'interno dell'oggetto sia inferiore a un raggio stabilito, sono posti a 1, altrimenti a 0.
- **Semisfera**: stabilita la lunghezza del raggio della semisfera che si intende rappresentare, gli elementi di f all'interno della regione semisferica sono posti a 1, altrimenti a 0.

2.3 Considerazioni geometriche

Di seguito vengono approfonditi i riferimenti geometrici necessari per l'implementazione dell'algoritmo di Siddon.

2.3.1 Posizionamento di sorgente, oggetto e rilevatore nello spazio

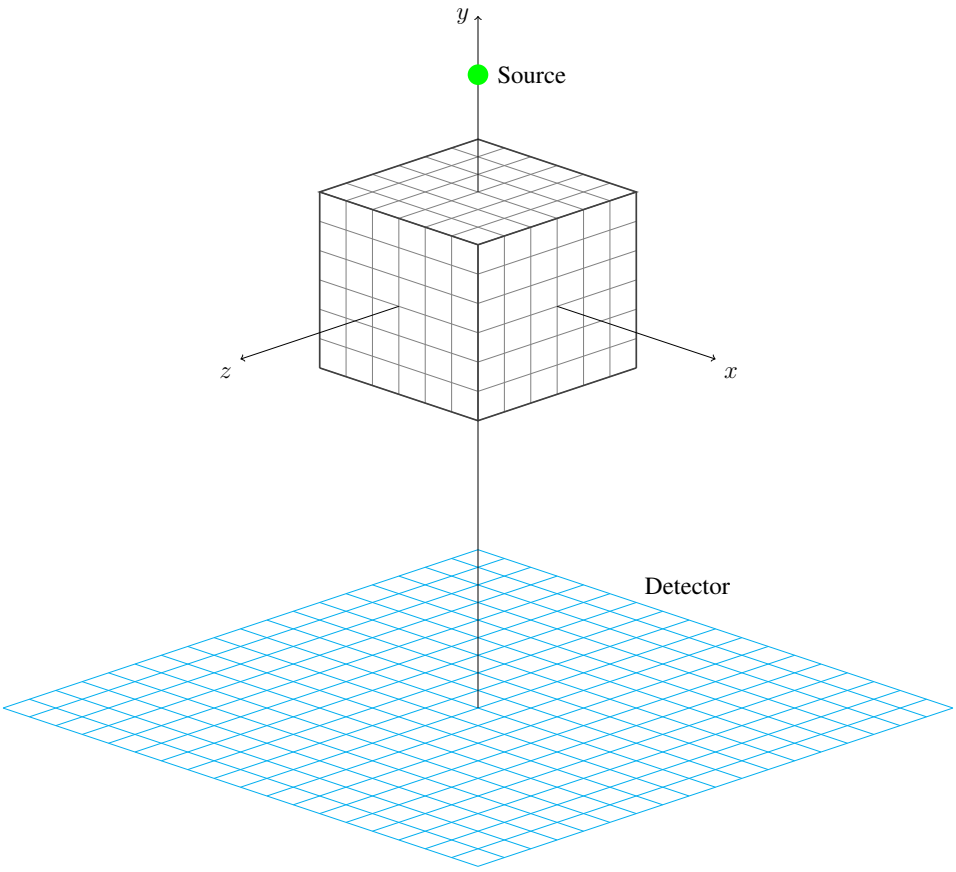


Figura 2.3: Rappresentazione dell'oggetto cubico, già illustrato in Figura 2.1, nel sistema di riferimento cartesiano adottato.

In Figura 2.3 sono rappresentati un oggetto di forma cubica, la sorgente e il rilevatore, tutti posti in uno specifico sistema cartesiano in cui il centro dell'oggetto corrisponde con l'origine degli assi.

La sorgente dei raggi e i pixel del rilevatore, verso cui sono diretti, ruotano lungo un'orbita circolare per scansionare il corpo da diverse angolazioni.

Quando la sorgente ruota in un senso, orario o antiorario, di un certo angolo nell'orbita, il rilevatore ruota coerentemente, quindi dello stesso angolo, nel senso opposto.

2.3.2 Le sorgenti

La sorgente è approssimata a un punto, che indica la posizione a partire dalla quale sono diretti i raggi. Le posizioni considerate sono più di una e, nell'implementazione trattata, sono distribuite

lungo una traiettoria circolare (o una sezione di essa) intorno all'oggetto. Occorre stabilire le coordinate cartesiane dei punti che rappresentano le sorgenti nello spazio.

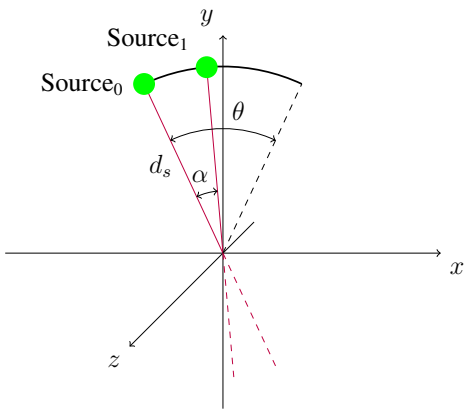


Figura 2.4: Posizionamento angolare delle prime due sorgenti.

Conoscendo l'estensione angolare θ della sezione della traiettoria considerata e la distanza angolare α tra ciascuna posizione, si è scelto l'uso delle coordinate polari, dove una posizione è individuata dalla distanza dall'origine e dalla distanza angolare dall'asse polare. In accordo con la Figura 2.4, viene considerato un sistema di coordinate polari il cui polo coincide con il centro dell'oggetto di studio, mentre si adotta l'asse y come asse polare. La distanza angolare è positiva in senso orario nel piano xy.

La posizione della sorgente può essere individuata nel seguente modo:

$$(d_s, -\theta/2 + \alpha i) \quad \text{per } i = 0, \ldots, N_\theta - 1 \tag{2.4}$$

dove:

- i è un indice che identifica una posizione della sorgente.
- d_s è la distanza sorgente-origine.
- N_θ è il numero di posizioni dato dal rapporto $N_\theta = \lfloor \theta/\alpha \rfloor + 1$.

Le coordinate cartesiane si ottengono a partire dalle coordinate polari nel seguente modo:

$$\begin{cases} X_i = \cos(\frac{\pi}{2} - (-\frac{\theta}{2} + \alpha i))d_s = \sin(-\frac{\theta}{2} + \alpha i)d_s \\ Y_i = \sin(\frac{\pi}{2} - (-\frac{\theta}{2} + \alpha i))d_s = \cos(-\frac{\theta}{2} + \alpha i)d_s \\ Z_i = 0 \end{cases} \qquad \text{per } i = 0, \ldots, N_\theta - 1 \tag{2.5}$$

dove sono state applicate le formule degli archi associati per seno e coseno per passare dal sistema con misurazione in senso orario a partire dal semiasse positivo delle y al sistema tradizionale con misurazione in senso antiorario a partire dal semiasse positivo delle x. Inoltre, Z_i è posta a 0 in quanto, per costruzione, la traiettoria delle sorgenti giace sul piano xy.

2.3.3 L'oggetto

Un oggetto è approssimato da un certo numero di voxel, ovvero parallelepipedi retti con dimensioni uniformi l'uno con l'altro. Le griglie generate da questi voxel, secondo l'algoritmo di Siddon, sono rappresentate da un certo numero di piani paralleli tra loro.

L'equazione cartesiana di un piano nello spazio è un'equazione di primo grado in tre incognite:

$$ax + by + cz + d = 0 \quad \text{con } a, b, c, d \in \Re$$

Come mostrato nella Figura 2.3, i piani coinvolti non solo sono paralleli tra loro, ma sono anche paralleli agli assi cartesiani. Pertanto, le loro equazioni sono nella forma:

$$\begin{aligned} ax + d = 0 \quad & \text{se parallelo al piano } yz \\ by + d = 0 \quad & \text{se parallelo al piano } xz \\ cz + d = 0 \quad & \text{se parallelo al piano } xy \end{aligned}$$

Poiché le distanze tra i piani paralleli sono uniformi, una volta determinata l'equazione di un piano, è possibile determinare l'equazione degli altri traslando più volte. L'equazione di ciascun piano successivo si ottiene a partire da quella del primo, utilizzando le seguenti formule:

$$\begin{aligned} X_{plane}(i) &= X_{plane}(1) + (i-1)d_x \quad \forall i \in 1, \ldots, N_x \\ Y_{plane}(j) &= Y_{plane}(1) + (j-1)d_y \quad \forall j \in 1, \ldots, N_y \\ Z_{plane}(k) &= Z_{plane}(1) + (k-1)d_z \quad \forall k \in 1, \ldots, N_z \end{aligned}$$

dove N_x, N_y e N_z sono il numero di voxel lungo gli assi x, y e z, mentre d_x, d_y e d_z sono a loro volta le dimensioni di un singolo voxel lungo i tre assi cartesiani.

2.3.4 I rilevatori

Nel caso tridimensionale, il rilevatore può essere rappresentato come una matrice di unità di misurazione (pixel), posizionata a una distanza d_r dal centro del corpo in esame.

L'approccio usato per relazionare i raggi con il rilevatore è detto *ray-driven*. Per ciascun unità del rilevatore si considera l'esistenza di un unico raggio passante, diretto dalla sorgente al centro del pixel stesso. Secondo questo approccio, il numero N_p di raggi utilizzati per il calcolo della proiezione corrisponde al numero di pixel del rilevatore.

Il rilevatore è sempre ortogonale all'asse che congiunge la sorgente con il centro del corpo.

Le coordinate cartesiane del centro di un'unità del rilevatore, al variare dell'angolo considerato, sono (X', Y', Z'), derivate dalla posizione (X, Y, Z), ricoperta nel caso in cui il rilevatore sia ortogonale all'asse y.

Considerando il semiasse negativo di y come asse polare e mantenendo il senso orario per gli angoli positivi, le coordinate polari del punto risultano:

$$(d_r, \delta)$$

dove δ è l'angolo che identifica l'unità del rilevatore.

