ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica - Scienza e Ingegneria - DISI

Corso di Laurea in Ingegneria e scienze informatiche

Implementazione CUDA dell'algoritmo FDK per la ricostruzione tomografica

Tesi di laurea in High Performance Computing

Relatore:
Chiar.mo Prof.
Moreno Marzolla

Correlatrice:
Chiar.ma Prof.ssa
Elena Loli Piccolomini

Presentata da:
Francesco Bittasi

Sessione di luglio 2025
Anno Accademico 2024/2025

Ai miei genitori e ai miei nonni

Sommario

La **Tomografia** è la tecnica di elaborazione delle immagini per la visione di sezioni trasversali di corpi e oggetti. Questa si realizza acquisendo proiezioni da diverse angolazioni tramite raggi X e ricostruendo l'interno del volume per retroproiezione, ovvero risolvendo il problema inverso della proiezione.

Grazie all'avvento dei computer la **Tomografia Computerizzata** è diventata un ampio campo di ricerca che sviluppa algoritmi e tecniche di ricostruzione sempre più dettagliate e veloci, per rendere più efficienti le diagnosi mediche non intrusive. Gli algoritmi che risolvono questo problema, ed in particolare i più recenti metodi iterativi, sono noti per essere computazionalmente molto impegnativi; per questo motivo, sfruttare le tecniche di ***High Performance Computing*** è fondamentale per consentire una ricostruzione delle immagini precisa e in tempi ridotti compatibili con il mondo ospedaliero.

I due processi fondamentali che caratterizzano questo problema sono la proiezione e la retroproiezione. Disporre di implementazioni efficienti di entrambi è fondamentale anche per lo sviluppo di algoritmi più avanzati che li utilizzano in molteplici iterazioni. Avendo già a disposizione un **proiettore** "*ray-driven, cone-beam*" efficiente, realizzato in una precedente tesi di laurea [1], questa tesi si propone di implementare un **retroproiettore** rapido e dello stesso tipo, sfruttando **CUDA** come API per calcolo parallelo su GPU.

Le retroproiezioni sono note generare immagini poco chiare e sfocate; perciò, ci siamo posti anche l'obiettivo di implementare i filtri dell'**algoritmo di Feldkamp-Davis-Kress (FDK)** [2] per consentire una ricostruzione più nitida e fedele del volume.

La tesi presenta una parte introduttiva alla Tomografia Computerizzata, all'High Performance Computing e a CUDA come interfaccia di programmazione delle GPGPU. Ne segue uno studio più approfondito del nostro caso di studio, delle tecniche di imaging adottate e dell'algoritmo FDK. Successivamente verrà descritta la nostra implementazione dell'algoritmo, le tecniche di programmazione parallela adottate per renderla il più efficiente possibile e, per concludere, verranno analizzati i risultati ottenuti e le prestazioni del programma.

Indice

Capitolo 1

Introduzione

In questo capitolo forniamo le nozioni fondamentali sulla Tomografia computerizzata e sul calcolo parallelo, approfondendo il calcolo su GPU tramite CUDA. Questa introduzione è necessaria per la comprensione del contesto e delle tecniche implementative che verranno adottate.

1.1 Fondamenti di Tomografia computerizzata

In seguito alla scoperta di Wilhelm Röntgen dei raggi X nel 1895 e la loro capacità di attraversare gli oggetti, i ricercatori hanno subito cercato di utilizzarli in medicina come strumento diagnostico non invasivo. Inizialmente veniva usata soltanto la radiografia planare, ovvero le singole proiezioni ottenibili dall'esposizione ai raggi: queste rilevano l'attenuazione che ciascun raggio subisce nell'attraversare il corpo, rivelando le diverse densità e proprietà dei tessuti attraversati.

Legge di Lambert-Beer

Il processo di attenuazione del raggio, ovvero la riduzione del numero di fotoni che lo compongono, è correlato allo spessore del volume attraversato e a un suo coefficiente di attenuazione [3]:

$$m(w + \Delta w) = m(w) - \mu(w)m(w)\Delta w \tag{1.1}$$

dove $m(w')$ rappresenta l'intensità del raggio nel punto $w' = w + \Delta w$, attenuato dall'attraversamento di un volume di spessore Δw e coefficiente di attenuazione costante $\mu = \mu(w)$.

Nel caso di spessori infinitamente piccoli ($\Delta w \to 0$) la **Legge di Lambert-Beer** esprime l'intensità del raggio nel punto w come:

$$m(w) = m_0 e^{-\mu w} \tag{1.2}$$

dove $m_0 = m(0)$ rappresenta l'intensità del raggio alla sorgente, come illustrato in Figura 1.1.

Espresso in forma integrale, questo modello è in grado di rappresentare l'attenuazione che il raggio subisce nell'attraversare un volume con una variazione continua del coefficiente di attenuazione μ. Il corpo umano corrisponde a questo modello in quanto diversi tessuti di diversa densità comportano una diversa capacità di assorbimento dei raggi X.

$$m(W) = m_0 e^{-\int_0^W \mu(w)dw} \tag{1.3}$$

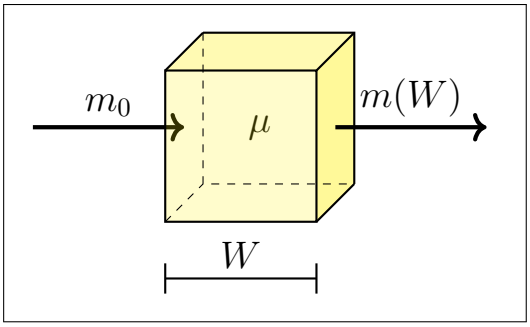


Figura 1.1: Attenuazione del raggio passante per un corpo di spessore W e funzione di attenuazione μ.

La Trasformata di Radon e la ricostruzione

È dal 1917 che, grazie al matematico Johann Radon, sarà possibile modellare matematicamente il concetto di ricostruzione tomografica: la **Trasformata di Radon** di un oggetto descritto da un coefficiente di attenuazione μ continuo è l'insieme delle proiezioni acquisite lungo una traiettoria circolare.

La ricostruzione tomografica rappresenta dunque il problema inverso della Trasformata di Radon: a partire dalle proiezioni ottenute da diverse angolazioni su una traiettoria circolare, cerchiamo di ottenere la funzione di attenuazione μ che descrive l'oggetto.

Grazie alle nozioni sulla legge di Lambert-Beer, riusciamo a modellare sia il problema diretto che quello inverso ma, a causa della sua natura, quest'ultimo è considerato matematicamente "mal posto". Questa condizione del problema rende impossibile ottenere una ricostruzione esatta del volume: le immagini ottenute sono infatti soggette a numerosi artefatti e richiedono molti dati, ovvero tante proiezioni, per ottenere buoni risultati.

I tomografi

Seguendo quando descritto dalla Trasformata di Radon, i tomografi realizzano la scansione di un oggetto da più angolazioni. Nel corso della storia sono stati sviluppati diverse generazioni di macchinari, nonché di algoritmi di ricostruzione, che trattano le proiezioni in maniera diversa.

Inizialmente limitata dalle capacità di calcolo, la ricostruzione è passata dalle singole sezioni a interi volumi tridimensionali, riducendo inoltre i tempi di acquisizione delle proiezioni. Un tomografo moderno effettua la proiezione dell'intero volume tramite

un singolo fascio di raggi a forma di cono: questa è la "*cone-beam projection*" e rappresenta l'estensione alla terza dimensione della geometria planare chiamata "*fan-beam projection*", entrambe rappresentate in Figura 1.2.

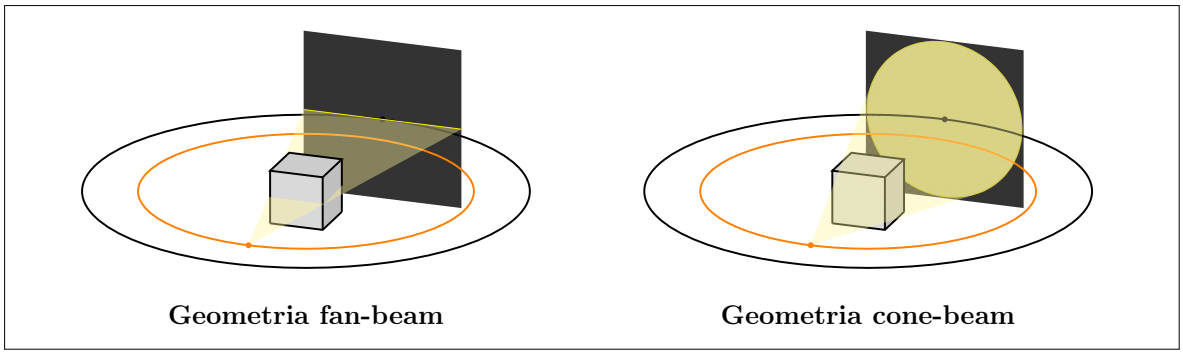


Figura 1.2: Rappresentazione della geometria fan-beam e dell'evoluzione cone-beam.

Formulazione discreta

Per poter realizzare la ricostruzione tramite l'uso di calcolatori è necessario discretizzare il problema e i modelli matematici usati.

Il volume in analisi è descritto come un insieme di piccole unità cubiche chiamate **Voxel** ("*volumetric picture element*", un'estensione 3D del concetto di pixel) caratterizzate da un coefficiente di attenuazione. La legge di Lambert-Beer diventa dunque esprimibile tramite una sommatoria dove la sequenza di voxel attraversati e la lunghezza dei singoli segmenti rappresentano il **percorso radiologico** del raggio nel volume:

$$m(w) = m_0 e^{-\sum_k \mu_k \Delta w_k} \tag{1.4}$$

dove $m(w)$ rappresenta l'intensità del raggio nel punto $w = \sum_k \Delta w_k$, dopo aver attraversato una serie di voxel di lunghezza Δw_k, ciascuno caratterizzato da un proprio coefficiente di attenuazione μ_k. Una sua rappresentazione è disponibile in Figura 1.3.

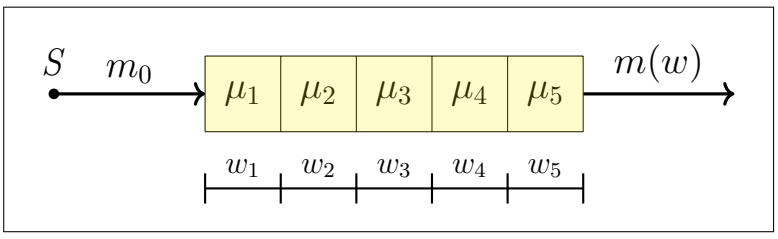


Figura 1.3: Attenuazione del raggio passante per un insieme di voxel.

Inoltre, il tomografo utilizza un rilevatore discreto: le immagini sono composte da diversi pixel, uno per singola unità di rilevazione. Nella modellazione del sistema è possibile adottare diversi metodi per individuare il contributo che ciascun voxel offre ai pixel del rilevatore. Le tecniche più semplici sono la "*pixel driven*" e la "*ray driven*" e sono rappresentate in Figura 1.4.

- **Pixel driven**: viene elaborato il contributo di ciascun voxel simulando un raggio passante per il suo centro.

- **Ray driven**: viene simulato un raggio per ogni pixel dell'immagine e si valutano i contributi dei voxel attraversati.

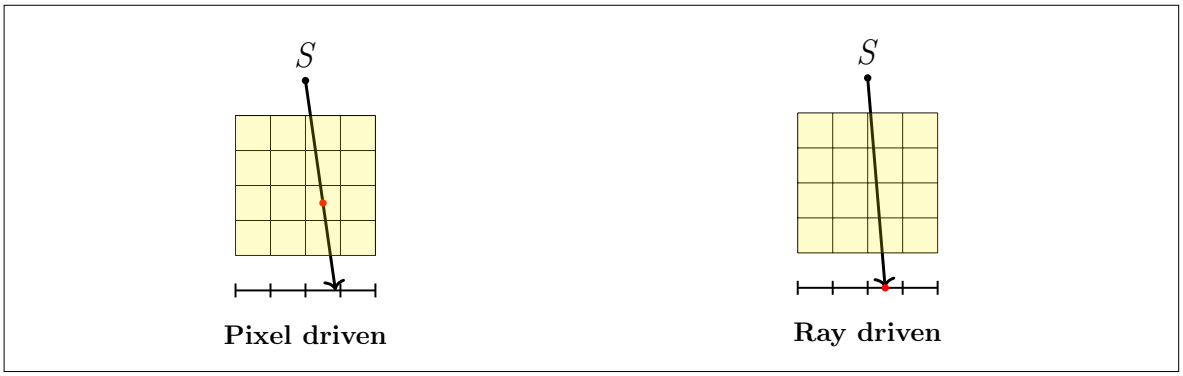


Figura 1.4: Confronto tra pixel driven e ray driven su una sezione del volume di voxel.

La metodologia ray driven sa essere più efficiente, ma comporterà alcuni difetti (descritti nella sezione 4.1) in quanto non garantisce di analizzare il contributo di tutti i voxel.

Il modello adottato

Per questo progetto è stata adottata una geometria composta da un volume 3D di voxel e da un proiettore a singola sorgente "cone-beam", che acquisisce le proiezioni su un rilevatore 2D posizionato frontalmente alla sorgente a distanza costante. Il proiettore adottato, e di conseguenza il retroproiettore che implementiamo, elabora le immagini col metodo "ray driven", simulando i raggi che attraversano il volume e calcolando il loro percorso radiologico con l'algoritmo di Siddon [4].

Per approfondire si rimanda a [5, 1]. Per approfondire l'evoluzione della tomografia computerizzata e le più recenti tecniche iterative si rimanda a [6, 7].

1.2 High Performance Computing

Con "*High Performance Computing*" si fa riferimento al campo dell'informatica che ha per obiettivo la risoluzione di problemi complessi in tempi ridotti e con elevata accuratezza, attraverso modelli di programmazione e architetture specializzate nel calcolo parallelo.

Programmazione parallela

Storicamente la *Legge di Moore* [8] garantiva il raddoppiamento del numero di transistor sui circuiti integrati ogni due anni, e con questo un costante aumento della loro capacità di calcolo. Ogni nuova generazione di processori era infatti basata su transistor più piccoli, ma oggi è diventato quasi impossibile rimpicciolirli ulteriormente a causa di limitazioni fisiche. Inoltre, aumentare la frequenza di calcolo dei processori non è sostenibile dal momento che l'eccessiva produzione di calore li renderebbe inaffidabili.

Con buona approssimazione è però possibile ottenere una riduzione del consumo energetico, riducendo dunque il calore ma mantenendo la stessa capacità di calcolo, utilizzando

più circuiti integrati a frequenza ridotta, al posto di uno solo. Grazie a questo ad oggi la *Legge di Moore* continua ad essere valida, non più per le dimensioni dei transistor, ma grazie all'integrazione di un numero crescente di **core**.

Programmare per architetture parallele richiede paradigmi di programmazione parallela dedicati. Un tipico programma parallelo segue la seguente struttura:

1. Il problema viene scomposto in sottoproblemi indipendenti tra loro.
2. I sottoproblemi vengono distribuiti e risolti dalle diverse unità di elaborazione.
3. I risultati parziali ottenuti vengono unificati per risolvere il problema totale.

Architetture parallele

Le architetture hardware vengono classificate tramite la *Tassonomia di Flynn* [9] in base al numero di flussi di esecuzione e dati gestibili simultaneamente:

- **SISD – Single Instruction Single Data stream**: rappresenta l'architettura di Von Neumann, un singolo flusso esecutivo che opera su un singolo dato alla volta;
- **SIMD – Single Instruction Multiple Data stream**: un'unica unità di controllo esegue la stessa operazione su differenti dati nello stesso momento;
- **MISD – Multiple Instruction Single Data stream**: applica diverse istruzioni su uno stesso dato, non è utilizzata in pratica;
- **MIMD – Multiple Instruction Multiple Data stream**: rappresenta le vere architetture parallele, opera diverse istruzioni su diversi set di dati nello stesso momento.

GPGPU

Le schede video (**GPU - Graphics Processing Unit**) sono nate per elaborare scene grafiche 3D, in particolare nell'ambito dei videogiochi.

All'inizio questi dispositivi di tipo "SIMD" seguivano una "pipeline" (una sequenza) di istruzioni fissa e non modificabile. Nel tempo, per consentire maggiore libertà creativa sulle immagini realizzate, le GPU hanno abbandonato la *Fixed Function Pipeline* per favorire una pipeline programmabile e personalizzabile.

Questi aggiornamenti hanno reso le schede video delle vere e proprie entità di calcolo con enormi capacità parallele e sono diventate parte integrante dell'High Performance Computing. Le **GPGPU (General Purpose GPU)** sono dunque dispositivi altamente parallelizzati, performanti e largamente programmabili.

1.3 CUDA

CUDA è l'API prioritaria NVIDIA per la programmazione *General Purpose* delle schede video.

L'algoritmo FDK si adatta bene al contesto della programmazione CUDA: il filtraggio delle proiezioni può essere elaborato indipendentemente per ogni pixel e anche la retroproiezione è altamente parallelizzabile [10]. Avendo già un'implementazione OpenMP

(API di parallelizzazione su CPU) del retroproiettore, sarà possibile vedere come, grazie alla notevole capacità di parallelismo delle GPGPU, programmare in CUDA porti a notevoli riduzioni dei tempi di esecuzione.

Struttura di un programma CUDA

Un programma CUDA è diviso in due componenti: l'**host program** e il **device program**. Il programma infatti opera da un lato su CPU ("*host*") e dall'altro sulla GPU ("*device*"), entrambi con propria memoria e comunicando tramite il PCIe BUS.

