UNIVERSITÀ DEGLI STUDI DI NAPOLI PARTHENOPE

Dipartimento di Scienze e Tecnologie

Corso di Laurea Triennale in Informatica

Tesi di Laurea

RIMOZIONE DEL RUMORE DA IMMAGINI PESATE IN DIFFUSIONE UTILIZZANDO IL METODO PCA LOCALE OVERCOMPLETE IN AMBIENTE GPU-CUDA

DIFFUSION WEIGHTED IMAGE DENOISING USING OVERCOMPLETE LOCAL PCA METHOD IN GPU-CUDA ENVIRONMENT

Relatore:
PROF. LIVIA MARCELLINO

Laureando:
IVAN OSATO 0124/250

ANNO ACCADEMICO 2013-2014

A mio padre che ora mi guarda da lassù, dedico con tutto il mio cuore questa tesi, che ho scritto in uno dei periodi più difficili della mia vita. In fondo se oggi sono qui a scrivere queste righe è anche per merito suo, perché nonostante tutte le difficoltà che ci hanno accompagnato nel corso degli anni, non mi ha mai fatto mancare nulla. Grazie di tutto papà!

A mia madre e ai miei fratelli, che mi hanno supportato in questi anni e a cui devo tanto se oggi sono quello che sono. Non potrò mai ringraziarli abbastanza, ma farò del mio meglio per ripagarli di tutti gli sforzi e i sacrifici che hanno fatto per me.

Abstract

Nel corso degli ultimi anni, è fortemente aumentata la necessità di processare grandi quantità di dati in maniera veloce e con tecniche sempre più sofisticate allo scopo di risparmiare tempo e denaro. Il calcolo ad elevate prestazioni (dall'inglese High Performance Computing, HPC) viene incontro a queste esigenze attraverso sistemi di elaborazione in grado di fornire delle prestazioni molto elevate, come ad esempio l'HPC su GPU (Graphics Processing Unit), destinato a ricoprire un ruolo da protagonista nel futuro prossimo. Lo scopo di questo lavoro di tesi è stato quello di evidenziare i vantaggi del computing accelerato dalle GPU nel campo dell'elaborazione delle immagini dove è noto che la maggioranza degli algoritmi ha una complessità di tempo elevata. In particolare, è stato preso in esame un metodo di enhancement che utilizza una tecnica di PCA (Principal Components Analysis) locale overcomplete (la stima dei voxel è ottenuta utilizzando patches sovrapposte) per rimuovere il rumore Rician da immagini di risonanza magnetica pesate in diffusione (Diffusion Weighted Image, DWI). A partire da questo metodo, quindi, è stata progettata e sviluppata una soluzione software parallela in ambiente GP-GPU in modo da ridurne drasticamente il tempo di esecuzione. L'obiettivo è stato raggiunto utilizzando CUDA che è l'architettura di elaborazione in parallello di NVIDIA e permette di avere dei netti aumenti di prestazioni del computing sfruttando la potenza di calcolo dell'unità di elaborazione grafica. Attraverso un confronto accurato tra le due versioni dell'algoritmo è stato possibile dimostrare che a fronte di una strategia parallela di riscrittura del codice ben progettata, si può far girare l'algoritmo a prestazioni decisamente più elevate, anche su di una scheda grafica a basso costo, risparmiando quindi tempo e denaro. I risultati sperimentali effettuati evidenziano un guadagno in termini di GFlops, addirittura pari al 91,6% e ad una riduzione del tempo di esecuzione (espresso in secondi) pari al 75%.

Over the last years, has greatly increased the need to process large amounts of data quickly and with ever more sophisticated techniques in order to save time and money. High Performance Computing meets these needs through processing systems able to provide very high performance, such as HPC on GPU (Graphics Processing Unit), destined to play a starring role in the near future. The purpose of this thesis was to highlight the advantages of GPU accelerated computing in the field of image processing, where it is known that algorithms have a high complexity time. In particular, it has been considered an enhancement technique that uses OLPCA method (Overcomplete Local Principal Components Analysis) to denoise Diffusion Weighted Images (DWI) from Rician noise. Using this method, has been developed a parallel software solution on GP-GPU environment, to reduce execution time. The goal was achieved using NVIDIA CUDA which is a parallel processing architecture able to increase computing performance using the power of graphics processing. Comparing the two versions of the algorithm, it was possible to demonstrate that with a parallel strategy well designed, you can get very high performance even on a low-cost graphics card, saving time and money. Experimental results show a performance increase in terms of GFlops equal to 91.6% and an execution time reduction of 75%.

Indice

Introduzione

L'imaging a risonanza magnetica (**Magnetic Resonance Imaging**, **MRI**) è una tecnica utilizzata in campo medico che consiste nella generazione di immagini ad alta definizione dell'interno del corpo umano per scopi medico-diagnostici. Tale tecnica è basata sul principio della risonanza magnetica nucleare (**NMR**) ma non è dannosa per il paziente, dal momento che quest'ultimo non è sottoposto a radiazioni ionizzanti come invece avviene nelle tecniche che fanno uso di isotopi radioattivi o di raggi X. Inizialmente l'MRI era una tecnica di imaging tomografico in grado di produrre una immagine del segnale di risonanza magnetica nucleare in una sottile fetta del corpo umano, poi successivamente si è evoluta in una tecnica di imaging volumetrico in grado di dare una rappresentazione in 3-D di una qualunque parte del corpo. Attraverso le immagini di risonanza magnetica è possibile la discriminazione tra tessuti sulla base della loro composizione biochimica ed inoltre si hanno immagini delle sezioni corporee su tre piani diversi (assiale, coronale, saggitale). Gli svantaggi dell'utilizzo di questa tecnica sono principalmente i costi e i tempi necessari all'acquisizione delle immagini.

Tra le diverse applicazioni dell'imaging a risonanza magnetica, si collocano le immagini pesate in diffusione (**DWI**, **Diffusion Weighted Image**) che nell'ultimo decennio hanno riscosso molta attenzione grazie alla loro capacità di misurare il moto microscopico delle molecole di acqua in un tessuto biologico e di permettere la conoscenza e quindi l'analisi della microstruttura della sostanza bianca cerebrale sia in condizioni normali che patologiche.
Queste immagini sono basate sul fenomeno della **diffusione** che non è altro che il moto caotico e disordinato delle molecole di un mezzo dovuto all'agitazione termica e l'elemento preponderante di tale processo è rappresentato dall'acqua che comprende il 65-90% in volume dei tessuti biologici e svolge la funzione di mezzo di trasporto dei composti biochimici, divenendo quindi l'elemento fondamentale di molte reazioni chimiche del corpo umano. Pertanto, strutture con diffusione normale sono convenzionalmente rappresentate più **scure**, perchè il segnale di risonanza è più attenuato mentre laddove la velocità di

diffusione è minore come nell'ischemia per esempio, vengono rappresentate più **chiare**.

Un parametro molto importante nella diffusione è rappresentato dal valore **b** (**b-value**) che rappresenta la velocità a cui si spostano le molecole e determina quindi la sensibilità in diffusione. Questo valore è influenzato dall'intensità e dalla durata di applicazione del gradiente ed al suo aumentare aumenta la sensitività e quindi la pesatura in diffusione ma si abbassa il rapporto segnale rumore.

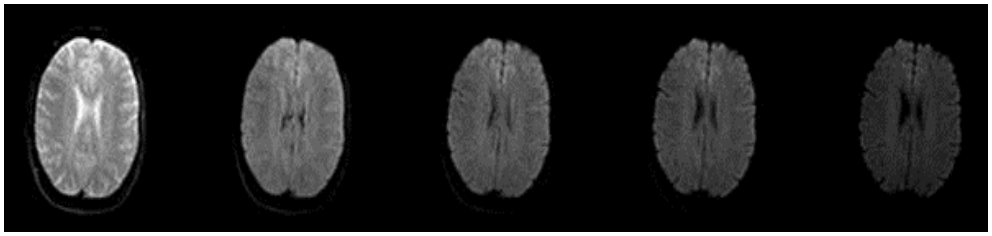


Figura 1: *Un esempio di immagine DWI in cui all'aumentare del valore b-value aumenta la pesatura in diffusione e si abbassa il rapporto segnale rumore*

Il più recente sviluppo della **DWI** è rappresentato dall'imaging in tensore di diffusione (**Diffusion Tensor Imaging**, **DTI**), che è una tecnica che mette in evidenza non soltanto l'entità della diffusione delle molecole di acqua nei tessuti ma anche la direzione consentendo di superare i limiti di direzionalità di DWI. Le immagini del tensore di diffusione si ottengo attraverso l'acquisizione di immagini pesate in diffusione lungo diverse direzioni tramite l'applicazione di una sequenza **EPI** (Echo Planar Imaging), cui viene fatta seguire in successione una serie di gradienti con orientazioni variabili [1].

Le immagini DWI, per loro natura, hanno un basso rapporto segnale/rumore (**SNR**) a causa della scarsa ampiezza del segnale ed il pronunciato rumore termico che è più evidente rispetto ad una convenzionale immagine di risonanza magnetica per via dell'utilizzo di strategie di acquisizione echo-planar estremamente rapide. Questo basso rapporto rende l'analisi delle immagini DWI estremamente complicata e per di più prediugica la stima dei parametri di diffusione quantitativi. Per aumentare il rapporto SNR è pratica comune fare la media di diverse acquisizioni in modo da ridurre la varianza del rumore senza rimuovere il rumore guidato dalla distorsione, ma tuttavia si tratta di una pratica che richiede tempo in termini di acquisizione e non è adeguata per le impostazioni cliniche in cui i pazienti devono rimanere per lunghi periodi di tempo.

Esistono diverse tecniche di denoising che possono essere applicate per migliorare la qualità dei dati DWI come step di post-elaborazione, così da non aumentare il tempo di scansione. Per esempio, Wiest-Daessle' et al. [2] hanno proposto una modifica del ben noto metodo **Non-local Means** (**NLM**) per trattare il rumore Rician che fu ulteriormente studiato da Descoteaux et al. [3]. Più recentemente, Tristan et al. [4] hanno proposto un approccio chiamato Errore Quadratico Medio Minimo Lineare (**LMMSE**) per affrontare il rumore Rician e la natura multicomponente delle immagini DWI.
Ancora più recente è il metodo di denoising di José V. Manjón e Pierrick Coupé et al. [5], oggetto di questo lavoro di tesi, che è basato sull'analisi delle componenti principali locale ed è progettato per tener conto della natura Rician del rumore presente nelle immagini DWI. In particolare, questo filtro sfrutta la natura multicomponente dei dataset di immagini DWI multi-direzionale utilizzando la decomposizione PCA locale (**LPCA**, **Local Principal Components Analysis**) per trarre vantaggio dalla ridondanza dell'andamento del segnale locale in contrapposizione ad altri metodi PCA che invece fanno utilizzo di un modello di ridondanza spaziale locale.

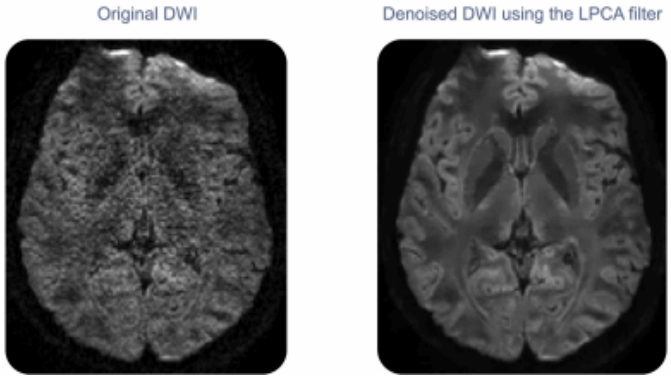


Figura 2: *A sinistra una immagine DWI con presenza di rumore, a destra l'immagine DWI dove il rumore è stato rimosso dal filtro LPCA*

Nonostante i suoi enormi vantaggi, il metodo **LPCA** implementato attraverso un algoritmo sequenziale è decisamente oneroso da un punto di vista computazionale, dal momento che bisogna eseguire l'analisi delle componenti principali per ogni voxel delle immagini DWI multicomponente e multidirezionali. Il lavoro svolto per questo progetto di tesi, si è posto come obiettivo la realizzazione di un software parallelo che implementasse il metodo **LPCA** in ambiente GPU-CUDA, con lo scopo di migliorare le prestazioni del filtro di rimozione del rumore, sfruttando le potenzialità dei processori grafici. Sono state sviluppate due versioni del software parallelo: una prima versione che fa un uso

maggiore della shared memory ma che a sua volta è limitata dalle dimensioni ridotte di questa memoria ed una seconda versione invece, che utilizza in modo combinato la shared memory con la global memory, per far fronte ai problemi di spazio della prima versione. Tra le due versioni dell'algoritmo parallelo, non sono state riscontrate differenze significative in termini di tempo d'esecuzione. Piuttosto, il confronto fra l'esecuzione dell'algoritmo eseguito su CPU e quello eseguito su GPU, ha evidenziato un notevole vantaggio in termini di prestazioni, dell'architettura CUDA. Infatti, i tempi di esecuzione del software parallelo si sono rivelati decisamente più bassi con un guadagno fino al 75% sull'algoritmo sequenziale, ottenendo oltretutto gli stessi risultati di accuratezza. Addirittura, il guadagno in termini di GFlops, è stato pari al 91,6%. Questi risultati incoraggianti, hanno permesso di dimostrare che a fronte di una strategia parallela di riscrittura del codice, è possibile far girare un algoritmo a prestazioni più elevate, anche su schede grafiche di medio/basso costo.

Nei capitoli che seguono si affronteranno i seguenti argomenti: nel capitolo uno verranno introdotti ed approfonditi due aspetti fondamentali di questo progetto di tesi: la struttura delle immagini DWI e la PCA sia da un punto di vista generale che da un punto di vista più applicativo nell'ambito della rimozione del rumore dalle immagini. A seguire una descrizione di due metodi di stima del rumore locale Rician (stimatori MUBE e SIBE). Il capitolo due è dedicato alla descrizione dell'algoritmo sequenziale con il quale è stato implementato il metodo LPCA per la rimozione del rumore da immagini DWI e rappresenta un importante punto di partenza per la progettazione e implementazione dell'algoritmo parallelo. Il capitolo tre è il nucleo di questa tesi ed è dedicato ad una introduzione dell'architettura CUDA e alla descrizione dettagliata dell'algoritmo parallelo realizzato in due differenti versioni che implementano le stesse operazioni ma utilizzando in maniera diversa le memorie (shared memory e global memory) della scheda video evidenziando quelli che sono i limiti di spazio e la scalabilità dell'algoritmo a seconda della scheda utilizzata. Nel capitolo quattro verranno illustrati i risultati sperimentali ottenuti con un'analisi completa sull'accuratezza dei risultati, un confronto delle performance tra CPU e GPU e i tempi di esecuzione dell'algoritmo parallelo con un confronto tra le due versioni realizzate. Infine il capitolo cinque contiene le conclusioni e gli sviluppi futuri.

Capitolo 1

Il metodo PCA locale per il denoising di immagini DWI

Il metodo PCA locale per la rimozione del rumore da immagini DWI sfrutta la ridondanza multi-direzionale delle immagini pesate in diffusione multi-componente per rimuovere il rumore casuale riducendo localmente le componenti principali meno significative attraverso un approccio overcomplete. Tale approccio è utilizzato dal momento che per ottenere l'elaborazione di ogni singolo voxel dell'immagine, si usano delle patch sovrapposte che portano ad ottenere più stime del singolo voxel processato. Si tratta di una tecnica che non richiede una ricerca delle patch simili all'interno dell'immagine ed è pertanto molto veloce.

1.1 Struttura di una immagine DWI

Un'immagine DWI è un'applicazione di imaging di risonanza magnetica ed in quanto tale è in grado di mostrare una rappresentazione tridimensionale di una qualunque parte del corpo suddivisa in sezioni. Ogni sezione (slice) è catturata perpendicolarmente attraverso il paziente e quindi più sezioni sono generate man mano che il tensore si sposta così che l'insieme di queste immagini compone la terza dimensione producendo quindi una rappresentazione 3D. In tal senso, la differenza con una MRI classica è rappresentata esclusivamente dalla presenza della diffusione. Però, nel loro più recente sviluppo in DTI, le immagini pesate in diffusione presentano una struttura multi-direzionale dove per direzione è intesa la direzionalità del moto microscopico delle molecole di acqua all'interno di un tessuto biologico. Pertanto, la natura direzionale conferisce all'immagine DWI una quarta dimensione (x, y, z, k) che ci consente

di visualizzare, fissata la terza dimensione (sezione specifica del tessuto), le k direzioni del moto microscopico all'interno di quella specifica sezione.
Le immagini DWI vengono principalmente digitalizzate nel formato **NIFTI** (Neuroimaging Informatics Technology Initiative) [6] ed i file hanno ".nii" come estensione. Questo formato è stato proposto da **NIfTI DFWG** (Neuroimaging Informatics Technology Initiative Data Format Working Group) [7] per facilitare le operazioni sui dati MRI. I file NIfTI possono essere visualizzati attraverso l'ausilio di diversi software come ad esempio **MRview** [8] oppure un altro dei tanti, **XMedCon** [9], che prevede anche diverse funzioni di conversione del formato da NIfTI a Raw Binary, DICOM, PNG, etc.
In figura 1.1 si può osservare un esempio di immagine DWI in cui viene evidenziata la natura multi-direzionale (4D) di queste immagini. Infatti, fissata la terza dimensione z, è possibile visualizzare una sezione specifica del cervello. Se invece ci muoviamo lungo la quarta dimensione k tenendo sempre fissata la terza dimensione z, è possibile osservare le direzioni di quella sezione del cervello.

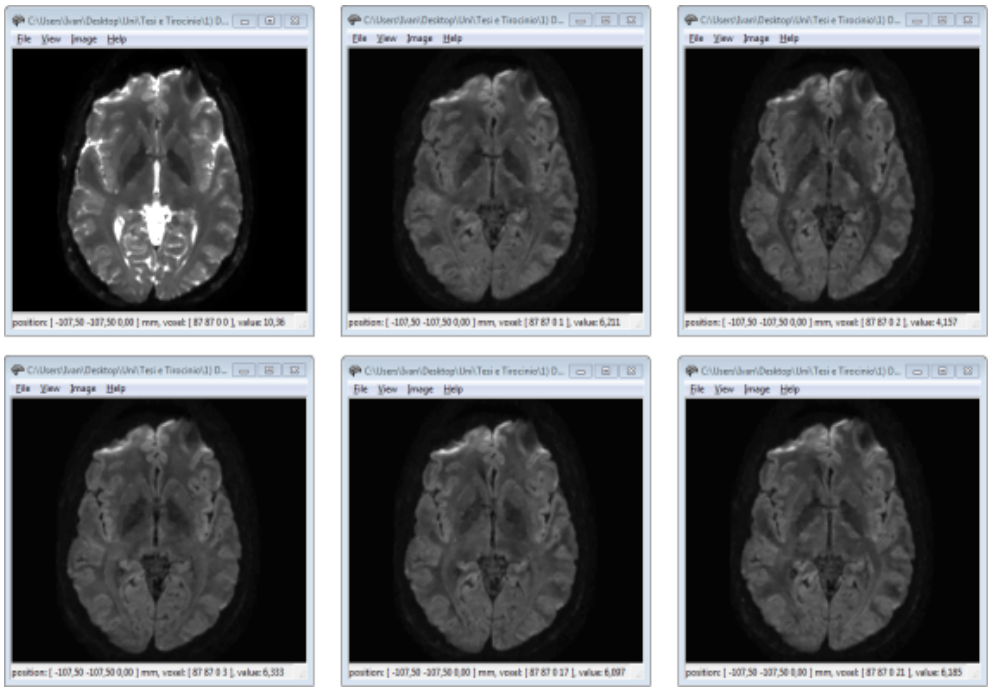


Figura 1.1: *Esempio di alcune immagini direzionali della sezione del cervello per z=0.*

1.2 PCA: Analisi delle Componenti Principali

La **Principal Component Analysis** (PCA, Analisi per Componenti Principali) è una tecnica per la semplificazione dei dati utilizzata nell'ambito della statistica multivariata. Fu proposta nel 1901 da Karl Pearson e sviluppata da Harold Hotelling nel 1933 [10].

1.2.1 Le componenti principali

La PCA è una trasformazione lineare ortogonale dei dati che vengono mappati in un nuovo sistema di riferimento ortogonale in modo da massimizzare la varianza (rispetto ad ogni altra proiezione) associata alla prima coordinata e poi quella associata alla seconda coordinata e così via. Le proiezioni, considerate nell'ordine, prendono il nome di prima componente principale, seconda componente principale, e così proseguendo fino ad arrivare all'ultima proiezione che è l'ultima componente principale. Quindi, tale tecnica, esprime i dati di partenza in un nuovo sistema di riferimento scelto in modo tale da catturare la maggior parte della variabilità dei dati. Non esiste nessun altro sistema di riferimento ortogonale in cui la proiezione dei dati abbia una varianza maggiore della proiezione dei dati sulla prima componente principale. La seconda componente principale, invece, è orientata in modo tale da catturare una variabilità dei dati che è seconda solo alla variabilità calcolata dalla prima componente principale. Questo concetto può essere quindi esteso per tutte le componenti.
In questo sistema ortogonale, il primo asse è orientato nella direzione di massima variabilità dei dati e cattura la maggior parte del fenomeno che essi descrivono. Le coordinate (le componenti principali) sono scelte in modo tale che la prima coordinata, cioè il versore del primo asse, massimizza la varianza dei dati proiettati in tale direzione, la seconda coordinata, cioè il versore del secondo asse, è la direzione in cui la varianza dei dati assume il secondo valore massimo, e così via.
Questo strumento risulta utile soprattuto quando si ha a che fare con un numero di variabili considerevole da cui si vogliono estrarre le maggiori informazioni possibili pur lavorando con un insieme più ristretto di variabili.