Le coordinate a seguito della traslazione saranno:

$$(d_r, \delta - \lambda)$$

dove $-\lambda$ è l'angolo di traslazione.

Le coordinate di tale posizione traslata sono date da:

$$\begin{cases} X' = \cos(\frac{3\pi}{2} - (\delta - \lambda))d_r = -\sin(\delta - \lambda)d_r \\ Y' = \sin(\frac{3\pi}{2} - (\delta - \lambda))d_r = -\cos(\delta - \lambda)d_r \end{cases} \tag{2.6}$$

dove, in modo analogo all'Equazione 2.5, sono state applicate le formule degli archi associati per seno e coseno per passare dal sistema con misurazione in senso orario a partire dal semiasse negativo delle y al sistema tradizionale con misurazione in senso antiorario a partire dal semiasse positivo delle x. In questo caso Z è costante, $Z' = Z$, in quanto, per costruzione, la traiettoria trasla sul piano xy, senza influire sulla coordinata nell'asse z, di conseguenza è omessa in questa discussione.

A partire dall'Equazione 2.6, applicando le formule di addizione degli archi per seno e coseno si ottiene:

$$\begin{cases} X' = -(\sin(\delta)\cos(-\lambda)d_r + \cos(\delta)\sin(-\lambda)d_r) \\ Y' = -(\cos(\delta)\cos(-\lambda)d_r - \sin(\delta)\sin(-\lambda)d_r) \end{cases}$$

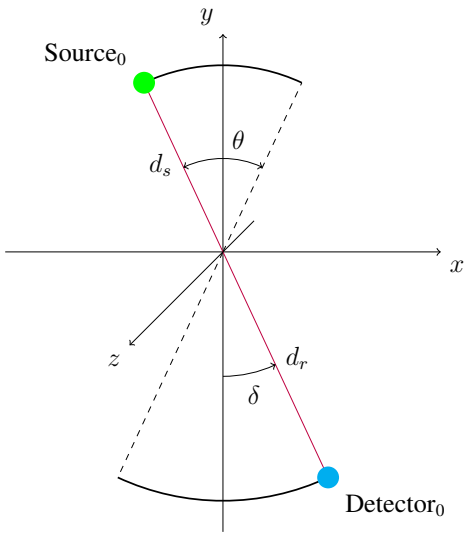


Figura 2.5: Vista angolare di un raggio diretto dalla prima sorgente verso il centro del pixel centrale nel rilevatore. Gli angoli corrispondono esattamente a quelli nell'esempio quando la matrice che costituisce il rilevatore ha un numero di pixel per lato dispari.

Osservando la Figura 2.5, dalle formule trigonometriche per i triangoli rettangoli risulta:

$$d_r = -\frac{X}{\sin(\delta)} = -\frac{Y}{\cos(\delta)}$$

sostituendo d_r e sfruttando le formule degli archi associati per seno e coseno si ottiene:

$$\begin{cases} X' = \cos(-\lambda)X + \sin(-\lambda)Y \\ Y' = \cos(-\lambda)Y - \sin(-\lambda)X \end{cases}$$

Analogamente a quanto avviene con la sorgente, la posizione di una unità del rilevatore varia, a intervalli regolari, di un angolo α su di una sezione di ampiezza regolare pari a θ. Quindi,

ricordando l'Equazione 2.4, si considera $-\lambda = -\theta/2 + \alpha i$. Le nuove posizioni di un'unità del rilevatore possono essere determinate a partire dalla posizione precedente utilizzando le seguenti equazioni:

$$\begin{cases} X_i' = \cos(-\theta/2 + \alpha i)X + \sin(-\theta/2 + \alpha i)Y \\ Y_i' = \cos(-\theta/2 + \alpha i)Y - \sin(-\theta/2 + \alpha i)X \end{cases} \qquad \text{per } i \in 0, \ldots, N_\theta - 1$$

2.3.5 I raggi

Un raggio, che rappresenta un'approssimazione delle radiazioni elettromagnetiche, è trattato come un segmento che collega la sorgente al centro di un pixel del rilevatore, secondo l'approccio ray-driven. Esso può essere descritto mediante l'equazione parametrica di una retta passante per due punti $Source = (X_1, Y_1, Z_1)$ e $Detector = (X_2, Y_2, Z_2)$:

$$\begin{aligned} X(a) &= X_1 + a(X_2 - X_1) \\ Y(a) &= Y_1 + a(Y_2 - Y_1) \\ Z(a) &= Z_1 + a(Z_2 - Z_1) \end{aligned}$$

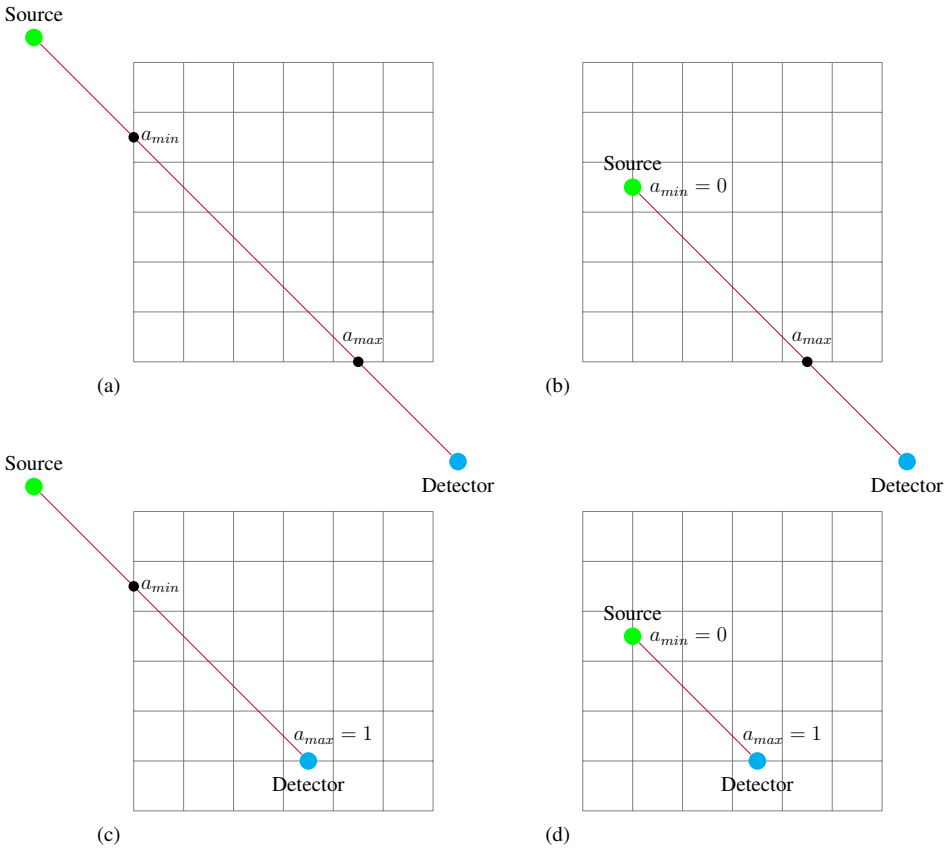


Figura 2.6: Le quantità a_{min} e a_{max} sono quelle che definiscono gli intervalli da considerare per ottenere solo le intersezioni con piani effettivamente all'interno della griglia di pixel (voxel nel caso 3D). Si può notare che a_{min} nelle figure *(a)* e *(c)* sarà maggiore di 0, così come a_{max} nelle figure *(a)* e *(b)* sarà minore di 1. Questo permette di non considerare le intersezioni in eccesso, esterne alla griglia di partenza, risolvendo il problema esposto nella Figura 2.2.

In questo modo:

- Per definire un raggio è sufficiente conoscere il punto di origine ($Source$) e il punto sul rilevatore ($Detector$) in cui avviene la misurazione.
- È possibile rappresentare un punto appartenente alla retta del raggio tramite un unico parametro a, in cui $a = 0$ coincide con il punto $Source$, $a = 1$ al punto $Detector$, e per i valori di a compresi nell'intervallo $[0, 1]$ si ottengono punti interni al segmento, come mostrato nella Figura 2.6.

Il punto di intersezione tra un raggio e un piano si ottiene determinando il valore di a che soddisfa il sistema lineare dato dall'equazione del piano e da una delle equazioni parametriche della retta, come mostrato di seguito:

$$\begin{cases} X = X_1 + a(X_2 - X_1) \\ X = X_{planes}(1) + (i-1)d_x \end{cases} \quad \text{se } i \in 1, \ldots, N_x$$

$$\begin{cases} Y = Y_1 + a(Y_2 - Y_1) \\ Y = Y_{planes}(1) + (i-1)d_y \end{cases} \quad \text{se } i \in 1, \ldots, N_y$$

$$\begin{cases} Z = Z_1 + a(Z_2 - Z_1) \\ Z = Z_{planes}(1) + (i-1)d_z \end{cases} \quad \text{se } i \in 1, \ldots, N_z$$

dove sono stati indicati i sistemi utilizzati nel caso in cui il piano sia parallelo al piano yz (primo sistema), al piano xz (secondo sistema) o al piano xy (terzo sistema).

Capitolo 3

Descrizione del progetto

Il progetto di tesi [Mar25] include diverse implementazioni scritte in linguaggio C, tra cui un programma per la generazione dei file di input, uno per il calcolo delle proiezioni basato sull'algoritmo di Siddon [Sid85] con OpenMP e una versione sviluppata con CUDA. Mentre le prime due derivano da una rielaborazione del lavoro di [Col24], l'implementazione CUDA rappresenta il fulcro di questo progetto.

Di seguito verrà riportata una breve introduzione alle tecniche di parallelizzazione adottate, seguita dalla descrizione delle implementazioni sviluppate.

3.1 Parallelizzazione

L'ottimizzazione di un algoritmo seriale è un processo complesso, che coinvolge molteplici strategie a seconda del contesto applicativo, delle caratteristiche hardware e degli obiettivi di prestazione. In generale, l'ottimizzazione può riguardare la riduzione della complessità computazionale, una migliore gestione della memoria, l'uso di strutture dati efficienti o l'impiego di tecniche avanzate per minimizzare i tempi di esecuzione.