Un tipico programma CUDA segue le seguenti quattro fasi:

1. L'host inizializza i dati in RAM e li carica sulla memoria del device.
2. L'host chiama le funzioni di API per far partire il calcolo sulla GPU (le funzioni che eseguono sul device vengono chiamate "*kernel*").
3. La GPU esegue i calcoli operando sulla propria memoria.
4. Il risultato dal device viene ricopiato sulla RAM per poter essere fornito in output.

Architettura

Le schede video sono dotate di un'architettura complessa: sono composte da numerosi blocchi di esecuzione SIMD che operano in modo indipendente tra loro, rendendole di fatto dei dispositivi MIMD a livello architetturale.

Nel dettaglio:

- una scheda video contiene più *Streaming Multiprocessor (SM)* indipendenti;
- ogni SM è in grado di eseguire istruzioni differenti a gruppi di 32 thread, detti "*Warp*", tramite gli *Streaming Processor* (le singole unità di calcolo);
- un *thread* è un singolo flusso di istruzioni eseguite in sequenza.

Come detto, i Warp a livello hardware sono un'architettura di tipo SIMD, ma CUDA fornisce un'astrazione chiamata **SIMT (Single Instruction, Multiple Thread)** che permette divergenza nel flusso di controllo. All'interno del Warp non è possibile eseguire istruzioni diverse simultaneamente, il warp scheduler però è in grado di fermare e alternare l'esecuzione dei thread con percorsi di controllo differenti. Questa architettura ha dunque massima efficienza quando tutti i thread eseguono la stessa operazione, ma garantisce flessibilità per risolvere problemi complessi.

Una GPU è inoltre dotata di memoria globale, diversi livelli di caching e registri locali ai thread, in modo da consentire accessi ottimizzati ai dati.

Divisione logica dei thread e accessi in memoria

Le librerie CUDA astraggono dall'hardware e organizzano il calcolo in blocchi di thread disposti all'interno di una griglia: la **griglia** rappresenta l'intero lavoro da svolgere, i **blocchi** rappresentano una porzione indipendente del calcolo e possono contenere al massimo 1024 **thread**.

Questa struttura, rappresentata in Figura 1.5, è puramente logica: serve al programmatore per organizzare il kernel e non dover ragionare a basso livello. Tutti i thread, infatti, eseguono il codice scritto nel kernel, e selezionano su quali dati operare in base alla loro disposizione nella griglia. È lo scheduler ad occuparsi della distribuzione dei blocchi sull'hardware, gestendo anche il caso in cui la griglia sia troppo grande per la scheda video utilizzata.

Inoltre, i blocchi consentono meccanismi di comunicazione tra i thread che ne fanno parte grazie ad una porzione di memoria condivisa detta "**shared**". Questa è più veloce della memoria globale e, se usata correttamente, permette di ottenere grande riduzione dei tempi di calcolo. L'accesso alla memoria globale è infatti il punto in cui viene perso più tempo nell'esecuzione su GPU; è importante strutturare il programma in modo da favorire accessi ai registri locali, alla memoria shared e accessi sincronizzati a dati vicini tra loro.

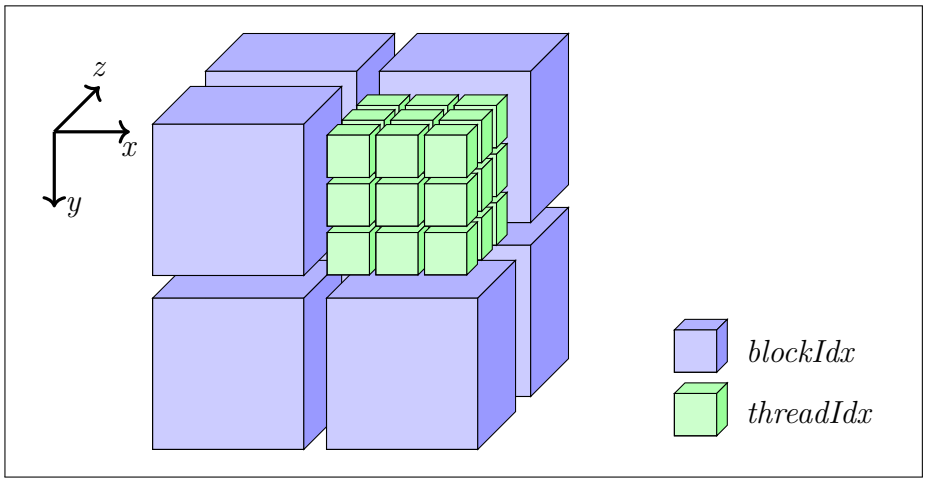


Figura 1.5: Organizzazione dei thread in una griglia di blocchi. I piccoli cubi verdi rappresentano i thread, quelli più grandi e blu i blocchi che formano la griglia.

Capitolo 2

La Filtered BackProjection (FBP) e l'algoritmo FDK

In questo capitolo descriviamo la **retroproiezione** ("*backprojection*") e la sua evoluzione nella "**Filtered Backprojection (FBP)**", studiando nello specifico l'**algoritmo FDK**.

Seguendo quanto definito dalla *trasformata di Radon*, la retroproiezione è il processo inverso della proiezione e permette di ottenere direttamente la ricostruzione di un volume: traccia al contrario i raggi della proiezione e diffonde le immagini nel corpo. Metodi differenti esprimono il volume come una matrice volumetrica di valori ignoti che vengono determinati iterativamente tramite sistemi di equazioni algebriche. Questi metodi permettono di ottenere risultati migliori e con un numero minore di immagini, consentendo la riduzione dell'esposizione ai raggi X e l'utilizzo di angolazioni limitate, ma sono stati a lungo troppo complessi computazionalmente per essere adottabili.

La retroproiezione è stata quindi per tanto tempo l'unica tecnica disponibile, e di conseguenza ne esistono tante varianti. Alcune di queste sfruttano iterativamente i processi di proiezione e retroproiezione per ottenere una ricostruzione migliore, ed è per questo motivo che è necessario disporre di implementazioni efficienti di entrambe le simulazioni.

Per approfondire le tecniche di ricostruzione si rimanda a [6] e [7].

2.1 La retroproiezione e la FBP

2.1.1 La proiezione

Per poter modellare la retroproiezione, è necessario prima comprendere come siano realizzate le proiezioni, ovvero il problema diretto.

Le proiezioni sono elaborate determinando per ogni pixel l'attenuazione totale del raggio che lo colpisce. Considerando il volume discreto, tale valore di attenuazione è calcolato come somma dei contributi che ciascun voxel offre all'assorbimento del raggio. In Figura 2.1 è riportata una rappresentazione schematica della proiezione di una sezione del volume. Per semplicità, viene mostrata una geometria a raggi paralleli.

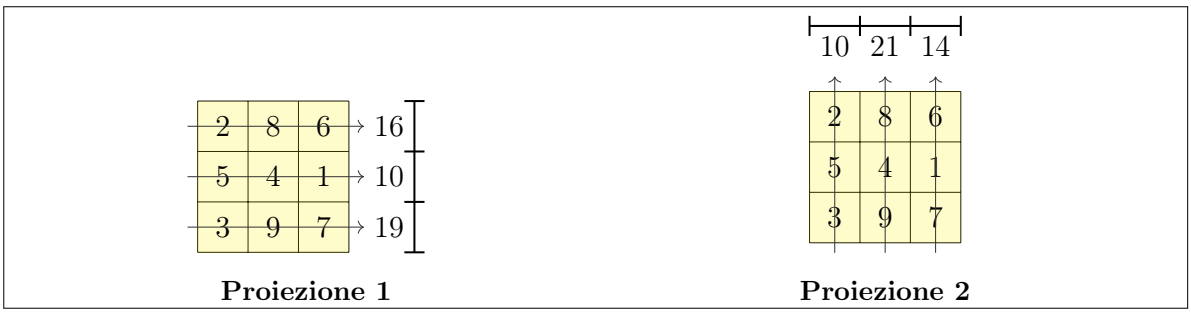


Figura 2.1: Rappresentazione schematica di una semplice proiezione di una sezione.

Nel caso specifico del modello adottato, definito nella sezione 1.1, il contributo di un voxel corrisponde al suo coefficiente di assorbimento, moltiplicato per la lunghezza del segmento di raggio interno al voxel [5]. I raggi simulati sono uno per pixel (viene usato il metodo ray driven), e ogni immagine viene realizzata da un'angolazione differente lungo una traiettoria circolare.

2.1.2 La retroproiezione

La retroproiezione deve svolgere il processo inverso di quello appena descritto: simulando gli stessi raggi, prende il valore del pixel corrispondente e lo distribuisce lungo i voxel attraversati. In Figura 2.2 è riportata la retroproiezione delle proiezioni ottenute in Figura 2.1.

Poiché l'obiettivo della tomografia non è ottenere l'esatto coefficiente di assorbimento di un punto del corpo, ma piuttosto distinguere correttamente le relative differenze di densità, è sufficiente diffondere il valore puro del pixel per realizzare una ricostruzione. Tipicamente viene applicata una normalizzazione o un'attenuazione dei valori distribuiti, ad esempio dividendoli per il numero di voxel attraversati, per prevenire errori causati dalla rappresentazione numerica finita nei calcoli computazionali. Nel caso illustrato in Figura 2.2, tale normalizzazione non è stata applicata per motivi di leggibilità.

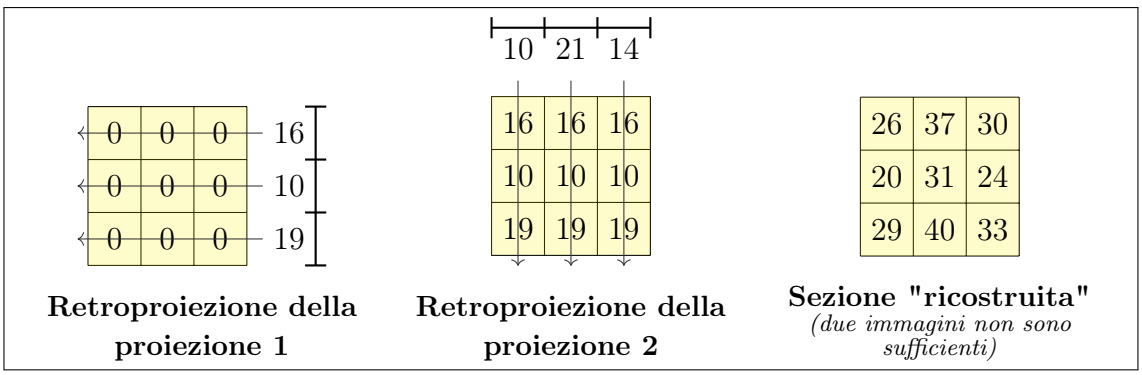


Figura 2.2: Rappresentazione schematica della retroproiezione delle proiezioni di Figura 2.1.

Il risultato mostrato in Figura 2.2 non rappresenta una buona ricostruzione a causa del numero limitato delle proiezioni disponibili. Il punto di forza dell'operazione risiede nel principio alla base della trasformata di Radon: applicando il processo inverso delle proiezioni su uno stesso volume, a partire da un insieme di immagini acquisite da an-

golazioni diverse lungo una traiettoria circolare, i contributi di ciascuna proiezione si sovrappongono in modo tale da ricostruire la struttura interna del volume.

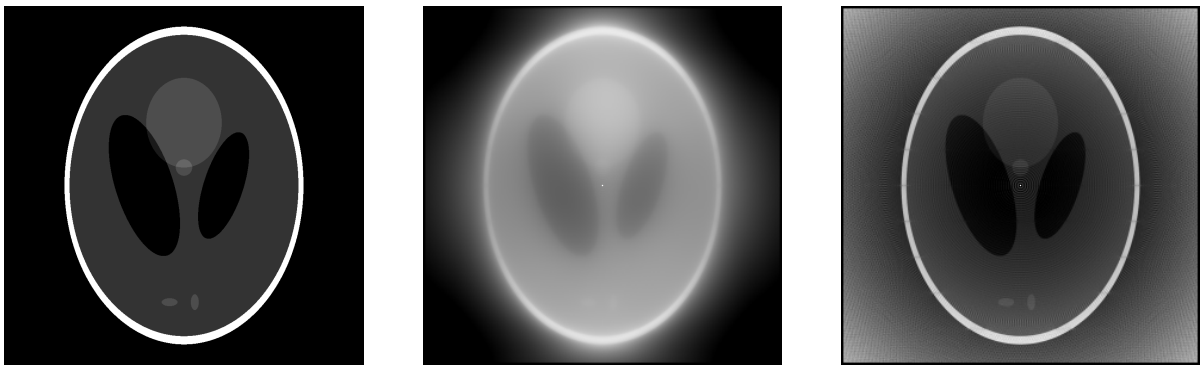
I risultati ottenuti dalla semplice retroproiezione delle radiografie permettono già di distinguere le caratteristiche macroscopiche del corpo, ma sono lontane dall'essere ottimali: la retroproiezione, infatti, è un'approssimazione del processo inverso desiderato, ma non lo può modellare esattamente. Questa discrepanza comporta la generazione di ricostruzioni sfocate e con maggiore luminosità nella zona centrale, a causa della maggiore sovrapposizione di immagini in quell'area. Un esempio di ricostruzione per retroproiezione è riportato nella Figura 2.3.

2.1.3 Filtered BackProjection

Per poter ottenere una ricostruzione più fedele è necessario dunque filtrarla per rimuovere la sfocatura e aumentare la nitidezza. Il filtro potrebbe essere applicato al volume ricostruito, ma è tipico applicarlo alle proiezioni prima di effettuarne la retroproiezione: il filtraggio di immagini bidimensionali ha una complessità computazionale inferiore rispetto a quello di volumi tridimensionali.

Questo processo descrive esattamente la **Filtered Backprojection (FBP)**: le proiezioni sono filtrate prima di effettuare la retroproiezione, così da ottenere una ricostruzione più nitida e chiara. In Figura 2.3 vengono confrontati i due metodi di ricostruzione col modello originale.

Esistono diversi filtri usati nella FBP; il filtro più semplice, nonché il più usato, è il **Ramp filter** e viene descritto nella sottosezione 2.2.3.



Modello | **Retroproiezione semplice** | **Ricostruzione con FDK (esempio di FBP)**

Figura 2.3: Confronto tra il modello e le ricostruzioni tramite semplice retroproiezione e FBP.

2.2 Filtrare un'immagine

2.2.1 Filtraggio nel dominio delle frequenze

Tramite la **Trasformata di Fourier** è possibile rappresentare una funzione come una combinazione di frequenze pesate. In questo modo, una funzione definita nel proprio

"*dominio spaziale*" può essere espressa nel cosiddetto "*dominio delle frequenze*", ovvero come integrale di componenti sinusoidali a frequenze diverse che, insieme, la descrivono.

Se consideriamo un'immagine come una funzione bidimensionale definita su un dominio discreto, tramite la **Trasformata di Fourier Discreta** è possibile esprimerla nel dominio delle frequenze. Le basse frequenze nella trasformata sono associate a componenti dell'immagine a bassa variabilità, ovvero alle feature di grandi dimensioni e le sue caratteristiche macroscopiche. Le frequenze elevate, invece, rappresentano i cambiamenti repentini di intensità, come i bordi e più generalmente i dettagli.

Filtrare un'immagine nel dominio delle frequenze consiste dunque nel modificare la Trasformata di Fourier moltiplicandola per una funzione filtro e poi applicare l'inverso della Trasformata per ottenere l'immagine filtrata. Filtri che attenuano le alte frequenze producono un effetto di sfocatura e sono chiamati "*low-pass filter*". I filtri che invece aumentano la nitidezza dell'immagine evidenziando le alte frequenze sono gli "*high-pass filter*" e sono quelli utilizzati nella FBP.

2.2.2 Filtraggio nel dominio spaziale per convoluzione

All'interno dell'implementazione presentata in questa tesi, però, le immagini sono trattate nel loro dominio spaziale e non sarebbe conveniente elaborare la Trasformata di Fourier e poi la sua inversa per realizzarne il filtraggio: grazie alle capacità parallele di CUDA è possibile filtrare un'immagine molto efficientemente.

I filtri lineari descrivono un pixel dell'immagine filtrata come la media pesata tra i pixel circostanti e un set di pesi. La natura del filtro è determinata dai pesi che lo definiscono e si dispongono in una maschera chiamata "*kernel convolutivo*". Il processo che applica i filtri lineari alle immagini prende il nome di "**convoluzione**": la maschera del filtro "scorre" lungo l'immagine per elaborare il risultato pixel per pixel. Applicando il pattern di programmazione parallela chiamato "*stencil*" è possibile realizzare la convoluzione molto efficientemente, applicando il kernel a ciascun pixel in parallelo.

Nel realizzare la convoluzione è necessario decidere come trattare i pixel ai bordi dell'immagine: non avendo pixel adiacenti, il filtro non sa quali valori usare nel calcolo. La tecnica più semplice consiste nel fissare i pixel esterni all'immagine a un valore costante, come se rappresentassero uno sfondo.

Tecnicamente la convoluzione inverte la maschera prima di applicarla. Dal momento però che useremo filtri simmetrici, salteremo questo passaggio e useremo la formula della **correlazione**. In Figura 2.4 è visibile una rappresentazione schematica della convoluzione e l'operazione svolta su ciascun pixel.

2.2.3 Ramp filter

Il **ramp filter** (noto anche come **Ram-Lak filter**) è un *high-pass filter* definito nel dominio delle frequenze come $H(\omega) = |\omega|$. In tomografia, è comune adottare una versione del filtro limitata ad un intervallo di frequenze, qui non riportata. Dalla definizione del filtro e dal grafico che lo rappresenta in Figura 2.5, vediamo come vengano intensificate frequenze di alto valore assoluto e attenuate quelle più vicine all'origine.