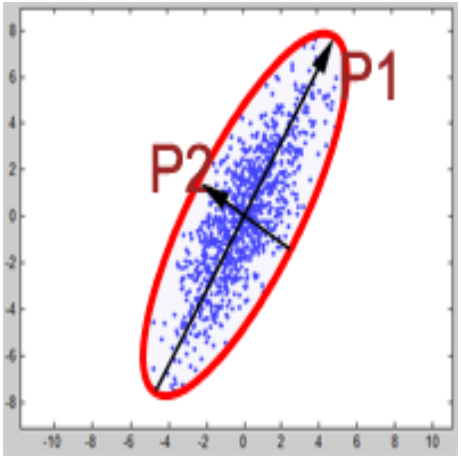


Figura 1.2: *Un esempio 2D raffigurante un sistema ortogonale a due assi, ovvero due vettori ordinati rispetto alla varianza dei dati.*

1.2.2 Funzionamento della PCA

I dati di partenza vengono organizzati in una matrice, indicata con X e definita come segue:

$$X = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_p \end{bmatrix} = \begin{bmatrix} X_{11} & X_{12} & \cdots & X_{1p} \\ X_{21} & X_{22} & \cdots & X_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ X_{p1} & X_{p2} & \cdots & X_{pp} \end{bmatrix} \tag{1.1}$$

con $i = 1, 2, ...p$ e $j = 1, 2, ..., p$ dove:

- le colonne rappresentano le p osservazioni effettuate.
- le righe sono le p variabili considerate per il fenomeno in analisi.

A partire dalla matrice X contenente per ogni $j - esima$ colonna, p variabili correlate tra loro, si vuole ottenere una matrice di nuovi dati Y, composta per ogni sua colonna $j - esima$ da p variabili incorrelate tra loro, che risultano essere una combinazione lineare delle prime.

La PCA dunque, consiste nel calcolo degli autovalori e degli autovettori della matrice di covarianza $\mathrm{X^T X}$ dei dati centrati:

$$X^T X P = P\Lambda \tag{1.2}$$

P è la matrice delle componenti principali contenente gli autovettori di X^TX, Λ invece è la matrice delle varianze associate alle componenti principali contenente gli autovalori di X^TX.

Il primo autovettore di X^TX è la prima componente principale, mentre il primo autovalore è la varianza associata a tale componente principale.

- La varianza associata alla prima componente principale è la massima possibile. Detto in altri termini, la prima componente principale cattura la maggior parte della variabilità dei dati.
- Il primo autovalore invece, indica la percentuale di informazione rappresentata dalla prima componente principale.

La varianza che rimane, è associata alle altre direzioni e quindi alle componenti principali che rimangono. Pertanto, il massimo di quello che resta, è associato al secondo autovettore con una percentuale pari al secondo autovalore e così via per tutte le altre direzioni.

1.2.3 La PCA nel contesto della rimozione del rumore

La filosofia di funzionamento del PCA nel contesto della rimozione del rumore è la seguente:

- Decomporre il segnale in componenti principali locali.
- Ridurre le componenti meno rilevanti con un'operazione di sogliatura.
- Ricostruire il segnale.

L'idea chiave di questo processo è che i modelli di immagine possono essere rappresentati come una combinazione lineare di un numero esiguo di immagini base mentre il rumore, non essendo sparso verrà propagato su tutte le componenti disponibili.
La rimozione del rumore basata su PCA può essere ottenuta utilizzando le informazioni globali di una serie di immagini (una componente per immagine) o localmente utilizzando finestre (patches) locali di immagine. Il primo approccio, sebbene efficace, richiede che il numero di immagini debba essere superiore al numero di componenti significative dell'immagine, ottenendo come risultato una rappresentazione meno sparsa. Questo problema può essere superato eseguendo la decomposizione PCA su piccole finestre locali invece che sull'intera immagine che produce una significativa rappresentazione sparsa.

1.3 Relazione tra PCA e fattorizzazione SVD

Tra la PCA e la fattorizzazione SVD sussiste una relazione, infatti la PCA è del tutto equivalente alla SVD. Nei paragrafi precedenti abbiamo detto che le componenti principali e le corrispondenti varianze sono gli autovettori e gli autovalori della matrice di covarianza dei dati centrati X^TX. Inoltre i vettori singolari di destra e i valori singolari di una generica matrice X nella SVD sono rispettivamente gli autovettori e la radice quadrata degli autovalori della matrice X^TX.

$$\left\{ \begin{array}{r} X^TXP = P\Lambda \\ X = U_n\Sigma_n V^T \\ V = autovettori(X^TX) \\ \Sigma_n = \sqrt{autovalori(X^TX)} \end{array} \right\} \Longrightarrow P = V, \lambda_i = \sigma_i^2$$

Quindi, le componenti principali della matrice dei dati centrati X sono i vettori singolari di destra di X e ogni componente singolare cattura una percentuale dei dati pari all'autovalore corrispondente. La prima colonna di V è la prima componente principale, σ_1^2 e σ_1 sono le corrispondenti varianza e deviazione standard associate a questa direzione e così via.

1.4 Rimozione del rumore locale PCA (LPCA)

L'immagine DWI è analizzata utilizzando un blocco scorrevole 4D locale e ad ogni posizione è applicata una decomposizione PCA per questo blocco in modo da trovare localmente la rappresentazione più ridotta di questi dati. L'eliminazione di informazioni superflue nei dati trasformati, riduce il rumore mentre preserva le principali caratteristiche dell'immagine.

1.4.1 La costruzione della matrice X

Per ogni punto x_p con $p = (i, j, z, k)$, del dominio dell'immagine $\Omega \times T \subset \Re^4$ con $\Omega \subset \Re^3$ e $T \subset \Re$, le patches 3D circostanti x_p in ogni immagine direzionale k sono riordinate come un vettore colonna di una matrice X che altro non è che una matrice di $N \times K$ componenti dove N corrisponde al numero di voxel della patch 3D intorno al punto di interesse (esempio: $N = 27 = 3 \times 3 \times 3$ voxels inclusi nella finestra 3D) e K corrisponde invece al numero delle immagini direzionali (può variare da 7 al numero di direzioni acquisite). Quindi ogni

vettore riga di questa matrice rappresenta il valore di un voxel x_p lungo tutte le k immagini di direzione (si veda fig. 1.3).

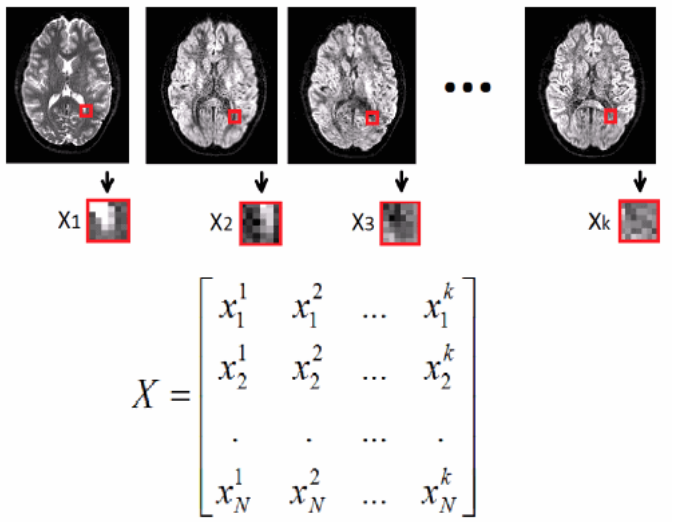


Figura 1.3: *Esempio di una matrice locale X formata da una serie di immagini DWI. Per ogni immagine viene estratta la finestra locale 3x3x3 del voxel x_p che poi è convertita in una colonna della matrice X.*

1.4.2 La decomposizione PCA locale

La PCA trasforma i vettori campione in un nuovo sistema dove alcune delle prime componenti rappresentano la maggior parte della variazione dei dati originali. Questa tecnica equivale a calcolare gli autovettori della matrice di covarianza di X, $C = X^T X$, dove X è stata precedentemente normalizzata sottraendo ad ogni colonna la propria media.

- Gli autovettori sono memorizzati come colonne in una matrice quadrata $K \times K$, che sarà indicata con W.
- Gli autovalori associati corrispondono alla quantità di variazione delle nuove componenti e sono memorizzati in una matrice diagonale di dimensione $K \times K$, che sarà indicata con $D = (\mathrm{d_{kk}})$.
- La decomposizione dell'autovalore di C si scrive come $C = WDW^T$.

Le nuove coordinate dei dati originali sono calcolate con un semplice prodotto righe per colonne tra matrici, come segue:

$$\mathbf{Y = XW} \tag{1.3}$$

1.4.3 La strategia di sogliatura

Gli algoritmi di sogliatura classici annullano i coefficienti di bassa grandezza. Questa tecnica può essere applicata numericamente modificando la matrice Y (definita in 1.3) in $\hat{\mathrm{Y}}$. Ogni valore di Y è annullato se la sua grandezza in valore assoluto è più bassa di un certo parametro $\tau = \gamma * \sigma$ dove σ è la deviazione standard di una stima del rumore locale, mentre γ è un fattore proporzionale a σ.

1.4.4 La ricostruzione di X senza rumore

Infine, la matrice di denoise è ottenuta calcolando la trasformazione PCA inversa per la ricostruzione dei dati di partenza epurati dal rumore iniziale:

$$\hat{\mathbf{X}} = \hat{\mathbf{Y}}\mathbf{W^{-1}} + \mu \tag{1.4}$$

Nella formula appena illustrata, $\hat{\mathrm{Y}}$ è la matrice delle nuove coordinate dei dati originali a cui è stata applicato il processo di sogliatura per annullare i coefficienti di bassa grandezza, W^{-1} è l'inversa della matrice delle componenti principali contenente gli autovettori, che è uguale alla trasposta W^{T} dal momento che si tratta di una matrice ortogonale. Ad ogni colonna k-esima della matrice $\hat{\mathrm{X}}$ è sommata la media μ_{k} della colonna che precedentemente era stata sottratta da X per la normalizzazione in modo tale da ripristinare i dati di partenza epurati dal rumore.

1.4.5 La media pesata per la stima dei voxel

La rimozione del rumore con PCA locale è realizzata in una maniera overcomplete dato che per ottenere l'elaborazione di ogni voxel del volume dell'immagine sono utilizzate patches sovrapposte. A causa di questa sovrapposizione di patch, si ottengono diverse stime per alcuni voxel poichè tutti i voxel sono processati indipendentemente (si veda fig. 1.4).

Per ogni voxel tutte le stime locali $\hat{\mathrm{x}}_{\mathrm{i}}(\mathrm{j})$ sono ottenute dalla combinazione di tutti i blocchi sovrapposti j alla posizione i utilizzando la seguente regola della media ponderata:

$$\hat{\mathbf{x}}_{\mathbf{i}} = \frac{\sum_{\mathbf{j=1}}^{\mathbf{V}} \theta_{\mathbf{j}} \hat{\mathbf{x}}_{\mathbf{i}}(\mathbf{j})}{\sum_{\mathbf{j=1}}^{\mathbf{V}} \theta_{\mathbf{j}}} \tag{1.5}$$

dove V è il numero di blocchi sovrapposti che contribuiscono a $\hat{\mathrm{x}}_{\mathrm{i}}$ e θ_j è il peso di ogni blocco j.

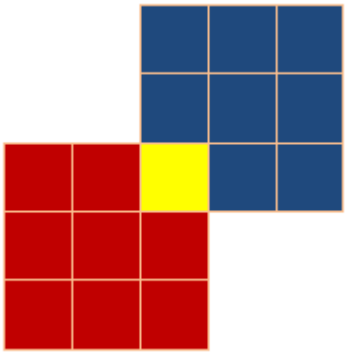


Figura 1.4: *Esempio semplificato di overlapping con due patch sovrapposte (rossa e blu). In giallo il voxel presente nell'intersezione tra le due patch e che deve essere stimato con la media dei due valori.*

1.5 La stima del rumore Rician nelle DWI

Il rumore nelle immagini di risonanza magnetica ed in particolar modo nelle immagini pesate in diffusione è di solito modellato con una distribuzione Rician. Sebbene esistano diversi metodi per stimare il livello di rumore Rician presente nelle immagini MRI [11],[12] pochi di loro sono capaci di stimare i modelli di rumore variabili spazialmente. In questo lavoro di tesi si è fatto riferimento ad un metodo presente in [5] che fornisce una stima della deviazione standard di rumore σ (come accennato nel par. 1.4.3) delle immagini Diffusion Weighted a seconda che esse abbiano una o più immagini con b-value=0. Ricordiamo che il b-value rappresenta la velocità a cui si spostano le molecole e determina quindi la sensibilità in diffusione. Le immagini con b-value=0 sono immagini non-DWI ovvero senza diffusione e generalmente sono quelle presenti nella direzione $k = 0$.

Come appena detto, questo metodo prevede quindi due tipi di stimatori:

- **Stimatore MUBE**: L'acronimo sta per *Multiple B=0 Estimator* e può essere applicato per stimare il rumore Rican nel caso in cui nel dataset di riferimento ci siano multiple immagini non-DWI con b-value=0. In questo progetto di tesi, è stato utilizzato tale stimatore dal momento che le immagini DWI utilizzate presentavano 21 immagini non-DWI in profondità (lungo la terza dimensione z) per la prima direzione $k = 0$.

- **Stimatore SIBE**: L'acronimo sta per *Single B=0 Estimator* e può essere utilizzato per stimare il rumore Rician nel caso in cui nel dataset di riferimento c'è solo una singola immagine non-DWI con b-value=0.

1.5.1 La stima del rumore con il metodo MUBE

Lo stimatore MUBE, introdotto nel precedente paragrafo, opera in questo modo: dato un insieme di n immagini con *b-value=0* ($n \geq 2$), si calcola per prima cosa la decomposizione PCA su una matrice formata da tante righe quante sono le immagini non-DWI e da tante colonne quanti sono i voxel componenti ciascuna immagine. In questo modo, le prime componenti saranno associate sia al segnale che al rumore mentre le ultime componenti conterranno principalmente un contributo di rumore. Quindi una stima del rumore può essere ottenuta calcolando la deviazione standard del rumore locale della componente meno significativa all'interno di regioni di $3 \times 3 \times 3$ voxels.

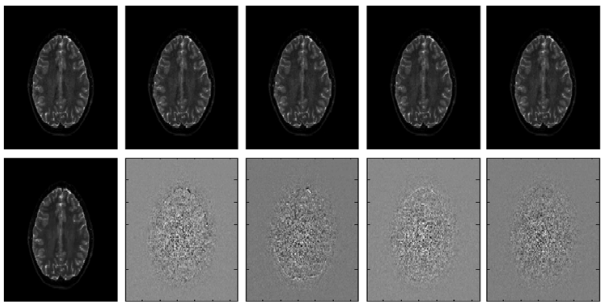


Figura 1.5: *In figura nella prima riga abbiamo 5 immagini non-DWI con b-value=0. In seconda riga è mostrato il risultato dell'applicazione del metodo MUBE sulle 5 immagini. Le ultime componenti mostrano prevalentemente un contributo di rumore piuttosto che di segnale.*

1.5.2 La stima del rumore con il metodo SIBE

Quando nel dataset di immagini utilizzato è disponibile soltanto una immagine non-DWI bisogna seguire una strategia differente per stimare il rumore locale Rician. Infatti, bisogna calcolare la decomposizione PCA sulle immagini gradiente piuttosto che sulle immagini non-DWI. Poi, in maniera del tutto simile al metodo MUBE, le prime componenti saranno associate sia al segnale che al rumore mentre le ultime componenti conterranno principalmente rumore. Così, una stima del rumore locale può essere calcolata nel medesimo modo in cui viene calcolata attraverso il metodo MUBE all'interno di regioni di $3 \times 3 \times 3$ voxels della componente meno significativa.

Capitolo 2

Implementazione sequenziale del metodo

Questo capitolo è dedicato alla descrizione e all'analisi dell'algoritmo sequenziale, che implementa il metodo OLPCA esaminato in maniera esaustiva nei paragrafi del capitolo precedente.

In questo lavoro di tesi, l'algoritmo sequenziale è stato sviluppato per diversi motivi: innanzitutto ha costituito un'ottima base di partenza per la progettazione della strategia che ha portato all'implementazione dell'algoritmo in ambiente parallelo. Inoltre, questa versione sequenziale è servita in un primo momento anche per testare l'accuratezza e la bontà del metodo OLPCA, nella rimozione del rumore dalle immagini pesate in diffusione e successivamente per confrontare i risultati con la versione parallela.

Il codice sorgente è stato sviluppato utilizzando il linguaggio C/C++ con l'ausilio della libreria grafica OpenCV. Nello specifico, è stata realizzata una classe *OLPCA* contenente tutti gli attributi e i metodi predisposti per la risoluzione del problema. Nelle sezioni che seguono, verranno illustrati i punti salienti dell'algoritmo:

1. Corruzione delle immagini DWI con rumore Rician
2. Lettura delle intensità dei voxel
3. Stima del rumore Rician (MUBE)
4. Denoising mediante PCA locale overcomplete
5. Post-processing con la tecnica di overlapping e media pesata

2.1 Corruzione delle immagini DWI con rumore Rician

Come detto nel capitolo precedente, le immagini pesate in diffusione sono disponibili in diversi formati ma quello standard è il formato **NIFTI** con estensione ".nii". Pertanto, una delle prime esigenze per l'elaborazione delle immagini Diffusion Weighted, è stata quella di poter accedere in lettura al contenuto di questo tipo di file. Tale scopo è stato raggiunto attraverso l'ambiente di calcolo numerico e di statistica **MATLAB** con l'ausilio del toolbox di analisi di immagini di risonanza magnetica *Nifti Toolbox for MATLAB*. Questo tipo di tool fornisce una serie di funzioni principali in grado di caricare, leggere, scrivere e visualizzare le immagini *DWI* in formato NIFTI in modo semplice ed intuitivo con poche righe di codice. In realtà, esistono anche altre librerie che si interfacciano con il linguaggio C/C++ come **ITK** e **niftilib** che sono finalizzate all'analisi delle immagini mediche ma la loro configurazione ed il conseguente utilizzo era meno immediato. Dal momento che l'obiettivo principale è stato quello di parallelizzare l'algoritmo sequenziale, si è preferita la stada più semplice per la lettura di queste immagini che è avvenuta quindi tramite MATLAB, anche se in seguito il contenuto è comunque stato salvato su un file binario, decisamente più semplice da leggere nel linguaggio C/C++ attraverso le funzioni di apertura/lettura file.

L'unica immagine Diffusion Weighted, in formato NIFTI, a cui si è avuto accesso è la *Ground-Truth* che è una immagine campione priva di alcun tipo di rumore. A partire da questa immagine, si sono poi aggiunte le diverse percentuali di rumore Rician ottenendo quindi le immagini rumorose da restaurare. La lettura dell'immagine Ground-Truth, l'aggiunta di rumore ed il salvataggio delle intensità dei voxel in un file binario per il linguaggio C/C++, sono state realizzate tramite il seguente codice MATLAB:

```
% lettura file NIFTI
out = MRIread('dwi_truth.nii');
outVolume = out.vol;

% corruzione con rumore Rician
NoisePerc = 3;
s = size(outVolume);
noiseStd = NoisePerc*max(max(outVolume(:,:,1,1)))/100;
noiseData = rndrice(outVolume, noiseStd);
IN = outVolume;

% sogliatura
for k = 1:22
```

```
  for z = 1:21
    for i = 1:176
      for j = 1:176
          if(outVolume(i,j,z,k) < 3)
             IN(i,j,z,k) = outVolume(i, j, z, k);
          else
             IN(i,j,z,k) = noiseData(i, j, z, k);
          end
      end
     end
   end
end

% salvataggio della struttura in un file binario
fileID = fopen('dwi_noise3.bin','w');
fwrite(fileID, outVolume, 'float');
fclose(fileID);
```

Attraverso la funzione *MRIread* si legge il contenuto del file NIFTI ottenendo in output la struttura *out*. Accedendo al campo *vol* della struttura *out*, si recupera il volume di dati 4D dell'immagine pesata in diffusione. I dati rumorosi sono ottenuti tramite la funzione *rndrice* che prende in input il volume di dati e la deviazione standard *noiseStd* dell'immagine dipendente dalla percentuale di rumore scelta, generando così la distribuzione di rumore Rician sull'immagine nella struttura dati *noiseData*, che ha la stessa dimensione di *outVolume*. L'immagine è stata corrotta con rumore Rician soltanto nella sezione raffigurante il cervello e non sullo sfondo nero, al fine di evitare di avere delle immagini troppo opache in seguito al processo di rimozione del rumore, dato che nel momento in cui lo sfondo viene corrotto è difficile che torni come prima, soprattutto con livelli di rumore elevati, influendo negativamente sul risultato finale. Questa interessante tecnica, è stata realizzata attraverso un processo di sogliatura in cui i valori di intensità bassi, vicini al colore nero dello sfondo, sono stati lasciati inalterati prendendo i valori dell'immagine Ground-Truth. Infine, la struttura 4D corrotta dal rumore, viene scritta in un file binario di float in modo da poterlo utilizzare negli algoritmi (sequenziale e parallelo) scritti in linguaggio C/C++.