Tuttavia, quando si passa a implementazioni ottimizzate per architetture specifiche come CPU e GPU, emergono strategie di ottimizzazione mirate, che sfruttano le peculiarità di ciascun tipo di hardware.

Nel seguito, ci concentreremo sulle tecniche di parallelizzazione adottate nelle implementazioni su CPU e GPU selezionate, analizzando le metodologie impiegate per massimizzare le prestazioni in ciascun caso.

3.1.1 CPU

La CPU (Central Processing Unit) è il componente principale di un computer responsabile dell'esecuzione delle istruzioni di un programma. Essa rappresenta l'unità di controllo ed elaborazione principale, in quanto gestisce i calcoli, l'elaborazione dei dati e il controllo delle operazioni.

Le CPU moderne, per motivi principalmente legati al risparmio energetico, dispongono di più core e supportano il multi-threading, consentendo l'esecuzione parallela di più attività.

OpenMP

OpenMP (Open Multi-Processing) è un'API (Application Programming Interface) che supporta la programmazione parallela su architetture multicore a memoria condivisa. Fornisce un insieme di direttive per il compilatore, librerie runtime e variabili d'ambiente che permettono di scrivere programmi paralleli. In questo progetto è stato utilizzato il linguaggio C.

OpenMP rientra nella categoria **MIMD** (Multiple Instruction, Multiple Data) secondo la tassonomia di Flynn, vedi [Fly66].

Le caratteristiche principali di OpenMP sono:

- Parallelismo a memoria condivisa: i processi condividono lo stesso spazio di indirizzamento.
- Programmazione basata su direttive: le sezioni di codice parallelo sono specificate tramite direttive `#pragma`.
- Scalabilità: adatta il numero di processi in base al numero di core disponibili.
- Facilità di utilizzo: non richiede riscrittura completa del codice, ma solo l'aggiunta di annotazioni.
- Sincronizzazione e gestione del carico: include meccanismi per bilanciare il carico tra i processi ed evitare race conditions.

3.1.2 GPU

La GPU (Graphics Processing Unit) è un processore specializzato progettato per eseguire calcoli in parallelo su grandi quantità di dati. Come si può intuire dal nome è stata originariamente sviluppata per il rendering grafico. Conseguentemente alla domanda del mercato, la GPU si è oggi ampiamente evoluta in un processore generale, che quindi può svolgere diverse tipologie di carichi di lavoro al di là del solo rendering grafico. Utilizzata in settori come intelligenza artificiale, elaborazione di immagini, deep learning e simulazioni scientifiche, grazie alla sua capacità di eseguire un elevato numero di operazioni in parallelo. Le GPU moderne, sebbene operino in generale a frequenze più basse delle CPU, sono molto più veloci di esse nell'eseguire i compiti in cui sono specializzate.

CUDA

CUDA (Compute Unified Device Architecture) è una piattaforma di calcolo parallelo e un modello di programmazione sviluppato da NVIDIA che consente di sfruttare le GPU NVIDIA per l'elaborazione generale, GPGPU (General-Purpose computing on Graphics Processing Units). In questo progetto è stato utilizzato il linguaggio 'CUDA-C' (C con estensioni NVIDIA).

A livello CUDA, per lavorare in modo efficiente, si astrae dal funzionamento hardware, considerando un parallelismo di tipo **SIMT** (Single Instruction, Multiple Thread), vedi [Lin+08], che è un'estensione del concetto SIMD (Single Instruction, Multiple Data) della tassonomia di Flynn, vedi [Fly66].

In realtà l'hardware delle GPU prevede esecuzione:

- SIMT, a livello di Warp (unità di schedulazione):
  - Le GPU eseguono gruppi di Thread chiamati Warp (tipicamente 32 Thread) che eseguono la stessa istruzione simultaneamente, ma su dati diversi. Questo è il comportamento tipico del paradigma SIMD.

  - Tuttavia, a differenza di un vero SIMD (dove tutti i dati sono rigidamente legati alla stessa istruzione), i Thread in un Warp possono, anche se inefficiente, divergere (ad esempio, con istruzioni condizionali), motivo per cui il modello è chiamato SIMT.
- MIMD, a livello di multiprocessore:
  - Ogni Streaming Multiprocessor (SM) della GPU può eseguire più Warp indipendenti, ognuno con istruzioni diverse.
  - Questo significa che, considerando l'intera GPU, essa opera in modo MIMD, poiché più SM possono eseguire istruzioni diverse su set di dati diversi.

Il meccanismo di computazione basato sulle unità Warp implica che le computazioni SIMT offrano le migliori prestazioni. Questo perché le divergenze, ovvero le istruzioni condizionali nel codice, causano la serializzazione dell'esecuzione dei Thread all'interno di un singolo Warp.

Le caratteristiche principali di CUDA sono:

- Parallelismo massivo: permette di eseguire migliaia di Thread in parallelo sulla GPU. Adatto a risolvere problemi del tipo 'Embarrassingly Parallel', quindi in cui la computazione può essere decomposta in task indipendenti, con poche divergenze e che richiedono poca comunicazione l'uno con l'altro.
- Memoria gerarchica: struttura di memoria ottimizzata per alte prestazioni (memoria globale, condivisa, registri).
- Elevata efficienza: permette di accelerare calcoli scientifici, AI, machine learning, rendering, simulazioni fisiche, e molto altro.

3.2 Generazione dell'input

L'implementazione proposta genera le configurazioni del vettore f già discusse nella sottosezione 2.2.2. Queste configurazioni vengono scritte su file. Per generare input di dimensioni opportune alle necessità, quando si esegue il programma è possibile specificare il numero di pixel per lato della matrice rilevatore. La strategia adottata per la scrittura è descritta di seguito.

```
unsigned Nx = N_VOXEL_X, Nz = N_VOXEL_Z, pNy = k;
// Dove k <= N_VOXEL_Y
for (unsigned y = 0; y < pNy; y++) {
  for (unsigned z = 0; z < Nz; z++) {
    for (unsigned x = 0; x < Nx; x++) {
      f[x + z*Nz + y*Nx*Nz] = 1.0;
    }
  }
}
```

Listato 3.1: Codice C per l'inizializzazione di f per un cubo.

Il Listato 3.1 mostra l'inizializzazione semplificata di f per un oggetto cubico denso.

Per comprendere l'indicizzazione è utile considerare che l'elemento nell'array 1D $f[x + zN_z + yN_xN_z]$, è equivalente a quello nella matrice 3D $\rho[x][z][y]$, in cui x, z e y sono fissati, vedi l'Equazione 2.3.

La caratteristica fondamentale di questa soluzione è che l'inizializzazione mostrata, di fatto, "srotola" la matrice ρ in un array f, in cui la coordinata y è indipendente rispetto alle altre. In questo senso le x e poi le z sono scritte più velocemente rispetto alla y, quindi dipendono

da essa. In altre parole, la y, nel codice del Listato 3.1, è l'unico contatore non annidato, di conseguenza quello che non dipende dagli altri.

Si osserva però che la specifica mappatura degli elementi scelta non è l'unica possibile, si sarebbero per esempio potute scambiare le varie coordinate tra di loro, invertendone i ruoli, e perciò le relative dipendenze.

Il Listato 3.1 rappresenta in realtà un'iterazione dell'inizializzazione di f. Infatti `pNy` non è assegnato a `N_VOXELS_Y`, cioè numero totale di voxel nell'asse y, ma a `k`. Quindi l'array f, limitato nelle dimensioni da `pNy`, rappresenta una partizione verticale dell'oggetto cubico di interesse e non l'oggetto completo così come considerato nel modello matematico, vedi la Figura 2.3. Tale partizione avrà un numero di voxel sull'asse y pari a `k`, con `k <= N_VOXELS_Y`. Nella soluzione proposta oltretutto la singola iterazione, simile a quella presentata, viene parallelizzata sfruttando OpenMP.

Seguendo questo schema, anche la lettura dell'input può essere fatta per partizioni sul numero totale di voxel nell'asse y. È essenziale riconoscere che tali partizioni possono essere lette con dimensioni arbitrarie, indipendentemente l'una dall'altra e, soprattutto, senza la necessità di conoscere in lettura la dimensione utilizzata per la scrittura.

Questa strategia di scrittura è essenziale per gestire input di grandi dimensioni. In particolare, nella computazione su CPU, è fondamentale considerare la quantità di memoria RAM disponibile sulla scheda madre, un fattore rilevante nella parallelizzazione con OpenMP. Nella computazione su GPU, invece, bisogna tenere conto sia della memoria RAM della scheda madre sia di quella disponibile sulla GPU, aspetto cruciale quando si parallelizza con CUDA. Ad esempio, supponiamo di dover analizzare una partizione di lunghezza `yVoxels`, con la condizione `yVoxels <= N_VOXELS_Y`. In questo caso, per quella partizione verranno letti solo `N_VOXEL_X * yVoxels * N_VOXEL_Z` valori, il che, per input di grandi dimensioni, può comportare un notevole risparmio di RAM.

3.3 Calcolo delle proiezioni

L'algoritmo per il calcolo delle proiezioni implementato si basa su quanto discusso nel Capitolo 2, in particolare tieni a mente l'Equazione 2.2 da cui siamo partiti. I passi dell'algoritmo sono i seguenti:

1. il vettore f dei coefficienti di attenuazione viene letto da file come input del programma;
2. si calcola il prossimo punto sorgente da analizzare, nel caso in cui siano finiti l'algoritmo termina;
3. si calcola il prossimo pixel del rilevatore (in realtà il punto centrale), nel caso in cui siano finiti si ritorna al punto 2;
4. per procedere si calcolano i valori parametrici di intersezione del raggio individuato dalla sorgente e dal pixel del rilevatore con il primo e l'ultimo piano di ogni dimensione, per verificare che il raggio attraversi l'oggetto, in caso contrario si ritorna al punto 3;
5. a questo punto si calcolano i valori parametrici dei piani estremi (il primo e l'ultimo) che devono essere considerati per calcolare il sottoinsieme delle intersezioni con il raggio che risultano dentro la griglia dei voxel, vedi la Figura 2.2;

6. poi si calcolano i valori parametrici delle intersezioni tra il raggio e i piani considerati;
7. si calcola la lunghezza del prossimo segmento sotteso ai voxel (valore di M), nel caso in cui siano finiti si ritorna al punto 3;
8. ora si calcola l'indice del voxel che sottende il segmento;
9. al pixel del rilevatore (pixel corrente di g) viene infine sommata l'attenuazione del voxel corrispondente, moltiplicata per la lunghezza del segmento considerato, si ritorna quindi al punto 7.

