Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Dipartimento di Fisica e Astronomia
Corso di Laurea Magistrale in Fisica

Parallelizzazione dell'algoritmo di ricostruzione di Feldkamp-Davis-Kress per architetture Low-Power di tipo System-On-Chip

Relatore:

Prof.ssa Maria Pia Morigi

Presentata da:

Omar Chehaimi

Correlatori:

Dott.ssa Rosa Brancaccio
Dott. Daniele Cesini
Dott.ssa Elena Corni

Anno Accademico 2016/2017

Sommario

In questa tesi, svolta presso il CNAF, si presentano i risultati ottenuti nel lavoro svolto per la parallelizzazione in CUDA dell'algoritmo di ricostruzione tomografica di Feldkamp-Davis-Kress (FDK), sulla base del software in versione sia sequenziale che parallela MPI, sviluppato presso i laboratori del *X-ray Imaging Group* di Bologna. Gli obbiettivi di questo lavoro sono principalmente due: ridurre in modo sensibile i tempi di esecuzione dell'algoritmo di ricostruzione FDK parallelizzando su *Graphics Processing Unit* (GPU) e valutare, su diverse tipologie di architetture, i consumi energetici. Le piattaforme prese in esame sono di due tipi: quelle note come SoC (*System-on-Chip*) *low-power*, architetture a basso consumo energetico ma a limitata potenza di calcolo, e quelle note come *High Performance Computing* (HPC), caratterizzate da un'elevata potenza di calcolo ma con un ingente consumo energetico. In particolare, si vuole mettere in risalto la differenza di prestazioni in relazione al tipo di architettura e rispetto al relativo consumo energetico. Infatti poter sostituire nodi HPC con schede SoC *low-power* presenta il vantaggio di ridurre i consumi, la complessità dell'*hardware* e la possibilità di ottenere dei risultati direttamente in loco.

I risultati ottenuti mostrano che la parallelizzazione di FDK su GPU sia la scelta più efficiente. Risulta infatti sempre, e su ogni architettura testata, più performante rispetto alla versione MPI, nonostante in quest'ultima venga parallelizzato tutto l'algoritmo. In CUDA invece si parallelizza solo la fase di ricostruzione. Inoltre si è risusciti a raggiungere un'efficienza di utilizzo della GPU del 100%.

L'efficienza energetica rapportata alle prestazioni in termini di tempo è migliore per le architetture SoC *low-power* rispetto a quelle HPC.

Si propone infine un approccio ibrido MPI unito a CUDA che migliora ulteriormente le prestazioni di esecuzione. Grazie al fatto che il filtraggio e la ricostruzione sono operazioni indipendenti fra loro si è pensato di utilizzare l'implementazione più efficiente per la data operazione, filtrare in MPI e ricostruire in CUDA.

Indice

Lista degli acronimi

CT Computed Tomography

FDK Feldkamp-Davis-Kress

FBP Filtered Backprojection

FFT Fast Fourier Transform

RAM Random Access Memory

CPU Central Processing Unit

GPU Graphics Processing Unit

GPGPU General Purpose Computing on Graphics Processing Units

UE Unit of Execution

PU Processing Unit

SP Streaming Processor

SM Streaming Multiprocessor

CUDA Compute Unified Device Architecture

MPI Message Passing Interface

HPC High Performance Computing

SoC System-on-a-chip

TK1 Jetson Tegra K1

TX1 Jetson Tegra X1

SISD Single Instruction Single Data

SIMD Single Instruction Multiple Data

MISD Multiple Instruction Single Data

MIMD Multiple Instruction Multiple Data

SIMT Single Instruction Multiple Threads

ROI Region of Interest

COSA Computing on SoC Architecture

openCL Open Computing Language

Introduzione

In questa tesi si presentano i risultati ottenuti dal lavoro svolto per la parallelizzazione in CUDA dell'algoritmo di ricostruzione tomografica di Feldkamp-Davis-Kress (FDK). Gli obbiettivi di questo lavoro, condotto presso i laboratori del CNAF, in collaborazione con l' *X-ray Imaging Group* del Dipartimento di Fisica e Astronomia di Bologna, che ha sviluppato sia la versione sequenziale che parallela MPI dell'algoritmo, sono principalmente due: migliorare le prestazioni di esecuzione dell'algoritmo di ricostruzione FDK, riducendo in modo sensibile i tempi di esecuzione grazie alla parallelizzazione su GPU (*Graphics Processing Unit*) e valutare, su diverse tipologie di architetture *hardware*, i consumi energetici. In particolare, sono stati presi in esame due tipi di architetture: quelle note come SoC *low-power*, architetture a potenza di calcolo e consumo energetico limitati, e quelle note come *High Performance Computing* (HPC), caratterizzate da un'elevata potenza di calcolo, ma con un notevole consumo energetico. Ciò che si vuole mettere in risalto è la differenza presente fra le due architetture in termini di prestazioni rispetto al relativo consumo energetico. A causa dei consumi energetici estremamente elevati dei sistemi HPC, studiare soluzioni per la risoluzione di problemi particolarmente onerosi in termini computazionali ha indubbi benefici. Infatti poter trovare sistemi che raggiungano un compromesso fra prestazioni elevate e basso consumo porta sicuramente il vantaggio di ridurre la complessità e la spesa in termini economici.

Ci si aspetta pertanto che i tempi di esecuzione dell'algoritmo parallelizzato siano nettamente più bassi per architetture HPC rispetto a quelle *low-power*, ma che il consumo energetico sia nettamente più elevato.

La tesi è stata organizzata nel modo seguente.

Nel primo capitolo si affronta il problema della tomografia computerizzata, descrivendo i principi fisici che ne stanno alla base, fornendo una breve panoramica delle varie tecniche di acquisizione e ricostruzione di immagini tomografiche attualmente presenti e soffermandosi in particolare sull'algoritmo di Feldkamp-Davis-Kress (FDK).

Nel secondo capitolo viene trattato il calcolo parallelo, come strumento generale per poter risolvere problemi computazionali in modo più performante. Vengono quindi affrontati dal punto di vista teorico i tipici problemi

legati al calcolo parallelo, approfondendo in particolare la parallelizzazione svolta su GPU NVIDIA con lo strumento CUDA.

Nel terzo capitolo, dopo una descrizione dettagliata delle architetture *hardware* per eseguire i test, viene introdotto lo stato dell'arte sulla parallelizzazione dell'algoritmo FDK e illustrato il modo in cui è stata eseguita la parallelizzazione in CUDA nell'ambito di questo lavoro di tesi.

Nel quarto capitolo sono presentati tutti i risultati ottenuti nei vari test effettuati. Vengono riportate le prove eseguite sulle varie architetture *hardware* utilizzate e le considerazioni conclusive sia legate alle prestazioni nell'esecuzione del codice che ai consumi energetici.

Nel quinto capitolo si riportano le conclusioni di questo lavoro di tesi.

Capitolo 1

Algoritmo di ricostruzione tomografica di Feldkamp-Davis-Kress

Nel seguente capitolo si descrivono i principi fisici alla base della tomografia computerizzata con raggi X, i principali algoritmi di ricostruzione tomografica, e, in particolare, l'algoritmo di Feldkamp-Davis-Kress (FDK) per la ricostruzione tomografica in geometria *cone-beam* come descritto in [1].

1.1 Introduzione alla tomografia computerizzata

La tomografia computerizzata (CT, *computed tomography*) è una tecnica di *imaging* che sfrutta le proprietà dei raggi X per ottenere delle immagini inerenti al campione analizzato. Questa tecnica è molto importante perché permette di acquisire informazioni sulla struttura interna degli oggetti in studio, sia di tipo morfologico che fisico, in modo non distruttivo e non invasivo. Pertanto questa tecnica trova innumerevoli applicazioni in svariati campi, come quello medico, industriale e di conservazione dei beni culturali.

Nella tomografia computerizzata l'immagine viene ricostruita a partire da radiografie dell'oggetto a diversi angoli, dette proiezioni, che possono essere di tipologie differenti a seconda della geometria in uso. Infatti si parla di *parallel beam* quando la sorgente di raggi X è un fascio a raggi paralleli, di *fan beam* quando la sorgente ha una apertura orizzontale a ventaglio, di *cone beam* se il fascio ha forma conica (apertura orizzontale e verticale). Esiste poi la geometria detta *helical* che si ottiene facendo ruotare la sorgente e traslare l'oggetto allo stesso tempo, ottenendo così una traiettoria a elica in cui le proiezioni sono definite per 180° e interpolate per ottenere i valori nel restante settore circolare [2]. In funzione della geometria del sistema

e dell'apparato di acquisizione si utilizzerà l'algoritmo di ricostruzione più appropriato.

1.1.1 Principi fisici della tomografia computerizzata

Nella CT la sorgente di radiazione utilizzata sono i raggi X, fotoni di energia superiore ai 100 eV. I fotoni possono interagire con la materia tramite i seguenti effetti:

- Effetto fotoelettrico, assorbimento totale
- Effetto Compton, urto elastico
- Reazioni fotonucleari, assorbimento totale
- Scattering Rayleigh, urto elastico

La sezione d'urto di tutti questi effetti dipende dal materiale con cui avviene l'interazione e dall'energia del fascio incidente.