2.2 Lettura delle intensità dei voxel

I file binari contenenti i valori di intensità dei voxel delle immagini con i diversi livelli di rumore e l'immagine ground-truth, sono utilizzati negli algoritmi

(sequenziale e parallelo), per il caricamento dei dati in apposite strutture atte a contenere la natura 4D delle immagini pesate in diffusione. Le strutture utilizzate sono dei semplici array *stacked* in grado di indicizzare tutti i voxel dell'immagine con una opportuna formula di *mapping* che tiene conto delle quattro dimensioni. Quindi, la lettura dei dati ed il conseguente caricamento nelle strutture avviene in maniera molto semplice tramite la classica gestione dei file binari nel linguaggio C/C++.

Di seguito alcuni snippet di codice sorgente utilizzati per il mapping dei dati ed il caricamento delle intensità dei voxel nella struttura.

```
// restituisce l'indice globale dell'immagine DW 4D
int maps(int i, int j, int z, int k)
{
   return (i * WIDTH) + j + (z * LENGTH * WIDTH) +
          (k * LENGTH * WIDTH * DEPTH);
}

// lettura voxel da file e caricamento nell'array image
for (int k = 0; k < DIRECT; k++) {
  for (int z = 0; z < DEPTH; z++) {
    for (int i = 0; i < LENGTH; i++) {
      for (int j = 0; j < WIDTH; j++) {
        fread(&voxel, sizeof(float),1, file);
        image[maps(i,j,z,k)] = voxel;
      }
    }
  }
}
```

La funzione *maps* prende in input gli indici per ogni dimensione ed applica la formula di mapping per generare l'indice globale per l'accesso all'elemento desiderato nell'array 4D, linearizzato come vettore stacked per velocizzare l'accesso e quindi la lettura dei dati. A seguire invece, la porzione di codice che effettua il caricamento dei dati nell'array stacked utilizzando quattro cicli for innestati per scansionare tutti i voxel dell'immagine nelle 4 dimensioni. La funzione *fread* legge un float per ogni iterazione e lo salva nella variabile voxel che poi è succesivamente assegnata all'array *image* tramite la funzione maps che indicizza ed accede all'elemento specifico (i, j, z, k).

2.3 La stima del rumore Rician (MUBE)

Il rumore Rician viene stimato utilizzando il metodo **MUBE** descritto nel capitolo precedente. Il primo passo prevede la costruzione della matrice X che ricordiamo, deve avere tante righe quante sono le immagini *non-DWI* e tante colonne quanti sono i voxel di ogni immagine. Dunque, si considerano soltanto le immagini non-DWI nella direzione ($k = 0$) e per ogni singola immagine lungo z, vengono caricate tutte le intensità dei suoi voxel in un vettore di float, poi ulteriormente convertito in un vettore riga del tipo Mat di *OpenCV* e tale riga è aggiunta in coda alla matrice X. Al termine di questo procedimento, si esegue la **PCA** sulla matrice X utilizzando la classe *PCA* della libreria OpenCV e quindi è determinata la matrice delle nuove coordinate dei dati originali Y (della stessa dimensione di X) attraverso il metodo *project* dell'oggetto *pca*. A partire da Y si recuperano le ultime 3 immagini sulle quali è poi costruita la matrice delle deviazioni standard del rumore *devStdMat*. Questa matrice ha size pari alla dimensione di una singola immagine DWI (176×176), pertanto per ogni suo elemento si estrae la finestra locale $3 \times 3 \times 3$ a partire dalle ultime 3 immagini di Y contenute nella matrice *YLSC* (***Y*** ***L****east* ***S****ignificant* ***C****omponent*) e di questa finestra locale se ne calcola la deviazione standard. La stima del rumore Rician contenuta nell'elemento (i, j) della matrice *sigma*, è utilizzata per ogni z e per ogni k.

Diamo un'occhiata al codice sorgente che implementa quanto appena detto:

```
Mat mubeEstimator()
{
   /* Costruzione della matrice X:
      1) X ha tante righe quante sono le immagini su z
      2) X ha tante colonne quanti sono i voxel di ogni immagine. */
   Mat X;
   vector<float> image;
   for (int z = 0; z < DEPTH; z++) {
      image.clear();
      for (int i = 0; i < LENGTH; i++) {
        for (int j = 0; j < WIDTH; j++) {
          image.push_back(imageInputNoise[maps(i,j,z,0)]);
        }
      }
      Mat riga=Mat(image).t();
      X.push_back(riga);
   }

   // esegue la PCA e ottiene le nuove coordinate Y
   PCA pca(X,Mat(),CV_PCA_DATA_AS_COL);
```

```
    Mat Y;
    pca.project(X, Y);

    // recupera le ultime 3 immagini elaborate in Y
    Mat YLSC(3,LENGTH*WIDTH, CV_64F);
    for (int i = 0; i < 3; i++) {
      for (int j = 0; j < LENGTH*WIDTH; j++) {
          YLSC.at<float>(i,j) = Y.at<float>(18+i,j);
      }
    }

    // costruisce la matrice delle deviazioni standard
    Mat devStdMat(176, 176, CV_64F);
    vector<float> localWindow;
    // estrazione della finestra locale 3x3x3 per ogni (i,j)
    for (int i = 0; i < LENGTH; i++) {
      for (int j = 0; j < WIDTH; j++) {
        localWindow.clear();
        for (int m = i-1; m <= i+1; m++) {
          for (int n = j-1; n <= j+1; n++) {
            if ((m >= 0 && m < LENGTH) && (n >= 0 && n < WIDTH)) {
              localWindow.push_back(YLSC.at<float>(0, m*WIDTH+n));
              localWindow.push_back(YLSC.at<float>(1, m*WIDTH+n));
              localWindow.push_back(YLSC.at<float>(2, m*WIDTH+n));
            }
          }
        }

        // media degli elementi della finestra locale
        float sum = 0, mean = 0;
        for (int k = 0; k < localWindow.size(); k++)
            sum += localWindow[k];
        mean = (float)sum/localWindow.size();

        // calcolo della deviazione standard del rumore
        sum = 0;
        for (int k = 0; k < localWindow.size(); k++)
           sum += pow(localWindow[k]-mean,2);

        devStdMat.at<float>(i,j) = sqrt((float)sum/localWindow.size());
      }
    }

    return devStdMat;
}
```

2.4 L'algoritmo di PCA locale overcomplete

L'algoritmo di *PCA locale overcomplete* sequenziale prevede la scansione di ogni singolo voxel lungo le tre dimensioni (i, j, z) attraverso tre cicli for innestati. Per ogni voxel, si esaminano tutte le k direzioni e quindi per ogni direzione si estrae la finestra locale $3x3x3$ (centrata nel voxel in esame) attraverso il metodo *getLocalWindow3D* che restituisce un vettore contenente i 27 elementi estratti. Il contenuto di questo vettore è poi convertito in un vettore riga di tipo Mat. Per ogni direzione, ciascuno di questi vettori riga è inserito in coda ad una matrice *xTemp*, successivamente trasposta al fine di formare la matrice X locale in cui la k-esima colonna rappresenta la finestra locale $3 \times 3 \times 3$ estratta nella direzione k e centrata nel voxel (i, j, z, k). Arrivati a questo punto, come descritto ampiamente nel capitolo precedente, a partire dalla matrice X viene applicata la PCA locale in cui l'unico dettaglio da menzionare rispetto a quello che è già stato detto, è che la matrice degli autovettori viene calcolata attraverso *SVD* con la funzione *compute* del namespace *SVD* della libreria *OpenCV*, dato che tra PCA e SVD sussiste una relazione di equivalenza. Una volta ottenuta la matrice XD restaurata, per ogni direzione k si estrae la colonna k-esima di XD, la si converte in un vettore di float che viene dato in input al metodo overlapping. L'overlapping, come verrà descritto in seguito, provvede a salvare tale colonna restaurata nella posizione corretta all'interno dell'immagine di output, dove in principio era stata estratta a partire dall'immagine rumorosa data in input.

Ecco il codice sequenziale che implementa il metodo PCA locale overcomplete appena illustrato:

```
void olpcaDenoising()
{
    vector<float> localWindow3D;
    Mat X, W, Y, XD, mean, C, w, vt;
    float sigmaValue = 0.0;

    // per ogni voxel (i, j, z)
    for (int z = 0; z < DEPTH; z++) {
      for (int i = 0; i < LENGTH; i++) {
        for (int j = 0; j < WIDTH; j++) {
          Mat xTemp;

          // per ogni direzione k costruzione di xTemp
          for (int k = 0; k < DIRECT; k++) {
            localWindow3D.clear();
            localWindow3D = getLocalWindow3D(i, j, z, k);
            Mat xCol=Mat(localWindow3D).t();
            xTemp.push_back(xCol);
```

```
            }

            // trasposizione di xTemp per ottenere X
            X  = xTemp.t();

            // operazioni di PCA locale overcomplete
            sigmaValue = sigma.at<float>(i,j);
            reduce(X, mean, 0, CV_REDUCE_AVG);
            centerX(X, mean);
            C = X.t() * X;
            SVD::compute(C, w, C, vt);
            W = C;
            Y = X * W;
            Y = doThresholding(sigmaValue, Y);
            XD = Y * W.t();
            restoreX(XD, mean);

            // per ogni direzione k fa overlapping
            for (int k = 0; k < DIRECT; k++) {
              Mat col = XD.col(k).t();
              vector<float> vec;
              col.copyTo(vec);
              overlapping(vec, i, j, z, k);
            }
          }
        }
      }
}
```

A seguire invece, il metodo *getLocalWindow3D* con il quale, a partire dagli indici del voxel in esame (i, j, z, k) forniti in input, estrae la finestra locale $3 \times 3 \times 3$ effettuando anche un opportuno controllo di bordi.

```
vector<float> getLocalWindow3D(int i, int j, int z, int k)
{
    vector<float> localWindow3D;

    // cicli for innestati per la finestra locale 3x3x3
    for (int l = z-1; l <= z+1; l++) {
      for (int m = i-1; m <= i+1; m++) {
        for (int n = j-1; n <= j+1; n++) {
          // controllo dei bordi di confine
          if ((l >= 0 && l < DEPTH) && (m >= 0 && m < LENGTH)
             && (n >= 0 && n < WIDTH))
            localWindow3D.push_back(imageInputNoise[maps(m,n,l,k)]);
          else
```

```
                localWindow3D.push_back(0);
            }
        }
    }

    return localWindow3D;
}
```

2.5 La tecnica di overlapping e la media pesata

La tecnica di overlapping costruisce l'immagine restaurata di output in maniera locale, collocando le colonne delle matrici XD restaurate, nelle locazioni dove in precedenza erano state estratte dall'immagine rumorosa di input per la costruzione della matrice X. Questa tecnica tiene conto delle sovrapposizioni tra le patch (finestre locali) considerando il valore di ogni voxel come un accumulatore contenente la somma delle sue stime presenti nelle finestre locali sovrapposte dei voxel adiacenti.

```
void overlapping(vector<float> window3D, int i, int j, int z, int k)
{
    int index=0;
    // accede agli elementi della finestra locale 3x3x3
    for (int l = z-1; l <= z+1; l++) {
      for (int m = i-1; m <= i+1; m++) {
        for (int n = j-1; n <= j+1; n++) {
          // controllo dei bordi
          if ((l >= 0 && l < DEPTH) && (m >= 0 && m < LENGTH)
          && (n >= 0 && n < WIDTH)) {
            imageOutput[maps(m, n, l, k)].value += window3D[index];
            imageOutput[maps(m, n, l, k)].overlaps++;
          }
          index++;
        }
      }
    }
}
```

L'algoritmo di overlapping prende in input una colonna della matrice XD, sottoforma di vettore, quindi a partire dal voxel centrale identificato dagli indici (i, j, z, k), determina attraverso tre cicli for innestati, gli indici della finestra

locale $3 \times 3 \times 3$ ed in queste posizioni colloca gli elementi del vettore sommandoli al contenuto precedente della cella, tenendo in considerazione l'eventuale possibilità di sovrapposizione. Ogni volta che un valore viene sommato al precedente, vuol dire che c'è una sovrapposizione in quella posizione, pertanto viene aggiornato anche il numero di sovrapposizioni (**overlaps**). Il tutto è gestito attraverso la rappresentazione in memoria dell'immagine di output come una struttura contenente due campi per ogni elemento: il campo *value* che memorizza la somma delle stime del voxel ed il campo *overlaps* che contiene il numero di sovrapposizioni stimate.

Come processo di post-elaborazione dell'algoritmo PCA locale overcomplete, viene eseguita una media pesata che consiste nello scansionare ogni voxel dell'immagine di output e per ognuno di essi dividere il valore contenente le sue stime per il numero di sovrapposizioni, in modo da avere un valore medio stimato.

```
void weightedAverage()
{
   // per ogni voxel calcola la media delle stime
   for (int k = 0; k < DIRECT; k++) {
     for (int z = 0; z < DEPTH; z++) {
       for (int i = 0; i < LENGTH; i++) {
         for (int j = 0; j < WIDTH; j++) {
           imageOutput[maps(i, j, z, k)].value /=
           imageOutput[maps(i, j, z, k)].overlaps;
         }
       }
     }
   }
}
```

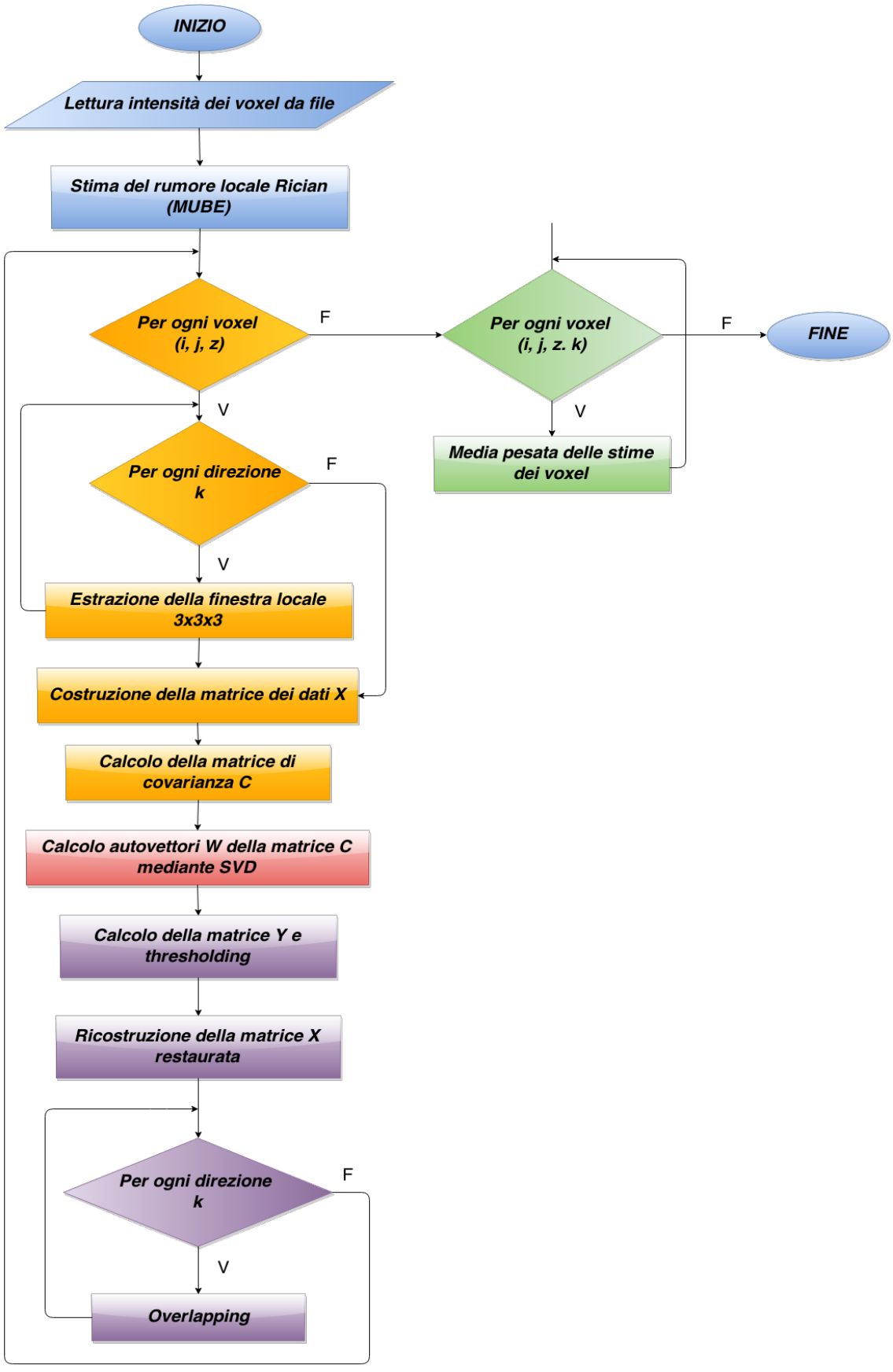


Figura 2.1: *Diagramma di flusso dei passi principali dell'algoritmo sequenziale del metodo PCA locale overcomplete per la rimozione del rumore Rician da immagini di risonanza magnetica pesate in diffusione.*

Capitolo 3

Implementazione parallela in ambiente GP-GPU

In questo capitolo inizialmente viene fatta una introduzione al *General-Purpose Computing On Graphics Processing Units* per poi esporre tutte le peculiarità dell'architettura hardware per l'elaborazione parallela *NVIDIA CUDA*. Successivamente si passerà alla descrizione del software parallelo che implementa il metodo PCA locale overcomplete in ambiente *GPU-CUDA*.

3.1 Introduzione al GP-GPU

3.1.1 Le Graphics Processing Unit

L'unità di elaborazione grafica (dall'inglese **G**raphics **P**rocessing **U**nit, **GPU**) [13] è una tipologia particolare di coprocessore composto da più multiprocessori paralleli organizzati secondo uno schema architetturale finalizzato al paradigma **SIMD** (*Single Instruction Multiple Data*). Queste unità nascono nell'ambito della *Computer Graphics* per l'esecuzione di operazioni grafiche (**rendering**) che lavorano su una grande quantità di dati e sono specializzate in applicazioni parallele estremamente esigenti in termini di potenza di elaborazione, dunque dedicano più transistor all'elaborazione dei dati piuttosto che alla loro memorizzazione e al controllo di flusso.

Non tutte le applicazioni traggono vantaggio dall'uso delle GPU. Le applicazioni con un'elevata logica di controllo del processo di calcolo vengono eseguite efficientemente in modo sequenziale dalle tradizionali CPU ma con pessime prestazioni dall'architettura parallela delle GPU (es. Gestione dei database, compressione dei dati, algoritmi ricorsivi).

Caratteristiche delle applicazioni che traggono vantaggio dall'uso delle GPU:

- Grande intensità aritmetica.
- Elevato grado di parallelismo (stesse operazioni ripetute su molti dati).
- Elevata richiesta di banda di memoria.
- Condizioni di controllo molto limitate.

3.1.2 Il computing accelerato dalle GPU

Il *computing accelerato dalle GPU* [14] consiste nell'affiancare una GPU ad una CPU per accelerare le applicazioni scentifiche, di analisi, tecniche o aziendali. Questa tecnica, offre prestazioni senza precedenti, demandando le porzioni più impegnative dei calcoli di ogni applicazione alle GPU, mentre la parte restante del codice viene eseguita dalla CPU. Per l'utente, l'unica differenza è che sostanzialmente le applicazioni sono molto più rapide.

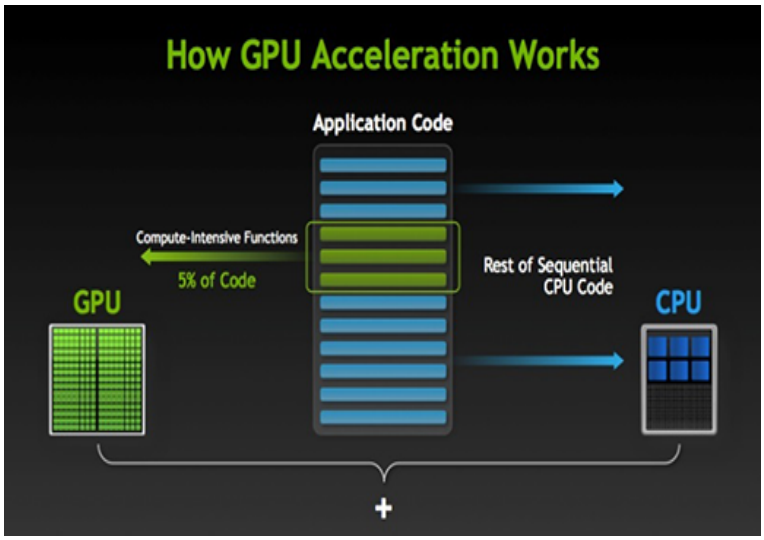


Figura 3.1: *In figura è mostrato il modo in cui GPU e CPU cooperano per accelerare il lavoro di un'applicazione.*

Un modo semplice per capire al volo la differenza tra una CPU e una GPU consiste nel mettere a confronto come elaborano le attività. Una CPU è costituita da diversi core ottimizzati per l'elaborazione sequenziale mentre una GPU è dotata di una fitta architettura parallela costituita da migliaia di core di minori dimensioni e di maggiore efficienza, progettati per la gestione simultanea di più operazioni.