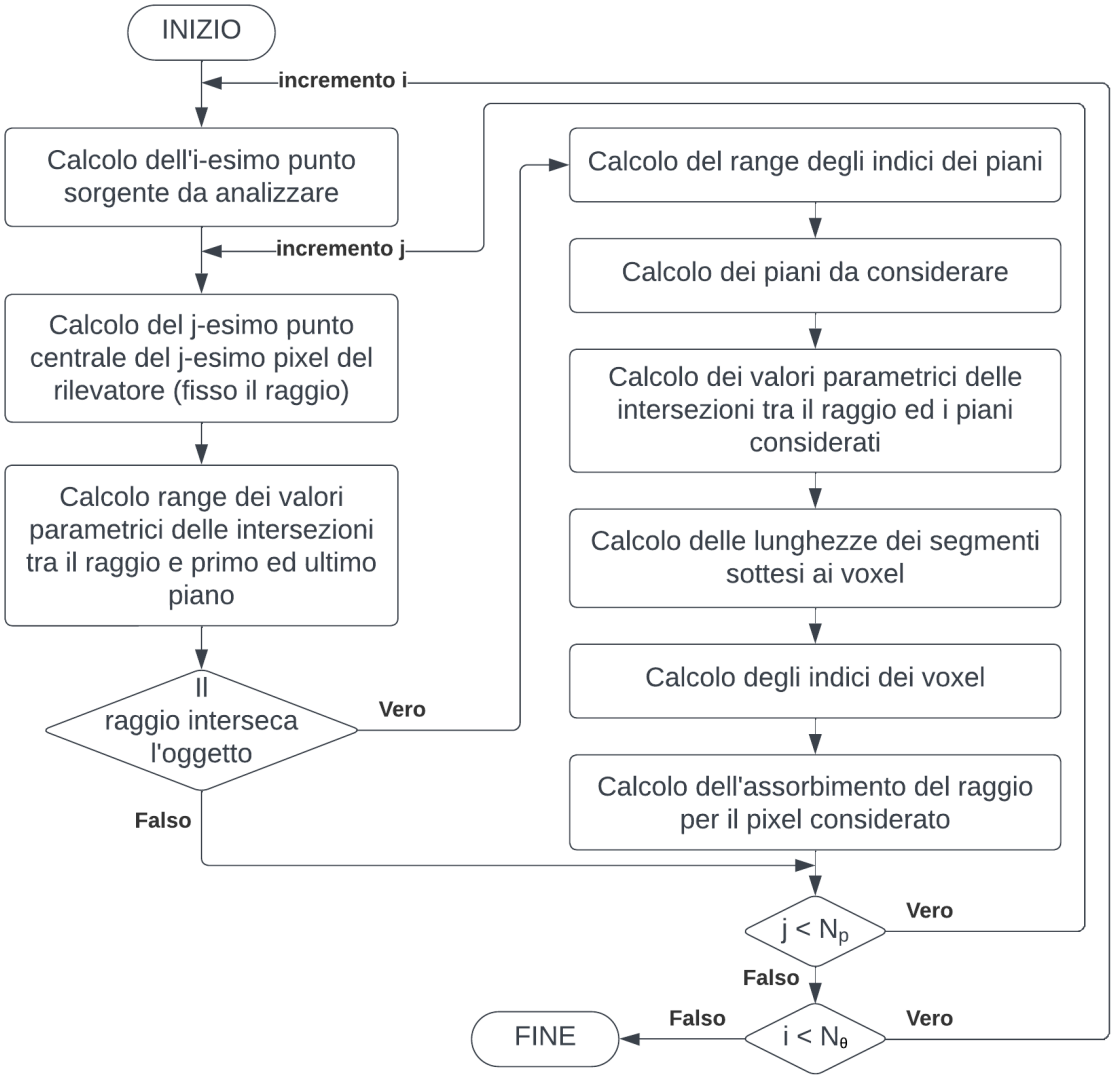


Figura 3.1: Diagramma di flusso del calcolo delle proiezioni.

In Figura 3.1 è presentato il diagramma di flusso relativo ai punti 1-9 dell'algoritmo di calcolo delle proiezioni.

3.3.1 Implementazione generale

1. Lettura dell'input

```
double *const f = (double *) malloc(sizeof(double) * gl_nVoxel[X] *
   yVoxels * gl_nVoxel[Z]);
...
for (unsigned short slice = 0; slice < gl_nVoxel[Y]; slice +=
   yVoxels) {
  unsigned short nOfSlices;
  if (gl_nVoxel[Y] - slice < yVoxels) {
    nOfSlices = gl_nVoxel[Y] - slice;
  } else {
    nOfSlices = yVoxels;
  }
  fread(f, sizeof(double), (size_t) gl_nVoxel[X] * nOfSlices *
     gl_nVoxel[Z], inputFilePointer);
  ...
  computeProjections(...);
}
```

Listato 3.2: Codice C per la lettura di f.

Il Listato 3.2 mostra la gestione dell'array f, che può costituire l'intero insieme di voxel o un suo sottoinsieme. La lettura dei dati viene quindi fatta considerando `nOfSlices` voxel alla volta sull'asse y, dove `yVoxels` è il numero massimo di voxel da considerare in ogni iterazione sull'asse y. Nella funzione `computeProjections` sono sviluppati tutti i calcoli necessari per ottenere le proiezione. Tutti i passi descritti successivamente sono svolti all'interno di tale funzione.

2. Individuazione della prossima sorgente

```
for (unsigned short positionIndex = 0; positionIndex < nTheta;
   positionIndex++) {
  const Point source = getSource(gl_sinTable, gl_cosTable,
     positionIndex);
  ... // Calcolo di ogni pixel del rilevatore
}
```

Listato 3.3: Codice C per l'individuazione di un punto sorgente.

Il Listato 3.3 mostra come avviene il calcolo di ogni punto sorgente in base all'angolo di rotazione. Il calcolo di ogni punto sorgente è ripetuto per ogni partizione considerata. I dettagli della funzione `getSource` non sono mostrati, il calcolo è lo stesso già presentato nella sottosezione 2.3.2.

3. Individuazione del prossimo pixel del rilevatore

```
... // Per ogni sorgente
for (unsigned r = 0; r < nSidePixels; r++) {
  for (unsigned c = 0; c < nSidePixels; c++) {
    ...
```

```
    const Point pixel = getPixel(gl_sinTable, gl_cosTable, r, c,
       positionIndex);
    ... // Calcolo dell'assorbimento per il pixel considerato
  }
}
```

Listato 3.4: Codice C per l'individuazione di un pixel del rilevatore.

Il Listato 3.4 mostra come avviene il calcolo di ogni punto centrale di un pixel del rilevatore in base all'angolo di rotazione. I dettagli della funzione `getPixel` non sono mostrati, il calcolo è lo stesso già presentato nella sottosezione 2.3.4.

4-9. Computazioni successive

Le computazioni seguenti sono eseguite facendo riferimento all'algoritmo di Siddon [Sid85].

Per l'implementazione di questi passaggi dell'algoritmo, viene analizzata una specifica scelta effettuata nel rispetto dell'algoritmo di Siddon. In particolare, l'attenzione è focalizzata sull'implementazione in C dei passaggi 6-9 del processo di calcolo delle proiezioni.

```
... // Si procede con un raggio che interseca l'oggetto
// Calcolo dei valori parametrici del primo e ultimo piano validi
Ranges indices[3];
indices[X] = getRangeOfIndex(source.x, pixel.x, isParallel, aMin,
   aMax, X);
indices[Y] = getRangeOfIndex(source.y, pixel.y, isParallel, aMin,
   aMax, Y);
indices[Z] = getRangeOfIndex(source.z, pixel.z, isParallel, aMin,
   aMax, Z);

// Si calcola la lunghezza degli array contenenti i valori
// parametrici delle intersezioni raggio-piani
const unsigned short lenX = max(0, indices[X].maxIndx - indices[X].
   minIndx);
const unsigned short lenY = max(0, indices[Y].maxIndx - indices[Y].
   minIndx);
const unsigned short lenZ = max(0, indices[Z].maxIndx - indices[Z].
   minIndx);
...
// Si calcolano i valori parametrici delle intersezioni raggio-piani
double aX[gl_nPlanes[X]];
double aY[gl_nPlanes[Y]];
double aZ[gl_nPlanes[Z]];
getAllIntersections(source.x, pixel.x, indices[X], aX, X);
getAllIntersections(source.y, pixel.y, indices[Y], aY, Y);
getAllIntersections(source.z, pixel.z, indices[Z], aZ, Z);

// Si fa un merge degli array che server per calcolare M
double aMerged[gl_nPlanes[X] + gl_nPlanes[Y] + gl_nPlanes[Z]];
const unsigned short lenA = merge3(aX, aY, aZ, lenX, lenY, lenZ,
   aMerged);

```

```
26 // Si calcola l'attenuazione totale del raggio considerato in g
27 const unsigned pixelIndex = positionIndex * nSidePixels *
      nSidePixels + r * nSidePixels + c;
28 g[pixelIndex] += computeAbsorption(slice, source, pixel, aMerged,
      lenA, f);
29 ...
```

Listato 3.5: Codice C per il calcolo dell'attenuazione.

Il Listato 3.5 mostra gli ultimi passi del calcolo dell'attenuazione nell'algoritmo descritto in [Col24], seguendo fedelmente l'implementazione di Siddon.