Detto x il punto in cui si vuole misurare l'attenuazione, I_0 l'intensità della sorgente e s la coordinata su cui si calcola l'integrale di linea, allora vale:

$$I(x) = I_0 e^{-\int_{line} \mu(s) ds} \tag{1.1}$$

dove μ il coefficiente di attenuazione lineare che dipende da σ, cioè dalla sezione d'urto totale ottenuta dalla somma di tutte le sezioni d'urto per tutti i fenomeni di interazione presenti fra la materia e i raggi X. In genere viene utilizzato il coefficiente di attenuazione massico, definito come il rapporto del coefficiente di attenuazione lineare con la densità del materiale (grafico in Figura 1.1. Calcolando il logaritmo del rapporto fra $I(x)$ e I_0 si ha:

$$ln\left(\frac{I(x)}{I_0}\right) = -\int_{line} \mu(s) ds \tag{1.2}$$

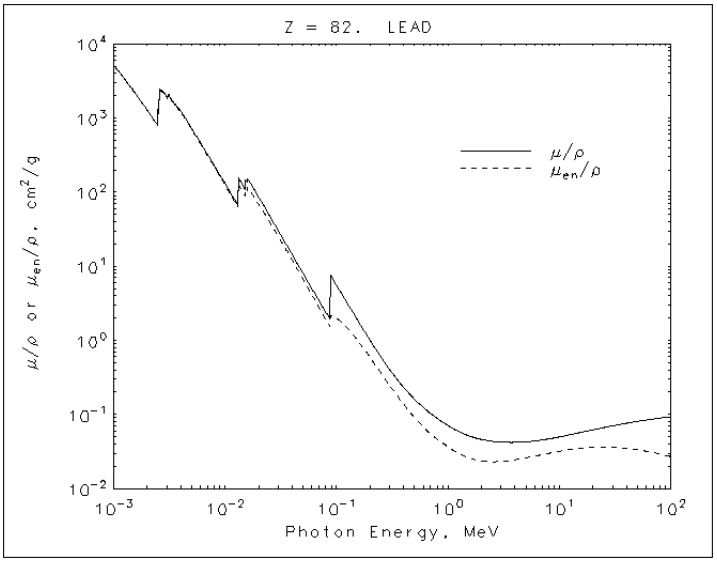

Figura 1.1: Coefficiente di attenuazione massico per il piombo, [3].

Nel caso di una radiografia si misura l'intensità del fascio attenuato a causa del percorso compiuto per raggiungere il *detector*. Quindi si ha che sul pixel x_d, y_d del rivelatore il fascio sarà attenuato per la somma di tante interazioni successive con la materia che attraversa. Matematicamente il valore del pixel è l'integrale di linea del coefficiente di assorbimento dei raggi X.

Figura 1.2: Esempio di radiografia acquisita durante prova in laboratorio.

Normalizzando il valore dell'intensità del pixel con il valore del fascio acquisito senza oggetto e calcolandone il logaritmo si ottiene l'integrale del

valore del coefficiente di assorbimento μ, che è funzione delle coordinate spaziali x, y, z. L'immagine che si ottiene mostra zone più scure e altre più chiare a prova del fatto che alcune aree dell'oggetto assorbono una maggiore quantità di raggi X piuttosto che altre a causa del diverso tipo di materiale di cui sono composte (figura 1.2).

1.1.2 Principali geometrie di acquisizione

Le geometrie di acquisizione di un sistema CT in generale dipendono dalla geometria del fascio di raggi X, dalle dimensioni dell'oggetto, quelle del rivelatore e dalle distanze reciproche che definiscono l'angolo di apertura verticale del fascio. Le diverse geometrie si suddividono in: tomografia a raggi paralleli (*parallel beam*), a ventaglio (*fan beam*) oppure a cono (*cone beam*). Nelle prime due, le sezioni (*slice*) da ricostruire sono indipendenti fra loro perché ognuna corrisponde a una riga delle proiezioni ai vari angoli, nella terza, le *slice* sono invece dipendenti e per ricostruirle è necessario elaborare tutte le righe di tutte le proiezioni a diversi angoli. Dal punto di vista degli algoritmi di ricostruzione sia il *parallel beam* che il *fan beam* sono molto più semplici da implementare e parallelizzare poiché permettono di ricostruire la *slice* partendo da un solo sinogramma (immagine risultante dall'insieme delle righe delle proiezioni ad una certa altezza sul rivelatore e per diversi angoli), al contrario invece del *cone beam*, in cui è necessario acquisire prima le radiografie dell'intero oggetto per poter iniziare il processo di ricostruzione tomografica.

1.2 Integrali di linea in due dimensioni

L'integrale di linea rappresenta l'integrale di una variabile lungo una retta. Nel campo degli algoritmi di ricostruzione tomografica gli integrali di linea sono fondamentali, la radiografia di un oggetto corrisponde fisicamente all'integrale sulla linea retta, nel caso di sorgenti non diffrangenti. Infatti, facendo riferimento alla figura 1.3, se B è la sorgente del fascio e la retta t il *detector*, il fascio interagisce con l'oggetto lungo tutta la retta AB.

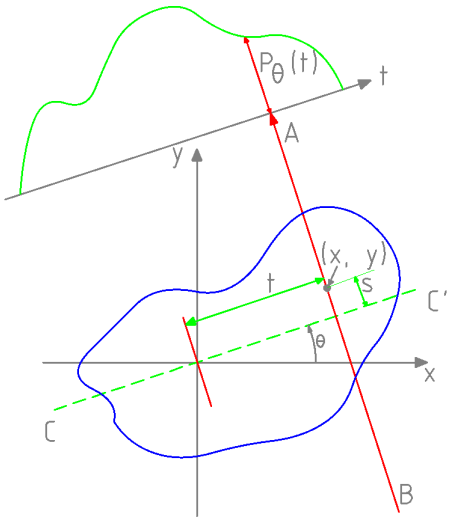

Figura 1.3: Proiezione del fascio di raggi X.

Pertanto l'interazione totale del fascio con l'oggetto, sintetizzato nel coefficiente di attenuazione lineare μ, è la somma di tutte le interazioni lungo la retta AB. Quindi indicando con θ l'angolo fra la retta CC$'$ e l'asse x e indicando con s la variabile posizione lungo AB, il valore misurato dal rivelatore nel punto t_1 sulla retta t, rappresentante il valore di attenuazione lineare, è la somma su tutti gli infinitesimi intervalli ds lungo AB, cioè:

$$P_\theta(t_1) = \int_{(\theta,t)line} f(x,y)ds \tag{1.3}$$

Riscrivendo quindi il valore della proiezione $P_\theta(t)$ per un punto generico t, utilizzando la funzione δ per ottenere lo spostamento infinitesimo ds funzione di (x,y) si ha:

$$P_\theta(t) = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} f(x,y)\delta(x\cos(\theta) + y\sin(\theta) - t)dxdy \tag{1.4}$$

Il valore $P_\theta(t)$ si dice trasformata di Radon. L'insieme dei valori di $P_\theta(t)$ per tutti i t è detto proiezione. Quindi il problema della ricostruzione tomografica risulta essere quello di trovare il valore della funzione $f(x,y)$, che nel caso della CT corrisponde al valore del coefficiente di attenuazione lineare nel punto (x,y). A questo punto sembrerebbe sufficiente invertire la trasformata di Radon per ottenere la funzione $f(x,y)$.

In realtà però la trasformata di Radon è valida solo per valori continui di $P_\theta(t)$ e per un numero di angoli continuo (numero infinito di proiezioni). È evidente che nella tomografia reale non è possibile acquisire un numero infinito di proiezioni, che, a loro volta essendo immagini digitali, non possono contenere un numero infinito di pixel. Per risolvere questi problemi di natura

numerica si utilizza l'algoritmo *filtered backprojection* (FBP), che consiste nel filtrare la proiezione eseguendo una convoluzione con un filtro e poi retroproiettare la proiezione filtrata per ricostruire la funzione $f(x, y)$ con il teorema della *slice* di Fourier. Questo algoritmo è il più utilizzato per problemi di ricostruzione tomografica e può essere applicato per qualunque tipo di geometria di acquisizione e del *detector*, come: *parallel beam*, *fan beam*, equispaziato o equiangolare o anche *cone beam* in condizioni di angolo di fan (apertura verticale del fascio di raggi X) adeguate.

1.3 Teorema della slice di Fourier

Questo teorema mostra nel caso di una geometria di tipo a fasci paralleli come sia possibile ottenere la funzione $f(x, y)$ partendo dalla trasformata di Fourier della proiezione $P_\theta(t)$. Esso stabilisce che la trasformata di Fourier della proiezione $P_\theta(t)$ equivale alla trasformata di Fourier della funzione $f(x, y)$. Quindi antitrasformando la trasformata di Fourier della proiezione si ottiene la funzione $f(x, y)$.

Si definiscono le seguenti trasformate:

$$F(u, v) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x, y) e^{-2\pi i(ux+vx)} dxdy \tag{1.5}$$

come la trasformata della funzione $f(x, y)$, e

$$S_\theta(w) = \int_{-\infty}^{+\infty} P_\theta(t) e^{-2\pi iwt} dt \tag{1.6}$$

come la trasformata della proiezione $P_\theta(t)$. Si dimostra il teorema nel caso più semplice per $\theta = 0$ (nel caso generale basterà applicare una rotazione mediante la matrice di rotazione alle variabili in uso). Si considera $v = 0$ (quindi ci si posiziona sull'asse orizzontale dello spazio di Fourier):

$$\begin{aligned} F(u, 0) &= \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x, y) e^{-2\pi iux} dxdxy \\ &= \int_{-\infty}^{+\infty} \left[\int_{-\infty}^{+\infty} f(x, y) dy \right] e^{-2\pi iux} dx \end{aligned} \tag{1.7}$$

Ma se $\theta = 0$ allora:

$$P_{\theta=0}(t) = \int_{-\infty}^{+\infty} f(x, y) dy \tag{1.8}$$

Che equivale a:

$$F(u, 0) = \int_{-\infty}^{+\infty} P_{\theta=0}(x) e^{-2\pi iux} dx = S_{\theta=0}(u) \tag{1.9}$$

La trasformata di Fourier della proiezione per $\theta = 0$ equivale alla trasformata della funzione $f(x, y)$ lungo l'asse orizzontale nello spazio delle frequenze (u, v).