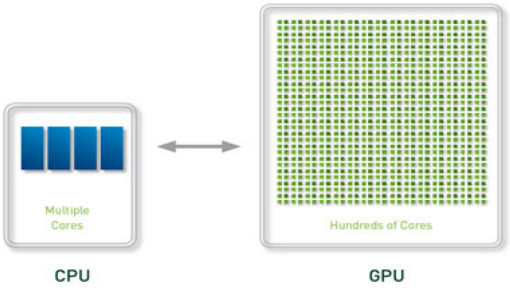


Figura 3.2: *In figura è evidenziata la differenza tra CPU e GPU in termini di core.*

3.1.3 Le General-Purpose GPU

La **GP-GPU**, sigla di **General-Purpose Computing on Graphics Processing Units** (calcolo a scopo generale su unità di elaborazioni grafiche) [15] si propone di impiegare le GPU, per applicazioni parallele *High Performance General Purpose*, sfruttando le loro elevate capacità di elaborazione in parallelo. In tale ambito di utilizzo la GPU viene impiegata per elaborazioni estremamente esigenti in termini di potenza di elaborazione, e per le quali le tradizionali architetture di CPU non hanno una capacità di elaborazione sufficiente. Tale tipo di elaborazioni sono, per loro natura, di tipo altamente parallelo e in grado quindi di beneficiare ampiamente dell'architettura tipica delle GPU.

Le applicazioni in grado di sfruttare in modo significativo la potenza di calcolo delle moderne GPU, rappresentano soltanto una piccola parte dell'intero panorama software, poichè per sfruttare a pieno le caratteristiche di queste architetture è necessaria un'elevata parallelizzazione del codice, una caratteristica che solo alcuni problemi scentifici possiedono. Esempi tipici di applicazioni che hanno questa caratteristica sono riscontrabili in maggior modo nel campo dell'elaborazione delle immagini, come il problema affrontato in questo elaborato di tesi sulla rimozione del rumore da immagine mediche.

I benefici dell'utilizzo di una GPU per svolgere elaborazioni di tipo diverso dall'ambito grafico, sono molteplici: innanzitutto le GPU offrono una enorme potenza teorica che porta alla riduzione del tempo di esecuzione rispetto alle elaborazioni svolte dalle CPU comportando quindi grossi *vantaggi a livello prestazionale* a fronte di una riscrittura ed ottimizzazione del software per il supporto di tale architettura. Inoltre, va considerato che il *costo* delle GPU è allineato a quello di una CPU che appartiene alla stessa fascia di mercato, pertanto ottimizzare i software per il corretto sfruttamento di queste architetture

consente di contenere efficacemente i costi migliorando l'efficienza dell'elaborazione. Le GPU hanno un *tasso di aggiornamento tecnologico* decisamente più elevato delle CPU, infatti le architetture relative alle GPU in genere durano soltanto dai 12 ai 18 mesi massimo e ad ogni nuova generazione si assiste al raddoppio puro della potenza elaborativa mentre le architetture delle CPU rimangono spesso immutate per anni ed in genere aumentano le prestazioni, a parità di clock, del 20-30% massimo. Va detto però, che i produttori di CPU hanno investito molte risorse sul fronte dell'efficienza energetica e del contenimento dei *consumi massimi*, cosa che i produttori di GPU sembrano aver proprio ignorato. Tuttavia, l'enorme potenza elaborativa teorica delle GPU compensa grandemente l'elevato livello di consumo energetico ed il rapporto consumo/prestazioni è nettamente a vantaggio delle GPU rispetto alle CPU.

Inizialmente la realizzazione di un'applicazione in ambiente GP-GPU consisteva nel programmare la GPU tramite le API di disegno 3D, ovvero linguaggi di programmazione grafici come *OpenGL* e *Cg*. Gli sviluppatori dovevano rendere le applicazioni scentifiche simili ad applicazioni grafiche: bisognava esprimere il problema da risolvere e i suoi dati in termini di vertici, triangoli e poligoni da manipolare richiamando le API. Ciò rendeva molto complicata la programmazione e limitava l'uso delle GPU. Successivamente **NVIDIA** ha modificato la GPU rendendola interamente programmabile per le applicazioni scentifiche e aggiungendovi il supporto per linguaggi ad alto livello come C e C++. Grazie all'ambiente di sviluppo **CUDA**, che approfondiremo nella prossima sezione, è possibile ricorrere a delle API apposite per la programmazione delle GPU piuttosto che alle API di disegno 3D.

3.2 Ambiente di sviluppo CUDA

CUDA (acronimo di ***C**ompute **U**nified **D**evice **A**rchitecture*) [16], [17] è un'architettura hardware creata da **NVIDIA** per l'elaborazione parallela che permette di avere dei notevoli aumenti di prestazioni del computing grazie allo sfruttamento della potenza di calcolo delle GPU. Tramite l'ambiente di sviluppo per CUDA, i programmatori di software possono scrivere applicazioni capaci di eseguire calcolo parallelo sulle GPU delle schede video NVIDIA. I linguaggi di programmazione disponibili nell'ambiente di sviluppo per CUDA, sono estensioni dei linguaggi più diffusi per scrivere programmi. Il principale è *CUDA-C* (C con estensioni NVIDIA), altri sono estensioni di *Python, Fortran, Java e MATLAB*.

CUDA ha molti vantaggi rispetto alle tecniche di computazione sulle GPU che usano le API grafiche:

- Letture temporali con le quali il codice può essere letto attraverso indirizzi arbitrari di memoria.
- Memoria condivisa: CUDA espone una veloce condivisione di memoria (una regione di 16KB di grandezza) che può essere condivisa fra i thread.
- Veloci downloads e riletture verso e dalla GPU.
- Supporto completo per divisioni intere e operazioni bit-a-bit, incluse strutture texture intere.

Ci sono anche delle limitazioni come:

- CUDA è un sottoinsieme del linguaggio C privo di ricorsione e puntatori a funzione, più alcune semplici estensioni. Comunque, un singolo processo deve essere eseguito attraverso multiple disgiunzioni di spazi di memoria, diversamente da altri ambienti di runtime C.
- La larghezza di banda e la latenza tra CPU e GPU può essere un collo di bottiglia.
- I thread devono essere eseguiti in multipli di 32 per ottenere migliori prestazioni, con un numero totale di thread nell'ordine di migliaia.
- GPU dotate di CUDA sono disponibili solo da NVIDA (*GeForce 8 serie superiori, Quadro e Tesla*).

3.2.1 Il modello di programmazione

Il modello di programmazione CUDA considera la CPU e la GPU come due macchine distinte e separate, dette rispettivamente **host** e **device**. Pertanto, il meccanismo di funzionamento di questa architettura, consiste nella combinazione di parti di codice sequenziale eseguite sull'host e parti di codice parallele eseguite sul device. Il codice parallelo viene chiamato **kernel** ed è l'host che lo richiama configurando il device per l'esecuzione in parallelo e passandogli i parametri opportuni mentre il device può eseguire soltanto un kernel per volta. Dal momento che l'host e il device sono due macchine distinte con spazi di indirizzamento separati, prima di chiamare un kernel bisogna effettuare una operazione di copia di tutti i dati necessari nella global memory del device. Viceversa, quando un kernel ha terminato la propria esecuzione, bisogna copiare sulla memoria dell'host i dati elaborati dal device.

In seguito a questa distinzione tra host e device le function CUDA hanno lo stesso prototipo delle usuali funzioni C ma è necessario anteporre ad esse un

qualificatore: `__host__` per le function canoniche eseguite su CPU, `__device__` per function richiamate ed eseguite su device e infine `__global__` per function richiamate da host ed eseguite su device, ovvero i kernel.

3.2.2 I kernel

Il kernel è un pezzo di codice parallelo richiamato dall'host dopo una opportuna configurazione dei suoi parametri ed eseguito quindi in parallelo sul device.

Queste parti di codice hanno determinate caratteristiche:

- devono avere void come tipo di ritorno.
- non possono essere ricorsivi.
- non possono avere un numero di parametri variabile.
- non possono usare variabili di tipo statico.
- possono accedere solo allo spazio di memoria della GPU.

Ecco l'istruzione per invocare un kernel CUDA:

```
nomeKernel<<<nBlocchi, nThreads, sizeShMem>>> (lista parametri);
```

dove *nBlocchi* è il numero di blocchi della griglia computazionale, mentre *nThreads* è il numero di thread per un singolo blocco (1D, 2D o 3D). Questi parametri sono di un tipo predefinito di CUDA, *dim3*, che rappresenta un vettore di 3 interi accessibili come campi con le notazioni .x, .y, .z utilizzati per indicare il numero di thread o blocchi in una determinata direzione. Infine *sizeShMem* è la dimensione da allocare per la shared memory se viene dichiarata extern.

3.2.3 L'unità fondamentale del parallelismo: i thread

L'unità fondamentale del parallelismo in CUDA è il thread: più thread eseguono lo stesso flusso di istruzioni su dati diversi a carico del device (*paradigma SIMT: Single Instruction Multiple Thread*).

- I thread vengono raggruppati in blocchi eseguiti sullo stesso multiprocessore e condividono la shared memory.
- I blocchi vengono invece raggruppati in una grid che indica la configurazione dell'esecuzione parallela di un kernel.

- Ogni kernel è organizzato in una grid di blocchi.

I thread eseguono parallelamente la stessa porzione di codice e lavorano con variabili predefinite:

- **threadIdx**: indice del thread all'interno del corrispondente blocco.
- **blockIdx**: indice del blocco d'appartenenza del thread all'interno della grid.
- **blockDim**: numero di thread contenuti in un singolo blocco.
- **gridDim**: numero di blocchi contenuti nella grid.

3.2.4 Organizzazione della memoria

Esistono diversi tipi di memorie con diverse latenze d'accesso all'interno della GPU:

- **Register memory**: spazio a bassa latenza, privato per ogni singolo processore e accessibile da un solo thread. I registri vengono suddivisi fra tutti i thread residenti.
- **Local memory**: spazio privato per ogni singolo thread in cui vengono memorizzate le variabili locali utilizzate durante l'esecuzione di un kernel.
- **Shared memory**: area a bassa latenza d'accesso condivisa fra tutti i thread di uno stesso blocco e da tutti i processori di uno stesso multiprocessore. Questa memoria è particolarmente veloce, i tempi d'accesso sono leggermente superiori a quelli dei registri. I thread di uno stesso blocco vedono la stessa shared memory grazie alla quale possono collaborare.
- **Constant memory**: area read-only per la lettura accelerata accessibile da tutti i thread. Spazio di memoria a latenza più bassa della global memory e fornito di una cache.
- **Texture memory**: area read-only ottimizzata per la lettura e accessibile da tutti i thread. E' una interfaccia alla memoria globale che offre un meccanismo di caching ottimizzato.
- **Global memory**: Area read/write esterna ai multiprocessori e condivisa fra i thread. Il tempo di accesso a questa memoria è centinaia di volte superiore rispetto a quello dei registri

Uno degli aspetti fondamentali per realizzare dei programmi CUDA con performance soddisfacenti è che non tutte le memorie sono uguali ma bisogna cercare di utilizzare al meglio ogni tipo di memoria presente. Il punto cruciale consiste nel minimizzare gli accessi alla memoria globale sfruttando il più possibile la memoria shared.

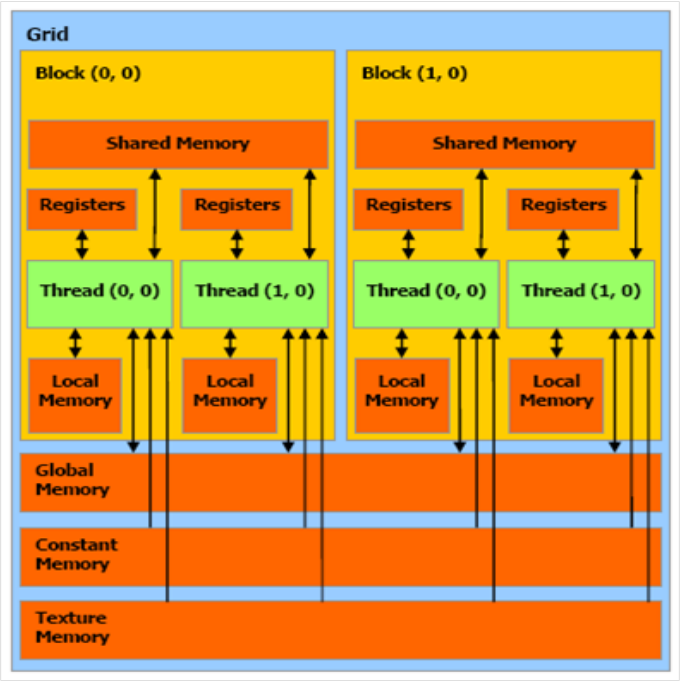


Figura 3.3: *In figura è mostrata l'organizzazione della memoria per ogni blocco di thread all'interno della griglia nell'architettura CUDA.*

3.2.5 Compilazione di un programma CUDA

nvcc è un compilatore che separa le parti per la CPU e per la GPU

- Il codice per la CPU viene compilato con il compilatore di sistema (*Visual C per Windows*, o *GCC per Linux/Unix*)
- Il codice per la GPU viene compilato con il CUDA compiler che genera un binario di tipo **Parallel Thread eXecution** (PTX) rappresentante l'Instruction Set Virtuale per le GPUNVIDIA, definendone anche il modello di programmazione, le risorse di esecuzione e lo stato.

- Un ulteriore traduttore (o compilatore stesso, o l'interprete a runtime di CUDA) trasforma il codice PTX nel codice binario dell'architettura fisica target.

3.3 Algoritmo di PCA locale overcomplete in ambiente GPU-CUDA

Come già detto nel paragrafo relativo alla struttura di una immagine *DWI*, questo tipo di immagini mostrano una rappresentazione tridimensionale di una sezione del corpo umano nella stessa maniera di una classica immagine di risonanza magnetica, quindi hanno di base una natura prettamente 3D. Però, la particolarità delle immagini pesate in diffusione è che presentano una struttura **multi-direzionale** che conferisce loro una quarta dimensione attraverso la quale è possibile osservare un voxel (i, j, z) in tutte le sue k direzioni. Pertanto, alla luce di quanto appena ribadito, l'algoritmo parallelo sviluppato, prevede una griglia computazionale tridimensionale grande almeno quanto la base 3D dell'immagine DWI. In questo modo ad ogni thread sarà associato un voxel (i, j, z, k) in tutte le sue k direzioni.

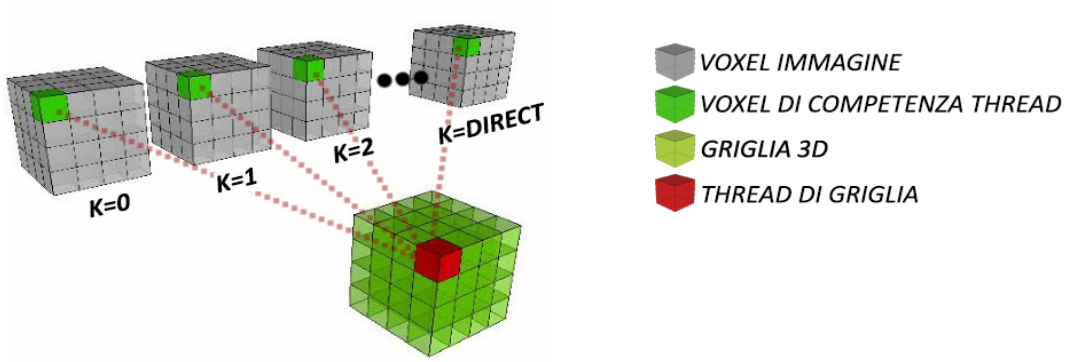


Figura 3.4: *Rappresentazione 3D dell'immagine DWI e della griglia computazionale che mostra la suddivisione dei dati dell'immagine per ogni thread della griglia. Nella griglia colorata in verde chiaro, è evidenziato un thread in rosso che ha come elementi di competenza i voxel colorati in verde scuro sui cubi grigi che rappresentano i voxel appartenenti alle immagini direzionali DWI.*

In figura si può osservare la griglia di thread tridimensionale di colore verde chiaro mentre l'immagine DWI 4D è rappresentata dai cubi in grigio chiaro dove ogni cubo rappresenta una direzione dell'immagine. Nella griglia è evi-

denziato un thread in rosso mentre in verde più scuro nell'immagine DWI sono evidenziati i voxel di sua competenza nelle k direzioni disponibili.

In questo lavoro di tesi sono state sviluppate 2 versioni dell'algoritmo parallelo, differenti per numero di kernel utilizzati, gestione della memoria disponibile su GPU (shared e global memory) e trasferimento dati host/device e viceversa.

- **Versione 1 - full shared memory:** La prima versione dell'algoritmo parallelo fa un uso totale della *shared memory* in tutte le principali operazioni di PCA locale: costruzione della matrice X, calcolo della media $Mean$, calcolo della matrice di covarianza C, calcolo della matrice Y, thresholding e ricostruzione dei dati nella matrice $\hat{X}$ con la PCA inversa. Utilizzando a pieno questo tipo di memoria, si sfrutta la sua bassa latenza d'accesso ottenendo quindi degli accessi più veloci ai dati ed un risparmio maggiore di tempo d'esecuzione con un guadagno in termini di prestazioni più elevato. Purtroppo, la grossa limitazione di questa versione parallela, in relazione al tipo di problema affrontato, è legata alla sua capacità ridotta in termini di spazio, che non consente di poter eseguire l'algoritmo su immagini con un numero troppo grande di direzioni dato che le strutture utilizzate per i calcoli di PCA locale, sono influenzate nella loro dimensione dal numero di direzioni dell'immagine. Per superare questo tipo di limitazione, che ha reso impossibile testare l'algoritmo con un numero certamente più idoneo di direzioni dell'immagine, è stata sviluppata una seconda versione.

- **Versione 2 - shared and global memory combination:** La seconda versione dell'algoritmo parallelo differisce principalmente dalla prima per la gestione più ridotta ed oculata della *shared memory* combinata all'utilizzo della *global memory* che è certamente una memoria più capiente. Con questa versione quindi, alcune strutture dati come $(X, mean, W)$ sono memorizzate nella shared memory mentre quelle più dipendenti dal numero di direzioni o comunque più grandi, sono allocate nella global memory. La limitazione di questa versione è senza dubbio legata ai tempi di accesso della global memory più alti rispetto a quelli della shared memory e quindi la combinazione di queste due memorie riduce senz'altro il vantaggio di utilizzare una memoria a bassa latenza. Però, come vedremo nel capitolo successivo relativo ai risultati sperimentali, questa differenza non è così netta e non incide più di tanto sul tempo d'esecuzione complessivo, almeno per il numero di direzioni testate. Il vantaggio evidente invece è che questa versione è meno soggetta ai limiti di spazio della shared memory perché si appoggia in parte ad una memoria decisamente più capiente, consentendo quindi di testare l'algoritmo su più

immagini direzionali per valuare la scalabilità e i vantaggi rispetto alla sua controparte sequenziale.

Un aspetto importante da considerare è legato al calcolo degli autovettori nella PCA locale. Infatti, per calcolare gli autovettori è necessario utilizzare una qualche libreria che disponga di una o più routine predisposte per questo tipo di operazione. Il calcolo andrebbe fatto all'interno del kernel in concomitanza con i prodotti tra matrici della PCA locale in modo da eseguirlo in parallelo, al fine di sfruttare il potenziale della GPU. Tecnicamente, è necessario invocare una *routine di libreria esterna per l'algebra lineare accelerata per GPU* (a tutti gli effetti un kernel), all'interno del kernel chiamante. Questa possibilità è stata introdotta da **CUDA** a partire dalla **versione 5.0** con il **Dynamic Parallelism** (*Parallelismo Dinamico*) [18] che è una estensione al modello di programmazione CUDA con la quale si può invocare un kernel figlio all'interno di un kernel padre, creando così una gerarchia di griglie computazionali. Il problema è che questa estensione è utilizzabile soltanto su schede video che possiedono una **compute capability uguale o superiore a 3.5** e purtroppo la scheda video utilizzata per gli esperimenti non soddisfa tali requisiti. Pertanto, non è stato possibile utilizzare questa tecnica decisamente vantaggiosa ma la forte limitazione è stata aggirata spostando il calcolo degli autovettori tramite *SVD* sull'host dove non c'è alcun tipo di problema nell'invocare routine di librerie esterne. Chiaramente, questo tipo di limite rappresenta un vero e proprio imbuto per l'applicazione parallela e porta diversi svantaggi:

1. Il calcolo viene effettuato in maniera sequenziale su *CPU* e non in parallelo, provocando un rallentamento che incide inevitabilmente sulle prestazioni globali dell'applicazione.

2. Un ulteriore rallentamento nei tempi d'esecuzione è dovuto al trasferimento dei dati tra host e device e viceversa, per consentire ai kernel di utilizzare gli autovettori calcolati sull'host.

Indipendentemente dalla versione sviluppata, l'algoritmo parallelo consiste di una fase di inizializzazione che resta sequenziale e prevede il caricamento delle intensità dei voxel da file su di opportune strutture dati e la stima del rumore Rician con il metodo *MUBE* (box colorati in azzurro nel diagramma di flusso in Figura 2.1). Poi, come detto, c'è una fase intermedia tra due kernel in cui avviene sequenzialmente il calcolo degli autovettori. Le due versioni dell'algoritmo parallelo sono articolate rispettivamente in 3 e 4 kernel. Il primo e l'ultimo kernel sono perfettamente identici in entrambe le versioni, che invece

differiscono nell'utilizzo dei kernel intermedi poichè la seconda versione splitta i due prodotti della PCA locale in due kernel differenti, al fine di ridurre il carico della shared memory.

Di seguito, nelle sezioni successive verranno illustrate le due versioni dell'algoritmo con un'ampia descrizione dei kernel sviluppati. Infine, il capitolo terminerà con un'analisi sul calcolo degli autovettori su host.