I dettagli relativi alle funzioni per il calcolo delle intersezioni e dell'assorbimento non sono riportati. Tuttavia, come verrà discusso in seguito, l'allocazione dei quattro array `aX`, `aY`, `aZ` e `aMerged` comporta un utilizzo di memoria che può diventare significativo quando un elevato numero di processi paralleli li alloca simultaneamente.

Anche senza considerare gli array di appoggio presenti nelle funzioni omesse, un singolo processo richiede comunque `2 * gl_nPlanes[X] * gl_nPlanes[Y] * gl_nPlanes[Z] * sizeof(double)` Byte di memoria. Se vengono eseguiti `p` processi, l'utilizzo complessivo diventa `p * 2 * gl_nPlanes[X] * gl_nPlanes[Y] * gl_nPlanes[Z] * sizeof(double)` Byte.

Ciò evidenzia come, all'aumentare di `p`, l'utilizzo di memoria cresca rapidamente, rappresentando un potenziale fattore limitante nelle esecuzioni altamente parallele.

3.3.2 Implementazione OpenMP

L'implementazione per CPU proposta sfrutta il parallelismo OpenMP.

Il calcolo della soluzione, per motivi legati all'utilizzo di memoria RAM, viene fatto in partizioni, considerando di default un numero massimo di 100 voxel sull'asse y.

Nella rielaborazione dell'implementazione di partenza è stata introdotta la possibilità di specificare il numero massimo di voxel quando si lancia il programma da linea di comando, ignorando quindi il valore di default.

Parallelizzazione

In questo contesto, nonostante alcune modifiche, il modello di parallelizzazione è stato mantenuto invariato rispetto a quello del codice originale.

```
1 ... // Per ogni sorgente
2 #pragma omp parallel for collapse(2) schedule(dynamic) ...
3 for (unsigned r = 0; r < nSidePixels; r++) {
4   for (unsigned c = 0; c < nSidePixels; c++) {
5     ... // Calcolo dell'assorbimento per il pixel considerato
6   }
7 }
```

Listato 3.6: Codice C per il calcolo dell'assorbimento di un determinato pixel del rilevatore.

Il Listato 3.6 mostra le direttive OpenMP utilizzate per parallelizzare il codice presentato nel Listato 3.4. Ogni processo OpenMP determina la posizione di un pixel del rilevatore e procede al calcolo dell'attenuazione che si verifica quando il raggio attraversa la partizione dell'oggetto in esame. La parallelizzazione viene eseguita considerando un carico di lavoro fortemente sbilanciato tra i diversi processi. Questo è dovuto al fatto che il numero di computazioni richieste per determinare l'attenuazione di ciascun raggio, relativo a una sorgente, varia in base al numero di voxel attraversati, portando così a una distribuzione del carico di lavoro potenzialmente molto disomogenea.

È stata mantenuta la stessa versione di partenza che adottava un partizionamento a grana fine seguendo il paradigma master-worker, in cui:

- Partizionamento a grana fine: si riferisce a una suddivisione del lavoro in compiti molto piccoli, assegnati ai vari processi per massimizzare il parallelismo e bilanciare il carico. In questo problema l'unità di parallelizzazione è il singolo valore fissato di `r` e `c`.
- Paradigma master-worker: è un modello di programmazione parallela in cui un processo master si occupa di assegnare i compiti ai processi worker. I worker eseguono i compiti assegnati e, una volta completato un task, passano immediatamente al successivo disponibile, seguendo l'ordine di completamento. Questo approccio consente una distribuzione dinamica ed efficiente del carico di lavoro, ottimizzando l'utilizzo delle risorse computazionali.

Nella parallelizzazione con OpenMP, l'allocazione degli array discussa nel Listato 3.5 non rappresenta un problema, poiché il numero di processi in esecuzione parallela è relativamente limitato, evitando così un utilizzo eccessivo di memoria. Infatti, il numero di core in una CPU è generalmente nell'ordine delle decine, impedendo un'eccessiva allocazione di risorse rispetto a scenari con un numero molto maggiore di processi.

3.3.3 Implementazione CUDA

L'implementazione per GPU NVIDIA proposta sfrutta il parallelismo CUDA.

Analogamente alla versione OpenMP, la soluzione viene calcolata per partizioni, al fine di ottimizzare l'utilizzo della memoria RAM. Tuttavia, in questo caso, il numero massimo di voxel lungo l'asse y viene determinato dinamicamente mediante un algoritmo euristico. Questo algoritmo stima rapidamente la memoria occupata sulla GPU, fornendo un valore approssimativo che potrebbe non essere necessariamente quello ottimale.

```
size_t freeMem, totalMem;
cudaMemGetInfo(&freeMem, &totalMem);
freeMem -= (totalMem * 2 / 8);
unsigned tmp = gl_nVoxel[Y];
size_t size = sizeof(double) * gl_nVoxel[X] * tmp * gl_nVoxel[Z];
while (size > freeMem) {
  tmp = tmp * 5 / 8;
  if (tmp == 0) {
    fprintf(stderr,
      "The voxels Y size is too small respect to the other sizes:\n"
      "- N voxels X: %u.\n"
      "- N voxels Y: %u.\n"
      "- N voxels Z: %u.\n"
      "Total size reduced to the minimum possible is %lu Bytes!\n"
```

```
      "This is too much for this GPU with %lu Bytes of usable "
      "global memory (of %lu Bytes total)!\n",
      gl_nVoxel[X], gl_nVoxel[Y], gl_nVoxel[Z],
      size, freeMem, totalMem);
    termEnvironment();
    exit(EXIT_FAILURE);
  }
  size = sizeof(double) * gl_nVoxel[X] * tmp * gl_nVoxel[Z];
}
yVoxels = tmp;
```

Listato 3.7: Codice CUDA-C per il calcolo del numero massimo di voxel sull'asse y.

Il Listato 3.7 mostra il funzionamento dell'algoritmo per il calcolo del numero massimo di voxel sull'asse y per ogni partizione dell'input da considerare. Dopo la sua individuazione il valore calcolato è assegnato a `yVoxels`.

Anche in questo caso è stata introdotta la possibilità di specificare il numero massimo di voxel quando si lancia il programma da linea di comando, ignorando quindi il valore generato dall'algoritmo euristico che agisce normalmente.

Parallelizzazione

Nelle discussioni seguenti, il termine host si riferirà all'esecuzione di codice seriale sulla CPU, mentre device indicherà l'esecuzione di codice parallelo sulla GPU.

```
double *const f = (double *) malloc(sizeof(double) * gl_nVoxel[X] *
   yVoxels * gl_nVoxel[Z]);
...
for (unsigned short slice = 0; slice < gl_nVoxel[Y]; slice +=
   yVoxels) {
  unsigned short nOfSlices;
  if (gl_nVoxel[Y] - slice < yVoxels) {
    nOfSlices = gl_nVoxel[Y] - slice;
  } else {
    nOfSlices = yVoxels;
  }
  fread(f, sizeof(double), (size_t) gl_nVoxel[X] * nOfSlices *
     gl_nVoxel[Z], inputFilePointer);
  ...
  asyncComputeProjections(...);
}
```

Listato 3.8: Codice CUDA-C per la lettura di f.

Il Listato 3.8, equivalente del Listato 3.2, mostra come viene svolta la computazione considerando partizioni dell'input.

In questo caso, a differenza di OpenMP, la funzione `computeProjections` non viene invocata direttamente, bensì viene chiamata un'altra funzione, `asyncComputeProjections`. La scelta è stata fatta per mantenere la stessa struttura di calcolo all'interno dello stesso identificatore di funzione, `computeProjections`, garantendo coerenza con il tipo di computazione eseguita.

Si noti che il prefisso `async` nella funzione `asyncComputeProjections`, chiamata dall'host, indica che l'esecuzione avviene in modo asincrono. Così si permette alla CPU di leggere simultaneamente la successiva partizione dell'input, introducendo un ulteriore livello di parallelismo tra le operazioni.

Tale meccanismo è reso possibile dalla separazione tra la memoria RAM della CPU e quella della GPU. Di conseguenza, una volta copiati i dati di input nella RAM della GPU, la CPU può immediatamente sovrascrivere il buffer contenente la partizione appena trasferita, subito dopo aver avviato il kernel in modalità asincrona. Inoltre, nel caso in cui non ci siano più dati da essere letti allora il buffer di lettura può essere deallocato, in tal caso l'host, inevitabilmente, attenderà il completamento del kernel per poter leggere i risultati ottenuti.

Questo approccio ha un impatto significativo sulla misurazione dei tempi di esecuzione: se il numero di iterazioni è elevato e la durata dell'esecuzione del kernel non è inferiore a quella della lettura dell'input da file, è come se dal tempo totale di esecuzione dei vari kernel (escluso l'ultimo) venisse sottratto il tempo di lettura da file. In pratica, ciò si traduce in una riduzione complessiva dei tempi di esecuzione.

```
1  void asyncComputeProjections(...)
2  {
3    // Si copia f dalla CPU nell'array corrispondente d_f sulla GPU
4    cudaMemcpy(d_f, f, sizeF, cudaMemcpyHostToDevice);
5    // Le dimensioni di griglia e blocco sono sempre le stesse
6    static dim3 block(BLKDIM_STEP, BLKDIM_STEP);
7    static dim3 grid((nSidePixels + BLKDIM_STEP - 1) / BLKDIM_STEP, (
       nSidePixels + BLKDIM_STEP - 1) / BLKDIM_STEP);
8    // Il kernel viene lanciato in modo asincrono
9    computeProjections<<<grid, block>>>(slice, nTheta, nSidePixels,
       d_f, d_g, isFirst);
10 }
```

Listato 3.9: Codice CUDA-C per il lancio del kernel.