Nello spazio di Fourier quindi una proiezione corrisponde ad una retta posta allo stesso angolo θ dello spazio reale, come in figura 1.4.

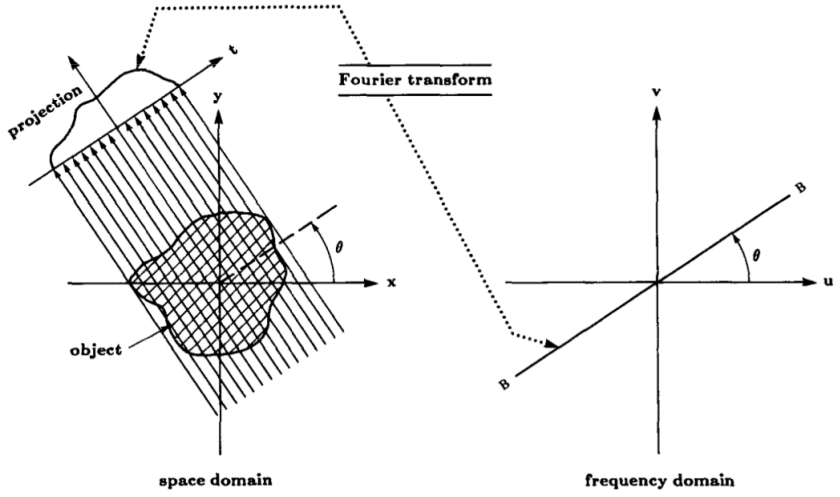

Figura 1.4: Rappresentazione grafica del teorema della slice di Fourier [1].

Quindi grazie a questo teorema prendendo infinite proiezioni, per infiniti angoli, e trasformandole per ottenere $F(u, v)$ nello spazio di Fourier, la sua antitrasformata allora sarà la funzione $f(x, y)$ nello spazio reale. Questo teorema sembrerebbe risolvere il problema della ricostruzione di un'immagine partendo dalle sue proiezioni. In pratica però la proiezione non è continua poiché è quantizzata in termini di *pixel* del rivelatore, quindi nello spazio di Fourier le linee rappresentanti le proiezioni sono rappresentate da una serie di punti appartenenti alla retta corrispondente a ciascuna proiezione. Per trovare la retta si interpola ottenendo così una stima della $F(u, v)$. Interpolando però si va incontro a problemi computazionali di stabilità dei sistemi di equazioni da risolvere per ottenere i coefficienti dell'interpolazione. Inoltre anche θ è discreto (numero non infinito di proiezioni) e quindi, a causa della simmetria radiale delle rette nello spazio di Fourier, i punti noti alle alte frequenze sono molto più distanziati rispetto a quelle più basse (figura 1.5). Per queste ragioni il teorema non può essere applicato direttamente al problema, ma occorrono diverse modifiche, che vengono introdotte con l'algoritmo FBP.

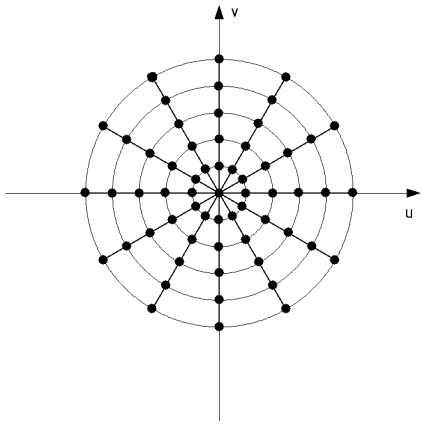

Figura 1.5: Rappresentazione grafica del problema del campionamento disomogeneo nello spazio delle frequenze.

1.4 Algoritmo filtered back projection

Come mostrato nella precedente sezione il teorema della slice non può essere applicato direttamente al problema della ricostruzione tomografica, ma necessita di correzioni particolari. Queste correzioni sono introdotte dall'algoritmo FBP e consentono di ottenere la ricostruzione dell'oggetto in modo omogeneo a tutte le frequenze. L'algoritmo consiste nel filtrare prima le proiezioni attraverso una convoluzione con un filtro e successivamente retroproiettare geometricamente, per ottenere il valore della funzione $f(x, y)$. Sia la fase del filtraggio che quella di retroproiezione dipendono dalla geometria del sistema (*fan beam*, equispaziato o equiangolare, o *cone beam*).

Si consideri l'antitrasformata di Fourier della funzione $f(x, y)$:

$$f(x,y) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} F(u,v) e^{2\pi i(xu+yv)} du dv \tag{1.10}$$

Riscrivendo l'antitrasformata con la seguente trasformazione di variabili:

$$u = w\cos(\theta), v = w\sin(\theta), dudv = wdwd\theta \tag{1.11}$$

e riscrivendo in coordinate polari, si ha:

$$\begin{aligned} f(x,y) &= \int_0^{2\pi} \int_0^{+\infty} F(w,\theta) e^{2\pi i w(x\cos\theta + y\sin\theta)} w dw d\theta \\ &= \int_0^{\pi} \int_0^{+\infty} F(w,\theta) e^{2\pi i w(x\cos\theta + y\sin\theta)} w dw d\theta \\ &+ \int_0^{\pi} \int_0^{+\infty} F(w,\theta+\pi) e^{2\pi i w[x\cos(\theta+\pi) + y\sin(\theta+\pi)]} w dw d\theta \end{aligned} \tag{1.12}$$

Invertendo gli estremi di integrazione del secondo integrale (quindi andando da $-\infty$ a 0) si ha che $F(w, \theta + \pi = F(-w, \theta)$ e fissando

$$t = x \cos\theta + y \sin\theta \tag{1.13}$$

si ha:

$$f(x, y) = \int_0^{\pi} \left[\int_{-\infty}^{+\infty} F(w, \theta)|w| dw \right] d\theta \tag{1.14}$$

Ma per il teorema della slice di Fourier $F(w, \theta) = S_\theta(w)$, quindi:

$$\begin{aligned} f(x, y) &= \int_0^{\pi} \left[\int_0^{+\infty} S_\theta(w)|w| e^{2\pi i w t} dw \right] d\theta \\ &= \int_0^{\pi} Q_\theta(x \cos\theta + y \sin\theta) d\theta \end{aligned} \tag{1.15}$$

Dove:

$$Q_\theta = \int_{-\infty}^{+\infty} S_\theta(w)|w| e^{2\pi i w t} dw \tag{1.16}$$

è la proiezione filtrata con filtro $|w|$.

Questi integrali sono validi nel caso continuo. È necessario quindi discretizzare i risultati ottenuti in precedenza. La prima discretizzazione è quella per calcolare la trasformata della proiezione $P_\theta(t)$ con l'algoritmo numerico della *Fast Fourier Transform* (FFT), che necessita di un numero di punti che sia una potenza di due, e pertanto occorre eseguire uno *zero padding* [1] dei valori non noti se necessario. Per grandi valori di $|t|$ la proiezione è nulla, allora considerando w come frequenza, le frequenze per le quali la trasformata è diversa da zero saranno nell'intervallo compreso fra -W e W, dove W è la frequenza massima. Per il teorema del campionamento il passo di campionamento spaziale lungo le proiezioni risulta T = 1/2W. Quindi: $P_\theta(t) = P_\theta(mT)$, con m che va da -N/2 a N/2-1. Allora la trasformata $S_\theta(w)$ usando la FFT diventa:

$$S_\theta(w) \simeq S_\theta\left(m\frac{2W}{N}\right) = \frac{1}{2W} \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} P_\theta\left(\frac{k}{2W}\right) e^{-2\pi i \frac{mk}{N}} \tag{1.17}$$

Calcolato $S_\theta(w)$ è ora possibile calcolare $Q_\theta(t)$ come:

$$\begin{aligned} Q_\theta(t) &= \int_{-W}^{W} S_\theta(w)|w| e^{2\pi i w t} dw \\ &\simeq \frac{2W}{N} \sum_{m=-\frac{N}{2}}^{\frac{N}{2}} S_\theta\left(m\frac{2W}{N}\right) \left| m\frac{2W}{N} \right| e^{2\pi i m \frac{2W}{N} t} \end{aligned} \tag{1.18}$$

[1] Lo *zero padding* consiste nell'aumentare il numero degli elementi di un vettore con degli zero per avere il numero di elementi desiderato.

Ora occorre trovare il valore della proiezione filtrata lungo i t campionati:

$$Q_\theta\left(\frac{K}{2W}\right) \simeq \sum_{m=\frac{N}{2}}^{\frac{N}{2}} \left(\frac{2W}{N}\right) S_\theta\left(m\frac{2W}{N}\right) \left|m\frac{2W}{N}\right| e^{2\pi i \frac{mK}{N}} \tag{1.19}$$

con k = -N/2 a N/2.