3.4 Algoritmo parallelo versione 1: full shared memory

La versione parallela **full shared memory** (*uso totale di memoria condivisa*) utilizza 3 kernel per implementare in ambiente *GPU-CUDA*, il metodo PCA locale overcomplete per la rimozione del rumore Rician da immagini pesate in diffusione. Come detto nell'introduzione a questo capitolo, la versione 1 esegue le operazioni dell'algoritmo utilizzando a pieno la *shared memory* e sfruttando quindi la bassa latenza di accesso di questa memoria.

Ecco di seguito i kernel sviluppati per questa versione:

1. **computeCovariance**$<<<-,-,->>>$: Costruisce localmente la matrice x, calcola la media di ogni colonna di x in *mean* e calcola quindi la matrice di covarianza *cov* (riquadro arancione in Figura 2.1).
2. **computePCA**$<<<-,-,->>>$: Calcola la Principal Component Analysis locale in maniera overcomplete (riquadro viola in Figura 2.1);
3. **weightedAverage**$<<<-,-,->>>$: Calcola la media pesata per la stima di ogni voxel nell'immagine di output restaurata (riquadro verde in Figura 2.1);

Di seguito vengono analizzati tutti i kernel nel dettaglio:

1. *computeCovariance*$<<<-,-,->>>$:

   Questo kernel effettua il calcolo della matrice di covarianza per ciascun voxel (i, j, z). La dimensione del blocco kernel è:

   $$nThreadPerBlocco.x * nThreadPerBlocco.y * nThreadPerBlocco.z$$

   mentre la dimensione della griglia è pari a:

   $$(\lfloor \frac{LENGTH}{nThreadPerBlocco.x} \rfloor \times \lfloor \frac{WIDTH}{nThreadPerBlocco.y} \rfloor \times \lfloor \frac{DEPTH}{nThreadPerBlocco.z} \rfloor)$$

con LENGTH, WIDTH e DEPTH rispettivamente il numero di righe, colonne e profondità dell'immagine DWI.

Il kernel prende in input l'immagine rumorosa ed una struttura atta a contenere la matrice di covarianza per ciascun voxel (i, j, z). Questa struttura allocata in memoria globale, è stata utilizzata per avere a disposizione uno spazio in cui ciascun thread responsabile di un voxel dell'immagine nelle k direzioni disponibili, possa copiare la matrice di covarianza (calcolata in precedenza in memoria condivisa) per renderla disponibile sull'host e quindi per calcolare le matrici di autovettori dei voxel (i, j, z).

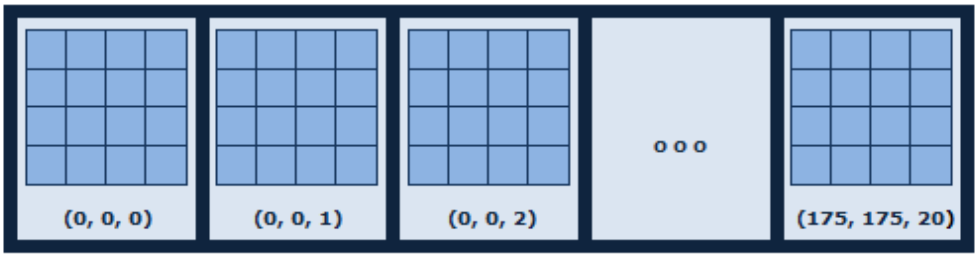


Figura 3.5: *Raffigurazione della struttura globale atta a contenere per ciascun voxel (i, j, z), la matrice di covarianza/autovettori (in questo esempio specifico 4x4).*

Si tratta fondamentalmente di un *array stacked 3D* che ha tanti elementi quanti sono i voxel di una singola direzione dell'immagine pesata in diffusione ($176 \times 176 \times 21$). Per ogni elemento è allocata una matrice di covarianza che in un primo momento serve a memorizzare le covarianze, ma in seguito al calcolo degli autovettori su host, sarà utilizzata per contenere questi ultimi.

Nella shared memory invece, sono dichiarate: la matrice x, il vettore di media *mean* e la matrice di covarianza *cov*. Poichè questo tipo di memoria è condivisa tra i thread di un blocco della griglia, le strutture utilizzate al suo interno hanno una dimensione tale da contenere i dati per ogni thread in tutte le direzioni k. Ad esempio la struttura x in shared memory contiene la matrice X per ogni thread del blocco, disposta in una opportuna locazione accessibile dal quel thread specifico secondo una regola di indicizzazione ben precisa. Questa strategia di memorizzazione,

è stata utilizzata per tutte le strutture che risiedono in shared memory all'interno dei diversi kernel sviluppati.

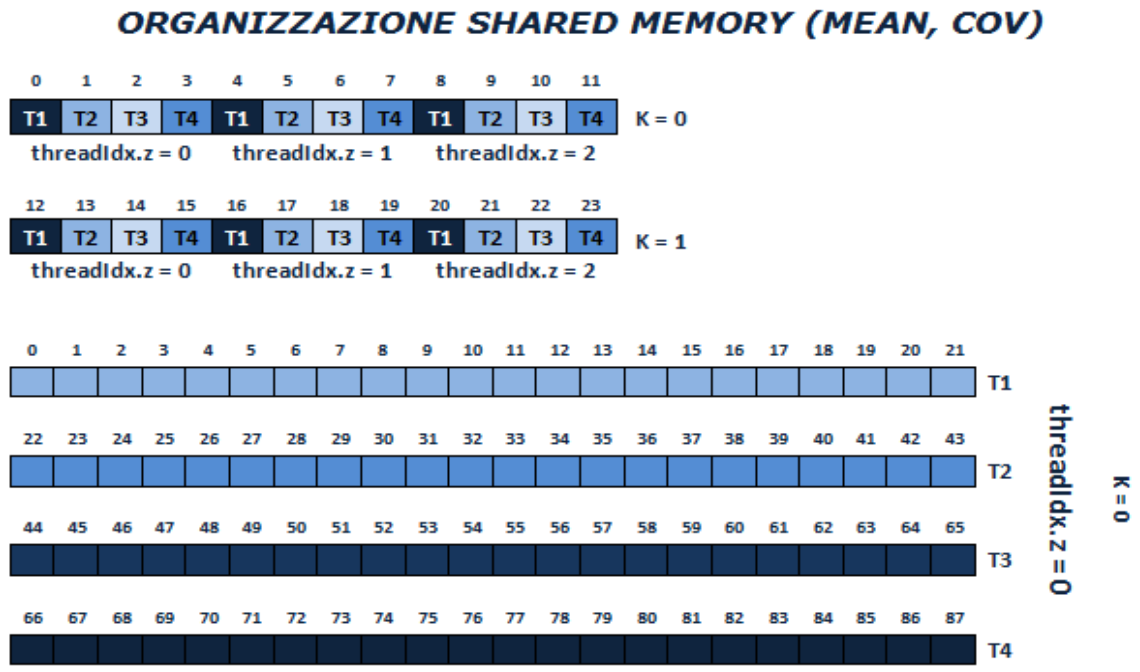


Figura 3.6: *Organizzazione delle strutture dati mean e cov nella shared memory. L'esempio riporta per semplicità solo una parte della memoria necessaria a memorizzare 2 direzioni nel caso della struttura Mean e 1 sola direzione per cov. Le celle di memoria sono colorate in maniera diversa per evidenziare gli elementi di competenza per ciascun thread. Nella struttura di cov (covarianza) sono riportati soltanto gli elementi di competenza del blocco di thread con threadIdx.z=0, per una questione di semplicità nella visualizzazione.*

Vediamo ora il funzionamento di questo kernel: per prima cosa, ciascun thread di un blocco copia alcuni dati dall'immagine rumorosa presente in memoria globale e li memorizza in memoria condivisa x, in modo tale che il blocco di thread abbia a disposizione in shared memory, le matrici X relative ai voxel di loro competenza (la strategia di copia verrà affrontata in dettaglio nel paragrafo successivo). A questo punto, i thread del blocco, calcolano in parallelo la media di ogni colonna della propria matrice X estratta da x e memorizzano queste medie nel vettore di media *mean* (l'organizzazione di questa struttura in shared memory è stata mostrata in Figura 3.6). Una volta costruita la X e calcolata la media

delle colonne, tutti i thread sono pronti a calcolare in parallelo la propria matrice di covarianza utilizzando i dati appena calcolati, secondo la formula $C = X^T X$ (l'organizzazione della matrice di covarianza in shared memory è mostrata in Figura 3.6). Per terminare, ogni thread accede alla propria matrice di covarianza appena calcolata in shared memory e copia i valori nella struttura globale adibita alla memorizzazione della covarianza per ogni voxel (La Figura 3.5 mostra il modo in cui è organizzata tale memoria).

Di seguito è mostrato l'algoritmo in pseudocodice del kernel *computeCovariance*, che schematizza quanto è stato detto (le operazioni mostrate sono da intendersi eseguite in parallelo da ogni thread di un blocco della griglia computazionale):

Algorithm 1 Pseudocodice del kernel computeCovariance

1: Dichiarazione dati shared memory per x, *mean* e *cov*
2: Identifica voxel (i, j, z) di competenza di un thread del blocco
3: Costruzione della matrice X in x
4: Calcolo della media di ogni colonna di X in *mean*
5: Calcolo della matrice di covarianza $C = X^T X$ in *cov*
6: Copia della matrice di covarianza da shared a global memory

2. *computePCA*$<<<-,-,->>>$:

   In questo kernel i thread per blocco e i blocchi nella griglia sono organizzati allo stesso modo del kernel *computeCovariance*.

   In input, il kernel prende l'immagine rumorosa, la struttura contenente le matrici di autovettori per ogni voxel (i, j, z) precedentemente calcolate sull'host in sequenziale, la matrice sigma (176×176) contenente la stima del rumore Rician per ogni voxel (i, j) lungo una immagine DWI (la stima è la stessa per ogni z e per ogni k, fissato (i, j)) e poi infine l'immagine di output da modificare tramite la tecnica di overlapping. In shared memory, sono dichiarate: la matrice x, il vettore di media *mean* e le matrici y, w e xd. Si ricorda, che le strutture utilizzate all'interno della shared memory, hanno una dimensione tale da contenere i dati per ogni thread del blocco ed in tutte le direzioni k.

   Vediamo ora il funzionamento di questo kernel: così come il kernel *computeCovariance* anche in questo caso c'è una prima fase in cui i thread di

ogni blocco della griglia, costruiscono le matrici X nella shared memory x copiando gli elementi dall'immagine rumorosa attraverso la strategia di copia che verrà illustrata a breve nel prossimo paragrafo. Successivamente i thread calcolano la media di ogni colonna della propria matrice X di competenza (indicizzata in maniera specifica per ogni thread) in *mean*. La fase di copia termina dopo che tutti i thread del blocco hanno copiato la matrice degli autovettori da global a shared memory in *w*. A questo punto in sequenza, ciascun thread del blocco calcola in parallelo, il prodotto fra matrici per il calcolo della matrice Y in *y*, il thresholding della matrice Y ed il prodotto righe per colonne per eseguire la *PCA inversa* che ricostruisce i dati originali restaurati $\hat{X}$ in *xd*. Infine, sempre in parallelo, ciascun thread effettua **overlapping** copiando i dati restaurati nell'immagine di output che si trova in memoria globale. Siccome la *tecnica di sovrapposizione* prevede l'aggiornamento dei voxel che ricoprono la funzione di *accumulatore*, è possibile che più thread accedano contemporaneamente ad una stessa locazione, causando **race condition**. Pertanto, questo tipo di operazione di aggiornamento della somma e dell'incremento del numero di sovrapposizioni (come si è mostrato nell'algoritmo sequenziale) è stata realizzata utilizzando la routine *atomicAdd* di *CUDA*, la quale consente di effettuare l'operazione in modo atomico (*in mutua esclusione*), evitando quindi i problemi di race condition che portano alle inconsistenze nei risultati.

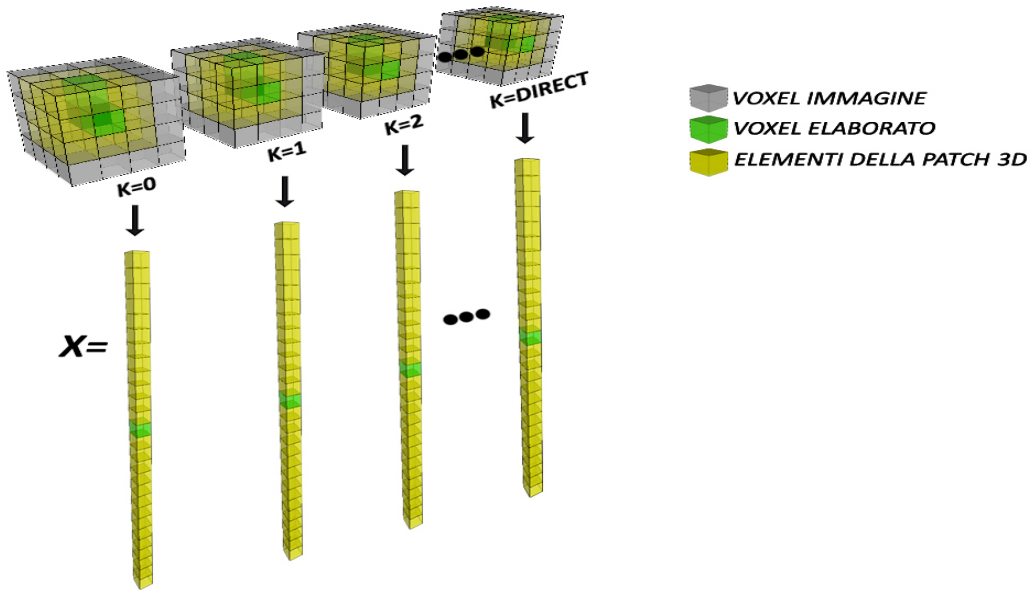


Figura 3.7: *Rappresentazione 3D del processo di costruzione della matrice X locale e del processo di overlapping, che altro non è che il processo inverso.*

Di seguito è mostrato l'algoritmo in pseudocodice del kernel *computePCA*, che schematizza quanto è stato detto (le operazioni mostrate sono da intendersi eseguite in parallelo da ogni thread di un blocco della griglia computazionale):

Algorithm 2 Pseudocodice del kernel computePCA

1: Dichiarazione dati shared memory per x, $mean$, y, w e xd
2: Identifica voxel (i, j, z) di competenza di un thread del blocco
3: Costruzione della matrice X in x
4: Calcolo della media di ogni colonna di X in $mean$
5: Copia degli autovettori da global a shared memory in w
6: Calcolo del prodotto righe per colonne $Y = X * W$ in y per la generazione delle nuove coordinate dei dati
7: Sogliatura sulla matrice Y per la rimozione del rumore Rician locale
8: Calcolo del prodotto righe per colonne $\hat{X} = Y * W^T$ in xd per la ricostruzione dei dati originali restaurati
9: Overlapping per la ricostruzione dell'immagine di output restaurata

3. *weightedAverage*$<<<-,-,->>>$:

   La struttura del kernel in termini di blocchi della griglia e di thread per blocco è uguale ai primi due kernel illustrati precedentemente.

   Il kernel prende in input soltanto l'immagine di output restaurata, che deve essere ulteriormente elaborata come processo di *post-elaborazione* della PCA. Non è utilizzata la shared memory in questo caso ma ciascun thread accede in memoria globale ai suoi k elementi di competenza nell'immagine di output e quindi esegue la media pesata dividendo il valore del voxel (pari alla somma di tutte le stime del voxel dovute alla sovrapposizione tra le patch delle finestre locali dei voxel limitrofi) per il numero di sovrapposizioni in ogni direzione k.

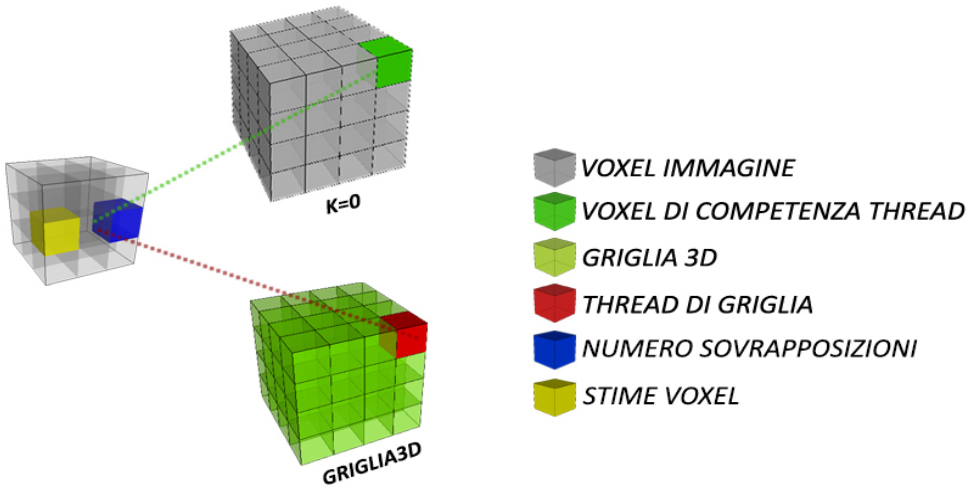


Figura 3.8: *Rappresentazione 3D dell'operazione di media pesata svolta da ogni thread della griglia computazionale. Ciascun thread, agisce sull'immagine di output per ogni voxel di sua competenza in tutte e K le direzioni. Per ogni direzione, effettua una divisione tra le stime dei voxel ottenute dal processo di overlapping e il numero di sovrapposizioni generate, calcolando così la stima finale per ciascun voxel.*

Di seguito il codice implementato per sviluppare questo kernel:

Algorithm 3 Pseudocodice del kernel weightedAverage

```
1: Identifica voxel (i, j, z) di competenza di un thread del blocco
2: for V_k (voxel) in tutte le direzioni K do
3:     Dividi il valore del voxel V_k per il numero di sovrapposizioni
4: end for
```

3.4.1 La strategia di copia dei dati per la costruzione della matrice X in shared memory

Quando si sviluppa un algoritmo parallelo in ambiente **GPU-CUDA**, nella maggior parte dei casi è possibile risolvere il problema attraverso varie strategie. A fare la differenza è il grado di ottimizzazione che mira ad avere risultati di qualità in termini di tempo e di spazio. Una delle possibilità per ottimizzare un algoritmo parallelo, è senza dubbio legata al coinvolgimento della *shared memory* nelle operazioni principali, in modo tale da sfruttare la bassa latenza

d'accesso di questa memoria al fine di ottenere dei tempi ancora più ridotti. L'utilizzo della shared memory comporta la progettazione di una strategia in termini di copia dei dati e di operazioni, che a volte può anche essere molto complessa ma ripaga il progettista nei risultati ottenuti. In questo lavoro di tesi da me svolto, una delle parti più complesse nella progettazione dell'algoritmo parallelo, è stata la costruzione della matrice X all'interno della shared memory.

Si ricorda che, la matrice X è costruita localmente per ogni voxel (i, j, z) all'interno del processo di *PCA locale*. La costruzione prevede l'estrazione per ogni direzione k, di una finestra locale $3 \times 3 \times 3$ centrata nel voxel (i, j, z, k). Ogni finestra locale compone una colonna della matrice X che avrà quindi tante colonne quante sono le immagini direzionali e un numero di righe pari alla dimensione della finestra locale, ovvero 27. Pertanto, per ogni voxel (i, j, z), la matrice X incapsula tutte le k direzioni. Questo significa che ogni thread della griglia che è responsabile di un voxel (i, j, z) in tutte le k direzioni, agirà sulla propria matrice X di competenza. Siccome la *shared memory* è una memoria condivisa tra i thread di un blocco, questa memoria deve contenere la matrice X per ogni thread del blocco stesso.