Il Listato 3.9 mostra l'inizializzazione del kernel CUDA. Il kernel `computeProjections`, che si occupa del calcolo delle proiezioni, non è più eseguito nell'host, ma nel device. Quindi l'esecuzione dei passi 2-9 dell'algoritmo considerato, cioè del kernel, avviene in parallelo sul device.

```
1  __global__ void computeProjections(...)
2  {
3    ...
4    // r e c mappano ogni Thread su un singolo pixel del rilevatore
5    const unsigned r = threadIdx.y + blockIdx.y * blockDim.y;
6    const unsigned c = threadIdx.x + blockIdx.x * blockDim.x;
7    ...
8
9    // Gli eventuali Thread in eccesso non eseguono calcoli
10   if (r < nSidePixels && c < nSidePixels) {
11     ...
12     // Calcolo dell'attenuazione di una coppia sorgente-pixel
13     for (unsigned short positionIndex = 0; positionIndex < nTheta;
         positionIndex++) {
```

```
14        const Point source = getSource(d_gl_sinTable, d_gl_cosTable,
              positionIndex);
15        const Point pixel = getPixel(d_gl_sinTable, d_gl_cosTable, r,
              c, positionIndex);
16        ... // Calcolo dell'assorbimento per il pixel considerato
17      }
18    }
19  }
```

Listato 3.10: Codice CUDA-C del kernel.

Nel Listato 3.10 viene mostrato come ogni CUDA Thread determini la posizione di un pixel del rilevatore per procedere al calcolo dell'attenuazione che si verifica quando il raggio attraversa la partizione dell'oggetto in esame.

Nel ciclo illustrato, ogni CUDA Thread analizza ciascuna sorgente in riferimento a uno specifico pixel, che è lo stesso per ogni iterazione sulle posizioni sorgente-rilevatore considerate. Tuttavia, la posizione del pixel viene ricalcolata a ogni passo, poiché subisce una traslazione di un angolo α.

```
1  __global__ void computeProjections(...)
2  {
3    __shared__ double l_gMins[BLKDIM];
4    __shared__ double l_gMaxs[BLKDIM];
5    const unsigned l_index = threadIdx.x + threadIdx.y * blockDim.x;
6    ...
7    l_gMins[l_index] = INFINITY;
8    l_gMaxs[l_index] = -INFINITY;
9
10   // Gli eventuali Thread in eccesso non eseguono calcoli
11   if (r < nSidePixels && c < nSidePixels) {
12     ...
13     for (unsigned short positionIndex = 0; positionIndex < nTheta;
           positionIndex++) {
14       ...
15       // Calcolo dell'attenuazione di una coppia sorgente-pixel
16       g[pixelIndex] += computeAbsorption(...);
17       l_gMins[l_index] = fmin(l_gMins[l_index], g[pixelIndex]);
18       l_gMaxs[l_index] = fmax(l_gMaxs[l_index], g[pixelIndex]);
19     }
20   }
21
22   unsigned b_size = blockDim.x / 2;
23   __syncthreads();
24   while (b_size > 0) {
25     if (l_index < b_size) {
26       if (l_gMins[l_index] > l_gMins[l_index + b_size]) {
27         l_gMins[l_index] = l_gMins[l_index + b_size];
28       }
29       if (l_gMaxs[l_index] < l_gMaxs[l_index + b_size]) {
30         l_gMaxs[l_index] = l_gMaxs[l_index + b_size];
31       }
```

```
32      }
33      b_size = b_size / 2;
34      __syncthreads();
35    }
36
37    if (l_index == 0) {
38      atomicMinDouble(&d_gMin, l_gMins[l_index]);
39      atomicMaxDouble(&d_gMax, l_gMaxs[l_index]);
40    }
41  }
```

Listato 3.11: Codice CUDA-C per effettuare la riduzione nel kernel.

Il Listato 3.11 mostra come avvengono le riduzioni, cioè il calcolo del minimo e massimo valore di attenuazione. In questa implementazione si fa uso della shared memory, cioè memoria condivisa tra tutti i thread di un blocco, per effettuare la riduzione in $O(\log_2 p)$ passi paralleli.

Il principale vantaggio di questa soluzione è la creazione di un unico kernel per ogni partizione dell'input, garantendo così il minor numero possibile di esecuzioni. Ciò consente di ridurre al minimo l'overhead associato alla creazione dell'ambiente di esecuzione del kernel. Inoltre, tutti i dati necessari per il calcolo ad eccezione dell'input f, che deve essere aggiornato per ogni partizione, vengono trasferiti tra CPU e GPU una sola volta, ottimizzando la gestione della memoria.

Tuttavia, poiché il calcolo dell'attenuazione di ciascun raggio è stato parallelizzato, come avviene con OpenMP, il carico di lavoro potrebbe risultare sbilanciato in alcuni scenari. Questo squilibrio potrebbe penalizzare le prestazioni di CUDA, causando una serializzazione parziale dell'esecuzione a causa del modello SIMT su cui si basa l'architettura delle GPU.

Un'ulteriore criticità nella parallelizzazione con CUDA riguarda l'allocazione degli array discussa nel Listato 3.5. A differenza della CPU, la GPU esegue migliaia di unità parallele simultaneamente, il che può portare a un utilizzo di memoria significativo. Inoltre, per ragioni di efficienza, la dimensione degli array deve essere definita a priori. Per questo motivo, è stato scelto un valore massimo di 1000 piani per asse, imponendo così un limite alla dimensione totale dell'input.

3.4 Generazione output

Il risultato dell'algoritmo di proiezione è l'array g di valori in virgola mobile, si faccia riferimento all'Equazione 2.2. Tali valori vengono mappati su scala di grigi, facilitando la conversione in formato `.pgm`, per poter poi essere visualizzati come immagini, convertendoli in formati visualizzabili graficamente come `.png`.

```
double *const g = (double *) calloc(nSidePixels * nSidePixels *
    nTheta, sizeof(double));
... // Calcolo di g con OpenMP o CUDA
FILE *const outputFilePointer = fopen(outputFileName, "w");
if (!outputFilePointer) {
  fprintf(stderr, "Unable to open file '%s'!\n", outputFileName);
  free(g);
  return EXIT_FAILURE;
```

```
8  }
9  // Stampa un valore nell'intervallo [0-255]
10 fprintf(outputFilePointer, "P2\n%d %d\n255", nSidePixels,
       nSidePixels * nTheta);
11 for (unsigned short positionIndex = 0; positionIndex < nTheta;
       positionIndex++) {
12   double angle = -(double) gl_angularTrajectory / 2 + (double)
         positionIndex * gl_positionsAngularDistance;
13   fprintf(outputFilePointer, "\n#%lf", angle);
14   for (unsigned i = 0; i < nSidePixels; i++) {
15     fprintf(outputFilePointer, "\n");
16     for (unsigned j = 0; j < nSidePixels; j++) {
17       const unsigned pixelIndex = positionIndex * nSidePixels *
             nSidePixels + i * nSidePixels + j;
18       int color = (g[pixelIndex] - gMinValue) * 255 / (gMaxValue -
             gMinValue);
19       fprintf(outputFilePointer, "%d ", color);
20     }
21   }
22 }
23 fclose(outputFilePointer);
```

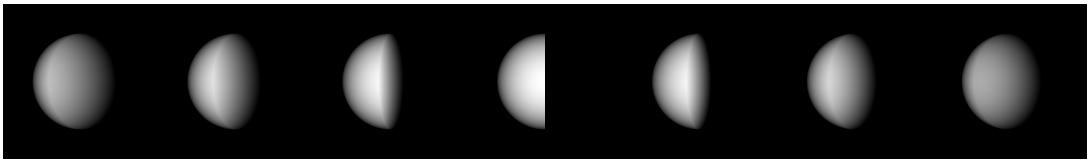
Listato 3.12: Codice C per la mappatura dell'output in immagini.

Il Listato 3.12 mostra come avviene la mappatura dei valori di g in un intervallo di valori 0-255 che individuano i colori nero-bianco nella scala di grigi.

In un'immagine generata a partire dalla matrice di proiezione, i colori rappresentano visivamente i diversi livelli di attenuazione. Un raggio che attraversa una regione più densa del corpo subirà una maggiore attenuazione, risultando in una tonalità più chiara (verso il bianco), mentre un raggio che attraversa una regione meno densa subirà una minore attenuazione, apparendo in una tonalità più scura (verso il nero).

(a) Cubo

(b) Cubo con cavità sferica



(c) Semisfera

Figura 3.2: Immagini delle proiezioni.

La Figura 3.2 mostra le immagini in formato `.png` delle proiezioni ottenute eseguendo la versione CUDA del programma. Si ricorda che le immagini ottenute non sono fotografie tradizionali dell'oggetto scansionato, piuttosto sono una rappresentazione della distribuzione di densità dell'oggetto scansionato. In questo caso l'input processato è stato generato specificando 2000 pixel per lato del detector, questo comporta che ciascuna proiezione qui mostrata è costituita da 2000×2000 pixel in scala di grigi. In particolare le immagini riportate, lette da sinistra a destra, corrispondono alle proiezioni del rispettivo oggetto ottenute considerando angoli di -45°, -30°, -15°, 0°, 15°, 30° e 45° in cui come asse polare si intende l'asse y e il senso orario di rotazione delle sorgenti. Le proiezioni sono state calcolate considerando come angoli di posizionamento della sorgente rispetto all traiettoria circolare $\theta = 90°$ e $\alpha = 15°$, vedi la Figura 2.4.

È importante inoltre considerare che, a causa delle differenze nelle architetture hardware di CPU e GPU, unitamente al fatto che l'ordine dei calcoli in virgola mobile eseguiti in OpenMP e CUDA può variare, i valori ottenuti dalle due versioni del programma non risultano equivalenti. Per verificare la correttezza dei risultati, si possono adottare due approcci:

- Confrontare direttamente le versioni binarie dei risultati, salvando la matrice g su file. Questo approccio garantisce la massima accuratezza.
- Confrontare i risultati in formato `.pgm`, dove g viene mappata da una matrice di valori in virgola mobile a un'altra di valori nell'intervallo 0-255. Questo metodo offre un'accuratezza inferiore rispetto al precedente.
- Confrontare visivamente i risultati in formato `.png`. Questo è il metodo meno preciso.