Il problema sembrerebbe ancora una volta risolto in questa forma, ma in pratica è necessario moltiplicare la proiezione filtrata per una funzione, detta finestra di Hamming, per due motivi: il primo e principale è che le formule ottenute in precedenza sono state ricavate nell'ipotesi che la banda di frequenza sia limitata e il secondo è che alle alte frequenze è presente il rumore. È quindi necessario correggere queste ipotesi di lavoro per rendere la ricostruzione più precisa possibile applicando appunto il filtro $H(w)$, in genere una funzione a rampa (figura 1.6).

La proiezione filtrata finale risulta essere:

$$Q_\theta\left(\frac{K}{2W}\right) \simeq \sum_{m=\frac{N}{2}}^{\frac{N}{2}} \left(\frac{2W}{N}\right) S_\theta\left(m\frac{2W}{N}\right) \left|m\frac{2W}{N}\right| H\left(m\frac{2W}{N}\right) e^{2\pi i \frac{mK}{N}} \tag{1.20}$$

Calcolato il valore discretizzato della proiezione filtrata è quindi possibile ricavare il valore ricercato, quello della funzione $f(x, y)$, come la somma su tutti i K angoli campionati θ_i:

$$f(x, y) = \frac{\pi}{K} \sum_{i=1}^{K} Q_{\theta_i}(x \cos\theta_i + y \sin\theta_i) \tag{1.21}$$

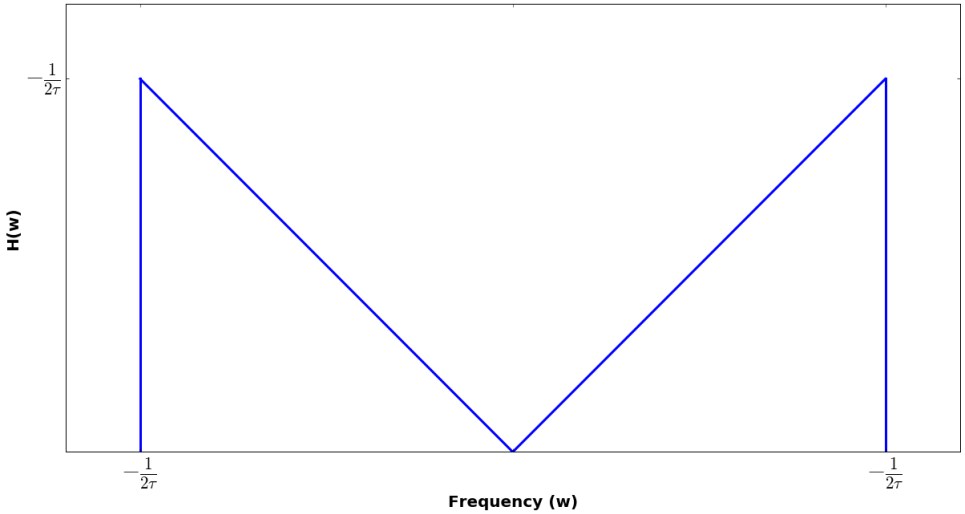

Figura 1.6: Filtro a rampa nello spazio delle frequenze.

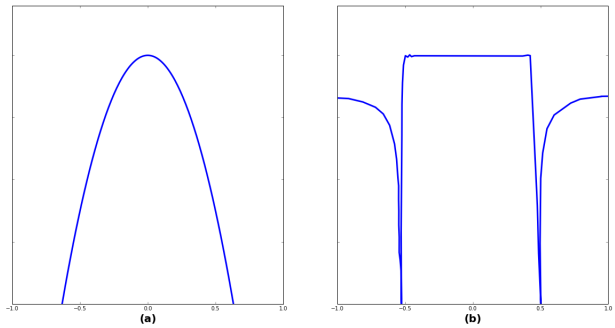

Figura 1.7: Proiezione non filtrata (a) e proiezione filtrata (b).

1.5 Algoritmo di Feldkamp-Davis-Kress

Nei paragrafi precedenti si è proposta una soluzione per la funzione $f(x, y)$ in generale, senza considerare la geometria dell'apparato di acquisizione. In questa sezione si descrive l'algoritmo di ricostruzione di Feldkamp-Davis-Kress (FDK) [4], nella situazione di *detector* equispaziato, come descritto in [1].

L'algoritmo FDK è detto anche *cone beam* perché risolve il problema in cui la sorgente di raggi X è a forma di cono. In questo algoritmo si utilizzano le proiezioni di un oggetto acquisite per vari angoli e poi si ricostruisce la funzione in tre dimensioni $f(x, y, z)$.

1.5.1 Proiezione in tre dimensioni

Prima di illustrare l'algoritmo FDK, occorre definire il concetto di proiezione in tre dimensioni. In tre dimensioni la proiezione di una funzione $f(x, y, z)$ sul piano del rivelatore con coordinate (r, t) è definita dall'integrale:

$$P_{\theta,\gamma}(t, r) = \int_{-sm}^{sm} f(t, s, r)ds \tag{1.22}$$

Dove $P_{\theta,\gamma}(t, r)$ indica la proiezione del fascio inclinato di θ rispetto al piano orizzontale (x, y) e di γ rispetto a quello verticale (y, z) che interseca perpendicolarmente il piano in (t, r) come mostrato in figura 1.8. Nel *cone beam* l'angolo di rotazione è indicato con β e gli integrali di linea che si acquisiscono sono descritti da $R_\beta(p', \xi')$. Poiché la trasformata di Radon vale per proiezioni a fasci paralleli, come per l'algoritmo FBP, è necessario ricavare la proiezione parallela equivalente. La trasformazione appropriata sulle coordinate per trasformare la proiezione $R_\beta(p', \xi')$ in quella equivalente a fasci paralleli $R_\beta(p, \xi)$, definendo D_{so} e D_{de} rispettivamente come la distanza sorgente oggetto e la distanza dal centro di rotazione al detector, si ottiene come:

$$p = \frac{p' D_{so}}{D_{so} + D_{de}}, \qquad \xi = \frac{\xi' D_{so}}{D_{so} + D_{de}} \tag{1.23}$$

$$t = p\frac{D_{so}}{\sqrt{D_{so}^2 + p^2}}, \qquad \theta = \beta + \arctan\left(\frac{p}{D_{so}}\right) \tag{1.24}$$

$$r = \xi\frac{D_{so}}{\sqrt{D_{so}^2 + \xi^2}}, \qquad \gamma = \arctan\left(\frac{\xi}{D_{so}}\right) \tag{1.25}$$

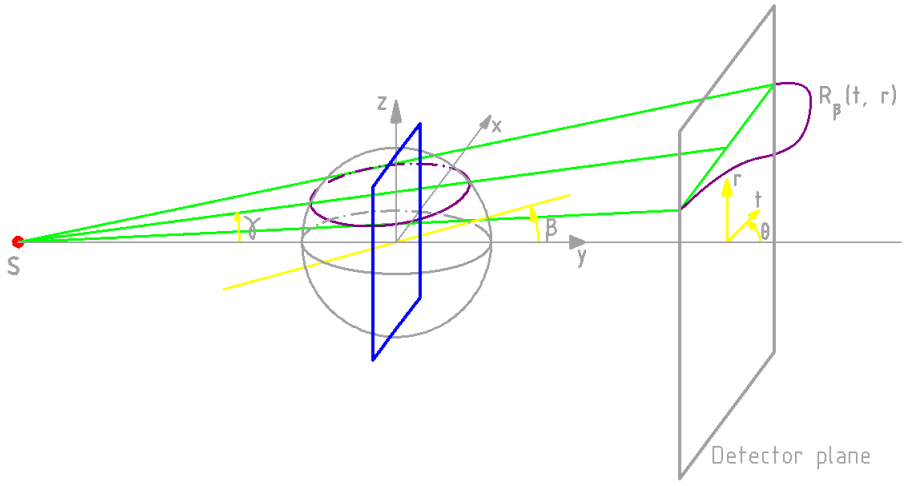

Figura 1.8: Proiezione in tre dimensioni. Il rettangolo in blu rappresenta la *slice* ricostruita utilizzando l'algoritmo FDK.

1.5.2 FBP per il caso cone beam, algoritmo FDK

In questo paragrafo si illustra l'algoritmo di FDK per la ricostruzione della funzione $f(x, y, z)$ a partire dalle proiezioni acquisite nella geometria *cone beam*. Il procedimento mostrato è quello ottenuto da Feldkamp [4], riportato come in [1].

L'idea alla base di questo algoritmo è quella di considerare ogni riga del piano del detector come una proiezione di un ventaglio di raggi X inclinato di un certo angolo, e di utilizzare i risultati ottenuti (si veda appendice A) nel caso di un *fan beam* per detector equispaziati. Data ogni proiezione inclinata rispetto al piano sorgente-rivelatore, è possibile ottenere il risultato della funzione nel punto (x, y, z) sommando il contributo di tutte le proiezioni. Poiché si considera il "cono" di raggi X definito da un insieme di "ventagli" di raggi X, tutti inclinati di un angolo differente fra loro, l'idea è quella di utilizzare i risultati ottenuti per la ricostruzione nel caso di un detector a fasci equispaziati. Poiché ogni ventaglio è inclinato occorre pesare opportunamente i suoi dati per poter utilizzare i risultati del *fan beam* a

fasci equispaziati. Riprendendo le formule ottenute in appendice A, si ha:

$$g(r,\phi) = \frac{1}{2}\int_0^{2\pi} \frac{1}{U^2}\left[\int_{-\infty}^{+\infty} R_\beta(p) h(p'-p) \frac{D_{so}}{\sqrt{D_{so}^2+p^2}} dp\right] d\beta \tag{1.26}$$

$$p' = \frac{D_{so} r\cos(\beta-\phi)}{D_{so}+r\cos(\beta-\phi)} \tag{1.27}$$

$$h(p) = \int_{-W}^{W} \mid w \mid e^{2\pi i w p} dw \tag{1.28}$$

$$U(r,\phi,\beta) = \frac{D_{so}+r\sin(\beta-\phi)}{D_{so}} \tag{1.29}$$

Dove r e ϕ rappresentano le coordinate polari piane sul piano x, y e $R_\beta(s)$ il valore della proiezione sul ventaglio inclinato (figura 1.9).