Consideriamo per semplicità il problema in due dimensioni con un blocco da 2×2 thread e vediamo di quanto deve essere la dimensione della shared memory: l'esigenza che ha ogni thread del blocco è quella di poter estrarre la finestra locale 3×3 centrata nel voxel di sua competenza. Ciò significa che, centrando la finestra locale sull'elemento di competenza di ogni thread, si vanno a considerare gli elementi che si trovano nell'intorno rettangolare di raggio 1 che possono essere sia elementi appartenenti all'immagine oppure elementi fuori bordo che in tal caso sono considerati pari a zero. Alla luce di questo ragionamento, considerando un blocco da 2×2 thread, la dimensione della shared memory deve essere pari a $(2+1) * (2+1)$ elementi, che corrisponde ad aggiungere un padding di 1 al blocco 2×2 di voxel di competenza dei thread del blocco stesso. La logica non cambia nel caso di utilizzo di blocchi da 3×3, 4×4, 5×5 thread, etc. Formalizzando questo concetto, la dimensione della shared memory (nel caso bidimensionale) deve essere pari a:

$$(nThreads.x + 2) * (nThreads.y + 2)$$

Quanto appena formulato, è sufficiente a memorizzare per ogni thread del blocco, una finestra locale 3×3 centrata sul proprio voxel di competenza. In realtà, ciascun thread è responsabile di un voxel in tutte le k direzioni disponibili, quindi è necessario memorizzare $k = DIRECT$ finestre locali 3×3 per ogni thread del blocco. La dimensione quindi diventa:

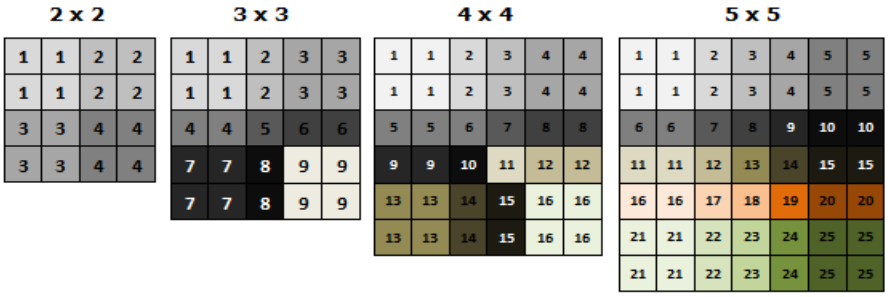


Figura 3.9: *Raffigurazione della shared memory bidimensionale, per diverse combinazioni di blocchi di thread: 2x2, 3x3, 4x4, 5x5. Nel problema 3D, questa shared memory rappresenta soltanto una porzione di immagine lungo un piano z.*

$$(nThreads.x + 2) * (nThreads.y + 2) * DIRECT$$

Bisogna infine considerare il problema dal punto di vista tridimensionale poichè ogni direzione è composta da 21 immagini. I blocchi di thread utilizzati sono quindi tridimensionali ed in questo caso particolare, la dimensione del blocco lungo l'asse z è stata scelta pari a tre in maniera tale da dividere perfettamente le 21 immagini in profondità. Ogni thread del blocco 3D deve prelevare k finestre locali di dimensione $3 \times 3 \times 3$ e per farlo ha bisogno di estrarre elementi dall'immagine $(z - 1)$ e dall'immagine $(z + 1)$ oltre che dall'immagine z dove è presente il suo voxel di competenza, il tutto appunto per ogni direzione k. Considerando che i blocchi di thread lungo la terza dimensione sono 3, per garantire che qualsiasi thread abbia a disposizione tutti gli elementi della finestra locale $3 \times 3 \times 3$ in tutte e k le direzioni, è necessario che la dimensione della shared memory sia 5 volte più grande rispetto all'ultima formulata, che equivale ad aggiungere un padding di $2 * (nThreads.x + 2) * (nThreads.y + 2)$ lungo la terza dimensione $(3 + 2 = 5)$. Quindi la dimensione definitiva per la shared memory, necessaria per la costruzione della matrice X per ogni blocco di thread tridimensionale è:

$$(nThreads.x + 2) * (nThreads.y + 2) * (nThreads.z + 2) * DIRECT$$

Esaminata la dimensione della shared memory, non resta che vedere qual è la strategia di copia utilizzata. La memoria è organizzata tenendo conto delle sovrapposizioni tra finestre locali, infatti il carico di lavoro per la copia degli elementi è bilanciato tra i thread, i quali si trovano quindi a copiare anche elementi che saranno utilizzati da altri thread appartenenti allo stesso blocco.

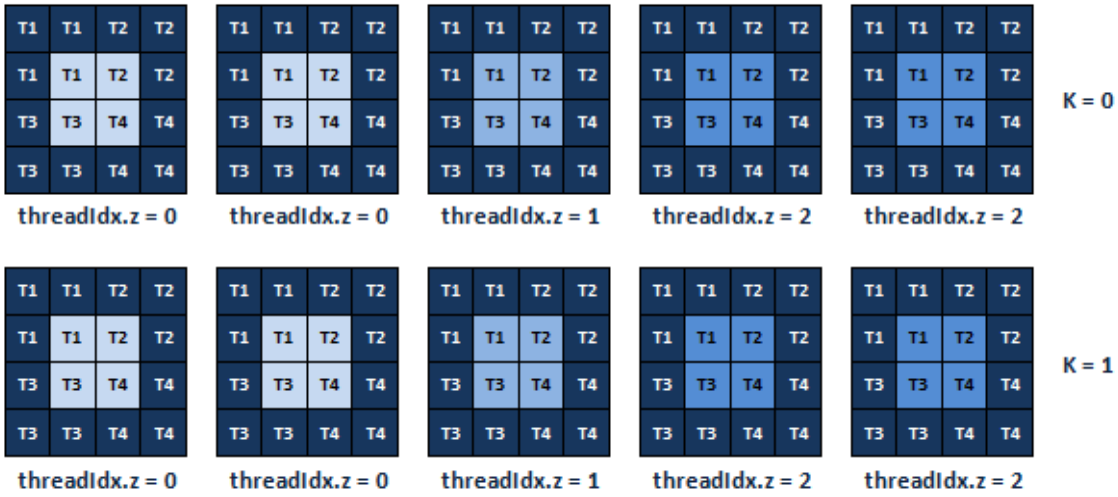


Figura 3.10: *Rappresentazione della matrice X in shared memory, per un blocco di thread. Nell'esempio è mostrata soltanto una parte della shared memory, necessaria a memorizzare le porzioni di immagini per due direzioni. I blocchi di thread con threadIdx.z differenti sono colorati con tonalità diverse per evidenziare i diversi piani z. In blu scuro sono riportati gli elementi di padding e all'interno di ogni cella è indicato l'identificativo del thread responsabile della copia dell'elemento secondo la strategia utilizzata.*

Ciascun thread del blocco copia innanzitutto il suo elemento di competenza per ogni direzione k. A questo punto mancano gli elementi del padding, gestiti in maniera differente tra angoli e il resto dei bordi rimanenti.

Gli elementi presenti nei 4 angoli di ogni piano della shared memory sono copiati da thread specifici:

- L'angolo in alto a sinistra è di competenza del thread con identificativo: $threadIdx.x = 0$ e $threadIdx.y = 0$
- L'angolo in alto a destra è di competenza del thread con identificativo: $threadIdx.y = blockDim.y - 1$ e $threadIdx.x = 0$
- L'angolo in basso a sinistra è di competenza del thread con identificativo: $threadIdx.y = 0$ e $threadIdx.x = blockDim.x - 1$
- L'angolo in basso a destra è di competenza del thread con identificativo: $threadIdx.x = blockDim.x - 1$ e $threadIdx.y = blockDim.y - 1$

Così come per i 4 angoli, anche gli elementi presenti nei bordi (escluso gli angoli) sono copiati da thread specifici:

- I thread con identificativo ($threadIdx.x = 0$), i cui elementi di competenza si trovano nella riga 2 (considerando la porzione comune ai thread di un piano z), copiano gli elementi presenti alla riga 1.
- I thread con identificativo ($threadIdx.x = blockDim.x - 1$), i cui elementi di competenza si trovano nella penultima riga, copiano gli elementi presenti nell'ultima riga.
- I thread con identificativo ($threadIdx.y = 0$), i cui elementi di competenza si trovano in colonna 2, copiano gli elementi presenti nella colonna 1.
- I thread con identificativo ($threadIdx.y = blockDim.y - 1$), i cui elementi di competenza si trovano in penultima colonna, copiano gli elementi presenti nell'ultima colonna.

Dato che la shared memory ha una profondità pari a 5 porzioni di immagini locali della dimensione di $(nThreads.x + 2) * (nThreads.y + 2)$, le 3 porzioni centrali sono state copiate dai thread con identificativi rispettivamente $threadIdx.z = 0, 1, 2$ poichè di loro competenza. Le altre due porzioni aggiunte come padding, non sono di competenza diretta di alcun thread del blocco 3D. Pertanto i thread con identificativo $threadIdx.z = 0$ ripetono le stesse operazioni appena descritte, anche per la prima porzione (elementi di immagine $z - 1$) mentre i thread con identificativo $threadIdx.z = 2$ ripetono le stesse operazioni anche per l'ultima porzione (elementi di immagine $z + 1$).

3.5 Algoritmo parallelo versione 2: shared and global memory combination

La seconda versione **shared and global memory combination** (*uso combinato di memoria condivisa e globale*) dell'algoritmo parallelo prevede un uso combinato di *shared e global memory*, al fine di ottimizzare lo spazio limitato della shared memory, superando per certi versi uno dei limiti della prima versione.

Ecco di seguito i kernel sviluppati per questa versione:

1. **computeCovariance**$<<<-,-,->>>$: Costruisce localmente la matrice x, calcola la media di ogni colonna di x in *mean* e calcola quindi la matrice di covarianza *cov* (riquadro arancione in Figura 2.1).

2. **computeProdYEXW**<<< −, −, − >>>: Calcola la matrice y ed effettua il thresholding su di essa (riquadro viola in Figura 2.1);

3. **computeProdXDEYWT**<<< −, −, − >>>: Calcola la PCA inversa ricostruendo la matrice $\hat{X}$ restaurata (xd) ed effettua overlapping (riquadro viola in Figura 2.1).

4. **weightedAverage**<<< −, −, − >>>: Calcola la media pesata per la stima di ogni voxel nell'immagine di output restaurata (riquadro verde in Figura 2.1);

E' possibile notare che il primo ed il quarto kernel sono perfettamente identici a quelli della prima versione, pertanto la descrizione verterà soltanto sui due kernel centrali. L'unica particolarità in termini di kernel per questa versione, è lo split del kernel *computePCA* della versione 1, nei due kernel *computeProdYEXW* e *computeProdXDEYWT* per ridurre appunto lo spazio utilizzato nella shared memory. Chiaramente questa suddivisione porta anche a più trasferimenti di dati da *host a device* e viceversa, per poter usufruire delle strutture nei diversi kernel. Nei primi tre kernel, le operazioni di inizializzazione che coinvolgono la memoria condivisa sono le stesse: la costruzione della matrice X, il calcolo della media di ogni colonna della matrice X nel vettore di media *mean* e la copia degli autovettori da global a shared memory. Queste operazioni sono ripetute perchè in tutti e tre i kernel si necessita della disponibilità di tali strutture, per le quali si preferisce ogni volta che siano ricalcolate sfruttando la velocità di accesso della shared memory piuttosto che salvarle nella global memory, occupando spazio di memoria per tutto il ciclo di vita dell'applicazione parallela.

Di seguito vengono analizzati i due kernel che rendono le versioni differenti:

- *computeProdYEXW*<<< −, −, − >>>:

  Nel kernel *computeProdYEXW* la configurazione della griglia è la medesima dei kernel della versione 1.

  In input, il kernel prende l'immagine rumorosa, la struttura contenente le matrici di autovettori per ogni voxel (i, j, z) precedentemente calcolate sull'host in sequenziale, la matrice sigma contenente la stima del rumore Rician per ogni voxel (i, j) lungo una immagine DWI (la stima è la stessa per ogni z e per ogni k, fissato (i, j)) e poi infine la struttura presente in global memory e atta a contenere le matrici Y per ogni voxel (i, j, z). Nella shared memory, sono dichiarate: la matrice x, il vettore di media

mean e la matrice degli autovettori w che come detto in precedenza, sono le uniche tre strutture utilizzate in shared memory, dal momento che sono fondamentali per effettuare le operazioni intermedie.

Vediamo ora il funzionamento di questo kernel: c'è una prima fase in cui i thread costruiscono le matrici X nella struttura condivisa x copiando gli elementi dall'immagine rumorosa (secondo la strategia già illustrata nella sezione dedicata alla prima versione), calcolano la media di ogni colonna della propria matrice X di competenza in *mean* e copiano gli autovettori w da global a shared memory. Terminata questa fase di inizializzazione dei dati, ciascun thread in parallelo effettua prima il prodotto tra matrici per generare la matrice Y e poi effettua il thresholding su di essa per rimuovere il rumore *Rician* localmente. Il prodotto righe per colonne in parallelo, come già detto più volte, fa un uso combinato della shared memory e della global memory, infatti: effettua il prodotto con le matrici x e w che sono in *shared memory* ed i risultati sono assegnati a y che invece è in global memory.

Di seguito è mostrato l'algoritmo in pseudocodice del kernel *computeProdYEXW*, che schematizza quanto è stato detto (le operazioni mostrate sono da intendersi eseguite in parallelo da ogni thread di un blocco della griglia computazionale):

Algorithm 4 Pseudocodice del kernel computeProdYEXW

1: Dichiarazione dati shared memory per x, *mean* e w
2: Identifica voxel (i, j, z) di competenza di un thread del blocco
3: Costruzione della matrice X in x
4: Calcolo della media di ogni colonna di X in *mean*
5: Copia degli autovettori da global a shared memory in w
6: Calcolo del prodotto righe per colonne $Y = X * W$ in y per la generazione delle nuove coordinate dei dati
7: Sogliatura sulla matrice Y per la rimozione del rumore Rician locale

- *computeProdXDEYWT*$<<<-,-,->>>$:

  Questo kernel ha la stessa configurazione di griglia del kernel precedente e prende in input più o meno anche gli stessi parametri: immagine rumorosa, struttura contenente gli autovettori, struttura contenente le matrici Y, struttura contenente le matrici XD e infine l'immagine di output sulla quale effettuare overlapping.

Vediamo ora il funzionamento del kernel: anche in questo caso c'è una prima fase di inizializzazione uguale al kernel precedente. Terminata questa fase, ciascun thread in parallelo effettua il prodotto righe per colonne per calcolare la matrice $\hat{X}$ in *xd*, alla quale contemporaneamente somma per ogni suo elemento della colonna k-esima il k-esimo valore di media della colonna di X. In seguito, i thread effettuano overlapping per copiare i dati restaurati nelle corrette locazioni dell'immagine di output. Il prodotto righe per colonne fa un uso combinato della shared memory e della global memory dato che le matrici *xd* e *y* sono collocate in *global memory* mentre la matrice *w* ed il vettore *mean* sono memorizzati in *shared memory*.

Di seguito è mostrato l'algoritmo in pseudocodice del kernel *computeProdXDEYWT*, che schematizza quanto è stato detto (le operazioni mostrate sono da intendersi eseguite in parallelo da ogni thread di un blocco della griglia computazionale):

Algorithm 5 Pseudocodice del kernel computeProdXDEYWT

1: Dichiarazione dati shared memory per x, *mean* e w
2: Identifica voxel (i, j, z) di competenza di un thread del blocco
3: Costruzione della matrice X in x
4: Calcolo della media di ogni colonna di X in *mean*
5: Copia degli autovettori da global a shared memory in w
6: Calcolo del prodotto righe per colonne $\hat{X} = Y * W^T$ in xd per la ricostruzione dei dati originali restaurati
7: Overlapping per la ricostruzione dell'immagine di output restaurata

3.6 Calcolo degli autovettori W sull'host

Come già detto in questo capitolo, il calcolo degli autovettori al momento rappresenta un collo di bottiglia per l'algoritmo parallelo di questo lavoro di tesi. Non è infatti stato possibile invocare all'interno di un kernel, una routine di libreria accelerata per GPU e predisposta per tale calcolo. L'ambiente parallelo *CUDA* non offre la possibilità di richiamare delle routine accelerate nei kernel, a meno che non si utilizzino schede video più recenti con compute capability uguale o superiore a 3.5, che supportino il *Dynamic Parallelism* di cui abbiamo già parlato in precedenza. Per il lavoro svolto, si è avuto a disposizione una scheda video *NVIDIA Quadro K5000*, che purtroppo non dispone di questa

caratteristica e pertanto è stato necessario trovare una soluzione che consentisse in ugual modo di portare a termine l'algoritmo parallelo. Tale soluzione, ha previsto il calcolo degli autovettori sull'host (in sequenziale), utilizzando la routine *compute* del namespace **SVD** della libreria per elaborazioni grafiche *OpenCV*.

Di seguito, il codice sorgente utilizzato per realizzare questo calcolo:

```
Mat w, vt; // matrici per il calcolo SVD
for (int z = 0; z < DEPTH; z++) {
  for (int i = 0; i < LENGTH; i++) {
    for (int j = 0; j < WIDTH; j++) {
        // assegna puntatore della matrice covHost ad A (tipo Mat)
        Mat A(DIRECT, DIRECT, CV_32F, covHost[cmaps1(i,j,z)].cov);
        // calcolo SVD per A (A memorizza anche gli autovettori U)
        SVD::compute(A, w, A, vt);
        // copia autovettori nella matrice di covarianza covHost
        copyUinCOV(A.t(), covHost, i, j, z);
    }
  }
}
```

Come è facile vedere dal codice sorgente, per ogni voxel (i, j, z) si assegna il puntatore della matrice cov (contenuta come campo nella struttura covHost) ad A, dove A è una matrice di OpenCV. Questa tecnica risulta utile quando si vogliono processare dati esterni usando OpenCV, evitando quindi di effettuare una copia dei dati che influirebbe negativamente sul tempo della computazione. La matrice covHost è un array lineare in cui ogni elemento è una matrice cov che contiene la matrice di covarianza relativa al voxel (i, j, z). A questo punto si esegue il calcolo SVD con la routine *compute* sfruttando la matrice A di input per memorizzare gli autovettori di output della matrice U. Infine gli autovettori vengono copiati nuovamente nell'array covHost.

Questa strategia sopperisce al problema del calcolo degli autovettori ma porta un deficit nei tempi d'esecuzione globali dell'algoritmo parallelo che è proporzionale all'aumentare del numero di direzioni dell'immagine pesata in diffusione. Come verrà mostrato nel prossimo capitolo, le prestazioni dell'algoritmo sono comunque notevolmente vantaggiose rispetto alla controparte sequenziale ma è chiaro che senza questa limitazione potrebbero essere ancora migliori.

Capitolo 4

Risultati Sperimentali

Nell'ambito della tesi da me svolta sono state implementate sia la versione sequenziale su CPU che le due versioni parallele su GPU dell'algoritmo di PCA locale overcomplete per la rimozione di rumore Rician da immagini pesate in diffusione. Lo sviluppo ed i test dell'algoritmo sequenziale sono stati effettuati su di un Intel Core i3 CPU M 370 (2.40 GHz) con sistema operativo MS Windows 7 Home Premium 64-bit SP1, 4,0GB di memoria RAM e scheda video ATI Mobility Radeon HD 5470 utilizzando l'ambiente di sviluppo NetBeans 8.0. Per quanto riguarda le due versioni dell'algoritmo parallelo invece, lo sviluppo ed i test d'esecuzione sono stati effettuati utilizzando una macchina CPU Intel Core i7 (2.8GHz, 8GB Cache) con GPU Nvidia Quadro K5000.

Riportiamo di seguito i test effettuati e i risultati ottenuti in termini di prestazioni, velocità e accuratezza.

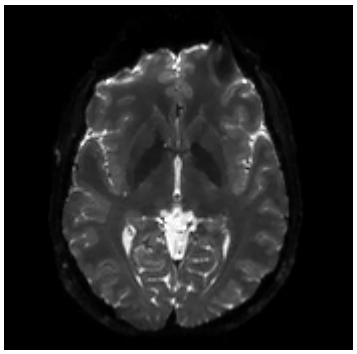
4.1 Analisi di accuratezza dei risultati

In questo paragrafo sono mostrati i test relativi all'accuratezza del metodo PCA locale overcomplete per la rimozione del rumore. In particolare, vengono presentati i risultati relativi all'errore di approssimazione commesso nell'immagine restaurata rispetto all'originale, in seguito al processo di denoising. L'errore viene calcolato attraverso degli indici che utilizzano come riferimento l'immagine ground-truth che è un'immagine campione, senza rumore, utilizzata per il confronto di qualità rispetto alla versione rumorosa e quella restaurata. I risultati ottenuti dai test di accuratezza, per l'algoritmo sequenziale e le due versioni dell'algoritmo parallelo, sono perfettamente identici tra loro.

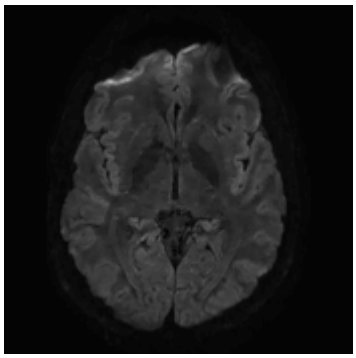
Le immagini DWI utilizzate in questa tesi sono dotate di 10 direzioni ed il metodo PCA locale è in grado di restaurare tutte le immagini per ogni z e per

ogni k, dove con z si indica la profondità dell'immagine pesata in diffusione e con k le sue direzioni. Data l'enorme mole di informazioni, nei test sperimentali relativi all'accuratezza, si è scelto di riportare i risultati ottenuti solo ad un fissato z, nelle direzioni $k = 0, 5, 9$.

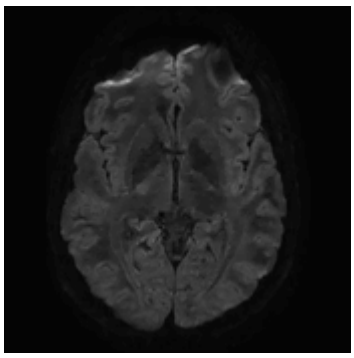
Nella figura 4.1, sono mostrate le 3 immagini ground-truth scelte:



(a) ground-truth (z=0, k=0)



(b) ground-truth (z=0, k=5)



(c) ground-truth (z=0, k=9)

Figura 4.1: Immagini di ground truth utilizzate

Gli indici di errore utilizzati sono:

- **Root Mean Square Error (RMSE)**: Radice dell'errore quadratico medio.
- **Peak Signal-Noise Ratio (PSNR)**: Picco del rapporto segnale rumore.

Nei successivi paragrafi verranno descritti i due indici di errore ed illustrati i relativi risultati di accuratezza.

4.1.1 Root Mean Square Error

In statistica, l'**Errore Quadratico Medio** (dall'inglese *Mean Squared Error, MSE*) misura la discrepanza quadratica media fra i valori dei dati osservati ed i valori dei dati stimati. La sua radice quadrata fornisce un ulteriore indice statistico chiamato **Radice dell'Errore Quadratico Medio** (dall'inglese *Root Mean Square Error, RMSE*) che corrisponde alla deviazione standard degli scarti. Nel contesto della rimozione del rumore dalle immagini, questo indice ci dice quanto è accurato l'algoritmo e quindi quanto è grande l'errore di approssimazione commesso. Più il valore del RMSE è prossimo a 0 e tanto più accurato è l'algoritmo.