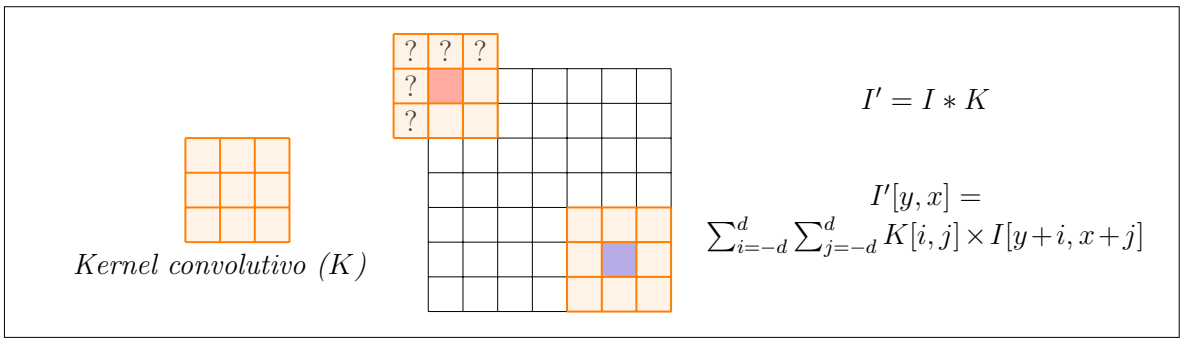


Figura 2.4: Convoluzione di un'immagine con un kernel simmetrico 3×3 ($d = \lfloor \frac{3}{2} \rfloor$). Il disegno mostra il posizionamento della maschera nei due pixel evidenziati. Per il pixel $(0, 0)$ bisogna scegliere come trattare i pixel segnati col simbolo '?'.

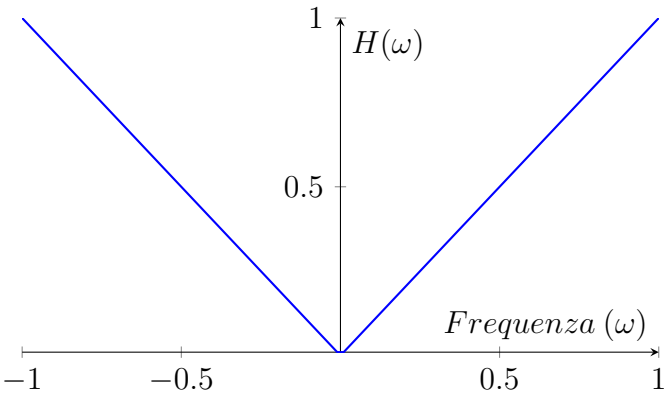


Figura 2.5: Ramp filter nel dominio delle frequenze, versione non limitata.

Applicando l'inversa della Trasformata di Fourier al ramp filter limitato, è possibile ottenere una definizione del filtro nel dominio spaziale [6]. Questa viene riportata nell'Equazione 2.1 e rappresentata in Figura 2.6.

$$h(n\tau) = \begin{cases} \frac{1}{4\tau^2} & n = 0 \\ 0 & n \text{ pari} \\ -\frac{1}{n^2\pi^2\tau^2} & n \text{ dispari} \end{cases} \tag{2.1}$$

dove τ è la dimensione di un'unità di rilevazione: considerando una riga di pixel, $n\tau$ rappresenta la posizione dell'n-esima colonna.

In [6] viene fornita anche l'Equazione 2.2 che definisce come tale filtro debba essere applicato all'immagine. Elaborando il filtro lungo l'asse orizzontale, i pixel di ascisse $n\tau$ sono elaborati tramite una media pesata tra la riga dell'immagine e il filtro h centrato sull'n-esima colonna, il tutto moltiplicato per τ.

$$\tilde{I}(y, n\tau) = \tau \sum_{k=0}^{N-1} h(n\tau - k\tau) I(y, k\tau), \quad n = 0, 1, 2, \ldots, N-1 \tag{2.2}$$

dove $\tilde{I}$ è l'immagine filtrata e I l'immagine originale.

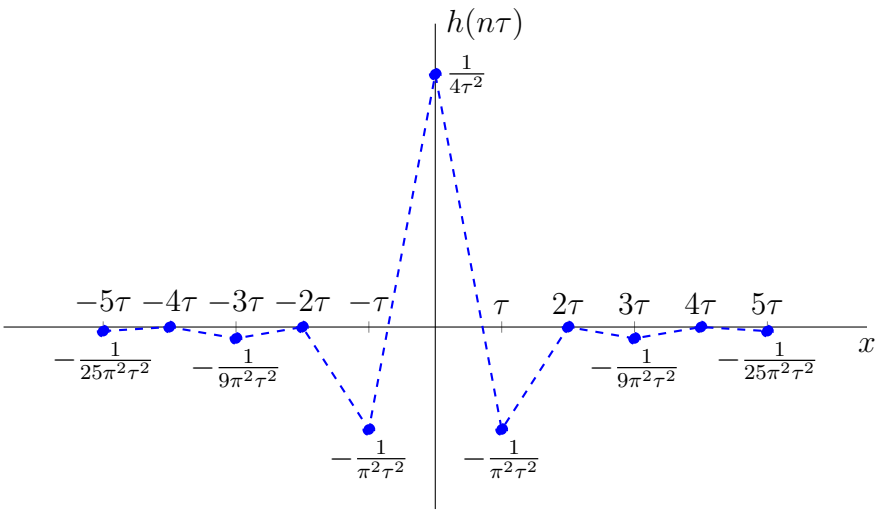


Figura 2.6: Ramp filter nel dominio spaziale.

Osservando l'Equazione 2.1 e la Figura 2.6, comprendiamo che la funzione h è pari, ovvero $h(x) = h(-x)$. Di conseguenza è possibile riformulare l'Equazione 2.2 invertendo h per facilitare la rappresentazione e l'implementazione del filtro:

$$\tilde{I}(y, n\tau) = \tau \sum_{k=0}^{N-1} h(k\tau - n\tau) I(y, k\tau), \quad n = 0, 1, 2, \ldots, N-1 \tag{2.3}$$

Realizzare questo calcolo, corrisponde quindi a realizzare una convoluzione con la maschera monodimensionale del ramp filter, i cui pesi sono definiti dall'Equazione 2.1, considerando posti a zero tutti i pixel esterni all'immagine. Una rappresentazione dell'applicazione del ramp filter ad un'immagine è visibile in Figura 2.7.

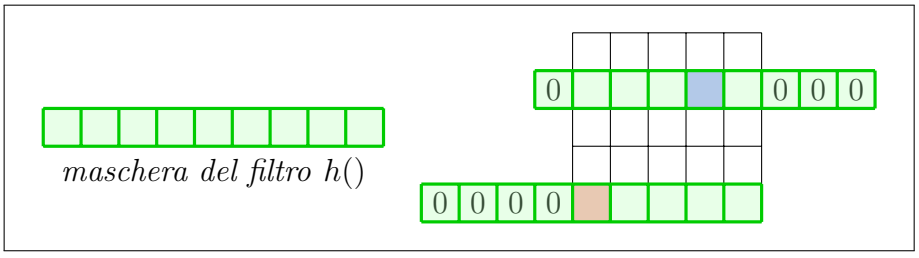


Figura 2.7: Applicazione della maschera monodimensionale del ramp filter a due pixel dell'immagine, considerando nulli i pixel esterni.

2.3 L'algoritmo FDK

L'algoritmo di Feldkamp, Davis e Kress (FDK) [2] è nato per essere un algoritmo di ricostruzione approssimativo, rapido ed efficiente, per la tomografia realizzata con geometria "cone-beam". Rappresenta di fatto un'estensione alla terza dimensione degli algoritmi di ricostruzione per geometrie planari "fan-beam".

La geometria utilizzata considera l'asse Z come asse verticale e posiziona la sorgente dei raggi sul piano XY. La distanza tra la sorgente e il centro di rotazione è indicata con

D. Le proiezioni sono realizzate su un piano rilevatore le cui coordinate saranno indicate con i simboli Y e Z, a rappresentare rispettivamente l'asse orizzontale e quello verticale.

2.3.1 Definizione dell'algoritmo

La formulazione continua proposta nell'articolo originale è quella riportata dalle equazioni 2.4 e 2.5: permette di calcolare il coefficiente di assorbimento di un singolo punto $r = (x, y, z)$ del volume, partendo da un set di immagini P.

$$f(r) = \frac{1}{4\pi^2} \oint d\Phi \frac{D^2}{(D + r \cdot \hat{x}')^2} \tilde{P}_\Phi[Y(r), Z(r)] \tag{2.4}$$

L'Equazione 2.4 modella la retroproiezione di tutte le immagini filtrate $\tilde{P}_\Phi$ sul punto r, dove Φ è l'angolazione di proiezione. Per farlo proietta r sulle immagini, individuando il corrispettivo punto sul rilevatore $[Y(r), Z(r)]$. Il risultato viene poi normalizzato per il numero di immagini retroproiettate che, essendo in una modellazione continua, corrisponde a $4\pi^2$.

Le immagini filtrate $\tilde{P}_\Phi$ sono ottenute dall'immagine originale applicando prima un peso $\cos\theta_{YZ}$ ad ogni pixel e poi i filtri g_y e g_z lungo i relativi assi. Il fattore $\cos\theta_{YZ}$ pesa il pixel in base all'angolazione del raggio che collega la sorgente al pixel (Y, Z): in questo modo corregge la geometria cone beam del proiettore e rimuove possibili distorsioni nella ricostruzione. L'Equazione 2.5 mostra il calcolo effettuato per determinare il punto (Y, Z) dell'immagine filtrata $\tilde{P}_\Phi$:

$$\tilde{P}_\Phi(Y, Z) = \int_{-\infty}^{\infty} dY' \int_{-\infty}^{\infty} dZ' g_y(Y - Y') g_z(Z - Z') \times P_\Phi(Y', Z') \cos\theta_{Y'Z'} \tag{2.5}$$

avendo:

$$\cos\theta_{YZ} = \frac{D}{\sqrt{D^2 + Y^2 + Z^2}} \tag{2.6}$$

dove D è la distanza tra la sorgente e il centro di rotazione, e Y, Z sono le coordinate spaziali del pixel sul rilevatore: $Y = \tau y$, con y indice della colonna (analogamente si sviluppa Z).

Come filtro abbiamo usato il ramp filter e all'interno di [2] viene dimostrato come la sua applicazione lungo l'asse verticale sia superflua. La formula che descrive il filtraggio delle immagini è dunque esprimibile nella seguente forma semplificata [11]:

$$\tilde{P}_\Phi(Y, Z) = \left(\frac{D}{\sqrt{D^2 + Y^2 + Z^2}} \cdot P_\Phi(Y, Z)\right) * h(Y) \tag{2.7}$$

L'Equazione 2.7 modella il calcolo implementato e descritto nella sezione 3.1: ciascun pixel viene prima pesato in base alla sua posizione nell'immagine e, successivamente, viene applicato il ramp filter monodimensionale lungo l'asse orizzontale dell'immagine. Per il calcolo dei pixel filtrati da h, è stata usata la formula definita dall'Equazione 2.3.

2.3.2 Retroproiettore FDK ray driven

Per trarre i benefici dei filtri definiti in FDK, non è importante il modo in cui viene effettuata la retroproiezione: siamo liberi di ignorare l'Equazione 2.4 e utilizzare invece un retroproiettore ray driven, come posto da obiettivo.

La nostra implementazione, dunque, adotterà i filtri di FDK e ne effettuerà la retroproiezione tramite un meccanismo ray driven. Nella sezione 4.1.2 osserveremo i difetti di ricostruzione causati da questa scelta.

Nel dettaglio, l'implementazione discreta che realizzeremo seguirà i seguenti passaggi:

1. vengono applicati ai pixel delle immagini i pesi definiti nell'Equazione 2.6;
2. viene applicato il ramp filter come da Equazione 2.3 lungo le righe delle immagini;
3. vengono tracciati i raggi di tutte le proiezioni da tutte le angolazioni;
4. viene elaborato il percorso radiologico di ciascun raggio;
5. ad ogni voxel attraversato viene sommato un contributo di assorbimento, determinato dal pixel generato dal raggio moltiplicato per la lunghezza del segmento interno al voxel.

Il passaggio cinque non diffonde il valore puro del pixel, ma lo rielabora per emulare più similmente il processo di proiezione descritto nella sottosezione 2.1.1. Nella sezione 3.2 verrà descritto nel dettaglio come viene elaborato il contributo sommato a ciascun voxel.

Capitolo 3

Implementazione parallela dell'algoritmo FDK

In questo capitolo descriviamo la nostra implementazione dell'algoritmo FDK, i pattern di programmazione parallela adottati e le tecniche usate per sfruttare al meglio l'architettura CUDA e le sue peculiari caratteristiche.

Questa implementazione rientra nello sviluppo di un repository dedicato alla ricostruzione tomografica; il nostro codice dunque sfrutta le strutture, le funzioni di utility, le librerie e le interfacce già definite ed implementate. Tutti i sorgenti sono disponibili pubblicamente sul repository "3D-CT-reconstruction".

Il programma è suddiviso in due fasi:

1. **filtraggio delle proiezioni** applicando i pesi e il ramp filter in parallelo per ciascun pixel;
2. **ricostruzione del volume tramite retroproiezione** delle immagini filtrate, simulando tutti i raggi in parallelo.

Poiché le due fasi sono implementate in due moduli distinti, è anche possibile eseguire direttamente la retroproiezione delle immagini non filtrate, saltando così la prima fase.

3.1 Filtraggio delle immagini

Il filtraggio opera i seguenti passi:

1. viene inizializzato l'ambiente CUDA allocando memoria, caricando i dati necessari e chiamando la generazione delle maschere dei filtri;
2. il corpo centrale del programma carica le proiezioni sulla GPU, chiama i kernel che applicano i filtri e scarica il risultato delle immagini in memoria RAM;
3. viene terminato l'ambiente CUDA liberando la memoria allocata e viene chiamata la retroproiezione delle immagini filtrate.

I valori utilizzati per filtrare le proiezioni sono costanti per tutte le immagini; per questo motivo li determiniamo una volta sola in fase di inizializzazione. I pesi hanno valore

costante e dipendono solamente dalla posizione del pixel, mentre il Ramp filter è un semplice filtro lineare, ovvero una media pesata con valori prefissati.

Quello descritto è un flusso di esecuzione semplificato: nel programma abbiamo implementato diversi meccanismi di gestione della memoria che descriviamo qui di seguito e che mostriamo nel diagramma di flusso in Figura 3.1.

- La GPU ha memoria limitata e perciò non può elaborare tutte le immagini nello stesso momento. Il punto due viene dunque eseguito in più turni: elaboriamo un blocco di proiezioni alla volta, determinando tramite delle euristiche lo spazio disponibile sulla scheda video.
- Le immagini filtrate vengono salvate su dei file di appoggio temporanei che verranno passati come input al retroproiettore. L'utilizzo dei file comporta perdita di tempo nella fase di retroproiezione, ma garantisce di elaborare il risultato anche in caso di memoria RAM di piccole dimensioni.

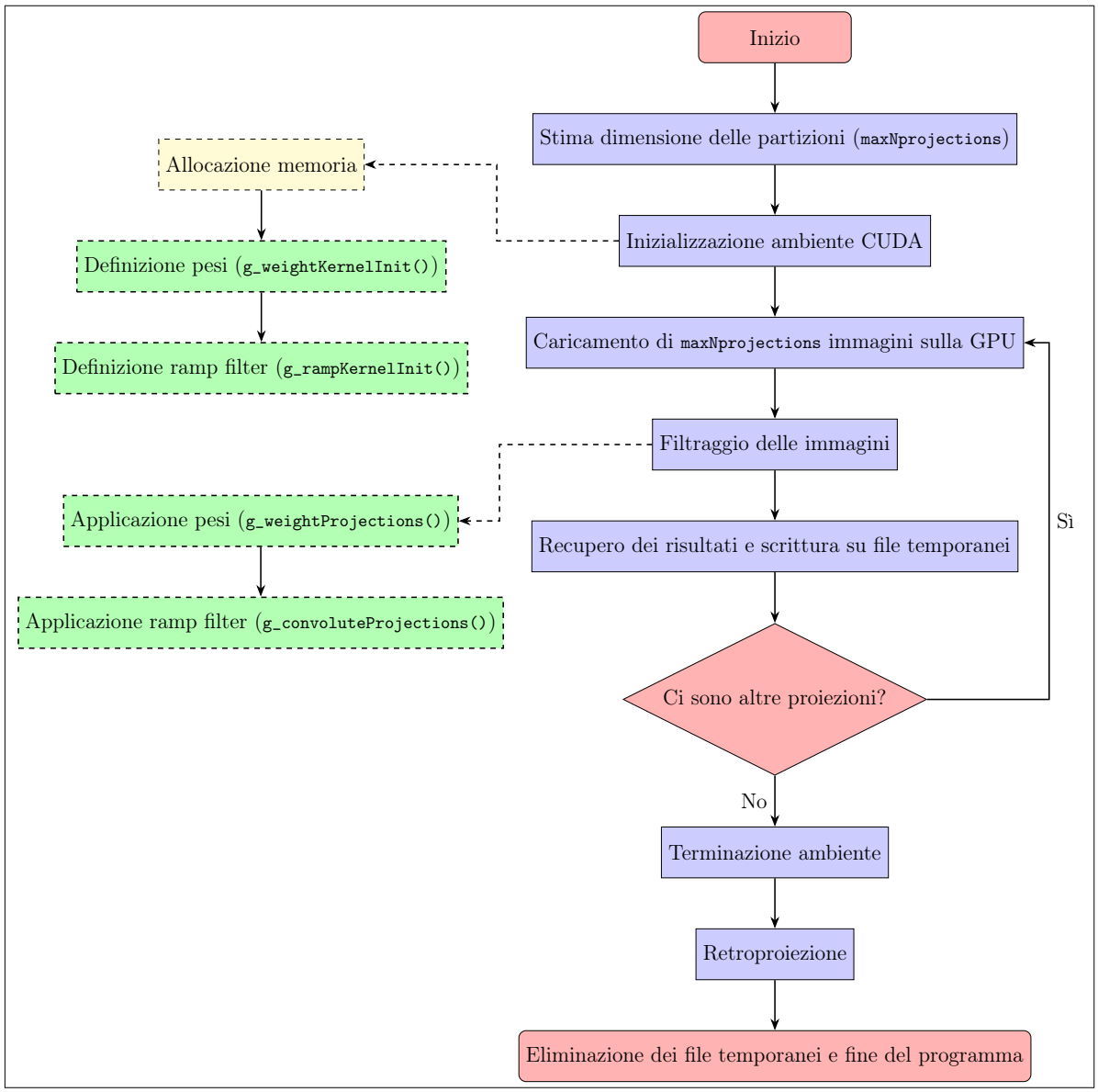


Figura 3.1: Diagramma di flusso del filtraggio. Nei sottoprocessi sono specificati i kernel eseguiti in parallelo.