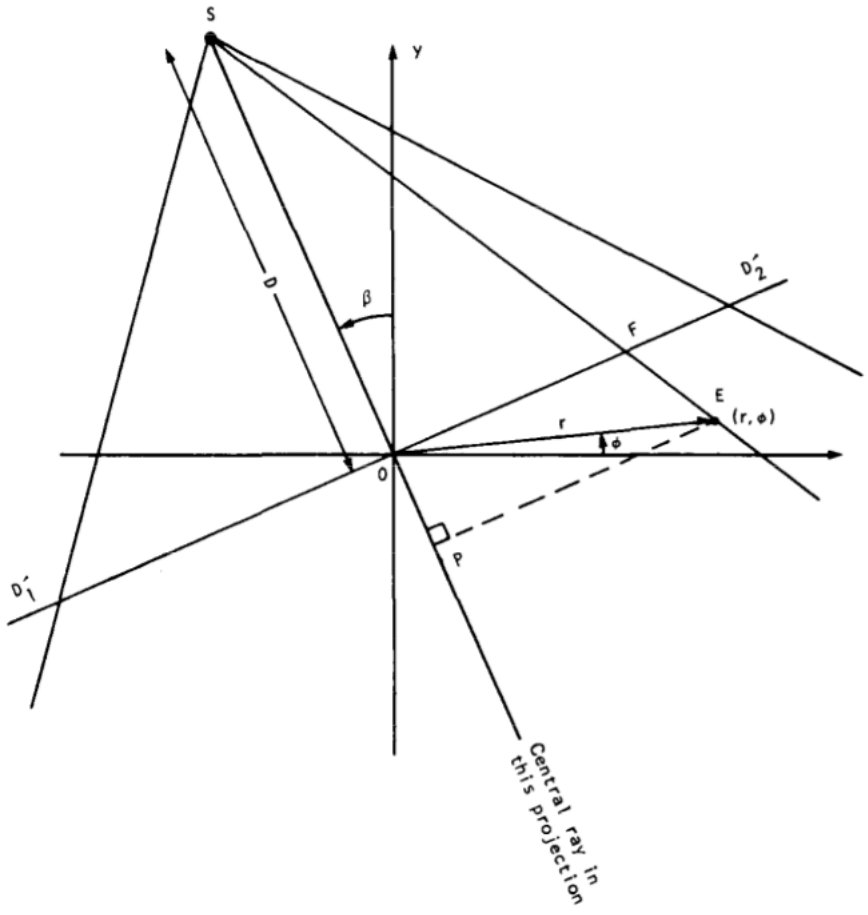

Figura 1.9: Ventaglio di raggi X [1].

Conviene riscrivere queste formule per il sistema (t, s), ruotato dell'angolo di rotazione fra la sorgente e il detector (figura 1.10) utilizzando:

$$t = x\cos\beta + y\sin\beta \tag{1.30}$$
$$s = -x\sin\beta + y\cos\beta \tag{1.31}$$
$$x = r\cos\phi \tag{1.32}$$
$$y = r\sin\phi \tag{1.33}$$

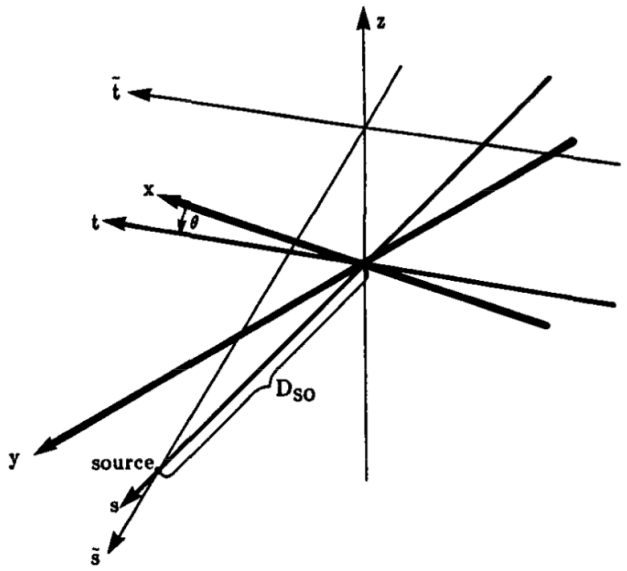

Figura 1.10: Sistema di riferimento ruotato [1].

Quindi l'integrale 1.26, riscritto nel sistema ruotato, risulta:

$$g(t,s) = \frac{1}{2}\int_0^{2\pi} \frac{D_{so}^2}{(D_{so}-s)^2} \left[\int_{-\infty}^{+\infty} R_\beta(p)h(\frac{D_{so}t}{D_{so}-s} - p)\frac{D_{so}}{\sqrt{D_{so}^2+p^2}}dp\right] d\beta \tag{1.34}$$

Ora occorre considerare che il ventaglio di raggi X è inclinato di un certo angolo e pertanto è necessario riscrivere l'integrale precedente nel nuovo sistema inclinato $(\tilde{t}, \tilde{s})$, che tiene conto del fatto che D_{so} è diversa per ogni ventaglio e dipende dall'angolo di inclinazione e anche da quello di rotazione β (figura 1.11).

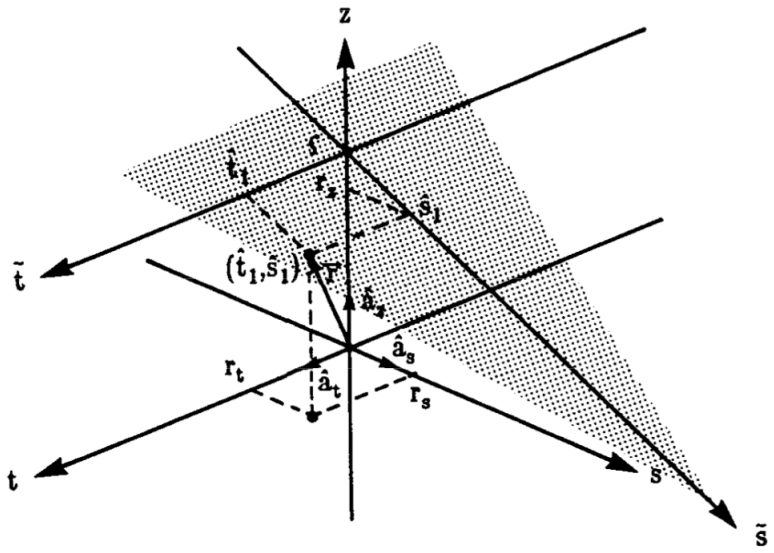

Figura 1.11: Sistema di riferimento inclinato [1].

La nuova distanza D'_{so} sarà quindi uguale a:

$$D'^2_{so} = D^2_{so} + \xi^2 \tag{1.35}$$

e l'incremento dell'angolo $d\beta'$ diventa

$$D_{so} d\beta = D'_{so} d\beta' \tag{1.36}$$

cioè

$$d\beta' = \frac{d\beta D_{so}}{\sqrt{D^2_{so} + \xi^2}} \tag{1.37}$$

Quindi riscrivendo l'integrale 1.34 nel sistema inclinato, sostituendo D_{so} con D'_{so} e β con β' e riscrivendo $R_\beta(p)$ con le nuove trasformazioni, si ottiene:

$$\begin{aligned} g(\tilde{t},\tilde{s}) = \frac{1}{2}\int_0^{2\pi} & \frac{D'^2_{so}}{(D'_{so}-\tilde{s}')^2} \\ & \left[\int_{-\infty}^{+\infty} R_{\beta'}(p,\xi) h\left(\frac{D'_{so}\tilde{t}}{D'_{so}-\tilde{s}} - p\right)\frac{D'_{so}}{\sqrt{D'^2_{so}+p^2}} dp\right] d\beta' \end{aligned} \tag{1.38}$$

Per ritornare al sistema (t,s) si applicano le seguenti trasformazioni:

$$\tilde{t} = t \tag{1.39}$$

$$\frac{\tilde{s}}{D'_{so}} = \frac{s}{D_{so}} \tag{1.40}$$

$$\frac{\xi}{D_{so}} = \frac{z}{D_{so}-s} \tag{1.41}$$

Quindi si ottiene:

$$g(t,s) = \frac{1}{2}\int_0^{2\pi} \frac{D_{SO}^2}{(D_{SO}-s)^2} \left[\int_{-\infty}^{+\infty} R_\beta(p,\xi) h\left(\frac{D_{SO}t}{D_{SO}-s} - p\right) \frac{D_{SO}}{\sqrt{D_{SO}^2+\xi^2+p^2}} dp\right] d\beta \tag{1.42}$$

Applicando questa formula si è in grado di risolvere il problema della ricostruzione di una funzione in tre dimensioni a partire dalle sue proiezioni. La precedente formula permette di ricostruire solo i punti all'interno di una sfera di raggio $D_{so}\sin\Gamma_m$, dove Γ_m rappresenta la metà dell'angolo massimo del cono di raggi X (figura 1.8).