Diamo un'occhiata alla formula che definisce l'indice RMSE:

$$\mathbf{RMSE} = \sqrt{\frac{\sum_{\mathbf{i=1}}^{\mathbf{n}} (\mathbf{x_i} - \hat{\mathbf{x}}_\mathbf{i})^\mathbf{2}}{\mathbf{n}}} \tag{4.1}$$

Va inoltre precisato che sia *MSE* che *RMSE* non sono quantità a-dimensionali per cui assumono l'unità di misura della grandezza considerata.

Di seguito riportiamo i valori di RMSE per le tre immagini campione utilizzate con $z = 0$ e $(k = 0, 5, 9)$ al variare del livello di rumore. Per ogni tabella, in prima colonna sono riportati i livelli di rumore presenti sull'immagine rumorosa che sono 3%, 5%, 7% e 9%. Nella seconda colonna abbiamo il fattore γ di threshold moltiplicato per la deviazione standard del rumore Rician locale σ con il quale abbiamo ottenuto i risultati illustrati. In terza colonna sono invece riportati i valori di RMSE ottenuti tra l'immagine ground-truth e quella rumorosa. L'ultima colonna invece, indica i valori di RMSE calcolati tra l'immagine ground-truth e quella restaurata.

Immagine k=0			
Livello rumore	**Fattore threshold**	**Truth-Noise**	**Truth-Denoise**
3%	3.5	2.74415	2.40756
5%	4	4.46429	4.03548
7%	5.5	5.98922	5.45814
9%	5.5	7.46004	5.79222

Immagine k=5			
Livello rumore	**Fattore threshold**	**Truth-Noise**	**Truth-Denoise**
3%	3.5	2.53526	1.21136
5%	4	3.78385	1.65182
7%	5.5	5.32402	1.98558
9%	5.5	6.80751	2.41109

Immagine k=9			
Livello rumore	**Fattore threshold**	**Truth-Noise**	**Truth-Denoise**
3%	3.5	2.48762	1.15479
5%	4	3.74874	1.59159
7%	5.5	5.24487	1.99902
9%	5.5	6.74083	2.38244

Guardando i risultati tabellari, è possibile notare che all'aumentare del livello di rumore, in tutte e tre le immagini, i valori della radice dell'errore quadratico medio tra l'immagine ground-truth e l'immagine rumorosa tendono ad aumentare poichè aumentando il rumore l'immagine risulta più degradata. Allo stesso modo, i valori di RMSE tra l'immagine ground-truth e l'immagine restaurata tendono certamente ad aumentare al crescere del livello di rumore ma il punto di confronto è con i valori di RMSE tra ground-truth e noise in quanto sono più bassi e quindi si avvicinano più velocemente a zero, segno che le immagini restaurate sono più accurate e più simili a quella ground-truth.

Un altro aspetto importante da notare è che le immagini gradiente restaurate per $k = 5$ e per $k = 9$, hanno dei valori di RMSE decisamente più bassi rispetto ai valori di RMSE dell'immagine rumorosa. Questo succede perchè le immagini gradiente hanno un rapporto segnale/rumore più basso rispetto alle immagini non-DWI e di conseguenza il miglioramento è più evidente sia da un punto di vista visivo ma soprattutto da un punto di vista numerico.

In definitiva, l'RMSE ci dimostra numericamente che l'algoritmo è abbastanza accurato e che effettivamente l'immagine restaurata ha subito un miglioramento rispetto alla controparte rumorosa.

4.1.2 Peak Signal Noise Ratio

Il **peak signal-noise ratio** (abbreviato con PSNR) è una misura che consente di valutare la qualità di un'immagine compressa rispetto all'originale. Si tratta di un indice di qualità delle immagini ed è definito come il rapporto tra la massima potenza di un segnale e la potenza di rumore. Il PSNR è solitamente espresso in termini di scala logaritmica di decibel e maggiore è il suo valore, tantopiù è maggiore la somiglianza con l'immagine originale da un punto di vista percettivo umano. I tipici valori assunti dal PSNR variano da 20 a 40.

Ma vediamo come viene definito questo indice:

$$\mathbf{PSNR} = \mathbf{10} * \mathbf{log_{10}}(\frac{\mathbf{L^2}}{\mathbf{MSE}}) \tag{4.2}$$

dove **L** è il massimo numero di livelli di grigio mentre **MSE** è l'errore quadratico medio calcolato sulle due immagini a confronto e definito come nella formula illustrata nel paragrafo precedente al di sotto della radice quadrata.

Di seguito riportiamo i valori di PSNR per le tre immagini campione utilizzate con $z = 0$ e $(k = 0, 5, 9)$ al variare del livello di rumore. Le tabelle sono organizzate nella stessa maniera dell'RMSE illustrato nel paragrafo precedente.

Immagine k=0			
Livello rumore	**Fattore threshold**	**Truth-Noise**	**Truth-Denoise**
3%	3.5	39.3627	40.4992
5%	4	35.1357	36.0129
7%	5.5	32.5834	33.3899
9%	5.5	30.676	32.8739

Immagine k=5			
Livello rumore	**Fattore threshold**	**Truth-Noise**	**Truth-Denoise**
3%	3.5	40.0504	46.4654
5%	4	36.5721	43.7716
7%	5.5	33.606	42.1731
9%	5.5	31.471	40.4865

Immagine k=9			
Livello rumore	**Fattore threshold**	**Truth-Noise**	**Truth-Denoise**
3%	3.5	40.2151	46.8808
5%	4	36.6531	44.0942
7%	5.5	33.7361	42.1145
9%	5.5	31.5565	40.5904

Osservando le tabelle si può notare che all'aumentare del livello del rumore, il PSNR tra ground-truth e noise tende a diminuire e per quanto detto prima, più diminuisce questo valore e più l'immagine sarà degradata come è logico che sia dal momento che l'immagine rumorosa è caratterizzata dall'aggiunta di rumore Rician rispetto all'immagine ground-truth. Nell'ultima colonna, vale a dire quella contenente i valori di PSNR tra ground-truth e denoise il fenomeno osservato è lo stesso e quindi si ha una diminuzione dei valori all'aumentare del livello di rumore. Quello che possiamo notare però è che questi valori di PSNR sono più alti rispetto ai valori di PSNR tra l'immagine ground-truth e l'immagine rumorosa poichè l'immagine restaurata è più simile in termini visivi all'immagine ground-truth rispetto a quella rumorosa.

Un'ultima cosa da notare è che i valori di PSNR dell'ultima colonna delle immagini per $k = 5$ e $k = 9$ sono più alti rispetto alle immagini $k = 0$ a parità

di rumore poichè come detto anche nel paragrafo precedente, le immagini gradiente hanno un rapporto segnale/rumore più basso di quelle non-DWI, quindi sono meno perturbate ed il processo di denoising porta un miglioramento più evidente.

In definitiva, così come accade per l'RMSE, anche il PSNR ci dimostra numericamente che l'algoritmo produce risultati di output con un buon grado di accuratezza.

4.1.3 Immagini Test

In questa sottosezione mostriamo i risultati di output che otteniamo dall'esecuzione dell'algoritmo per le immagini con $z = 0$ e $k = 0, 5, 9$ con i livelli di rumore 3%, 5%, 7% e infine 9%.

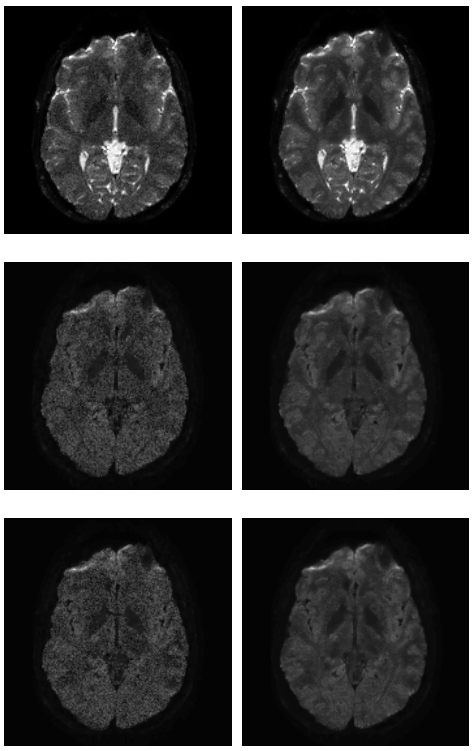


Figura 4.2: La tabella mostra i test sulla qualità dei risultati per il livello di rumore 3%. In prima colonna ci sono le immagini rumorose per z=0 e k=0,5,9 e in seconda colonna le corrispettive immagini restaurate.

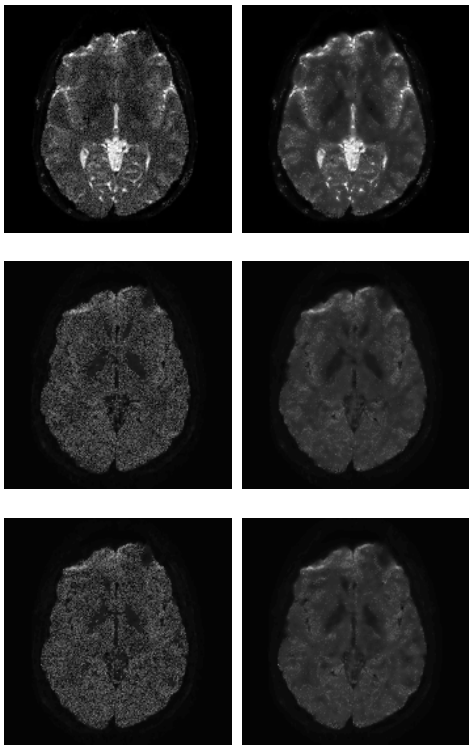


Figura 4.3: La tabella mostra i test sulla qualità dei risultati per il livello di rumore 5%. In prima colonna ci sono le immagini rumorose per z=0 e k=0,5,9 e in seconda colonna le corrispettive immagini restaurate.

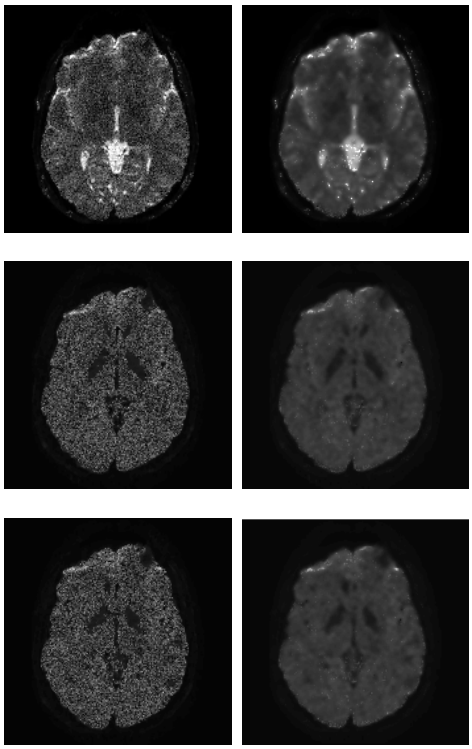


Figura 4.4: La tabella mostra i test sulla qualità dei risultati per il livello di rumore 7%. In prima colonna ci sono le immagini rumorose per z=0 e k=0,5,9 e in seconda colonna le corrispettive immagini restaurate.

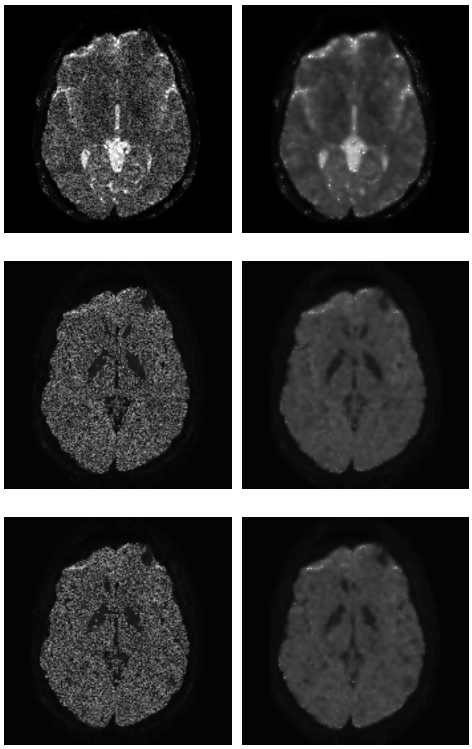


Figura 4.5: La tabella mostra i test sulla qualità dei risultati per il livello di rumore 9%. In prima colonna ci sono le immagini rumorose per z=0 e k=0,5,9 e in seconda colonna le corrispettive immagini restaurate.

4.2 Caratteristiche di esecuzione attraverso profiling

Lo strumento di profiling **nvprof** [19] consente di raccogliere e visualizzare i dati relativi alle caratteristiche di esecuzione di un software parallelo eseguito su GPU in ambiente CUDA. Attraverso questo tool eseguibile da riga di comando, possiamo effettuare un controllo di integrità per capire se l'esecuzione del nostro software è andata a buon fine e se si è fatto un uso corretto della memoria a disposizione.

Utilizzando il comando *nvprof* come segue:

```
nvprof --print-gpu-trace ./myApp
```

è possibile visualizzare una lista completa di tutti i kernel eseguiti nel software e di tutte le copie di memoria da host a device e viceversa. Inoltre per ogni kernel si può tenere traccia dello spazio occupato nella shared memory, il numero di

thread per blocco, la dimensione della griglia, il tempo di esecuzione del kernel e tante altre informazioni interessanti.

Di seguito sono mostrati i risultati di output del tool, ottenuti eseguendo in un primo momento la versione 1 dell'algoritmo parallelo su 10 direzioni con l'unica configurazione possibile di $(2, 2, 3)$ thread per blocco e $(88, 88, 7)$ blocchi nella griglia. In un secondo momento è stata eseguita invece la versione 2 dell'algoritmo sempre su 10 direzioni ma questa volta variando le configurazioni possibili di thread per blocco e quindi di blocchi della griglia. Nelle tabelle che seguono, sono state riportate soltanto alcune informazioni salienti: il tempo di esecuzione kernel, la dimensione della griglia, il numero di thread per blocco, la dimensione della shared memory dinamica allocata per blocco ed il nome del kernel eseguito.

Ecco i risultati per la versione 1 dell'algoritmo parallelo:

Duration	Grid size	Block size	DSMem	Name
1.10s	(88,88,7)	(2,2,3)	8.48KB	computeCovariance
7.21s	(88,88,7)	(2,2,3)	34.40KB	computePCA
7.32ms	(88,88,7)	(2,2,3)	0KB	weightedAverage

Di seguito, invece abbiamo i risultati del tool per la versione 2 dell'algoritmo parallelo:

Duration	Grid size	Block size	DSMem	Name
1.10s	(88,88,7)	(2,2,3)	8.48KB	computeCovariance
1.04s	(88,88,7)	(2,2,3)	8.48KB	computeProdYEXW
5.42s	(88,88,7)	(2,2,3)	8.48KB	computeProdXDEYWT
7.27ms	(88,88,7)	(2,2,3)	0KB	weightedAverage
Duration	Grid size	Block size	DSMem	Name
1.74s	(59,59,7)	(3,3,3)	16.88KB	computeCovariance
1.91s	(59,59,7)	(3,3,3)	16.88KB	computeProdYEXW
11.71s	(59,59,7)	(3,3,3)	16.88KB	computeProdXDEYWT
5.29ms	(59,59,7)	(3,3,3)	0KB	weightedAverage

Duration	Grid size	Block size	DSMem	Name
1.82s	(44,44,7)	(4,4,3)	28.32KB	computeCovariance
2.33s	(44,44,7)	(4,4,3)	28.32KB	computeProdYEXW
15.07s	(44,44,7)	(4,4,3)	28.32KB	computeProdXDEYWT
3.67ms	(44,44,7)	(4,4,3)	0KB	weightedAverage
Duration	Grid size	Block size	DSMem	Name
1.44s	(36,36,7)	(5,5,3)	42.80KB	computeCovariance
1.57s	(36,36,7)	(5,5,3)	42.80KB	computeProdYEXW
9.93s	(36,36,7)	(5,5,3)	42.80KB	computeProdXDEYWT
4.82ms	(36,36,7)	(5,5,3)	0KB	weightedAverage

Esaminando i dati tabellari possiamo innanzitutto notare da un punto di vista pratico quello che si è già detto in maniera esaustiva nel capitolo 3, vale a dire che nella versione 1 dell'algoritmo parallelo per come è strutturato e per il tipo di gestione della memoria, richiede soltanto l'esecuzione di tre kernel mentre la versione 2 che fa un uso più massiccio della memoria globale richiede certamente più copie da host a device e viceversa ed inoltre per ottenere gli stessi risultati della versione 1 utilizza ben quattro kernel.

Nella versione 2 possiamo inoltre notare che la dimensione occupata per la shared memory dinamica è uguale per i primi tre kernel dal momento che come operazione preliminare tutti e tre fanno la stessa operazione e quindi utilizzano lo stesso spazio di shared memory per blocco. L'ultimo kernel *weightedAverage* in entrambe le versioni non usa la shared memory in quanto effettua l'operazione direttamente su global memory.

In termini di tempo di esecuzione, per quanto riguarda la seconda versione dell'algoritmo possiamo vedere che aumentando il numero di thread per blocco i tempi sembrano non giovarne almeno per quanto riguarda i primi tre kernel nel quale si fa uso combinato tra shared memory e global memory e quindi sembra appunto che la configurazione più conveniente sia proprio quella di $(2, 2, 3)$ thread per blocco. Anche se la configurazione $(5, 5, 3)$ ha dei tempi inferiori alle configurazioni $(3, 3, 3)$ e $(4, 4, 3)$ lasciando presagire una buona scalabilità dell'algoritmo all'aumentare delle direzioni. Invece per il quarto kernel, dal momento che i thread operano solo su global memory, aumentando il numero di thread per blocco il tempo tende a diminuire.

Facendo invece un confronto tra le due versioni solo sulla configurazione $(2, 2, 3)$ possiamo certamente dire che nonostante ci sia una gestione e quindi un utilizzo diverso delle due memorie (shared e global memory) le due versioni non differiscono moltissimo a livello di tempi.

4.3 Performance CPU vs GPU

In questo paragrafo sono messe a confronto le performance dell'algoritmo sequenziale e dell'algoritmo parallelo in termini di GFlops. **Flops** è l'abbreviazione di ***FL****oating point* ***O****perations* ***P****er* ***S****econd* ed indica il numero di operazioni in virgola mobile eseguite in un secondo dalla CPU ($1GFlops = 10^9 Flops$).

Ecco la formula attraverso la quale sono state calcolate le performance dei due algoritmi:

$$\textbf{GFlops} = \textbf{Cores} * \frac{\textbf{Flops}}{\textbf{Time}} \tag{4.3}$$

I termini della formula indicano:

- **Cores:** è il numero di thread utilizzati, che nel caso dell'algoritmo sequenziale assume il valore di 1.
- **Flops:** è il numero di operazioni in virgola mobile eseguite nel metodo implementato.
- **Time:** è il tempo di esecuzione dell'algoritmo sequenziale espresso in secondi.

Sono stati effettuati diversi test, variando per ogni esecuzione le direzioni dell'immagine DWI (a partire da 2 fino ad arrivare alla massima direzione gestibile entro i limiti di memoria della macchina utilizzata per questo lavoro di tesi) e quindi il numero di voxel processati. Nella tabella che segue abbiamo quindi in prima colonna la dimensione del dataset, in seconda colonna le performance della CPU e in terza ed ultima colonna le performance della GPU.

Dataset size (voxel)	GFlops CPU	GFlops GPU
176x176x21x2 (1300992)	0,03	0,33
176x176x21x3 (1951488)	0,05	0,63
176x176x21x4 (2601984)	0,13	1,6
176x176x21x5 (3252480)	0,31	3,8
176x176x21x6 (3902976)	0,7	8,16
176x176x21x7 (4553472)	1,41	17
176x176x21x8 (5203968)	2,6	31
176x176x21x9 (5854464)	4,3	51,51
176x176x21x10 (6504960)	7	83,14

Prendendo in considerazione l'ultima riga della tabella, che riguarda le 10 direzioni massime sperimentabili, si può osservare che le performance della CPU sono pari a 7 GFlops mentre le performance della GPU arrivano a 83,14 GFlops. Le performance della GPU sono all'incirca 12 volte maggiori rispetto a quelle della CPU ottenendo un guadagno prestazionale in termini di percentuali pari a:

$$\textbf{GUADAGNO} = \frac{(83,14 - 7)}{83,14} = \frac{76,14}{83,14} = 0,916 * 100 = 91,6\%$$

Il guadagno delle performance dell'algoritmo parallelo rispetto a quello sequenziale, è decisamente elevato e questo non fa altro che confermare quanto sia importante e fortemente consigliato l'utilizzo dell'architettura GPU che ben si presta alla risoluzione di questo tipo di problemi in cui il carico di lavoro è estremamente elevato.

Di seguito è mostrato il grafico che evidenzia le differenze di performance tra CPU e GPU all'aumentare delle direzioni dell'immagine DWI. Come si può vedere, aumentando le direzioni, la curva delle performance della GPU assume un andamento esponenziale mentre la curva di performance della CPU tende quasi ad essere costante.

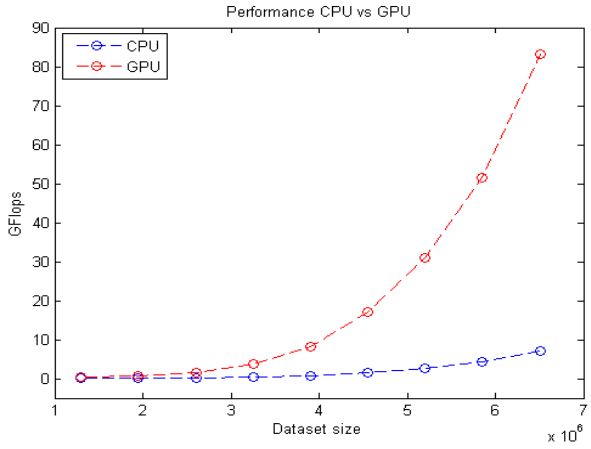


Figura 4.6: *In figura il grafico che riporta un confronto tra i GFlops effettuati dalla CPU e quelli effettuati dalla GPU*

4.4 Analisi dei tempi

In questa sezione si confrontano i tempi di esecuzione delle due versioni del software parallelo rispetto ai tempi dell'algoritmo sequenziale eseguito su CPU.