Per i risultati generati, oltre alla verifica visiva (cioè il terzo metodo), è stato utilizzato anche il secondo metodo come conferma. Su Linux, questo confronto può essere effettuato tramite il comando:

```
compare -metric RMSE PGM1 PGM2 OUTPUT
```

dove:

- `PGM1` va sostituito con il percorso del primo file in formato `.pgm` da considerare.
- `PGM2` va sostituito con il percorso del secondo file in formato `.pgm` da considerare.
- `OUTPUT` va sostituito con il percorso del file in cui verranno scritte le differenze tra i due file considerati.

Si deve inoltre considerare che, se la metrica RMSE (Root Mean Square Error) ottenuta è un valore sotto a 1000, i due risultati possono essere considerati altamente simili.

Capitolo 4

Valutazione delle prestazioni

Nella valutazione delle prestazioni dell'implementazione CUDA trattata in questo progetto di tesi, le metriche di analisi considerate sono il tempo di esecuzione, il throughput e lo speedup rispetto alla versione OpenMP.

Si inizierà analizzando la complessità computazionale dell'algoritmo implementato, per poi presentare le statistiche ottenute dai test pratici eseguiti su un server.

Sia per OpenMP che per CUDA è stata utilizzata la versione di default dei programmi. Per OpenMP si considera un numero massimo di 100 voxel sull'asse y, mentre per CUDA questo valore è determinato dall'algoritmo euristico descritto in precedenza.

4.1 Statistiche

Prima di analizzare i risultati ottenuti è utile individuare la complessità computazionale dell'algoritmo considerato. Il limite superiore del tempo di esecuzione dell'algoritmo viene già individuato in [Col24]. Tale complessità computazionale è qui riproposta:

$$O(N_\theta \cdot N_{pixel}^2 \cdot (N_x + N_y + N_z)) \tag{4.1}$$

dove:

- O detta o-grande, indica che si tratta di un limite superiore del tempo di esecuzione, relativo al caso in cui un raggio interseca tutti i piani.
- N_θ è il numero di proiezioni calcolate, cioè il numero di posizioni della sorgente considerate.
- N_{pixel} è il numero di unità di misurazione per lato del rilevatore.
- N_x è il numero di voxel lungo l'asse x.
- N_y è il numero di voxel lungo l'asse y.
- N_z è il numero di voxel lungo l'asse z.

L'`isi-raptor03.csr.unibo.it` è il server di riferimento utilizzato per effettuare i test riportati successivamente. La CPU utilizzata è un `Intel Xeon CPU E5-2603 v4 @ 1.70 GHz` con 12 core fisici e 12 core logici (quindi senza Hyper-Threading), dotata di 64 GB di RAM. Nei test OpenMP riportati di seguito sono stati sfruttati tutti i 12 core disponibili. La

GPU impiegata nei test è una `NVIDIA GeForce GTX 1070`, dotata di 8 GB di RAM. Va ricordato che, su una GPU NVIDIA, l'esecuzione parallela dei processi avviene per blocchi, unità di lavoro indipendenti che possono essere eseguite in qualsiasi ordine. Nei test CUDA condotti, il numero di CUDA Thread per blocco è stato ridotto in modo significativo (da 1024 a 512) a causa dell'utilizzo dei registri dovuto all'implementazione adottata. Tuttavia, quando possibile, è consigliabile utilizzare il numero massimo di CUDA Thread per blocco disponibile al fine di ottimizzare le prestazioni.

Gli input considerati sono i file `CubeWithSphericalHole{400-1500}.dat`, cioè del tipo: cubo con cavità sferica. Il numero di pixel per lato della matrice rilevatore varia da 400 a 1500 con intervalli di 100 pixel tra un input e il successivo, per un totale di 12 input diversi.

Configurazione	N_θ	N_{pixel}	N_x	N_y	N_z
1	7	400		168	
2	7	500		210	
3	7	600		252	
4	7	700		294	
5	7	800		336	
6	7	900		378	
7	7	1000		420	
8	7	1100		462	
9	7	1200		504	
10	7	1300		546	
11	7	1400		588	
12	7	1500		630	

Tabella 4.1: Dimensione del problema per ogni configurazione considerata.

La Tabella 4.1 mostra quali sono la dimensione del problema. La configurazione indica le dimensioni dei parametri dell'input per ogni istanza testata. N_θ è costante perché identificato dagli angoli fissati $\theta = 90°$ e $\alpha = 15°$. Il numero di voxel sui vari assi N_x, N_y e N_z è uguale perché viene considerato un oggetto cubico. Tale valore del numero di voxel per asse viene calcolato dal generatore di input in base al numero di pixel per lato del rilevatore scelti. Il valore N_{pixel} viene fissato per ogni configurazione in modo empirico, cioè in base alle prestazioni del calcolatore a disposizione; pertanto, questa configurazione è specifica per il server utilizzato.

4.1.1 Tempo di esecuzione

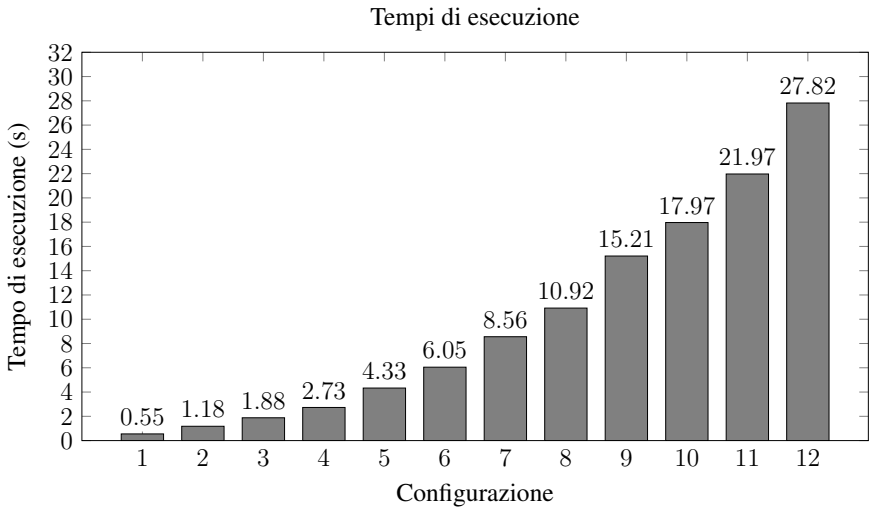
Il tempo di esecuzione potrebbe essere semplicemente considerato come il tempo totale impiegato dal programma per completare l'elaborazione. Tuttavia, ciò includerebbe anche i tempi di lettura dell'input e di scrittura dell'output, influenzando significativamente la misurazione. Per questo motivo, nel seguito verrà considerato il tempo di esecuzione dato da:

- Allocazione dati: per la CPU, comprende l'allocazione delle strutture dati e la loro inizializzazione, mentre per la GPU include anche il trasferimento dalla RAM dell'host alla RAM del device.

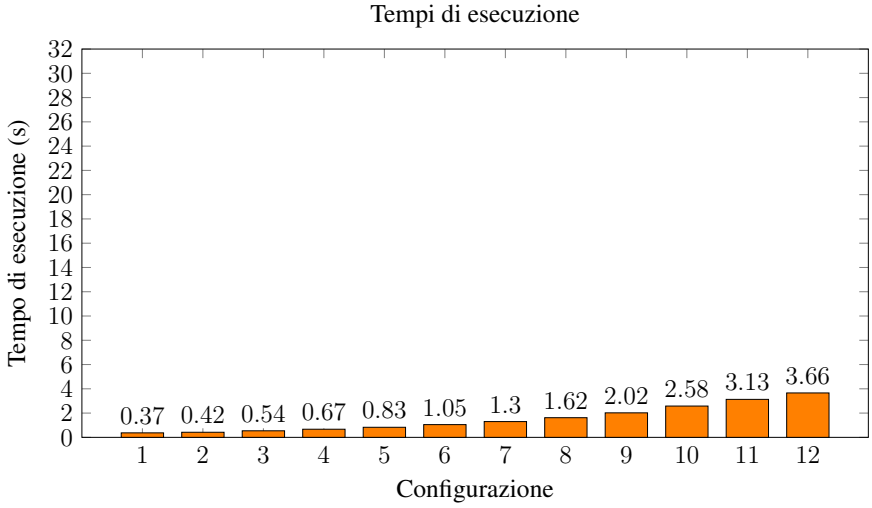
- Calcolo dei valori di attenuazione: comprende ogni chiamata alla funzione che calcola le proiezioni.
- Deallocazione dati: per la CPU, si riferisce semplicemente alla deallocazione delle strutture dati, per la GPU include anche il trasferimento dalla RAM del device alla RAM dell'host.

Non saranno presi in considerazione:

- Il tempo di lettura dell'input da file, poiché le operazioni sono identiche in entrambe le versioni.
- Il tempo di scrittura dell'output su file, in quanto, sebbene i risultati delle due versioni possano differire leggermente a causa dei calcoli in virgola mobile, le operazioni eseguite rimangono le stesse.



(a) OpenMP



(b) CUDA

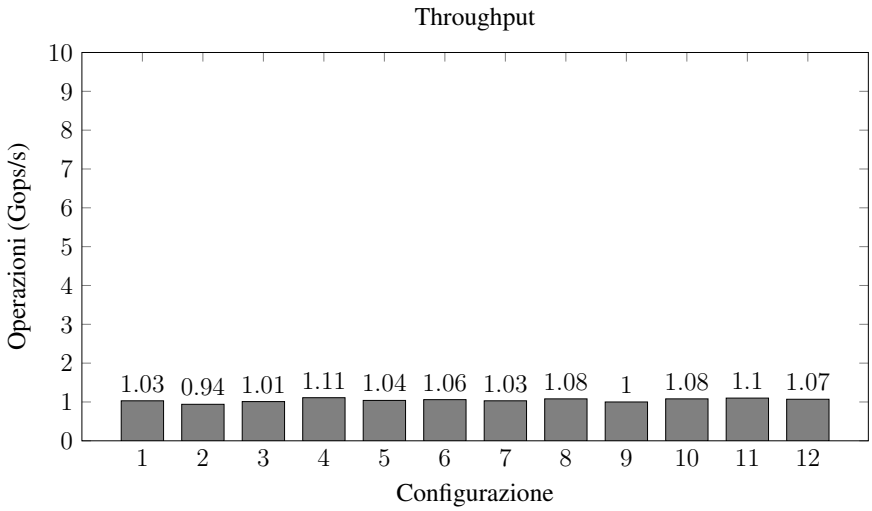
Figura 4.1: Tempi di esecuzione a confronto.