1.5.3 Passi dell'algoritmo FDK

L'implementazione dell'algoritmo viene eseguita seguendo i seguenti passi:

1 Calcolo della proiezione pesata $R'_\beta(p,\xi)$ a partire dalla proiezione acquisita $R_\beta(p,\xi)$ con:

$$R'_\beta(p,\xi) = \frac{D_{SO}}{\sqrt{D_{SO}^2+\xi^2+p^2}} R_\beta(p,\xi) \tag{1.43}$$

2 Calcolo della convoluzione della proiezione pesata con $\frac{1}{2}h(p)$. Questo passo può essere computato per ogni elevazione ξ del ventaglio indipendentemente:

$$Q_\beta(p,\xi) = R'_\beta(p,\xi) * \frac{1}{2}h(p) \tag{1.44}$$

3 Infine retroproiezione geometrica per trovare il valore del pixel nello spazio (t, s, z):

$$g(t,s,z) = \int_0^{2\pi} \frac{D_{SO}^2}{(D_{SO}-s)^2} Q_\beta\left(\frac{D_{SO}t}{D_{SO}-s}, \frac{D_{SO}z}{D_{SO}-s}\right) d\beta \tag{1.45}$$

Capitolo 2

Calcolo parallelo

In questo capitolo vengono introdotti i concetti principali del calcolo parallelo, e, in particolare, quelli riguardanti la parallelizzazione su GPU NVIDIA con CUDA (Compute Unified Device Architecture).

2.1 Introduzione

Il calcolo parallelo è una parte dell'informatica che studia come risolvere simultaneamente più problemi, separandoli in sottoproblemi indipendenti. Questa tecnica in genere permette la risoluzione di problemi molto complessi in tempi di gran lunga minori rispetto ad una soluzione del problema effettuata in modo sequenziale, svolgendo cioè le singole operazioni una di seguito all'altra. Infatti dalla metà degli anni ottanta fino ai primi anni duemila[1] la tecnica per ridurre il tempo di esecuzione di programmi era quella di aumentare la frequenza del processore di numero di operazioni eseguite al secondo. Il problema di questo approccio è però legato al consumo di energia, infatti la potenza segue la legge:

$$P = CV^2F \tag{2.1}$$

dove C è la capacità, proporzionale al numero di transistor il cui stato è cambiato, V il voltaggio e F la frequenza di operazioni al secondo [5]. Aumentare quindi la frequenza è sì utile per aumentare le prestazioni della CPU (*Central Processing Unit*) ma aumenta considerevolmente anche il consumo energetico. Per risolvere questo problema i costruttori di CPU hanno cominciato a proporre sistemi con più core per CPU e più CPU per *device*, potendo allora eseguire più operazioni simultaneamente. Questo nuovo tipo di processori si denominano *multi-core*.

[1] Intel nel 2004 ha deciso di interrompere lo sviluppo del processore a singolo core che stava sviluppando in quel periodo, quello noto come Tejas and Jayhawk, a favore di quelli a *multicore*.

A partire dagli inizi degli anni duemila sono state introdotte le GPU (*Graphics Processing Unit*) per scopi diversi rispetto a quelli della *computer graphics*, creando un nuovo ambito noto come GPGPU (*General-Purpose Computing on Graphics Processing Units*). Le GPU sono componenti elettronici specializzati nella gestione di grafica in due o tre dimensioni, costituiti da migliaia di processori equivalenti alle CPU per natura, ma di grandezza e potenza, in termini di frequenza, molto minori. Avere un numero molto elevato di processori è estremamente efficiente nel caso in cui si debba lavorare su moltissimi dati, infatti processori poco potenti, ma che in parallelo eseguano compiti uguali (condizione nota come SIMD *Single Instruction Multiple Data*), permettono di ridurre notevolmente il tempo di esecuzione del codice. Utilizzare CPU o GPU dipende esclusivamente dal tipo di problema da risolvere, perché in certe situazioni è più conveniente utilizzare le CPU piuttosto che le GPU o viceversa.

2.2 Definizioni

2.2.1 Concorrenza

Si definisce il concetto di concorrenza come la caratteristica di un problema di poter essere scomposto in sottoproblemi indipendenti gli uni dagli altri e che possono essere quindi eseguiti simultaneamente [6]. Per risolvere un problema in parallelo occorre come primo passo mettere in evidenza la concorrenza del problema.

2.2.2 Task

Il *task* è una sequenza di istruzioni che può essere eseguita indipendentemente e contemporaneamente ad altre sequenze e corrisponde ad una parte dell'algoritmo [6]. Dividere in *task* un problema è il primo passo per poterlo eseguire in parallelo.

2.2.3 Unit of execution

Le *Unit of Execution* (UE) sono unità di esecuzione delle operazioni aritmetiche, presenti all'interno delle CPU, che svolgono le istruzioni del programma. Nel contesto del calcolo parallelo si hanno più UE differenti e occorre mappare i vari *task* nelle varie UE come processi o *thread*. I processi sono un insieme di operazioni presenti nel programma che sono eseguite dalle UE. I processi a loro volta sono costituiti di *thread*, le operazioni elementari e concorrenti, eseguibili separatamente in parallelo dei processi [6].

2.2.4 Race conditions

Le *race conditions* è un fenomeno che si presenta nei sistemi concorrenti e avviene quando, in un sistema basato su processi multipli, il risultato finale dell'esecuzione dei processi dipende dalla temporizzazione o dalla sequenza con cui vengono eseguiti. I problemi dovuti alle *race conditions*, in genere, non sono risolvibili perché non direttamente controllabili dal programmatore. Esistono tuttavia una serie di misure che permettono di diminuire il danno dovuto alla gestione delle risorse, come sincronizzare il codice, in modo che più o meno tutti i *task* concorrenti eseguiti in parallelo impieghino circa lo stesso tempo. In questo modo non si verifica quella situazione in cui alcuni *core* sono in attesa che altri finiscono e di rendere indipendenti il più possibile i singoli *thread*, in modo da evitare che a causa di un problema nell'esecuzione di uno di questi gli altri possano continuare indisturbati il loro svolgimento [6].

2.3 Architettura dei sistemi per il calcolo parallelo, tassonomia di Flynn

Nel 1966 Michael J. Flynn (Ingegnere della IBM) scrisse un articolo [7] in cui definì i vari tipi di architetture possibili per sistemi di calcolo. Questa classificazione è nota come tassonomia di Flynn e classifica i vari sistemi di calcolo parallelo in base al flusso di operazioni che possono processare in ogni istante e al flusso di dati su cui essi possono operare simultaneamente.

2.3.1 Single instruction stream, single data stream (SISD)

Facendo riferimento alla Figura 2.1, in questa architettura si processa nella *Processing Unit* (PU) un singolo flusso di operazioni presenti, nell'*instruction pool* (area della memoria in cui sono salvate le istruzioni da eseguire), applicato ad un singolo flusso di dati, presenti nel *data pool* (area della memoria in cui sono salvati i dati). Essa è anche nota come architettura di Von Neumann ed è attualmente in uso sui PC precedenti all'avvento dei *multi-core*.

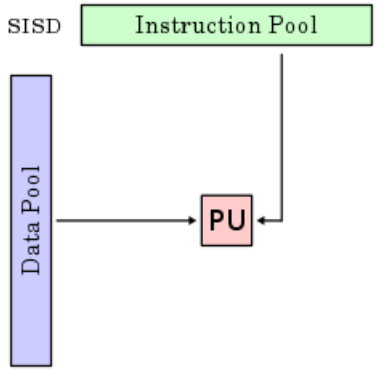

Figura 2.1: Schema dell'architettura SISD, Wikipedia.

2.3.2 Single instruction stream, multiple data streams (SIMD)

In questa architettura si processa un singolo flusso di istruzioni applicato ad una moltitudine di dati caricati su più processori (Figura 2.2). Questa architettura è quella presente in array di CPU o nelle GPU.

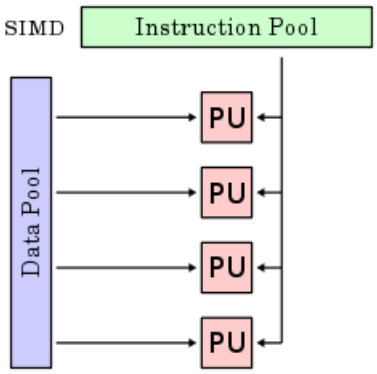

Figura 2.2: Schema dell'architettura SIMD, Wikipedia.

2.3.3 Multiple instruction streams, single data stream (MISD)

In questa architettura si hanno diversi flussi di istruzioni applicati ad un singolo flusso di dati (Figura 2.3). Questa architettura non è molto comune e viene utilizzata su sistemi eterogenei di calcolo; ad esempio, per controllare che il risultato di un particolare calcolo sia corretto, lo si esegue su più sistemi e se ne osservano le eventuali discrepanze, come nel sistema di controllo di volo dello Space Shuttle [8].

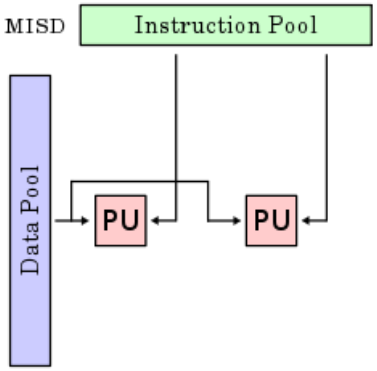

Figura 2.3: Schema dell'architettura MISD, Wikipedia.