Ogni pixel viene elaborato indipendentemente dagli altri e per questo motivo l'intero processo di filtraggio può essere classificato come "**Embarrassingly parallel**". Con questa terminologia si intendono tutti i problemi per i quali la risoluzione può essere banalmente scomposta in sotto-problemi totalmente indipendenti e dunque estremamente parallelizzabili.

Nella lettura dei listati e delle figure è importante tenere presente che, nel *Computer Imaging*, i pixel sono solitamente ordinati prima lungo l'asse verticale e poi lungo quello orizzontale, e che l'origine del sistema di coordinate si trova in alto a sinistra.

3.1.1 Applicazione dei pesi

Come descritto nella sezione 2.3 il filtraggio delle immagini è diviso in due fasi: la pesatura e l'applicazione del ramp filter. Qui mostriamo come abbiamo realizzato la prima fase.

Il peso applicato a ciascun pixel, come definito dall'Equazione 2.6, dipende unicamente dalle coordinate del pixel stesso sul rilevatore. È dunque sufficiente definire questi pesi una volta sola e applicare successivamente la maschera ad ogni immagine tramite una semplice moltiplicazione tra il valore del pixel e il peso corrispondente per tale posizione. La maschera viene generata dal kernel CUDA `g_weightKernelInit()` mostrato nel Listato 3.1. Questo kernel viene eseguito da una griglia quadrata con sufficienti blocchi 2D affinché ogni thread si occupi di un solo pixel; tutti i pesi sono dunque definiti in parallelo.

```
1  static dim3 w_block(SQUARE_BLOCK, SQUARE_BLOCK);
2  static dim3 w_grid(
3      (width + SQUARE_BLOCK - 1) / SQUARE_BLOCK,
4      (height + SQUARE_BLOCK - 1) / SQUARE_BLOCK
5  );
6  g_weightKernelInit<<<w_grid,w_block>>>(*device, distanceSourceOrigin,
       pixelSize, width, height);
7
8  __global__ static void g_weightKernelInit(d_paramPointers device, const
       int distanceSourceOrigin, const int pixelSize, const int width,
       const int height) {
9      // init local indexes and pixel sizes
10     // Y is the vertical axis on the detector
11     [...]
12
13     if (xIdx < width && yIdx < height) {
14         // Pixel position on planar detector
15         const double xPos = xIdx * l_pixelSize - (double)width / 2 +
               halfPixel;
16         const double yPos = yIdx * l_pixelSize - (double)height / 2 +
               halfPixel;
17         // FDK under equation (34)
18         const double weight = (double)distanceSourceOrigin /
19             sqrt((double)distanceSourceOrigin * distanceSourceOrigin
20             + xPos*xPos + yPos*yPos );
21         // Storing value
22         device.weightsKernel[xIdx + yIdx * width] = weight;
23     }
```

```
}
```

Listato 3.1: Codice per l'inizializzazione del peso.

L'applicazione della maschera alle immagini segue una struttura molto simile, ma la griglia è estesa alla terza dimensione: come rappresentato in Figura 3.2, l'asse z indica ai thread quale immagine elaborare. Abbiamo dunque sempre un thread per ogni pixel e il kernel `g_weightProjections()` (riportato nel Listato 3.2) si occupa unicamente di applicare il peso al pixel corrispondente nell'immagine indicata. In questo modo, su una GPU ideale con infinite capacità di calcolo parallelo, tutti i pixel di tutte le immagini sono elaborati in contemporanea. Le immagini sono caricate sulla GPU in un array monodimensionale chiamato `device.pixels[]`: permette una facile gestione dei file e un rapido caricamento dei dati. Ogni thread accede al pixel adeguato conoscendo le dimensioni delle immagini e il loro posizionamento nell'array.

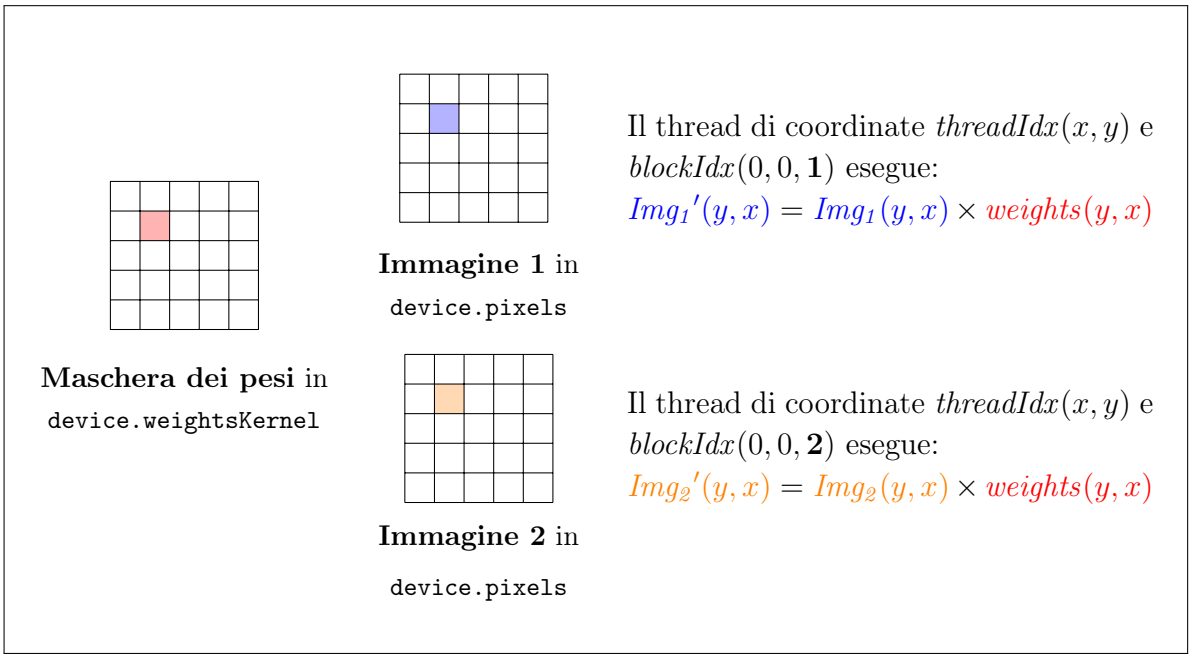


Figura 3.2: Applicazione dei pesi a due immagini.

```
static dim3 w_block(SQUARE_BLOCK, SQUARE_BLOCK);
static dim3 w_grid(
    (width + SQUARE_BLOCK - 1) / SQUARE_BLOCK,
    (height + SQUARE_BLOCK - 1) / SQUARE_BLOCK,
    nProjections
);
g_weightProjections<<<w_grid,w_block>>>(device, width, height);

__global__ static void g_weightProjections(d_paramPointers device,
   const int width, const int height) {
    const int xIdx = threadIdx.x + blockIdx.x * blockDim.x;
    const int yIdx = threadIdx.y + blockIdx.y * blockDim.y;
    const int projIdx = blockIdx.z;
    const int pixelIdx = yIdx * width + xIdx;
    // device.pixels contains all the pixels of all the loaded images
    const int global_pixelIdx = pixelIdx + projIdx * width * height;
```

```
16
17      if (xIdx < width && yIdx < height) {
18          // apply weight
19          device.pixels[global_pixelIdx] = device.pixels[global_pixelIdx]
                * device.weightsKernel[pixelIdx];
20      }
21  }
```

Listato 3.2: Codice per l'applicazione del peso.

Teoricamente tutte le proiezioni caricate sulla GPU sono elaborate in contemporanea ma, a causa delle grosse dimensioni delle immagini, è più probabile che lo scheduler esegua dei blocchi in tempi diversi. Abbiamo provato ad utilizzare un solo set di thread con griglia 2D che applicasse il peso alle immagini in un ciclo, ma è risultata più conveniente la tecnica sopra descritta.

3.1.2 Convoluzione tramite Ramp filter

Anche il Ramp filter, come il peso, viene elaborato una volta sola per poi essere riutilizzato. Il filtro applicato è monodimensionale e ne vengono utilizzate porzioni diverse in base alla posizione dei pixel; il calcolo applicato rientra dunque nella categoria dei filtri lineari e questa implementazione può essere considerata appartenere al pattern di programmazione parallela chiamato "**Stencil**".

Definizione della maschera e sua applicazione

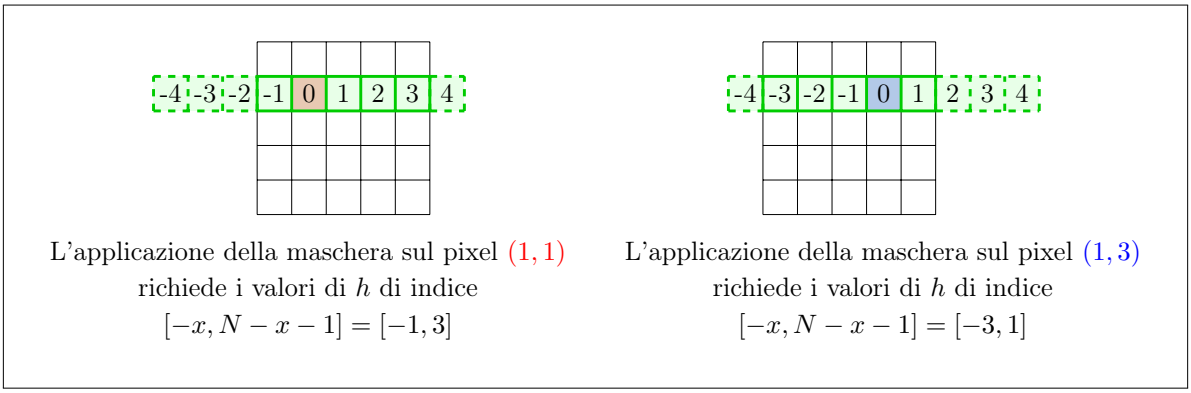
Considerando la larghezza N dell'immagine e l'Equazione 2.3, capiamo che il sottoinsieme del dominio di h utile per l'applicazione del filtro ad un pixel risiede in uno specifico intervallo: ogni pixel di ascissa x richiede l'intervallo $[-x, N-x-1]$ (Figura 3.3). Considerando l'immagine nel suo complesso, per applicarle il filtro necessitiamo dunque dei valori di h di indice $[-N+1, N-1]$. La maschera viene quindi costruita su un array di dimensioni $2N-1$ e per ogni pixel ne verrà utilizzata una sottoporzione di dimensione N. Il kernel CUDA `g_rampKernelInit()` (riportato nel Listato 3.3) si occupa di creare la maschera del ramp filter seguendo l'Equazione 2.1. Anche in questo caso il kernel viene eseguito con una struttura tale da avere un thread per ogni singolo valore ($2N-1$ thread), perciò l'intera maschera viene costruita in parallelo.

```
1  static dim3 f_block(ROW_BLOCK);
2  static dim3 f_grid(((width * 2 - 1) + ROW_BLOCK - 1) / ROW_BLOCK);
3  g_rampKernelInit<<<f_grid,f_block>>>(*device, pixelSize, width);
4
5  __global__ static void g_rampKernelInit(d_paramPointers device, const
       int pixelSize, const int width) {
6      const int idx = threadIdx.x + blockIdx.x * blockDim.x;
7      // location 0 is centered in the array
8      const int posIdx = idx - (width);
9
10     const double pixelSize_cm = MICRO_TO_CM(pixelSize);
11     if (idx < width * 2 - 1) {
12         // Equation (61), chapter 3, Principles of CTI
13         device.rampKernel[idx] = (posIdx % 2 == 0) ? 0 : -(double)1 /
```

```
                    (posIdx * posIdx * M_PI * M_PI * pixelSize_cm *
                       pixelSize_cm);
        if (posIdx == 0) {
            device.rampKernel[idx] = (double)1 / (4 * pixelSize_cm *
               pixelSize_cm);
        }
    }
}
```

Listato 3.3: Codice per l'inizializzazione del Ramp filter.

L'applicazione del filtro invece, implementata nel kernel `g_convoluteProjections()` come mostrato nel Listato 3.4 e rappresentato in Figura 3.3, esegue per ogni pixel di ascissa x il calcolo descritto dall'Equazione 2.3: una media pesata tra la riga di pixel in `device.pixels[]` e i pesi dell'intervallo $[-x, N-x-1]$ salvati in `device.rampKernel[]`. Questo kernel si occupa anche di individuare i valori massimi e minimi dell'immagine, ma questo processo verrà descritto successivamente.



L'applicazione della maschera sul pixel $(1,1)$ richiede i valori di h di indice $[-x, N-x-1] = [-1, 3]$

L'applicazione della maschera sul pixel $(1,3)$ richiede i valori di h di indice $[-x, N-x-1] = [-3, 1]$

Figura 3.3: Applicazione del ramp filter su due pixel della stessa riga di lunghezza N=5. Il rettangolo evidenziato rappresenta la maschera di h e i relativi indici.

```
static dim3 f_block(ROW_BLOCK);
static dim3 f_grid(
    ((width * 2 - 1) + ROW_BLOCK - 1) / ROW_BLOCK,
    height,
    nProjections
);
g_convoluteProjections<<<f_grid,f_block>>>(device, width, height,
   pixelSize);

__global__ static void g_convoluteProjections(d_paramPointers device,
   const int width, const int height, const int pixelSize) {
    // pixel indexes
    const int p_xIdx_n = threadIdx.x + blockIdx.x * blockDim.x;
    const int p_yIdx = threadIdx.y + blockIdx.y * blockDim.y;
    const int projIdx = blockIdx.z;
    const int global_row = p_yIdx * width + projIdx * width * height;
    const int global_pIdx = global_row + p_xIdx_n;
    // block indexes
    const int thread_Idx = threadIdx.x;
    // result
    double updatedPixel = 0;
```

```
20
21      // Equation (66), chapter 3, Principles of CTI  - (n and k are
            swapped: h() is even)
22      for (int k = 0; k < width; k++) {
23          // current pixel longitudinal index is k
24          // - position in image is k + global_row
25          // h filter index is k - p_xIdx_n
26          // - position in filter array is offset by width
27          updatedPixel += (device.rampKernel[k - p_xIdx_n + width] *
                device.pixels[k + global_row]);
28      }
29      // multiply by the lenght of the pixel
30      updatedPixel *= MICRO_TO_CM(pixelSize);
31
32      // update global pixel value
33      if (p_xIdx_n < width) {
34          device.pixels[global_pIdx] = updatedPixel;
35      }
36
37      // find min and max values
38      [...]
39  }
```

Listato 3.4: Codice per l'applicazione del Ramp filter.

Come in `g_weightProjections()` ogni thread si occupa di un singolo pixel e dunque idealmente vengono tutti elaborati in parallelo. Il kernel `g_convoluteProjections()` si differenzia però in quanto, dal momento che il filtro viene applicato lungo le righe, i blocchi usati sono monodimensionali in modo da poter sfruttare la memoria shared con la tecnica descritta nel prossimo paragrafo.

Ottimizzazione tramite memoria shared

Il ramp filter richiede per ogni pixel la lettura dell'intera riga e di N valori della maschera. Come discusso nella sezione 1.3, gli accessi in memoria globale sono molto costosi e, dal momento che thread della stessa riga necessitano dei medesimi dati, è utile sfruttare la memoria shared.

La funzione `g_convoluteProjections()` può essere dunque modificata, come mostrato nel Listato 3.5, in modo da sfruttare due buffer in memoria shared (`l_pixels[ROW_BLOCK]` e `l_h[ROW_BLOCK * 2]`) in ogni blocco di `ROW_BLOCK` thread. Il kernel scorre la riga a partizioni di `ROW_BLOCK` pixel caricandoli in `l_pixels[]` e mettendo all'interno di `l_h[]` la sottoporzione del filtro h necessaria per il calcolo. Per la partizione di pixel $[k_{min}, k_{max}]$ (nel codice [a,a+ROW_BLOCK-1]), salvati in `l_pixels[]`, nel caso di un blocco di thread che elabora pixel con ascisse nell'intervallo $[x_{min}, x_{max}]$, è necessario caricare in `l_h[]` i pesi delle posizioni appartenenti all'intervallo $[k_{min} - x_{max}, k_{max} - x_{min}]$. Questo processo viene rappresentato schematicamente nella Figura 3.4, mostrando due iterazioni del ciclo di riga 10 del Listato 3.5, nel caso di un'immagine 6×6 con blocchi di dimensioni `ROW_BLOCK=3`.

Il partizionamento della riga è necessario a causa delle limitate dimensioni della memoria shared, ma grazie al parallelismo il caricamento dei dati è praticamente immediato e i numerosi accessi alla memoria sono ottimizzati dalla sua rapidità.