La Figura 4.1 mostra i tempi di esecuzione di OpenMP e CUDA a confronto. Come si può osservare, i tempi di esecuzione in CUDA sono fino a 7 volte inferiori rispetto a quelli ottenuti

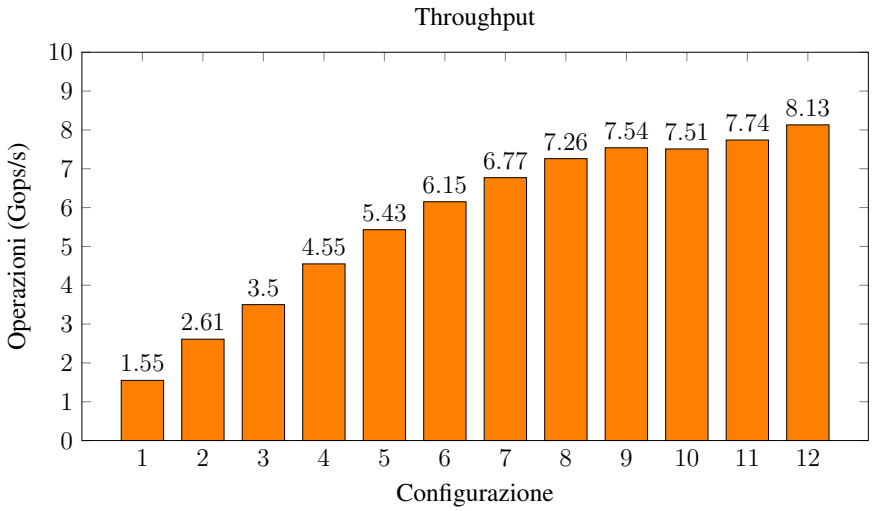
in OpenMP.

4.1.2 Throughput

Il throughput, è il numero di operazioni al secondo in funzione della dimensione dell'input. La dimensione dell'input è approssimata con il lavoro svolto dall'algoritmo, vedi l'Equazione 4.1.



(a) OpenMP



(b) CUDA

Figura 4.2: Throughput a confronto.

Il throughput è calcolato come $\frac{Numero\ di\ operazioni}{Tempo\ di\ esecuzione\ (s)}$. Per stimare il limite superiore del numero di operazioni al secondo eseguite dai programmi sono stati utilizzati i dati riportati nella Tabella 4.1, i risultati sono visibili nella Figura 4.2. Tale numero di operazioni è approssimato a miliardi di operazioni per secondo. In questo caso, il numero di operazioni al secondo in CUDA è fino a 8 volte superiore rispetto a quello in OpenMP.

4.1.3 Speedup

Lo speedup di CUDA è inteso come il confronto dei tempi di esecuzione della versione CUDA rispetto a quella OpenMP. Indica quanto l'implementazione per la GPU è mediamente più veloce rispetto a quella per la CPU, al variare della dimensione dell'input.

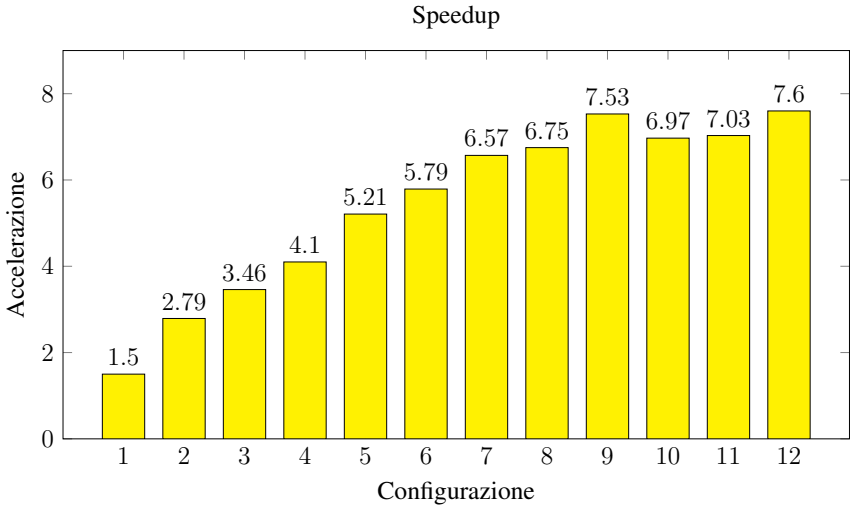


Figura 4.3: Accelerazione di CUDA rispetto a OpenMP.

La Figura 4.3 mostra il confronto delle prestazioni tra la versione CUDA e quella OpenMP. Per input di dimensioni ridotte, l'accelerazione della versione CUDA è limitata, mentre aumenta con l'aumentare della dimensione dell'input, stabilizzandosi intorno alle 6 volte, con un picco di 7,6 volte in un caso favorevole.

Capitolo 5

Conclusioni e sviluppi futuri

In questa tesi è stato affrontato il problema diretto della tomografia computerizzata 3D, con l'obiettivo di sviluppare una versione ottimizzata dell'algoritmo di Siddon [Sid85] per GPU NVIDIA utilizzando CUDA, partendo da una versione parallela per CPU basata su OpenMP [Col24].

Il lavoro è stato articolato in diverse fasi, ciascuna delle quali ha contribuito al raggiungimento degli obiettivi prefissati.

Nell'introduzione sono stati presentati l'evoluzione della tomografia computerizzata e i principi fondamentali della tecnica.

Di seguito sono stati presentati i fondamenti matematici e geometrici alla base della risoluzione del problema diretto. In particolare, l'equazione di Lambert-Beer è stata esaminata matematicamente, evidenziando il ruolo della matrice M e del vettore f nel processo di proiezione. Inoltre, il posizionamento della sorgente, dell'oggetto e del rilevatore ha permesso di definire il sistema di riferimento geometrico adottato.

Successivamente, nel capitolo dedicato alla descrizione del progetto di tesi, sono state illustrate le versioni del generatore di input e del proiettore OpenMP utilizzate come punto di partenza. Si è quindi discusso il processo di parallelizzazione adottato per lo sviluppo della versione CUDA, evidenziando i vantaggi e le criticità dell'implementazione proposta. Infine, è stata descritta la struttura dell'output generato e il metodo di verifica della correttezza dei risultati ottenuti.

Nella fase di valutazione delle prestazioni la versione CUDA è stata confrontata con l'implementazione OpenMP iniziale, analizzando i risultati attraverso diverse metriche. I test hanno confermato un notevole miglioramento delle prestazioni grazie all'uso della GPU.

In conclusione, questa tesi ha approfondito le tecniche di parallelizzazione di un algoritmo di proiezione tomografica su GPU. I risultati ottenuti dimostrano che l'implementazione CUDA sviluppata offre un'efficienza superiore rispetto a quella della versione OpenMP di partenza.

Tuttavia, rimangono alcune aree di miglioramento e possibili sviluppi futuri, tra cui:

- Determinare dinamicamente il numero massimo di piani da trattare sull'asse y, ad esempio tramite un algoritmo euristico, considerando la quantità di memoria RAM disponibile sulla scheda madre, sia per la versione OpenMP che per quella CUDA.

- Ottimizzare l'utilizzo di memoria nei CUDA Thread nella GPU, valutando alternative all'algoritmo classico di Siddon per ridurre la necessità di array di appoggio.
- Modificare la strategia di scrittura e lettura dell'input per supportare input con una distribuzione non uniforme di voxel lungo i vari assi.
- Riorganizzare il codice in modo da unificare le parti comuni delle due implementazioni, ad esempio attraverso lo sviluppo di librerie distinte con un'interfaccia condivisa.

Questi miglioramenti potrebbero contribuire a rendere l'algoritmo ancora più efficiente e flessibile, ampliandone le potenziali applicazioni in ambito tomografico.

Bibliografia

[Mar25] Enrico Marchionni. *3D-CT-projection-cuda*. 2025. URL: https://github.com/EnryMarch10/3D-CT-projection-cuda.git.

[Col24] Lorenzo Colletta. «Implementazione Parallela di un Algoritmo di Proiezione Tomografica». Tesi di laurea triennale. Ingegneria e Scienze Informatiche, Università di Bologna, Cesena, 2024. URL: https://amslaurea.unibo.it/id/eprint/33715/.

[MP21] Elena Morotti e Elena Loli Piccolomini. «Sparse Regularized CT Reconstruction: An Optimization Perspective». In: *Handbook of Mathematical Models and Algorithms in Computer Vision and Imaging: Mathematical Imaging and Vision*. A cura di Ke Chen et al. Cham: Springer International Publishing, 2021, pp. 1–34. ISBN: 978-3-030-03009-4. DOI: 10.1007/978-3-030-03009-4_123-1. URL: https://doi.org/10.1007/978-3-030-03009-4_123-1.

[FPN11] Lorenzo Faggioni, Fabio Paolicchi e Emanuele Neri. *Elementi di tomografia computerizzata*. Vol. 4. Springer Science & Business Media, 2011.

[Lin+08] Erik Lindholm et al. «NVIDIA Tesla: A Unified Graphics and Computing Architecture». In: *IEEE Micro* 28.2 (2008), pp. 39–55. DOI: 10.1109/MM.2008.31.

[Sid85] Robert L. Siddon. «Fast calculation of the exact radiological path for a three-dimensional CT array». In: *Medical Physics* 12.2 (1985), pp. 252–255. DOI: https://doi.org/10.1118/1.595715. eprint: https://aapm.onlinelibrary.wiley.com/doi/pdf/10.1118/1.595715. URL: https://aapm.onlinelibrary.wiley.com/doi/abs/10.1118/1.595715.

[Fly66] M.J. Flynn. «Very high-speed computing systems». In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909. DOI: 10.1109/PROC.1966.5273.