2.3.4 Multiple instruction streams, multiple data streams (MIMD)

In questa architettura ogni flusso di istruzioni agisce sul proprio flusso di dati. Questa architettura è la più generale possibile e può essere a sua volta separata in altre a seconda dell'organizzazione della memoria (Figura 2.4).

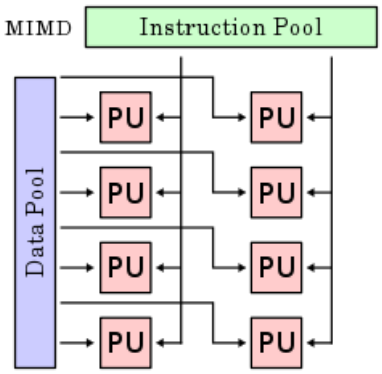

Figura 2.4: Schema dell'architettura MIMD, Wikipedia.

2.3.5 Single instruction, multiple threads (SIMT)

Questa architettura non è presente nella tassonomia di Flynn ma è una sua estensione nata grazie all'utilizzo delle architetture SIMD con il *multithreading*. Nella SIMT si utilizza il fatto che una PU possa eseguire più *thread*. Logicamente equivale alla SIMD, ma ad ogni PU non è associata una sola operazione bensì più operazioni.

2.4 Analisi quantitativa del calcolo parallelo

Il calcolo parallelo nasce per ridurre i tempi di esecuzione di un programma, pertanto è necessario svolgere uno studio più analitico e rigoroso per capire in che modo il calcolo parallelo possa essere sfruttato in maniera ottimale.

Sono due le leggi che meglio sintetizzano lo studio delle prestazioni dell'esecuzione di un programma: la legge di Moore e quella di Amdahl-Gustavson.

2.4.1 Legge di Moore

La legge di Moore, proposta da Gordon Moore (cofondatore di Intel) nel 1965, è data da un'osservazione empirica, basata sul fatto che il numero di transistor presenti in un circuito integrato duplica ogni anno. Successivamente David House (ingegnere della Intel) la corresse dicendo che il numero di transistor di un circuito integrato duplica ogni diciotto mesi. Il numero di transistor all'interno di un processore è legato alle sue prestazioni in termini di operazioni al secondo che possono essere svolte, quindi questa legge prevede come la potenza di calcolo evolva positivamente nel tempo. Naturalmente non può essere vera all'infinito, infatti nei primi anni duemila la potenza di calcolo ha iniziato a saturare a causa del fatto che aumentare il numero di transistor porta ad un aumento di potenza dissipata (relazione 2.1).

Tuttavia, inserire su uno stesso microprocessore più processori, che lavorano in parallelo, permette di aggiornare la legge di Moore in modo che sia ancora valida nella formulazione: il numero di *core* raddoppia ogni due anni (Figura 2.5)

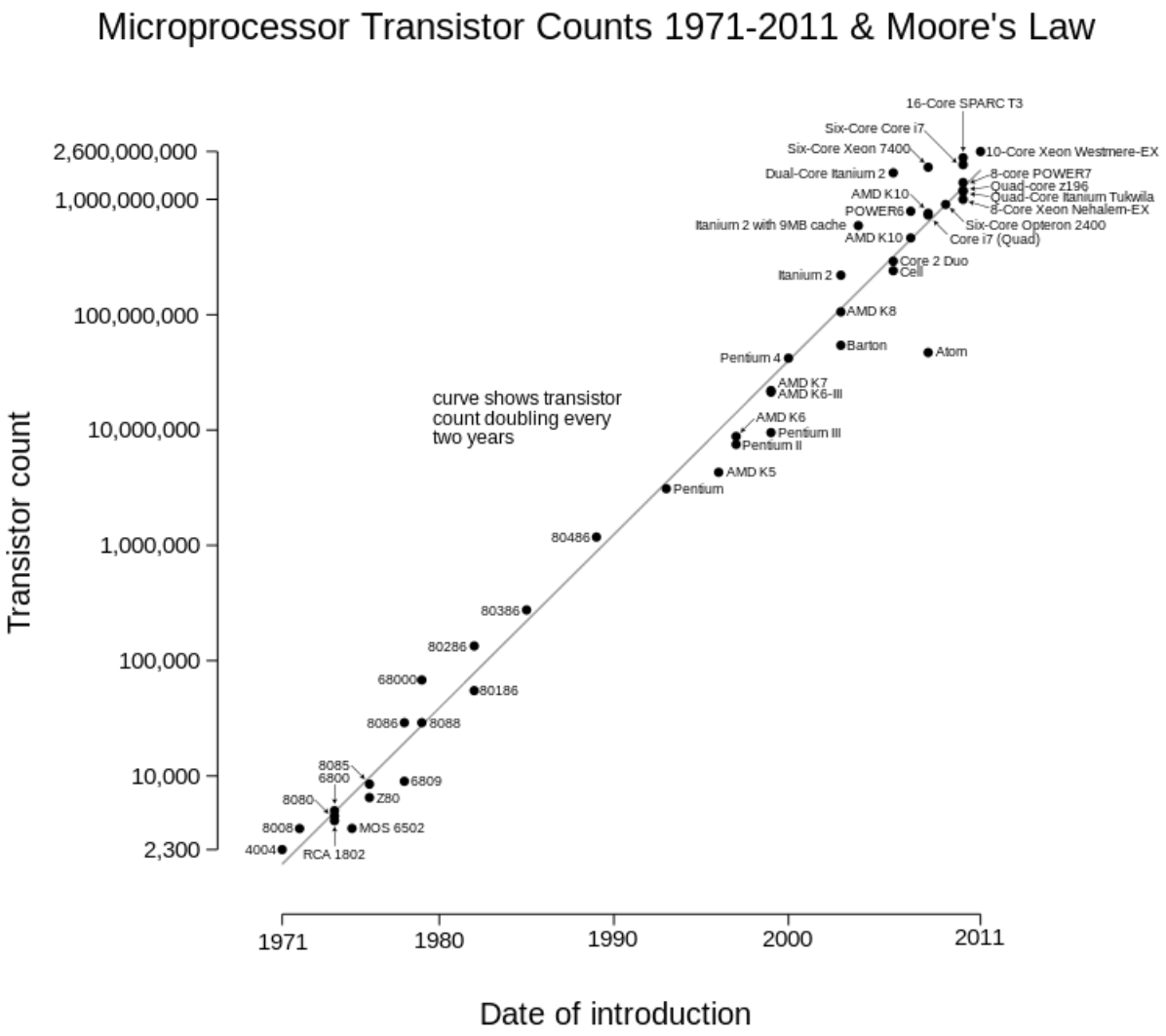

Figura 2.5: Andamento del numero di transistor presenti in un microprocessore e Legge di Moore, Wikipedia.

2.4.2 Legge di Amdahl e legge di Gustafson–Barsis

Si considerano ora le prestazioni, in termini di tempo impiegato, di un codice eseguito in parallelo. Nell'esecuzione di un programma le principali fasi temporali che si possono riscontrare sono tre: il tempo di *setup* (T_{setup}), il tempo computazionale ($T_{compute}$) e il tempo di finalizzazione ($T_{finalization}$). Il tempo totale di esecuzione su un solo processore sarà allora la somma dei

tre tempi [6].

$$T_{total}(1) = T_{setup} + T_{compute} + T_{finalization} \tag{2.2}$$

Se si considerano più processi, l'unica fase che è possibile eseguire su tutti i processi allo stesso tempo è quella computazionale. Allora il tempo totale di esecuzione di un programma nel caso di P processi sarà:

$$T_{total}(P) = T_{setup} + \frac{T_{compute}(1)}{P} + T_{finalization} \tag{2.3}$$

Si definisce lo *speedup* S come il rapporto fra il tempo di esecuzione per un processo con quello per P:

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)} \tag{2.4}$$

e l'efficienza E come lo *speedup* normalizzato al numero totale di processi P:

$$E = \frac{S(P)}{P} \tag{2.5}$$

La situazione in cui lo *speedup* è equivalente a P si dice linearità perfetta e sarebbe la condizione ideale per un sistema a calcolo parallelo, ma nel caso reale questa condizione non si verifica quasi mai a causa dei tempi di *setup* e di finalizzazione che non scalano con il numero di processori P. Conviene allora inserire questi due tempi in un fattore γ, detto fattore seriale, che considera i tempi delle operazioni non riscrivibili in parallelo:

$$\gamma = \frac{T_{setup} + T_{finalization}}{T_{total}(1)} \tag{2.6}$$

Quindi $(1-\gamma)$ rappresenta la frazione di tempo parallelizzabile. Riscrivendo la 2.3 con il coefficiente γ appena definito, si ha:

$$T_{total}(P) = \gamma T_{total}(1) + \frac{(1-\gamma)T_{total}(1)}{P} \tag{2.7}$$

Riscrivendo anche il valore dello *speedup* S(P), si ottiene la legge di Amdahl [9]:

$$S(P) = \frac{T_{total}(1)}{\left(\gamma - \frac{1-\gamma}{P}\right) T_{total}(1)} = \frac{1}{\left(\gamma - \frac{1-\gamma}{P}\right)} \tag{2.8}$$

Nel caso limite in cui P vada all'infinito, S sarà $\frac{1}{\gamma}$. Quindi lo *speedup* viene regolato dalla parte di codice seriale, mostrando il vero limite alla parallelizzazione. Questa relazione è molto importante poiché permette di stabilire se conviene oppure no parallelizzare un algoritmo; infatti se una frazione considerevole del programma è seriale non ha senso compiere grandi sforzi per parallelizzare l'intero codice.