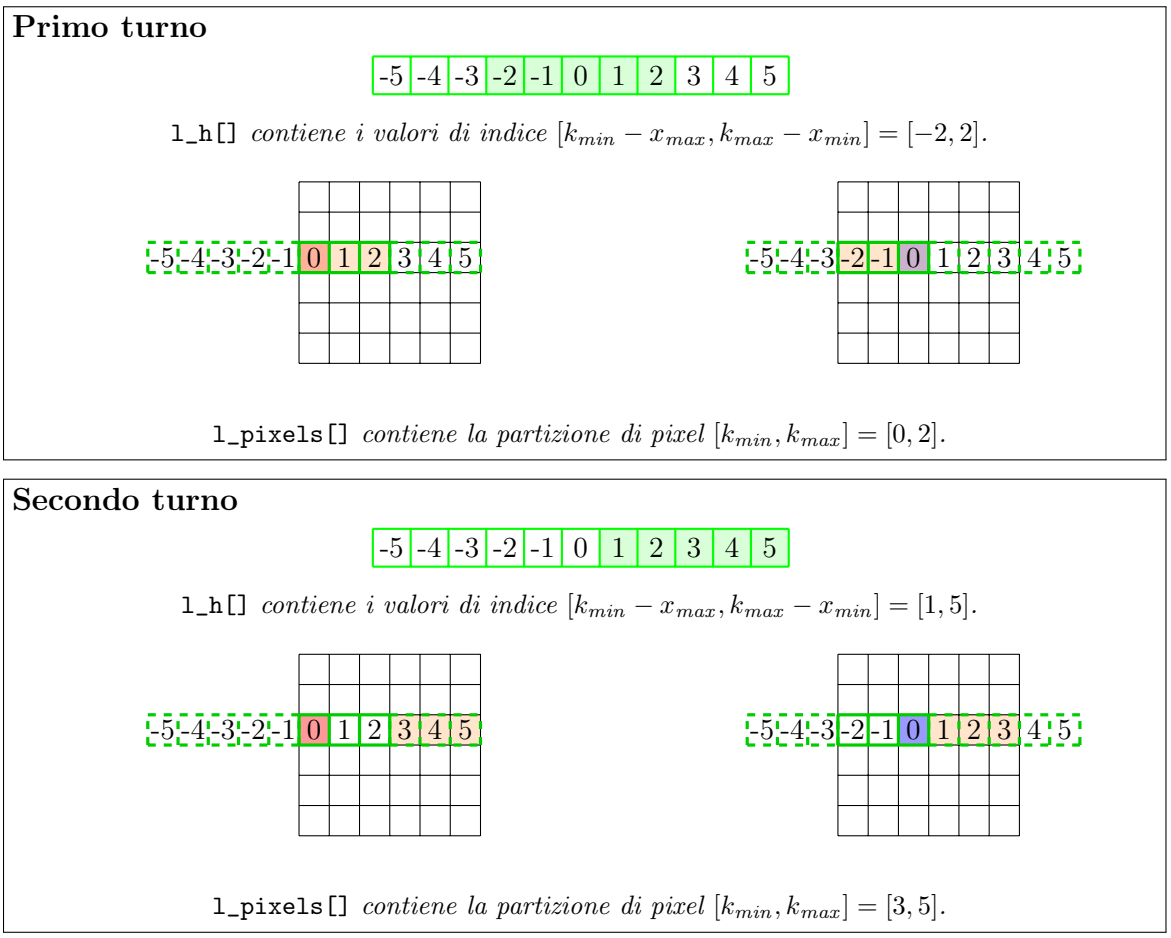


Figura 3.4: Applicazione del ramp filter tramite memoria shared su un blocco con `ROW_BLOCK=3`, che elabora i pixel di una riga con ascisse nell'intervallo $[0, 2]$.

```
__global__ static void g_convoluteProjections(d_paramPointers device,
    const int width, const int height, const int pixelSize) {
    // indexes retrival
    [...]

    // load first half of the kernel
    int h_offset = 0 - max_pxIdx_inBlock;
    int loadedIdx = h_offset + thread_Idx + width;
    l_h[ROW_BLOCK + thread_Idx] = (loadedIdx >= 0 && loadedIdx < width
        *2 - 1) ? device.rampKernel[loadedIdx] : 0;
    // iterates over chunks of the row
    for (int a = 0; a < width; a += ROW_BLOCK) {
        // Shared data update
        __syncthreads();

        /// load pixel chunk [a,a + ROW_BLOCK]
        l_pixels[thread_Idx] = (a + thread_Idx < width) ? device.pixels
            [a + thread_Idx + global_row] : 0.0;

        /// load kernel second half chunk
        /// l_h will contain [a - max_pxIdx_inBlock, (a + ROW_BLOCK) -
            min_pxIdx_inBlock] (k_min - n_max, k_max - n_min)
        // move second half to first half
```

```
20          l_h[thread_Idx] = l_h[ROW_BLOCK + thread_Idx];
21          // load new half
22          int loadedIdx_i = h_offset + ROW_BLOCK + thread_Idx + width;
23          l_h[ROW_BLOCK + thread_Idx] =
24              (loadedIdx_i >= 0 && loadedIdx_i < width*2 - 1) ?
25                  device.rampKernel[loadedIdx_i] : 0;
26          __syncthreads();
27
28          // Every thread computes the filtering for its own pixel
29          // Equation (66), chapter 3, Principles of CTI  - (n and k are
                swapped: h() is even)
30          if (p_xIdx_n < width) {
31              for (int k = a; k < a + ROW_BLOCK && k < width; k++) {
32                  updatedPixel += (l_pixels[k-a] * l_h[k - p_xIdx_n  -
                        h_offset]);
33              }
34          }
35          // move offset -> change loaded h() chunk
36          h_offset += ROW_BLOCK;
37      }
38      // multiply by the lenght of the pixel
39      updatedPixel *= MICRO_TO_CM(pixelSize);
40      // update pixel in the image
41      if (p_xIdx_n < width) {
42          device.pixels[global_pIdx] = updatedPixel;
43      }
44
45      // find min and max values of each image
46      [...]
47  }
```

Listato 3.5: Codice per l'applicazione del Ramp filter sfruttando la memoria shared.

Min-Max reduction

Durante la retroproiezione e la visualizzazione delle immagini è necessario normalizzare i valori dei pixel. Questa operazione richiede la conoscenza del valore massimo e minimo presenti in ciascuna immagine. Per calcolarli in modo efficiente, è stato utilizzato un pattern ben noto nella programmazione parallela chiamato "**Reduction**". Una riduzione è l'applicazione di un operatore binario associativo (ad esempio gli operatori *min* e *max*) agli elementi di un array. È facilmente parallelizzabile in quanto l'operatore viene applicato a coppie di elementi in contemporanea, riducendo di volta in volta l'array, fino a giungere ad un unico risultato.

L'implementazione parallela della *min-max reduction*, riportata nel Listato 3.6 e rappresentata schematicamente dalla Figura 3.5, è la stessa descritta in [1]: sfrutta array shared per effettuare una riduzione all'interno di ogni singolo blocco per poi confrontare, tramite operazioni atomiche, i risultati di ciascun blocco e ottenere il risultato finale. Il kernel CUDA elabora il massimo e il minimo di ciascuna immagine: sarà poi compito del programma host ottenere i valori assoluti a partire da quelli relativi del kernel, ma, data la quantità ridotta di proiezioni, non è necessaria un'implementazione parallela.

Gli array contenenti i risultati (`device.min` e `device.max`) si trovano in memoria globale e tutti i blocchi della stessa immagine cercano di modificare la stessa cella. Una **race condition** si verifica quando due processi cercano di modificare lo stesso valore nello stesso istante; questo provoca un errato aggiornamento della cella di memoria con conseguenti errori di calcolo. Per evitare questo problema, si utilizzano **operazioni atomiche** che assicurano un accesso sequenziale ai dati, anche se ciò comporta un rallentamento a causa dei tempi di attesa.

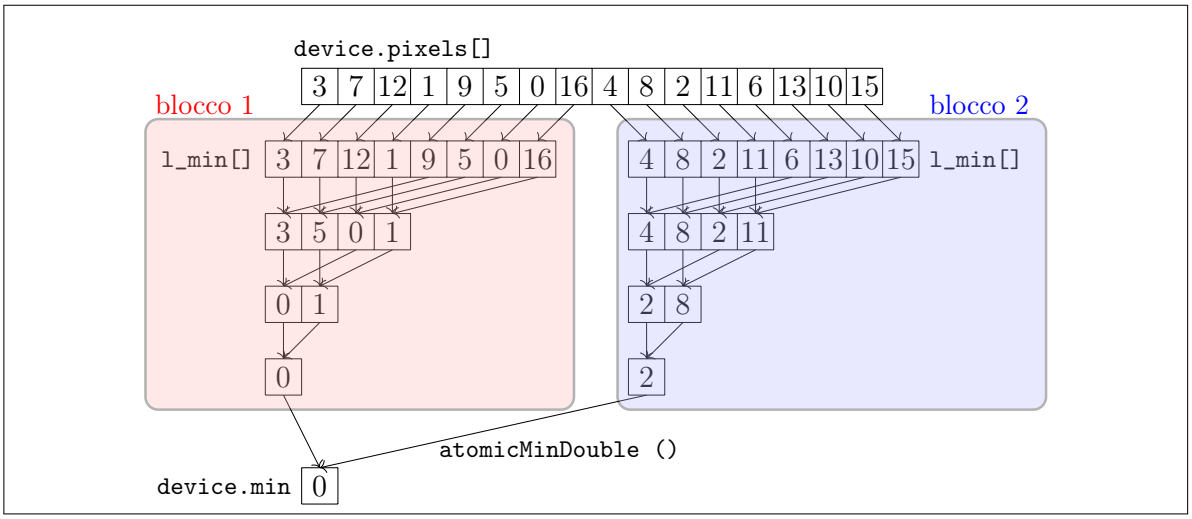


Figura 3.5: *Min reduction* di un array su due blocchi da 8 thread (la *max reduction* segue un processo analogo).

```
// min-max reduction via shared data
__shared__ double l_min[ROW_BLOCK];
__shared__ double l_max[ROW_BLOCK];
int bsize = ROW_BLOCK / 2;
l_min[thread_Idx] = (p_xIdx_n < width) ? updatedPixel : INFINITY;
l_max[thread_Idx] = (p_xIdx_n < width) ? updatedPixel : -INFINITY;
__syncthreads();
while ( bsize > 0 ) {
    if ( thread_Idx < bsize ) {
        if (l_min[thread_Idx + bsize] < l_min[thread_Idx]) {
            l_min[thread_Idx] = l_min[thread_Idx + bsize];
        }
        if (l_max[thread_Idx + bsize] > l_max[thread_Idx]) {
            l_max[thread_Idx] = l_max[thread_Idx + bsize];
        }
    }
    bsize = bsize / 2;
    __syncthreads();
}
if (thread_Idx == 0) {
    atomicMinDouble(&(device.min[projIdx]), l_min[0]);
    atomicMaxDouble(&(device.max[projIdx]), l_max[0]);
}
```

Listato 3.6: Codice per la riduzione min-max sfruttando la memoria shared.

3.2 Elaborazione della retroproiezione ray-driven

Come posto da obiettivo, abbiamo scelto di implementare un retroproiettore di tipo "ray-driven": l'algoritmo consiste nel simulare i raggi della proiezione al contrario e diffondere il valore dei pixel corrispondenti lungo i voxel attraversati.

Come discusso nella sottosezione 2.3.2, questo approccio differisce da quello adottato in FDK [2] che invece riproietta ogni voxel sulle immagini per valutarne il contributo, ma favorisce continuità con il proiettore adottato per la futura implementazione di algoritmi iterativi.

Struttura generale e tecnica adottata

In [10] è stato analizzato quali fossero gli approcci più convenienti per parallelizzare la retroproiezione ray driven e ne è stata realizzata un'implementazione con OpenMP. La nostra implementazione CUDA parte da questi studi e sfrutta il parallelismo massivo offerto dalle GPU per ridurre i tempi di elaborazione.

Il problema può essere affrontato in diversi modi, ma le prestazioni migliori si ottengono elaborando i raggi in parallelo [10]: ogni raggio è indipendente dagli altri e questo rende il processo parallelizzabile. La versione OpenMP, limitata dal numero di core del processore, deve scegliere se elaborare contemporaneamente diverse immagini, gestendo sequenzialmente i raggi di ciascuna, oppure se simulare, per ciascuna posizione, tutti i raggi in parallelo, processando in sequenza le varie angolazioni. Al contrario, CUDA non soffre di tali limitazioni e permette di simulare in parallelo tutti i raggi di tutte le immagini. Le due versioni restano comunque molto simili e si basano entrambe sull'algoritmo di Siddon [4] per il calcolo del percorso radiologico, la cui implementazione è stata ereditata da precedenti progetti di tesi [5, 10, 1]. In Figura 3.6 sono rappresentati i tre diversi approcci paralleli adottabili nella retroproiezione, con una vista dall'alto del volume.

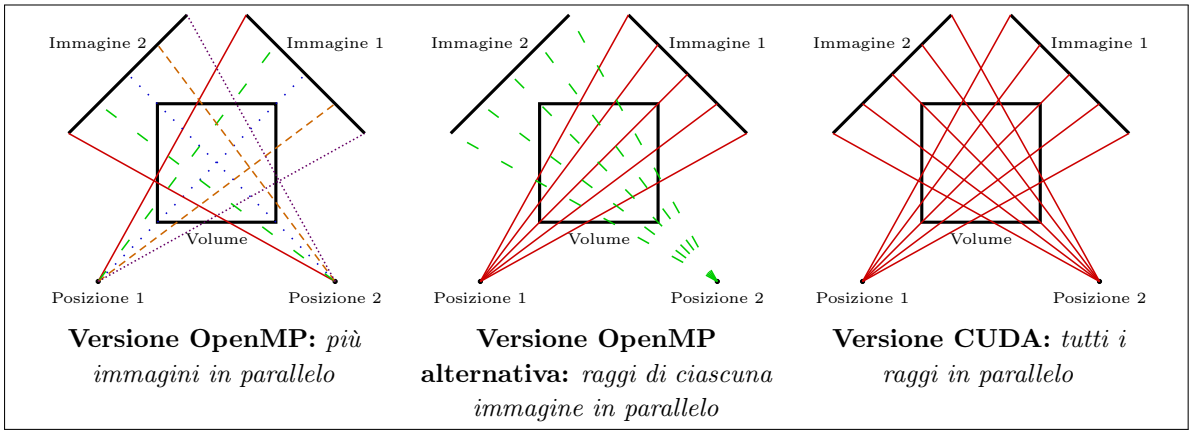


Figura 3.6: Confronto tra i diversi approcci paralleli per la simulazione dei raggi: raggi con lo stesso colore e stesso tratto sono elaborati in parallelo.

Come ogni programma CUDA, il retroproiettore è diviso in una fase di preparazione dell'ambiente della GPU, seguita dalla chiamata del kernel di calcolo e terminata dal trasferimento dei risultati sulla memoria RAM. Similmente al processo di filtraggio, è

necessario dividere il lavoro in diversi turni, per via delle limitazioni della memoria. Vengono dunque elaborate diverse partizioni del volume, rappresentate in Figura 3.8, su diversi sottoinsiemi di immagini per volta: sulla stessa partizione di volume, il kernel retroproiettore `g_computeBackprojections()` viene eseguito più volte con un set di immagini differenti. Questo complica un po' il flusso di esecuzione del programma, riportato in Figura 3.7, e ha richiesto ulteriori euristiche per stimare le dimensioni appropriate delle partizioni.

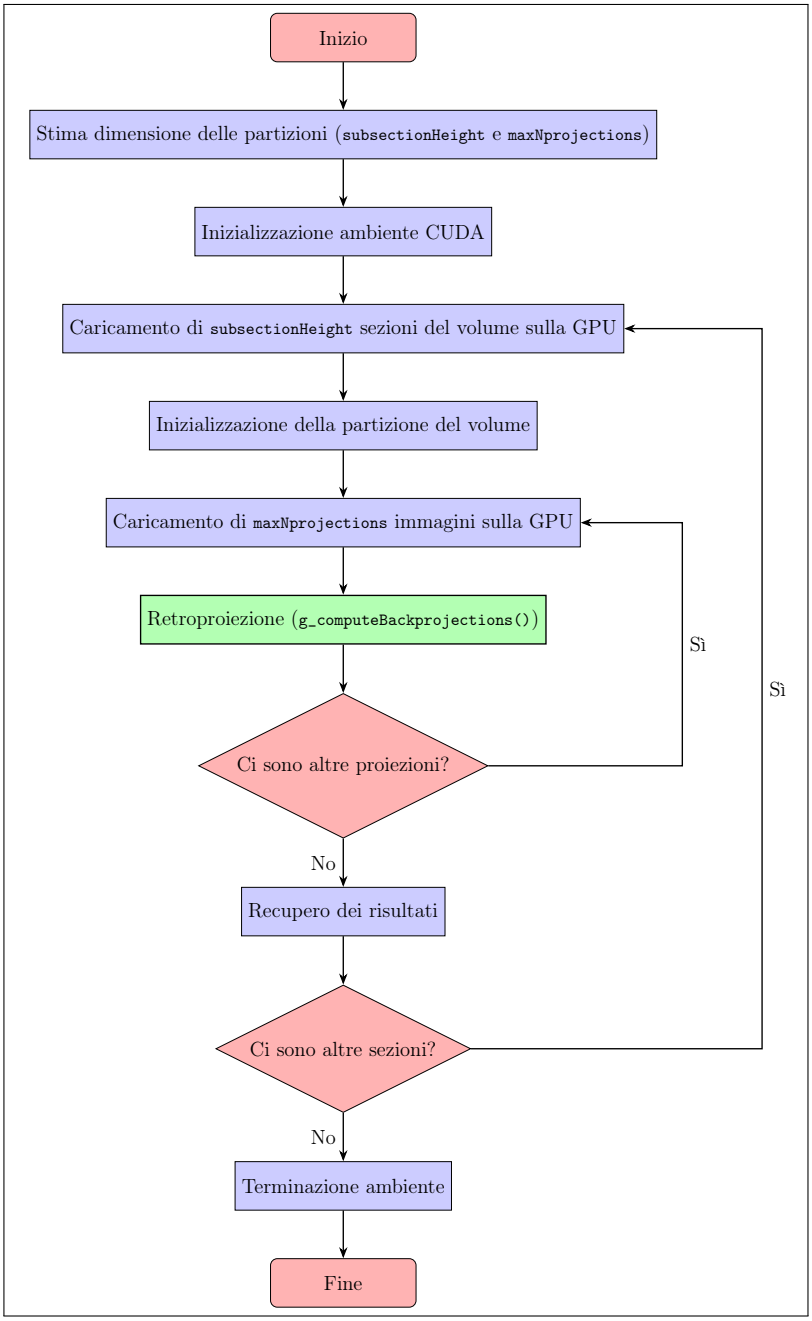


Figura 3.7: Diagramma di flusso della retroproiezione. È specificato il kernel `g_computeBackprojections()` eseguito in parallelo.