Nei diversi test effettuati, per ogni esecuzione si è variato il numero di direzioni dell'immagine DWI (a partire da 2 fino ad arrivare alla massima direzione gestibile entro i limiti di memoria della macchina utilizzata per questo lavoro di tesi) e quindi il numero di voxel processati. In tabella, abbiamo quindi in prima colonna la dimensione del dataset, in seconda colonna il tempo di esecuzione dell'algoritmo sequenziale eseguito su CPU ed in terza e quarta colonna c'è il tempo di esecuzione dell'algoritmo parallelo rispettivamente nella versione 1 e 2. Le due versioni parallele sono state eseguite con la stessa ed unica configurazione confrontabile per ragioni di spazio a disposizione, vale a dire $(2, 2, 3)$ thread per blocco e $(88, 88, 7)$ blocchi nella griglia.

Dataset size (voxel)	**Time CPU**	**Time GPU v1**	**Time GPU v2**
176x176x21x2 (1300992)	17	1,04	1,35
176x176x21x3 (1951488)	25	1,72	2,03
176x176x21x4 (2601984)	31	2,82	3,1
176x176x21x5 (3252480)	39	4,23	4,61
176x176x21x6 (3902976)	49	6,6	6,7
176x176x21x7 (4553472)	57	9,1	9,74
176x176x21x8 (5203968)	68	15,01	12,8
176x176x21x9 (5854464)	82	18,4	17,4
176x176x21x10 (6504960)	95	23,55	24

Prendendo in considerazione l'ultima riga della tabella che riguarda le 10 direzioni massime sperimentabili, si può osservare che il tempo d'esecuzione impiegato dalla CPU è di 95 secondi mentre quello della GPU si riduce a 23,55 secondi per la versione 1 e 24 secondi per la versione 2. Approssimativamente le due versioni dell'algoritmo parallelo impiegano lo stesso tempo pari a 24 secondi e quindi il tempo impiegato dalla GPU è all'incirca 4 volte minore rispetto al tempo impiegato dalla CPU ottenendo quindi un guadagno di tempo in termini di percentuali pari a:

$$\textbf{GUADAGNO} = \frac{(95 - 24)}{95} = \frac{71}{95} = 0,75 * 100 = 75\%$$

Il guadagno in termini di tempo da parte del software eseguito su GPU rispetto al software eseguito su CPU è soddisfacente e questo ci porta a concludere

che l'algoritmo parallello è organizzato in maniera tale da produrre risultati in tempi decisamente minori rispetto a quelli ottenuti su CPU, confermando ancora una volta di come sia conveniente risolvere questo tipo di problema attraverso l'ausilio dell'HPC ed in particolare dei processori grafici.

Di seguito il grafico mostra le curve dei tempi di esecuzione dell'algoritmo sequenziale su CPU e delle due versioni dell'algoritmo parallelo su GPU. E' possibile notare a primo impatto che le curve delle due versioni parallele si sovrappongono quasi perfettamente e questo conferma che la differente gestione delle memorie sull'architettura GPU per le due versioni non influisce più di tanto almeno per quanto riguarda il tempo d'esecuzione. Invece i tempi di esecuzione dell'algoritmo sequenziale sono decisamente più alti della controparte parallela e questa differenza diventa sempre più marcata all'aumentare della dimensione del dataset elaborato.

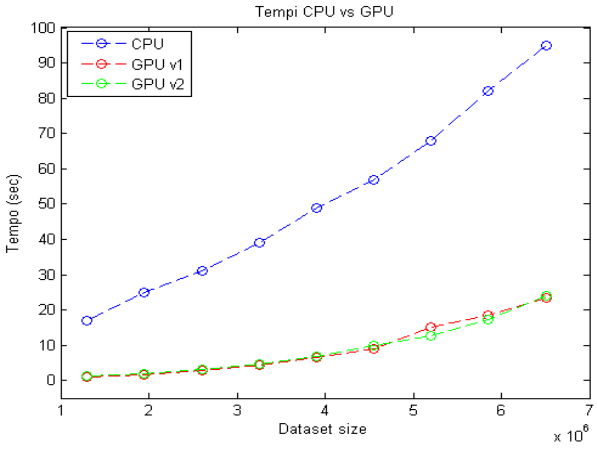


Figura 4.7: *In figura il grafico che riporta un confronto dei tempi di esecuzione dell'algoritmo sequenziale eseguito su CPU rispetto alla controparte parallela nelle due differenti versioni eseguite su un'architettura GPU.*

Nei test che seguiranno nelle prossime sottosezioni verranno esaminati i tempi di esecuzione della seconda versione dell'algoritmo parallelo, a seguire i tempi di esecuzione a confronto tra le due versioni e infine i tempi di esecuzione per il calcolo degli autovettori su CPU, che costituiscono una vera e propria limitazione in termini di tempo per l'algoritmo parallelo.

4.4.1 Tempi d'esecuzione algoritmo parallelo (vers. 2)

In questo esperimento vengono esaminati i tempi di esecuzione della seconda versione dell'algoritmo parallelo che è la versione che prevede un uso combinato di global memory e shared memory all'interno dei kernel al fine di superare i limiti di spazio per la shared memory. Nei vari test effettuati, sono state variate le direzioni dell'immagine DWI e per ogni direzione l'algoritmo è stato eseguito con le diverse configurazioni di griglia nei limiti dello spazio a disposizione.

Nella tabella che segue, per ogni test è riportato in prima colonna il numero di thread per blocco utilizzato, in seconda colonna il numero di direzioni dell'immagine DWI ed infine in terza colonna il tempo di esecuzione in secondi rilevato.

Thread per blocco	**Direzioni**	**Tempo**
2x2x3	2	1,35
3x3x3	2	1,43
4x4x3	2	1,45
5x5x3	2	1,5
2x2x3	5	4,61
3x3x3	5	4,94
4x4x3	5	4,91
5x5x3	5	5,1
2x2x3	7	9,8
3x3x3	7	10,6
4x4x3	7	10,7
5x5x3	7	10,8
2x2x3	9	18
3x3x3	9	21
4x4x3	9	20,4
5x5x3	9	23,3
2x2x3	10	24
3x3x3	10	31,8
4x4x3	10	35,6
5x5x3	10	29,30

Osservando la tabella si può notare che a prescindere dalla configurazione utilizzata, aumentando il numero di direzioni i tempi di esecuzione, come è logico che sia, aumentano dal momento che il carico di lavoro è sempre maggiore. Inoltre, per ogni direzione, la configurazione migliore è sempre quella di $(2, 2, 3)$ thread per blocco ad ottenere i tempi più bassi e subito dopo la confi-

gurazione $(4, 4, 3)$ thread per blocco, almeno per quanto riguarda le direzioni provate.

Di seguito il grafico mostra i tempi di esecuzione dell'algoritmo in funzione delle direzioni utilizzate ed al variare della configurazione dei kernel. Si può notare che per le direzioni 2, 5 e 7 il tempo di esecuzione è pressapoco costante a prescindere dalla configurazione utilizzata. Aumentando le direzioni utilizzate invece diventa più evidente che la migliore configurazione è quella che prevede $(2, 2, 3)$ thread per blocco. Anche se, osservando l'ultima curva relativa alle 10 direzioni si nota un calo del tempo di esecuzione corrispondente alla configurazione $(5, 5, 3)$ che lascia ben sperare che aumentando le direzioni potrebbe essere utile distribuire il carico di lavoro su più thread per ottenere tempi di esecuzione minori.

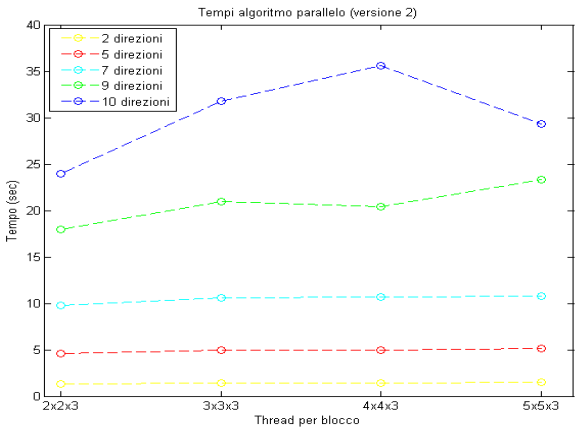


Figura 4.8: *In figura il grafico che riporta i tempi di esecuzione della seconda versione dell'algoritmo parallelo al variare del numero di direzioni dell'immagine DWI e al variare del numero di thread per blocco.*

4.4.2 Tempi d'esecuzione a confronto tra le due versioni

I seguenti test mettono a confronto le due versioni dell'algoritmo parallelo in termini di tempo d'esecuzione, al variare del numero di thread per blocco e fissando le direzioni dell'immagine DWI a 2 per i motivi già ampiamenti discussi, relativi ai limiti di spazio della shared memory utilizzabile.

Esaminando i dati tabellari si percepisce quasi subito che in termini di secondi, la differenza nei tempi di esecuzione tra le due versioni è molto bassa. Questa differenza è certamente dovuta alla velocità di accesso della shared memory

Thread per blocco	Time GPU v1	Time GPU v2
2x2x3	1,04	1,35
3x3x3	1,1	1,43
4x4x3	1,04	1,45
5x5x3	1,27	1,5

che è molto più veloce della global memory e a trarne beneficio è proprio la versione 1.

Di seguito il grafico mostra i tempi di esecuzione a confronto, variando il numero di thread per blocco. Dalle due curve, si riscontra che nella prima versione dell'algoritmo è più evidente il vantaggio nell'utilizzare le configurazioni $(2, 2, 3)$ e $(4, 4, 3)$ che hanno i picchi minimi di tempo mentre la configurazione $(5, 5, 3)$ è quella meno adatta dal momento che raggiunge il picco più elevato. Nella seconda curva rossa, quella relativa ai tempi di esecuzione della seconda versione dell'algoritmo parallelo, l'andamento in termini di tempo è crescente, pertanto questo ulteriore test non fa altro che confermare quanto visto nel test precedente, vale a dire che la configurazione migliore nel caso della seconda versione dell'algoritmo, è certamente quella che prevede $(2, 2, 3)$ thread per blocco.

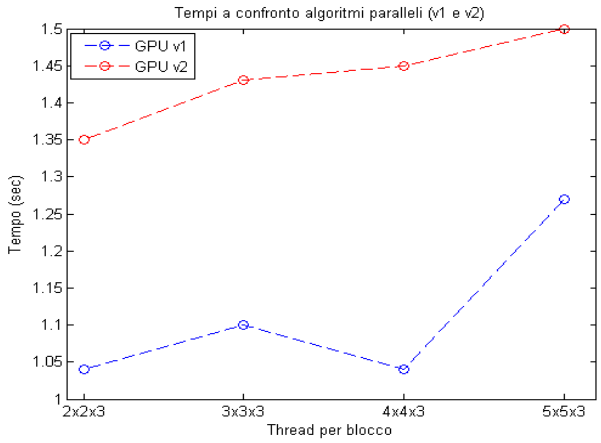


Figura 4.9: *In figura il grafico che riporta un confronto dei tempi di esecuzione tra le due versioni dell'algoritmo parallelo.*

4.4.3 Tempi d'esecuzione per il calcolo degli autovettori

In questo esperimento è riportato per ogni test, il tempo di esecuzione necessario alla computazione degli autovettori attraverso Singular Value Decompo-

sition. Come detto nel capitolo 3, il calcolo degli autovettori rappresenta il principale deficit (in termini di tempo di esecuzione) per l'algoritmo parallelo sviluppato in questo progetto di tesi, perchè è stato necessario effettuare l'operazione sull'host a causa dell'impossibilità di poter richiamare le routine di libreria nel kernel CUDA per i limiti di *compute capability* della scheda video montata sulla macchina dove sono stati effettuati gli esperimenti e questo chiaramente ha inficiato sulle prestazioni generali.

Attraverso i dati tabellari che seguono, è possibile vedere come questo tempo d'esecuzione aumenti all'aumentare delle direzioni dell'immagine DWI da elaborare. Questo comportamento è dovuto al numero di direzioni che incidono sulla dimensione della singola matrice di autovettori ($DIRECT \times DIRECT$) e quindi di conseguenza la routine SVD impieghierà più tempo per il calcolo.

Dataset size (voxel)	**Direzioni**	**Time W**
176x176x21x2 (1300992)	2	0,9
176x176x21x3 (1951488)	3	1,31
176x176x21x4 (2601984)	4	2
176x176x21x5 (3252480)	5	3,1
176x176x21x6 (3902976)	6	4.45
176x176x21x7 (4553472)	7	6,4
176x176x21x8 (5203968)	8	8,25
176x176x21x9 (5854464)	9	11,30
176x176x21x10 (6504960)	10	15

Il grafico seguente conferma a livello visivo ciò che è stato detto prima e infatti il tempo di esecuzione per il calcolo degli autovettori assume un andamento *quadratico*.

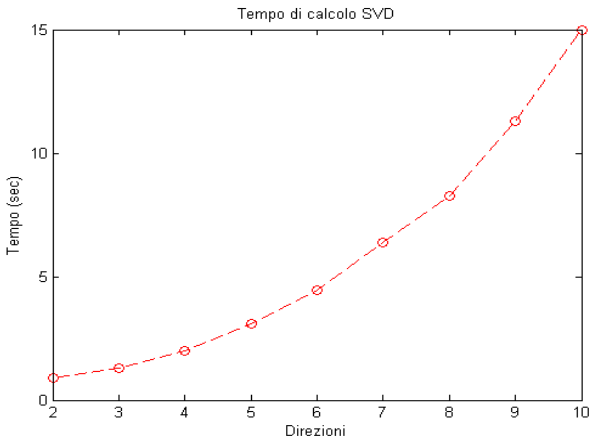


Figura 4.10: *In figura il grafico che riporta i tempi di esecuzione necessari alla computazione degli autovettori mediante SVD.*

Capitolo 5

Conclusioni e Sviluppi Futuri

Con il lavoro svolto in questa tesi si è innanzitutto esaminato il metodo PCA locale overcomplete per la rimozione di rumore Rician da immagini pesate in diffusione, proposto in [4]. Successivamente è stato implementato l'algoritmo sequenziale per tale metodo e quindi sono stati effettuati dei test d'esecuzione per esaminare le caratteristiche del software. Gli esperimenti hanno evidenziato che l'algoritmo è in grado di rimuovere il rumore dalle immagini DWI producendo dei buoni risultati di output sia da un punto di vista qualitativo dell'immagine che di accuratezza. Sfortunatamente, come era già stato previsto, i tempi di esecuzione si sono rivelati eccessivamente lunghi se consideriamo che questa tipologia di operazione di enhancement, su queste immagini mediche, dovrebbe essere realizzata quasi istantaneamente al fine di mettere in condizione il medico di esaminare il materiale ed effettuare la diagnosi senza dover aspettare per troppo tempo. Quindi, con lo scopo di ridurre i tempi d'esecuzione dell'algoritmo pur ottenendo i medesimi risultati di accuratezza, si è realizzata una implementazione parallela in ambiente GPU-CUDA in modo da sfruttare l'enorme potenza di calcolo dell'architettura parallela delle GPU. Per questo lavoro di tesi sono state sviluppate due versioni del software parallelo: una prima versione che fa un uso più massiccio della shared memory ma che è limitata proprio dalle dimensioni ridotte di questa memoria e invece una seconda versione che combina accessi alla shared memory con accessi alla global memory aggirando i problemi di spazio della prima versione entro comunque certi limiti. Dai risultati sperimentali è emerso che le due versioni non differiscono di molto in termini di tempo di esecuzione, almeno per le direzioni testate. Il confronto fra l'esecuzione dell'algoritmo eseguito su CPU e quello eseguito su GPU ha evidenziato un notevole vantaggio, in termini di prestazioni, dell'architettura CUDA. Infatti, i tempi di esecuzione del software parallelo sono molto più bassi con un guadagno fino al 75% ottenendo comunque gli stessi risultati di accuratezza del software sequenziale.

Questi aspetti rappresentano un importante punto di partenza per eventuali sviluppi futuri del lavoro di tesi. Si potrebbe pensare ad esempio, di realizzare un'implementazione che sia in grado di sfruttare contemporaneamente la potenza di calcolo di multiple GPU per incrementare ulteriormente il guadagno in termini prestazionali. In tal senso, bisognerebbe splittare l'immagine DWI lungo la terza dimensione in modo da generare più immagini, ognuna da caricare su una GPU diversa e quindi in questo caso il codice verrebbe eseguito parallelamente ma in maniera indipendente su ogni GPU. Alla fine del processo, bisogna effettuare un merge per ricomporre i dati per ottenere l'immagine restaurata. Un altro aspetto ancora più interessante, potrebbe essere quello di adottare tecniche di parallelismo più avanzate come il Dynamic Parallelism che è un'estensione al modello di programmazione CUDA introdotto a partire dalla versione 5.0 e utilizzabile solo su schede video con compute capability superiore o uguale a 3.5. Il Dynamic Parallelism abilita un kernel ad invocare al suo interno un kernel figlio e a sincronizzarsi con esso portando l'immediato vantaggio di ridurre la necessità di trasferire i dati dall'host al device e viceversa. Attraverso questa estensione si potrebbero utilizzare le API dinamiche delle librerie di algebra lineare accelerate su GPU come CULA o CUBLAS per effettuare calcoli più complessi direttamente all'interno del kernel senza dover spostare i dati sull'host ed eseguire le routine su CPU. Questo accorgimento potrebbe ridurre notevolmente i tempi di esecuzione dell'algoritmo parallelo da me sviluppato, dal momento che uno dei principali limiti come detto più volte nei capitoli precedenti, è rappresentato dall'impossibilità di poter richiamare la routine per il calcolo degli autovettori all'interno del kernel, causando quindi un aumento del tempo di esecuzione e una perdita di guadagno in termini di prestazioni. Infine, data la scalabilità dell'algoritmo implementato, sarebbe certamente utile sia ai fini sperimentali ma anche nell'ottica di un utilizzo in un contesto reale, eseguire il software parallelo su schede video con memorie più capienti in modo tale da eliminare i limiti di spazio che non hanno consentito in questo caso particolare, di elaborare oltre le 10 direzioni e quindi verificare fin dove possono arrivare i guadagni.

Bibliografia

[1] *'Imaging con tensore di diffusione'* - http://it.wikipedia.org/wiki/Imaging_con_tensore_di_diffusione

[2] *'Non-local means variants for denoising of diffusion-weighted and diffusion tensor MRI. MICCAI2007.'* - Wiest-Daesslé N, Prima S, Coupé P, Morrissey SP, Barillot C, 2007.

[3] *'DWI filtering using joint information for DTI and HARDI. Med Image Anal.'* - Tristán-Vega A, Aja-Fernández S, 2010.

[4] *'Impact of Rician adapted non-local means filtering on HARDI. MICCAI 2008.'* - Descoteaux M, Wiest-Daesslé N, Prima S, Barillot C, Deriche R, 2008.

[5] *'Diffusion Weighted Image Denoising Using Overcomplete Local PCA. Plos One.'* - José V. Manjón, Pierrick Coupé, Luis Concha, Antonio Buades, D. Louis Collins, Montserrat Robles, 2013.

[6] *'Neuroimaging Informatics Technology Initiative: un formato per facilitare le operazioni di analisi delle immagini mediche MRI'* - http://nifti.nimh.nih.gov/.

[7] *'Neuroimaging Informatics Technology Initiative Data Format Working Group'* - http://nifti.nimh.nih.gov/dfwg/.

[8] *'Software per la visualizzazione di immagini mediche in diversi formati.'* - http://www.brain.org.au/software/mrtrix/general/mrview.html.

[9] *'Un toolkit open source per la conversione e visualizzazione di immagini mediche in diversi formati.'* - http://xmedcon.sourceforge.net/.

[10] *'Principal component analysis. New York: Springer-Verlag (1986).'* - I.T. Jolliffe.

[11] *'Restoration of DWI data using a Rician LMMSE estimator. IEEE Trans Med Imaging.'* - Aja-Fernandenz S, Niethammer M, Kubick M, Shenton E, Westin CF, (2008).

[12] *'Robust Rician Noise Estimation for MR Images. Medical Image Analysis.'* - Coupé P, Manjón JV, Gedamu E, Arnold D, Robles M, et al. (2010).

[13] *'Graphics Processing Unit'* - http://www.nvidia.com/object/gpu.html.

[14] *'Il computing accelerato dalle GPU'* - http://www.nvidia.it/object/gpu-computing-it.html.

[15] *'Il GPGPU'* - http://it.wikipedia.org/wiki/GPGPU.

[16] *'CUDA'* - http://it.wikipedia.org/wiki/CUDA.

[17] *'Parallel Computing con CUDA'* - http://www.nvidia.it/object/cuda-parallel-computing-it.html.

[18] *'Parallelismo Dinamico di CUDA'* - http://on-demand.gputechconf.com/gtc/2012/presentations/S0338-GTC2012-CUDA-Programming-Model.pdf.

[19] *'Profiler CUDA nvprof'* - http://docs.nvidia.com/cuda/profiler-users-guide/.