Simulazione dei raggi

La griglia che definisce il calcolo è tridimensionale ed è composta da blocchi bidimensionali: anche in questo caso, come descritto nell'applicazione dei filtri nella sottosezione 3.1.1,

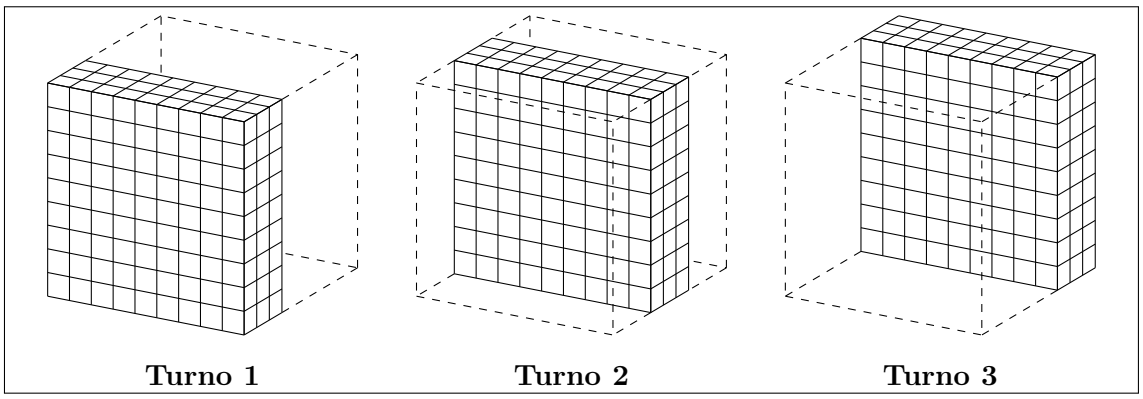


Figura 3.8: Rappresentazione del partizionamento del volume in 3 turni.

il terzo asse indica ai thread quale immagine devono retroproiettare. Ogni thread elabora dunque un singolo raggio, originato dalla sorgente in una delle posizioni, verso un determinato pixel.

Per migliorare la qualità delle immagini, è possibile simulare più raggi distribuiti uniformemente, come mostrato in Figura 3.9. Il numero di raggi da simulare lungo un lato corrisponde al numero di pixel per lato dell'immagine, moltiplicato per un fattore k (nel Listato 3.7 chiamato `RAY_M_FACTOR`). Se $k \neq 1$ i raggi non intersecano più il rilevatore nel punto centrale del pixel: il valore retroproiettato viene determinato dunque tramite interpolazione lineare dei pixel più vicini.

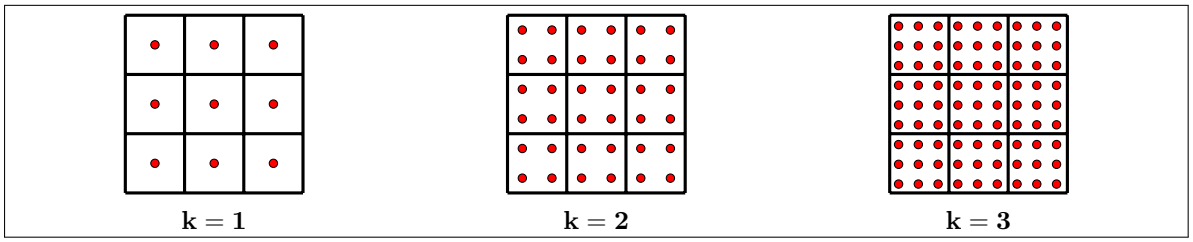


Figura 3.9: Distribuzione dei raggi sull'immagine al variare di k: i punti rappresentano l'intersezione tra i raggi e il rilevatore, dove una cella è una singola unità di rilevazione (un pixel).

Ogni thread opera quindi nel seguente modo:

1. prende le coordinate spaziali della sorgente relativa alla posizione da elaborare e del punto sull'immagine da retroproiettare;
2. definisce il raggio passante per quei due punti ed elabora il suo percorso radiologico;
3. diffonde nei voxel attraversati il valore di assorbimento `voxelAbsorptionValue`.

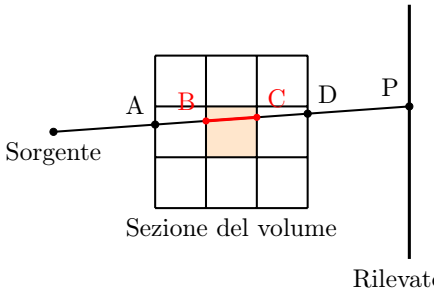
Il valore di assorbimento è determinato per ciascun voxel dal colore del punto intersecato sull'immagine, normalizzato (`pixelColor`) e pesato con la lunghezza del segmento di raggio interno al voxel (`segmentLength`): più il segmento è lungo, più il voxel ha contribuito all'assorbimento del raggio. Nel caso di retroproiezione di immagini filtrate con FDK il valore viene anche attenuato per il numero di immagini (`nAngles`): questo passaggio è in realtà superfluo, ma è stato tenuto per normalizzare i dati come fatto nell'Equazione 2.4.

Nel caso in cui le proiezioni non siano state filtrate è necessario tenere conto anche della geometria conica dei raggi: `segmentLength` viene normalizzato rispetto alla lunghezza

totale del raggio all'interno del volume (`radioPath.rayLenghtinVol`). Questo passaggio in FDK viene gestito dall'applicazione dei pesi e perciò non è necessario in questa fase.

Il valore di assorbimento ottenuto viene dunque sommato al valore del voxel tramite l'operazione atomica `atomicAddDouble` per evitare race condition, dal momento che più raggi possono attraversare lo stesso voxel.

Una rappresentazione schematica di una sezione del volume, che riporta anche le formule di aggiornamento di un voxel attraversato dal raggio qui descritte, è riportata in Figura 3.10. Per quanto riguarda l'individuazione delle coordinate spaziali del punto sorgente e dell'unità di rilevazione, nonché il calcolo del percorso radiologico, si è fatto affidamento alle funzioni già implementate dai precedenti progetti di tesi nei quali è stata definita la geometria [5, 10, 1].



Nel caso di immagine filtrata (FDK): $VoxelValue \mathrel{+}= \frac{pixelColor \times \overline{BC}}{nAngles \times k}$

Nel caso di immagine non filtrata: $VoxelValue \mathrel{+}= \frac{pixelColor \times \overline{BC}}{\overline{AD} \times k}$

Dove $\overline{AD}$=`radioPath.rayLenghtinVol`, $\overline{BC}$=`segmentLength`, k=`RAY_M_FACTOR`, $nAngles$ è il numero di immagini e $pixelColor$ il colore normalizzato dell'immagine nel punto P.

Figura 3.10: Calcolo del valore di assorbimento del voxel evidenziato attraversato dal raggio. La figura mostra solo una sezione 2D del volume.

```
1  static __global__ void g_computeBackprojections(d_paramPointers
       pointers, unsigned int startProj, unsigned int endProj, const double
        absMax, const double absMin) {
2      // idx retrival
3      [...]
4
5      // for each projection each pixel is backprojected
6      // each thread is assigned to one pixel of one projection
7      if (pixel_rowIdx <= height - 0.5 && pixel_colIdx <= width - 0.5) {
8          // Gets source and pixel positions
9          const point3D source = getSource(sinTable, cosTable, projIdx,
10             distanceObjectSource);
11         const point3D pixel = getPixel(sinTable, cosTable,
12             pixel_rowIdx, pixel_colIdx, projIdx, scanner);
13
14         // Gets pixel color
15         double pixelColor;
16         if (RAY_M_FACTOR == 1) {
17             // ray hits the center of the pixel
```

```
18              pixelColor = get_pixelColor(projection->pixels, width,
                    height, pixel_colIdx, pixel_rowIdx);
19          } else {
20              // Linear interpolation of the 4 adjacent
21              //   pixels if the ray isn't centered
22              [...]
23              pixelColor = interpolatedColor;
24          }
25
26          // creates the ray
27          const ray ray = {.source=source, .endPoint=pixel};
28
29          // compute radiological path of the ray inside the volume
30          path radioPath;
31          radioPath.intersections = intersectionBuffer;
32          computePath(ray, &radioPath);
33
34          // compute absorption value
35          const double normalizedPixelValue = (pixelColor - absMin) /
36              (absMax - absMin);
37          double adjustmentFactor;
38          switch (l_absType)
39          {
40          case raw_abs:
41              // Compensating the geometry
42              adjustmentFactor = (1 / (double)(radioPath.rayLenghtinVol *
                    RAY_M_FACTOR));
43              break;
44          case fdk_abs:
45              // The filtered images already compensate the geometry
46              adjustmentFactor = (1 / (double)(scanner->nAngles *
                    RAY_M_FACTOR));
47              break;
48          }
49
50          computeAbsorption(volume, normalizedPixelValue,
                adjustmentFactor, subsection, &radioPath, ray, l_absType);
51      }
52  }
53
54  static __device__ double  computeAbsorption(volume* volume, const
        double normalizedPixelValue, double adjustmentFactor, const range
        subsection[3], const path* radioPath) {
55      // For each voxel crossed, spread the absorption value
56      for (int i = 0; i < radioPath->nIntersections; i++) {
57          intersection it = radioPath->intersections[i];
58          // Voxels are referred to the total volume, we adapt the
                indexes to the subsection.
59          [...]
60          const int voxelIndex = voxelX + voxelZ * subZ + voxelY * subX *
                subZ;
61
62          const double voxelAbsorptionValue = normalizedPixelValue * it.
                segmentLength * adjustmentFactor;
63          // updates voxel absorption
```

```
64          atomicAddDouble(&(volume->coefficients[voxelIndex]),
                voxelAbsorptionValue);
65      }
66  }
```

Listato 3.7: Codice per la retroproiezione.

I voxel attraversati cambiano per ogni raggio e questo rende il calcolo non ottimale per CUDA: ogni thread esegue istruzioni diverse e si verificano accessi alla memoria disallineati all'interno del Warp. Inoltre, gli aggiornamenti ai voxel richiedono di essere gestiti tramite operazioni atomiche e non è possibile sfruttare la memoria shared a causa delle grosse dimensioni del volume. Le prestazioni che questa architettura offre permettono comunque di ottenere ottimi risultati; ci siamo dunque limitati a minimizzare il più possibile gli accessi alla memoria globale, favorendo invece quelli ai registri locali.

Capitolo 4

Analisi dei risultati

In questo capitolo analizziamo le immagini ottenute e le prestazioni dell'implementazione del retroproiettore con filtro FDK presentato in questa tesi: studiamo gli artefatti, il comportamento del programma in diverse situazioni e la sua efficienza.

Come caso di studio abbiamo utilizzato il **modello di Shepp-Logan**, inizialmente proposto in [12], nella sua estensione tridimensionale riportata in Figura 4.1. Questo modello è definito come una somma di ellissoidi aventi un certo coefficiente di assorbimento: rappresenta in maniera schematica una testa, ed è formulato in questo modo per permettere di calcolare rapidamente delle proiezioni esatte sulle quali testare gli algoritmi di ricostruzione [6]. Nonostante questa proprietà del modello di Shepp-Logan, abbiamo comunque scelto di utilizzare la simulazione discreta della proiezione [1], integrando il modello discreto nel programma di generazione dei volumi di [5], per ottenere le immagini di cui effettuare la retroproiezione.

Figura 4.1: Sezioni del modello di Shepp-Logan discretizzato e una sua visualizzazione in 3D.

4.1 Risultati visivi

In questa sezione osserviamo le immagini ottenute ignorando i tempi necessari per elaborarle. Mantenendo costanti le caratteristiche del tomografo simulato (numero di angoli, rapporto tra le dimensioni dei voxel e dei pixel e fattore k di raggi simulati) si ottengono risultati molto simili anche con volumi o immagini di più bassa risoluzione. I parametri del tomografo usato per realizzare queste immagini sono riportati nella Tabella 4.1.

Fase	Descrizione	Dimensioni/Quantità
Modello originale	Volume 3D iniziale	$1000 \times 1000 \times 1000$ voxel
Proiezioni generate e filtrate	Proiezioni 2D ottenute dal modello e filtrate	360 proiezioni 2352×2352 pixel
Retroproiezione	Ricostruzione 3D da proiezioni	$300 \times 300 \times 300$ voxel Fattore di raggi simulati $k = 5$

Tabella 4.1: Parametri per la generazione delle immagini.

Ci focalizzeremo prima sull'analisi delle proiezioni usate per la ricostruzione e degli effetti che i filtri hanno su di loro: osserveremo gli artefatti causati dal proiettore discreto e le caratteristiche del filtro FDK. In seguito, studieremo le sezioni del volume ricostruito, confrontandole con il modello originale e con la retroproiezione senza filtri, per osservare i miglioramenti apportati e gli artefatti causati dal retroproiettore "ray-driven", nonché dalla retroproiezione effettuata con un numero ridotto di immagini.

I programmi operano al loro interno tramite file binari per mantenere massima precisione dei dati. Per permettere la visualizzazione dei risultati, le immagini e i volumi sono convertibili in altri formati file noti:

- Le proiezioni sono normalizzate nel range [0-255] e convertite in un file `.pgm` [13] in scala di grigi. È possibile usare comuni programmi di elaborazione delle immagini come 'Gimp' [14] per visualizzarle, ma a causa delle grandi dimensioni potrebbe non essere possibile aprire il file. Abbiamo dunque creato un piccolo programma che ci permettesse di suddividere il file delle proiezioni in più file (uno per immagine), rendendone più semplice la visualizzazione e la conversione in altri formati.
- I volumi sono salvati in file `.nrrd` [15] visualizzabili tramite 'ImageJ' [16]: un programma di imaging molto usato in ambito medico. Per la visualizzazione tridimensionale è stato usato 'ITK viewer' [17]: una web app che permette di osservare in 3D il contenuto dei file `.nrrd` di piccole dimensioni.

4.1.1 Proiezioni filtrate

Le proiezioni del modello di Shepp-Logan sono state ottenute utilizzando il proiettore [1] su un volume di 1000 voxel per lato e un rilevatore di 2352×2352 pixel. È possibile osservare un piccolo sottoinsieme delle 360 immagini usate per la retroproiezione nella Figura 4.2: zone più luminose corrispondono a porzioni del volume più dense.

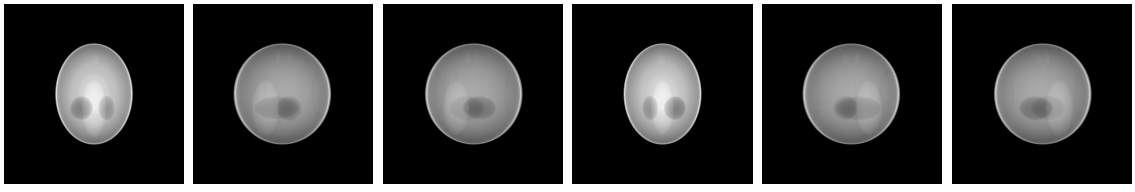


Figura 4.2: Sei proiezioni del modello effettuate da angolazioni equispaziate di 60 gradi.

Applicando i filtri di FDK alle proiezioni sopra mostrate, otteniamo le nuove immagini riportate in Figura 4.3. Come previsto dall'applicazione del ramp filter, è possibile notare come le uniche informazioni riportate siano i dettagli più fini, codificati con colori più scuri e chiari rispetto al grigio di sfondo.

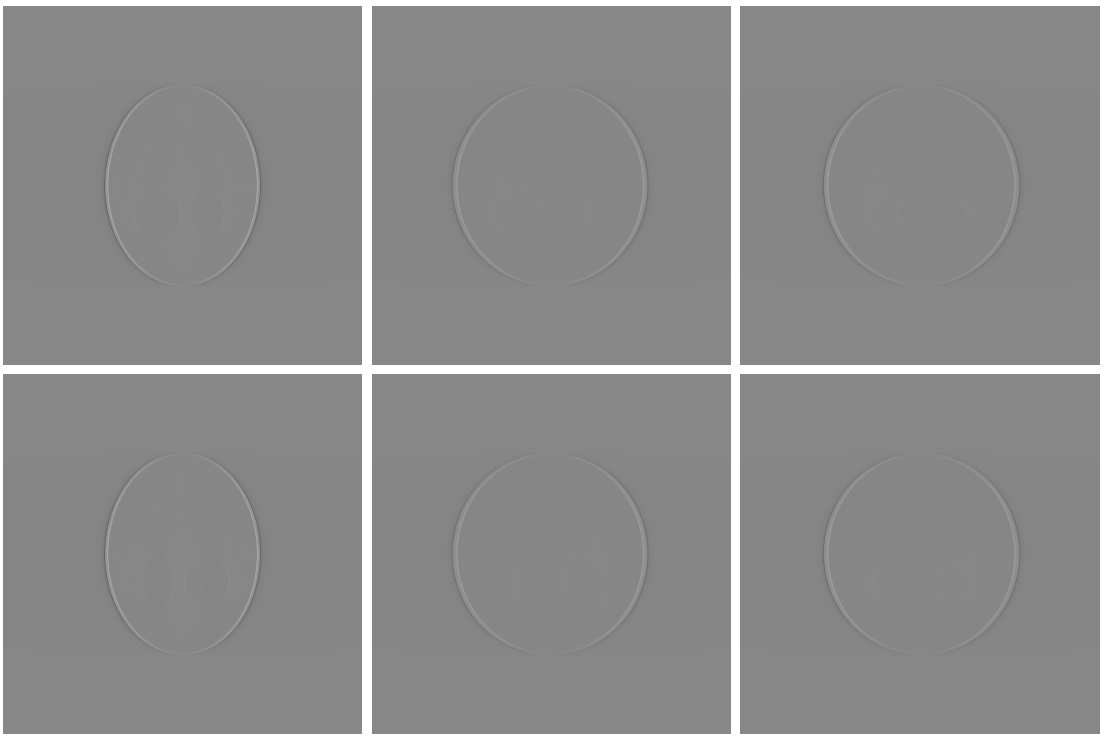


Figura 4.3: Le proiezioni di Figura 4.2 filtrate con i filtri di FDK.

Dal momento che vengono accentuati i dettagli, diventano più evidenti anche i difetti della proiezione dovuti alla rappresentazione discreta del modello. È infatti possibile distinguere i singoli voxel che compongono gli ellissoidi, con la comparsa di "*gradini*" che definiscono la sagoma del volume. Sempre a causa della discretizzazione del modello, è possibile notare anche delle fasce scure nella proiezione originale. Queste risultano meno visibili nell'immagine filtrata, ma avranno effetto sull'immagine ricostruita (riportata in Figura 4.5) come piccoli tratti scuri sul bordo esterno.

4.1.2 Volumi ricostruiti

Ricostruzioni

Osservando la ricostruzione in Figura 4.4 effettuata senza i filtri, possiamo distinguere le caratteristiche macroscopiche del volume. Queste immagini risultano molto sfocate e di poca praticità per distinguere propriamente i diversi elementi e tessuti che compongono il corpo: tutti difetti noti e descritti nella sezione 2.1.

Le immagini del volume ricostruito tramite FDK, riportate in Figura 4.5, invece permettono di distinguere nitidamente gli elementi che compongono il volume: effettuando un confronto diretto con il modello originale, è possibile verificare la corretta dimensione, forma e valore di densità di ciascun ellissoide.

Le immagini ricostruite sono state ottenute usando le proiezioni del paragrafo precedente, ma retroproiettate su un volume di più bassa risoluzione (300 voxel per lato) e con un moltiplicatore di raggi simulati pari a 5 (2352×5 raggi retroproiettati per ciascun lato dell'immagine).

Figura 4.4: Ricostruzione del volume con le immagini non filtrate.

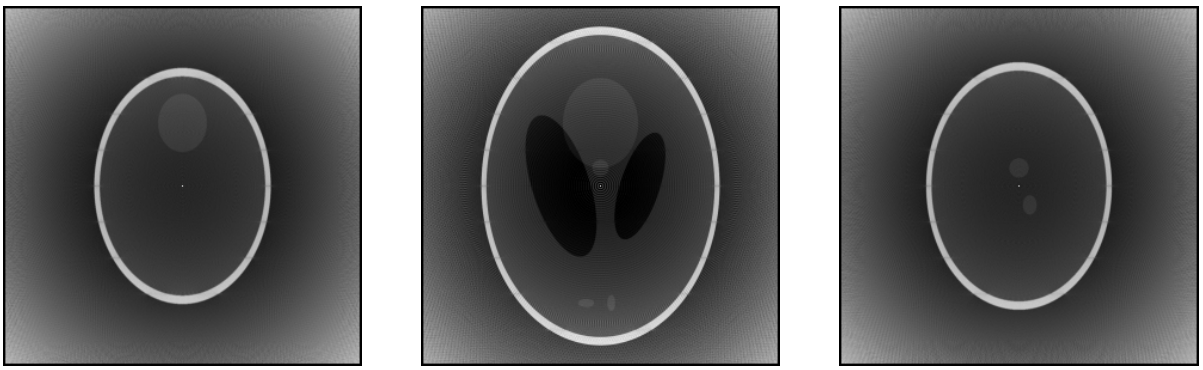


Figura 4.5: Ricostruzione del volume tramite FDK.

Artefatti della retroproiezione ray driven

A causa della natura "ray driven" del retroproiettore, e per via del numero finito di raggi simulati, è possibile che alcuni voxel vengano attraversati di meno e di conseguenza risultino in pixel più scuri nella ricostruzione. Questo comportamento porta alla formazione di artefatti concentrici radiali visibili in Figura 4.6 che, per via dell'addensamento dei raggi, aumentano di intensità nelle sezioni centrali del volume. Questi sono difetti noti di questa categoria di retroproiettori e sono riportati anche in [10]. Inoltre, i voxel della colonna centrale sono attraversati da raggi in ogni angolazione: questo provoca un addensamento di valori su di essi, che si traduce in un pixel centrale bianco in ogni sezione del volume.

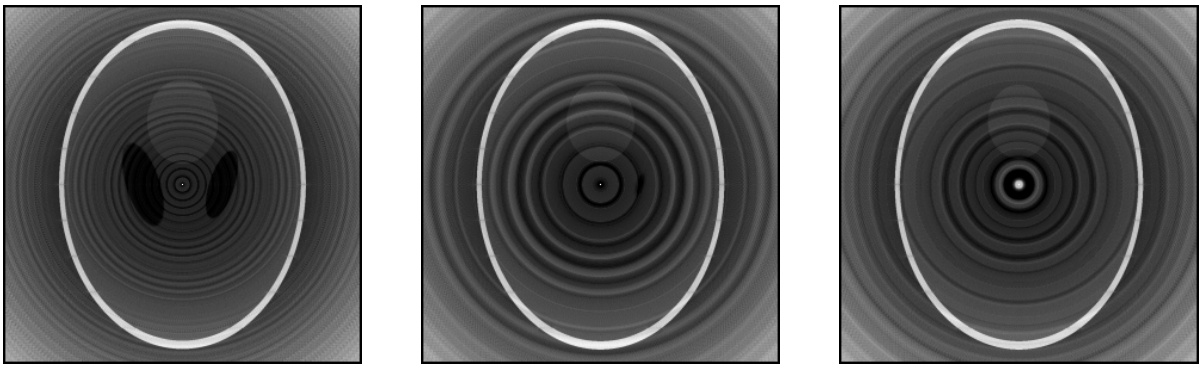


Figura 4.6: Artefatti concentrici della retroproiezione ray driven in sezioni centrali del volume.

Questi difetti non dovrebbero causare problemi per l'utilizzo di queste ricostruzioni, generate anche con meno dati, in algoritmi che utilizzano il proiettore e il retroproiet-

tore iterativamente: entrambi i programmi simulano gli stessi raggi e di conseguenza analizzano il contributo degli stessi voxel.

I difetti qui riportati non sono presenti in altri esempi di ricostruzione FDK in quanto l'Equazione 2.4, come già descritto nella sezione 2.3, descrive la formula di ricostruzione di un singolo voxel: per effettuare la ricostruzione del volume, FDK analizza dunque equamente tutti i voxel. Una conseguenza negativa dei filtri sul nostro retroproiettore è che, accentuando i dettagli, il ramp filter tende ad accentuare anche gli artefatti: nella retroproiezione di immagini non filtrate, come visibile in Figura 4.7, gli artefatti studiati sono presenti, ma meno evidenti.

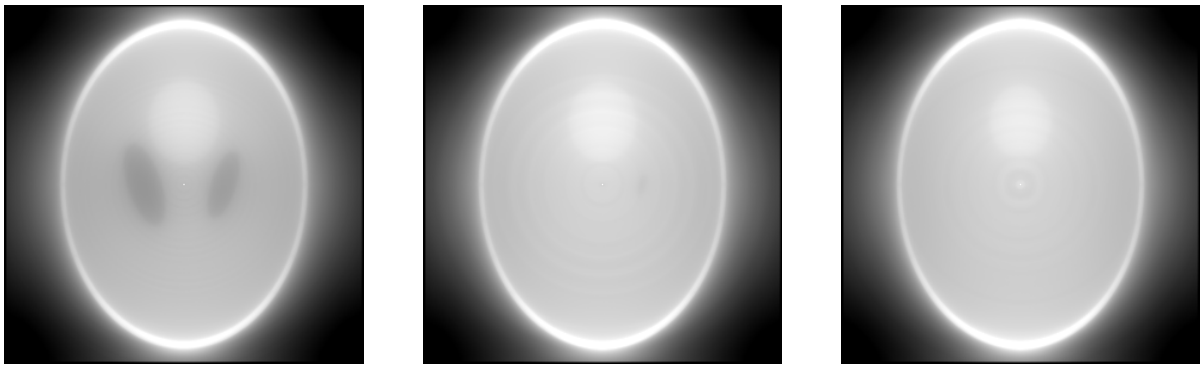
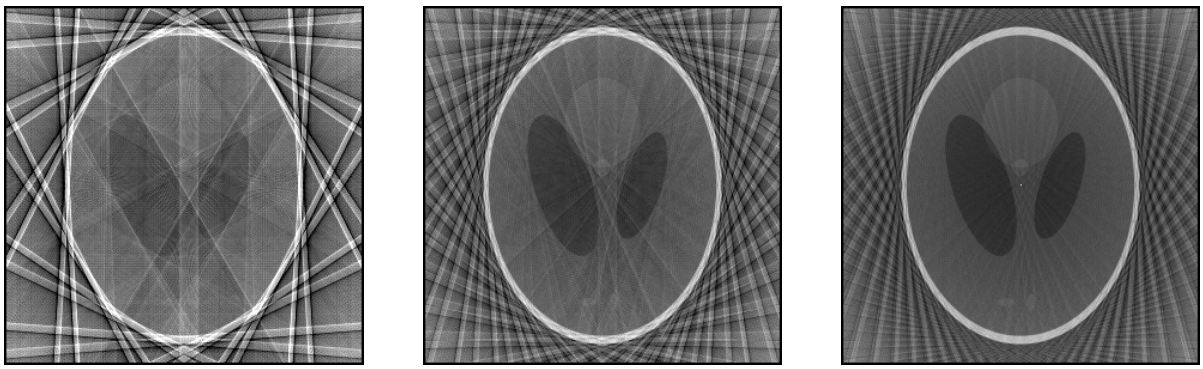


Figura 4.7: Artefatti concentrici della retroproiezione ray driven di immagini non filtrate.

Artefatti della retroproiezione con poche immagini

In Figura 4.8 è possibile osservare gli artefatti generati da una retroproiezione effettuata con poche immagini: queste sono acquisite comunque lungo una traiettoria circolare, ma con passo angolare ampio. A differenza degli artefatti precedenti, questi si presentano sotto forma di striature rettilinee, orientate lungo le direzioni delle proiezioni usate.



12 immagini **30 immagini** **60 immagini**

Figura 4.8: Artefatti della retroproiezione con poche immagini.

4.2 Analisi delle prestazioni

In questa sezione del capitolo studiamo il comportamento del programma in termini delle sue prestazioni. I dati relativi alla macchina e ai compilatori usati sono riportati nella Tabella 4.2. I tempi rilevati corrispondono alla differenza tra il tempo di fine e di

inizio del calcolo: l'unità di misura usata è il "*Wall Clock Time*", ovvero il tempo umano di attesa.

Categoria	**Dettagli**	**Note**
Processore	Intel(R) Xeon(R) CPU E5-2603 v4 @ 1.70GHz (6 core, 6 thread)	Cache 15 MB Intel® Smart Cache
GPU	NVIDIA GeForce GTX 1070 (8GB GDDR5, 1920 CUDA cores, CC 6.1)	Clock: 1.80GHz; Mem: 4004MHz, 256-bit; Driver/Runtime: 12.2 / 8.0
RAM	64GB	
Compilatore (C/C++)	GCC 9.4.0	Flag: `-std=c99 -Wall -Wpedantic -Werror -O2 -D_XOPEN_SOURCE=600 -DNDEBUG -DRELEASE`
Compilatore (CUDA)	nvcc 12.2.140	Flag: `-Xcompiler "-Wall -Werror" -Wno-deprecated-gpu-targets -DRELEASE -DNDEBUG -DNO_CUDA_CHECK_ERROR -x cu -dc`
Sistema operativo	Ubuntu 20.04.6 LTS	

Tabella 4.2: Dettagli del sistema e ambienti di compilazione.

4.2.1 Il Throughput

Dal momento che in CUDA non è possibile effettuare il calcolo con un minor numero di thread senza alterare le dimensioni del problema, è impossibile utilizzare i concetti di speedup ed efficienza tipicamente usati per la valutazione dei programmi paralleli. Usiamo dunque come metrica di valutazione il *Throughput*:

$$Throughput = \frac{numero\ di\ operazioni\ totali}{tempo\ di\ esecuzione} \tag{4.1}$$

dove il numero di operazioni svolte viene stimato conoscendo la dimensione dei dati e la complessità computazionale degli algoritmi implementati.

L'operazione di filtraggio esegue un calcolo per ogni pixel: considerando un'immagine di dimensioni $h \times w$, l'applicazione dei pesi ha complessità $O(hw)$, mentre il ramp filter, dal momento che legge una maschera lunga w per ogni elemento dell'immagine, avrà complessità $O(hw^2)$. Nel complesso, l'intera fase di filtraggio ha dunque come limite superiore:

$$O(hw^2) \tag{4.2}$$

Nel caso invece della retroproiezione, possiamo prendere come riferimento la complessità della proiezione individuata in [5]: i due algoritmi eseguono le stesse operazioni, solo che nel nostro caso l'output è il volume. Dobbiamo però tenere conto del fattore k definito per simulare più raggi: questo non è stato implementato nel proiettore e ne aumenta la complessità di un fattore k^2. Nel caso di un rilevatore quadrato, infatti, vengono simulati $nPixel \times k$ raggi per lato. La complessità computazionale della retroproiezione diventa quindi:

$$O(N_\theta \times (N_{pixel} \times k)^2 \times (nVoxel_X + nVoxel_Y + nVoxel_Z)) \tag{4.3}$$

con N_θ il numero di immagini retroproiettate, N_{pixel} il numero di pixel per lato dell'immagine e $nVoxel$ il numero di voxel per lato del volume.

4.2.2 Prestazioni del filtro

Per quanto riguarda le prestazioni del filtraggio, abbiamo effettuato un confronto tra la versione che sfrutta la memoria shared e quella che si appoggia unicamente alla memoria globale della GPU.

Potendo controllare l'ambiente ed essendo il resto del programma identico tra le due versioni, abbiamo escluso dalle misure i tempi necessari al caricamento delle immagini e abbiamo considerato soltanto i tempi di esecuzione dei kernel.

Sono state analizzate le prestazioni in 8 configurazioni di dimensioni crescenti: il numero di immagini filtrate è sempre 360, ma aumenta la loro risoluzione. Il grafico in Figura 4.9 mette a confronto il throughput delle due implementazioni nei diversi casi analizzati e mostra come, sfruttare la memoria shared, comporti una notevole riduzione dei tempi di calcolo.

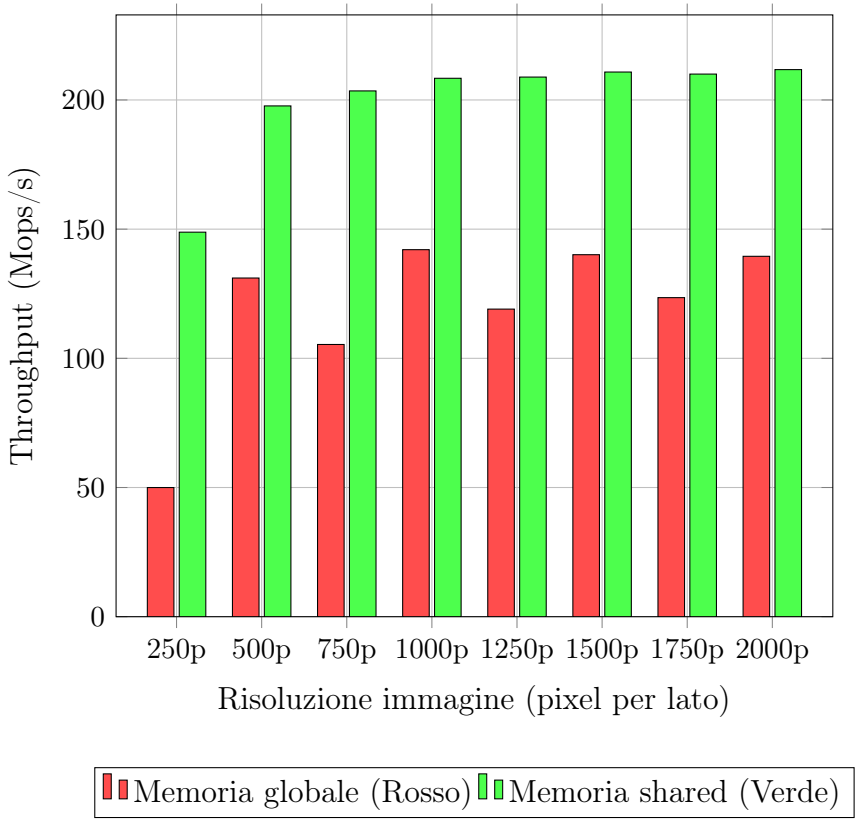


Figura 4.9: Confronto throughput tra il filtraggio solo su memoria globale e con memoria shared, su immagini a risoluzione crescente. Valori più alti indicano prestazioni migliori.

4.2.3 Prestazioni della retroproiezione

La retroproiezione, a causa dell'implementazione dell'algoritmo di Siddon usata e ad altre strutture necessarie per il calcolo del percorso radiologico, richiede tanta memoria nei registri dei thread. Sarebbe ottimale usare sempre blocchi CUDA di massima dimensione (1024 thread per blocco) ma la GPU usata per effettuare queste misure non ha sufficiente memoria e per questo motivo abbiamo usato blocchi di dimensione 16×16.

Mostriamo prima un confronto tra l'implementazione CUDA e OpenMP della retroproiezione, per poi analizzare come il fattore k dei raggi simulati influenzi le prestazioni del

programma. In queste misure consideriamo il tempo totale di esecuzione, anche quello passato in lettura dei file: dal momento che l'asincronicità del calcolo su GPU permette di guadagnare tempo anche in questa fase, considerare anche il tempo di lettura ci permette di confrontare interamente le due versioni. Per coerenza, anche le misure relative al fattore k considerano gli stessi tempi.

Cuda e OpenMP a confronto

Dal momento che la versione su CPU della retroproiezione non ha implementato il moltiplicatore di raggi simulati, il fattore k non viene usato ed è posto ad uno.

Il primo confronto, riportato in Figura 4.10, effettua la retroproiezione di immagini con risoluzione fissa (500×500 pixel) su un volume di 210 voxel per lato, usando un numero crescente di immagini.

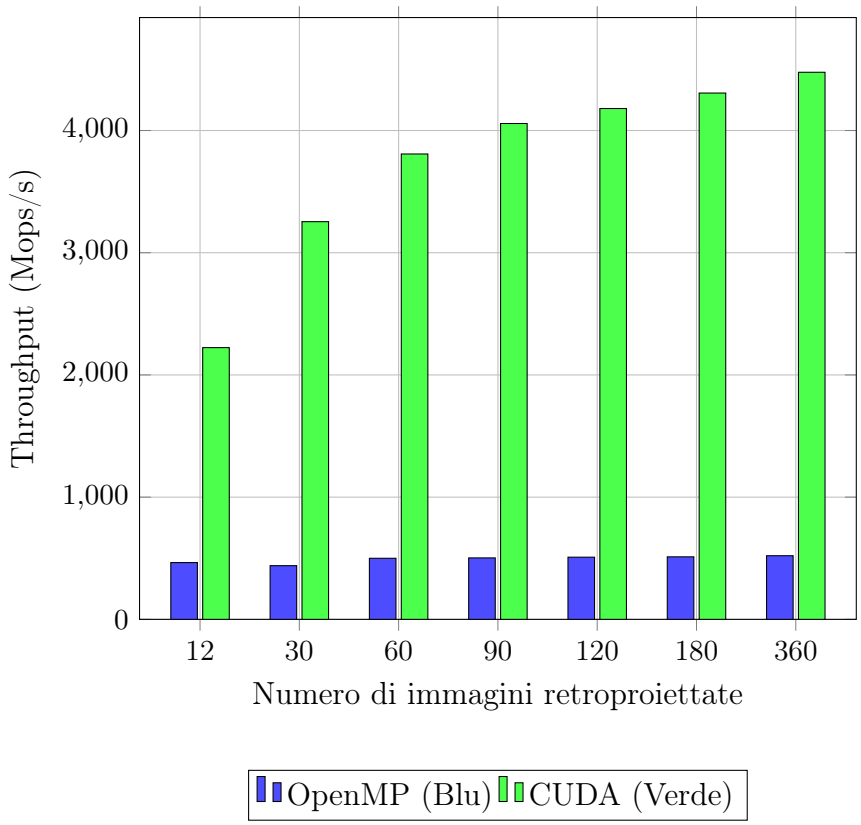


Figura 4.10: Confronto throughput tra retroproiettore Cuda e OpenMP per immagini di 500×500 pixel su un volume di 210 voxel per lato, con numero crescente di immagini retroproiettate. Valori più alti indicano prestazioni migliori.

Il secondo confronto invece, riportato in Figura 4.11, studia l'efficienza allo scalare della risoluzione delle immagini. Il numero di proiezioni usate è fisso a 12, ma immagini di maggiore qualità vengono retroproiettate su volumi con più voxel, nelle configurazioni riportate nella Tabella 4.3.

Configurazione	Numero di pixel per lato delle immagini	Numero di voxel per lato
1	250p	105v
2	500p	210v
3	750p	315v
4	1000p	420v
5	1250p	525v
6	1500p	630v
7	1750p	735v
8	2000p	840v
9	2352p	1000v

Tabella 4.3: Configurazioni con numero di pixel e voxel per lato

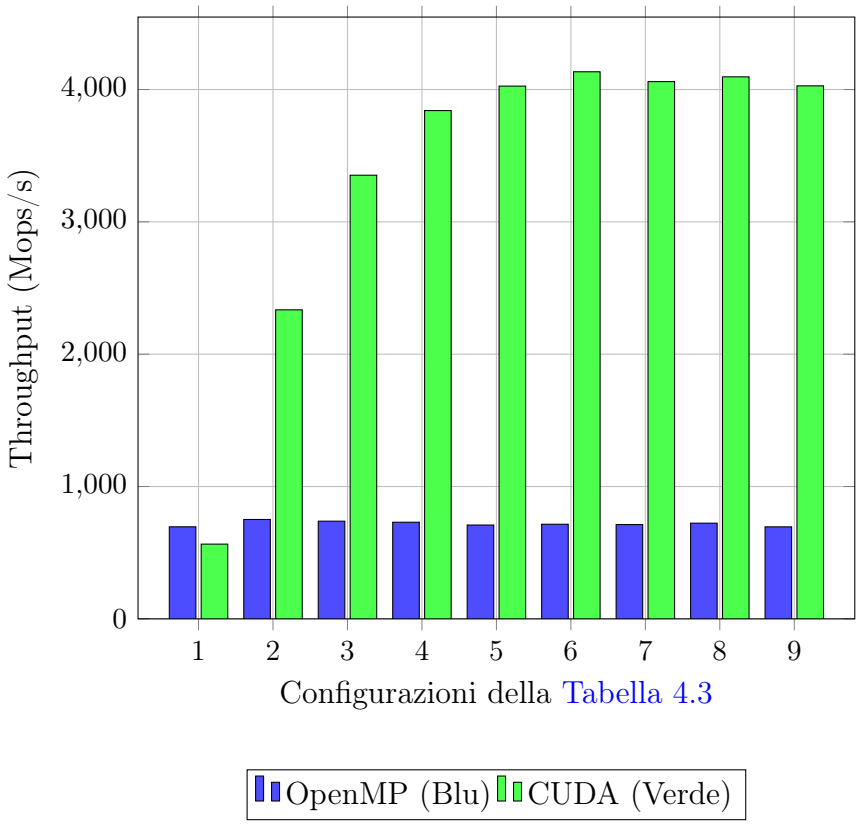


Figura 4.11: Confronto throughput tra retroproiettore Cuda e OpenMP per la retroproiezione di un numero fisso di immagini a risoluzione crescente, su volumi con un crescente numero di voxel (configurazioni della Tabella 4.3). Valori più alti indicano prestazioni migliori.

Retroproiezione CUDA con molteplici raggi

Per l'analisi dell'impatto del fattore k sulle prestazioni del programma, abbiamo effettuato la retroproiezione di 360 immagini di 500×500 pixel su volumi a risoluzione costante di 210 voxel per lato. Le misure sono state effettuate prendendo di volta in volta un valore di k crescente, in un range da 1 a 6. I risultati, riportati in Figura 4.12, evidenziano come, per quanto ci si potesse aspettare una riduzione delle prestazioni a causa del numero maggiore di race condition da gestire, questa sia limitata e non comporti grossi cambiamenti all'efficienza del programma.

I tempi di calcolo, dunque, crescono quadraticamente all'aumentare di k, come descritto dalla complessità del problema nell'Equazione 4.3. Di conseguenza, nel caso si effettui una retroproiezione di immagini ad alta risoluzione su un volume con tanti voxel, una

leggera variazione del fattore k comporta una rapida crescita dei tempi di calcolo.

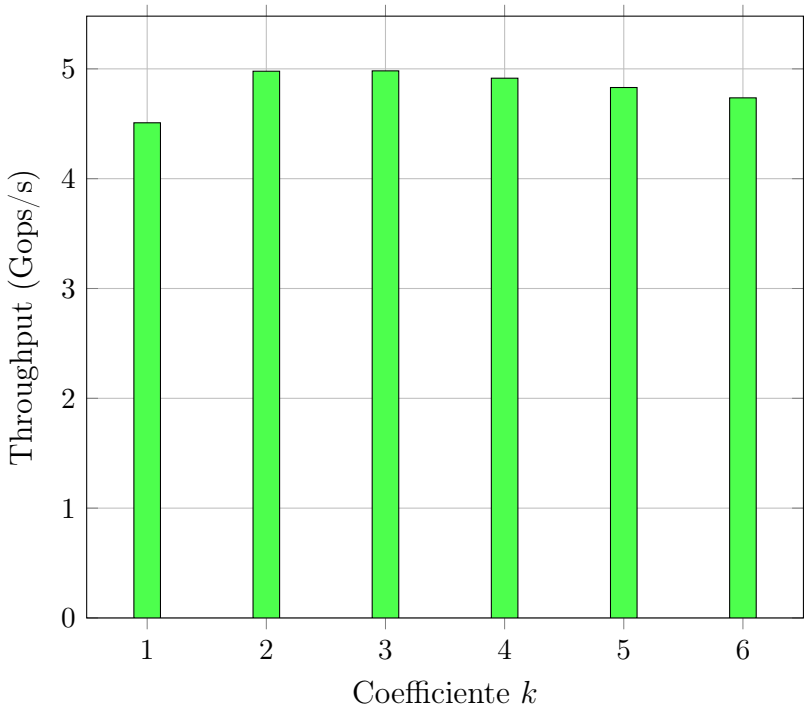


Figura 4.12: Throughput del retroproiettore CUDA con un fattore k di raggi simulati crescente. Valori più alti indicano prestazioni migliori.

Capitolo 5

Conclusioni e sviluppi futuri

In conclusione, gli obiettivi posti sono stati raggiunti: abbiamo ottenuto un programma di ricostruzione rapido ed efficiente, migliore di quello realizzato con OpenMP, che includesse i filtri dell'algoritmo FDK per una generazione di immagini più nitide e utili in ambito diagnostico. Abbiamo osservato come il calcolo parallelo su GPU permetta una notevole riduzione dei tempi di calcolo rispetto al calcolo su CPU, ma che alcune limitazioni, come la gestione della memoria, ci impediscano di trarre sempre il meglio da questa architettura.

Le immagini di ricostruzione ottenute sono lontane dall'essere perfette, e sono tante le possibili vie di sviluppo che potrebbero migliorare i risultati. Segue qualche spunto su possibili sviluppi futuri utili a migliorare la qualità delle ricostruzioni:

- Per eliminare gli artefatti causati dalla retroproiezione ray driven, è possibile implementare la retroproiezione dei singoli voxel in parallelo, come descritto dal retroproiettore FDK nella sezione 2.3: per realizzarlo è sufficiente modificare la chiamata del kernel di retroproiezione e il kernel stesso.
- Per attenuare invece l'effetto degli artefatti, ma senza alterare la natura del retroproiettore, sarebbe possibile tentare di applicare dei filtri di regolarizzazione. Questi riducono il rumore delle immagini, a costo però della loro nitidezza.
- Sarebbe possibile importare nel proiettore il fattore k dei raggi simulati o utilizzare metodologie differenti, come la "*distance driven*" o la "*separable trapezoid footprint*" [7], in modo da diminuire gli artefatti nelle proiezioni.
- Si potrebbe implementare un programma di proiezione esatta per il modello di Shepp-Logan, così da valutare quanto gli artefatti di ricostruzione dipendano dalle proiezioni discrete.

Per quanto riguarda invece possibili sviluppi atti a ridurre ulteriormente i tempi di calcolo, sarebbe possibile implementare algoritmi di elaborazione del percorso radiologico più efficienti rispetto a quello di Siddon (ad esempio [18]). Altrimenti, sarebbe interessante esplorare la programmazione *multi-GPU*: il programma potrebbe elaborare differenti partizioni del volume su diverse GPU in parallelo, come proposto in [19].

Bibliografia

[1] Enrico Marchionni. «Implementazione CUDA di un algoritmo di proiezione tomografica». URL: https://amslaurea.unibo.it/id/eprint/34502/.

[2] L. A. Feldkamp, L. C. Davis e J. W. Kress. «Practical cone-beam algorithm». In: *J. Opt. Soc. Am. A* 1.6 (giu. 1984), pp. 612–619. DOI: 10.1364/JOSAA.1.000612. URL: https://opg.optica.org/josaa/abstract.cfm?URI=josaa-1-6-612.

[3] Thorsten M. Buzug. «Computed Tomography». English. In: *Springer Handbook of Medical Technology*. A cura di Rüdiger Kramme, Klaus-Peter Hoffmann e Robert S. Pozos. Springer Handbooks. Springer Berlin Heidelberg, 2011, pp. 311–342. ISBN: 978-3-540-74657-7. DOI: 10.1007/978-3-540-74658-4_16.

[4] Robert L. Siddon. «Fast calculation of the exact radiological path for a three-dimensional CT array». en. In: *Medical Physics* 12.2 (mar. 1985), pp. 252–255. ISSN: 0094-2405, 2473-4209. DOI: 10.1118/1.595715. URL: https://aapm.onlinelibrary.wiley.com/doi/10.1118/1.595715.

[5] Lorenzo Colletta. «Implementazione Parallela di un Algoritmo di Proiezione Tomografica». URL: https://amslaurea.unibo.it/id/eprint/33715/.

[6] Avinash C. Kak e Malcolm Slaney. *Principles of Computerized Tomographic Imaging*. Society for Industrial e Applied Mathematics, 2001. DOI: 10.1137/1.9780898719277. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898719277. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898719277.

[7] Elena Morotti e Elena Loli Piccolomini. «Sparse Regularized CT Reconstruction: An Optimization Perspective». eng. In: *Handbook of Mathematical Models and Algorithms in Computer Vision and Imaging*. GBR, 2022. ISBN: 9783030030094. DOI: 10.1007/978-3-030-03009-4_123-1. URL: https://cris.unibo.it/handle/11585/917488.

[8] Gordon E. Moore. «Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.» In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 33–35. DOI: 10.1109/N-SSC.2006.4785860.

[9] 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.

[10] Emanuele Borghini. «Parallelizzazione di un algoritmo Ray-Driven per la Ricostruzione Tomografica con OpenMP in C». URL: https://amslaurea.unibo.it/id/eprint/33729/.

[11] Hui Miao et al. «Implementation of FDK Reconstruction Algorithm in Cone-Beam CT Based on the 3D Shepp-Logan Model». In: *2009 2nd International Conference*

on Biomedical Engineering and Informatics. 2009, pp. 1–5. DOI: 10.1109/BMEI.2009.5304987.

[12] L. A. Shepp e B. F. Logan. «The Fourier reconstruction of a head section». In: *IEEE Transactions on Nuclear Science* 21.3 (1974), pp. 21–43. DOI: 10.1109/TNS.1974.6499235.

[13] *PGM Format Specification*. URL: https://netpbm.sourceforge.net/doc/pgm.html (visitato il giorno 22/06/2025).

[14] *GIMP*. en. URL: https://www.gimp.org/ (visitato il giorno 22/06/2025).

[15] *Teem: nrrd: Definition of NRRD File Format*. URL: https://teem.sourceforge.net/nrrd/format.html (visitato il giorno 22/06/2025).

[16] *ImageJ Wiki*. URL: https://imagej.github.io/ (visitato il giorno 22/06/2025).

[17] Kitware Inc. *Overview*. en. Feb. 2025. URL: https://kitware.github.io/itk-vtk-viewer/docs/index.html (visitato il giorno 22/06/2025).

[18] Hao Gao. «Fast parallel algorithms for the x-ray transform and its adjoint». en. In: *Medical Physics* 39.11 (nov. 2012), pp. 7110–7120. ISSN: 0094-2405, 2473-4209. DOI: 10.1118/1.4761867. URL: https://aapm.onlinelibrary.wiley.com/doi/10.1118/1.4761867 (visitato il giorno 22/06/2025).

[19] Shunli Zhang, Guohua Geng e Jian Zhao. «Fast parallel image reconstruction for cone-beam FDK algorithm». en. In: *Concurrency and Computation: Practice and Experience* 31.10 (mag. 2019), e4697. ISSN: 1532-0626, 1532-0634. DOI: 10.1002/cpe.4697. URL: https://onlinelibrary.wiley.com/doi/10.1002/cpe.4697 (visitato il giorno 22/06/2025).

Ringraziamenti

Inizio ringraziando il Prof. Moreno Marzolla e la Prof.ssa Elena Loli Piccolomini per avermi guidato e permesso di lavorare su un argomento così interessante. Tra i vari professori che ho avuto, vorrei anche ringraziare la mia prof.ssa di matematica delle superiori, per avermi fornito gli strumenti necessari per affrontare questo percorso.

Ringrazio i miei genitori e i miei nonni che, grazie al loro lavoro e ai loro sacrifici, mi permettono di studiare e perseguire le mie passioni. Un grazie anche ai miei fratelli, Letizia e Filippo, per essere sempre presenti e disponibili per me.

Ci tengo a ringraziare i miei amici di casa per colorare la mia vita, quelli della Scuola Ortogonale e Boost per il supporto e l'ispirazione che mi danno, e le persone che ho conosciuto in questo campus e che mi hanno arricchito nel profondo.

Per concludere, un ringraziamento simbolico va a tutti gli artisti che in questi tre anni hanno fatto da colonna sonora ai miei viaggi sul 149 e a tutte le ore passate davanti al computer.