Questa legge però non considera tutti i fattori possibili che potrebbero minare l'efficienza dell'esecuzione parallela di un codice. Il principale fattore trascurato è la competizione alle risorse disponibili, che in genere conduce ad una perdita di efficienza in termini di tempo di esecuzione.

È interessante svolgere anche altri tipi di considerazioni. Ad esempio, ci si potrebbe chiedere come sia possibile aumentare la dimensione di un problema aumentando il numero di processori P in modo da mantenere costante lo *speedup*. Per rispondere a questa domanda la legge di Amdahl non è efficace in questa forma e occorre riscrivere lo *speedup* in funzione dei processi P variabili.

$$T_{total}(1) = T_{setup} + PT_{compute}(P) + T_{finalization} \tag{2.9}$$

Definendo il fattore scalere seriale γ_{scaled} come:

$$\gamma_{scaled} = \frac{T_{setup} + T_{finalization}}{T_{total}(P)} \tag{2.10}$$

Allora il tempo totale diventa:

$$T_{total}(1) = \gamma_{scaled}T_{total}(P) + P(1 - \gamma_{scaled})T_{total}(P) \tag{2.11}$$

Quindi lo *speedup*, in funzione del numero di processi P, vale:

$$S(P) = P + (1 - P)\gamma_{scaled} \tag{2.12}$$

Questa forma è nota come legge di Gustafson–Barsis [10] e permette di valutare, a tempo fissato di esecuzione, come varia lo *speedup* all'aumentare della dimensione del problema. Questa relazione è ottenuta nell'ipotesi che il tempo seriale non vari all'aumentare della dimensione del problema. Nel limite di P che tende all'infinito si ottiene il risultato della legge di Amdahl.

2.4.3 Tempi di latenza

Un altro aspetto molto importante da tenere in considerazione è il tempo di latenza, ovvero il tempo impiegato da un messaggio per essere trasferito da una parte all'altra del sistema. Infatti nel calcolo parallelo si hanno più processori i quali in genere devono, una volta processate le informazioni, inviarle all'*otuput* oppure ad altri processori. Quindi nel tempo di latenza sono incluse le operazioni di I/O (Input/Output) e i tempi di trasferimento dei dati fra i vari processori. Questi tempi dipendono da tempi fisici del sistema e dalla grandezza utilizzata per quantificare la velocità di trasferimento di un'informazione, in termini di byte per unità di tempo, che è detta *bandwidth*.

In termini quantitativi si ha che il tempo di trasferimento di un messaggio è dato da:

$$T_{message-transfer} = \alpha + \frac{N}{\beta} \tag{2.13}$$

dove α è la latenza, ovvero il tempo che impiega il sistema a trasferire un messaggio vuoto, β la velocità di trasferimento, ovvero la *bandwidth*, e N la lunghezza del messaggio.

Questo tempo dipende da tanti fattori, sia *hardware* che *software* e necessita di grande attenzione nella fase di progettazione di un sistema in parallelo.

2.5 Calcolo parallelo su GPU

Le GPU (*Graphic Processing Unit*) sono architetture volte al processamento di dati per la visualizzazione grafica, caratterizzate da una struttura di esecuzione delle operazioni altamente parallelizzata. Questa loro caratteristica ha interessato il mondo scientifico, che ha iniziato a sfruttarle per il calcolo parallelo anche per problemi non tipici della *computer graphics*, aprendo così un nuovo settore di ricerca: il *General Purpose computing on Graphics Processing Unit* (GPGPU).

Le GPU sono processori ottimizzati per avere il maggior numero possibile di operazioni per unità di tempo (*throughput*), mentre le CPU sono ottimizzate per ottenere dei tempi di latenza minimi, passando così da un'operazione all'altra nel minor tempo possibile. Le GPU hanno migliaia di processori, poco potenti e con una memoria *cache* piccola, in cui il tempo di latenza viene nascosto dal fatto che è possibile eseguire molti processi allo stesso tempo. Mentre sulle CPU si hanno pochi processori, molto potenti e con una memoria *cache* grande (Figura 2.6).

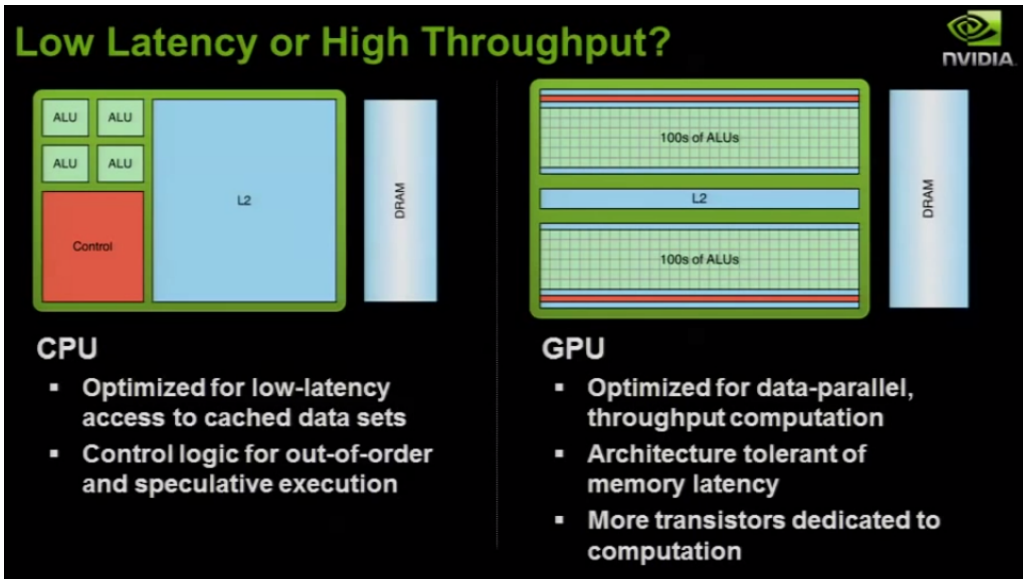

Figura 2.6: Differenze d'architettura tra CPU e GPU [11].

Quindi le GPU sono processori molto efficienti se utilizzati per eseguire le stesse operazioni su moltissimi dati differenti (SIMD). Infatti, come si osserva dalla Figura 2.7, sulle CPU il tempo di latenza risulta essere molto più piccolo rispetto a quello sulle GPU, ma data la presenza di moltissimi processori, nel caso in cui si debba eseguire un numero molto elevato di operazioni questi tempi vengono nascosti dalla possibilità di eseguire le operazioni in parallelo. È proprio dall'analisi di questi tempi che si comprende se un algoritmo è parallelizzabile in modo più efficiente sulle CPU piuttosto che sulle GPU o viceversa.

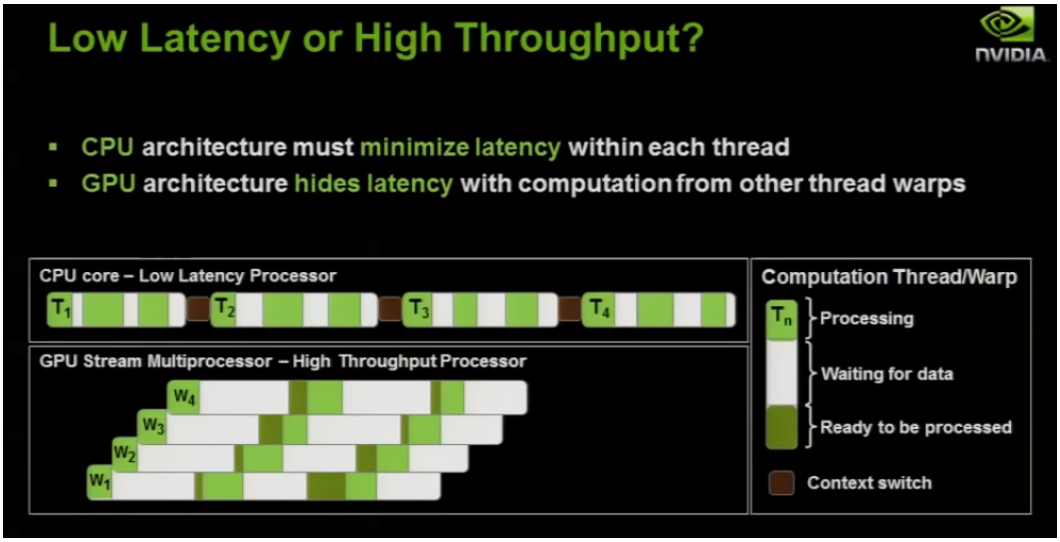

Figura 2.7: Confronto dei tempi di latenza delle CPU e delle GPU [11].

L'elevata potenza di parallelizzazione delle GPU è data da vari fattori: disposizione dei *core* d'esecuzione, gli *Streaming Processor* (SP), in grado di eseguire sequenzialmente un *thread*, organizzati a loro volta in un *array* di multiprocessori, gli *Streaming Multiprocessor* (SM), i quali lavorano contemporaneamente e indipendentemente gli uni dagli altri e da diversi tipi di memoria, uno per ogni livello di granularità di parallelizzazione. La GPU, chiamata *device*, essendo un coprocessore deve essere collegata a un processore, chiamato *host*, per poter eseguire delle istruzioni. Il flusso di dati che permette di eseguire un codice sulla GPU è il seguente (Figura 2.8):

1 Trasferimento dei dati dalla CPU alla GPU

2 Caricamento del programma da eseguire sulla GPU ed esecuzione

3 Trasferimento dei dati ottenuti dalla GPU alla CPU

Capire come questo flusso di dati si verifica, in particolare i tempi con cui avvengono i trasferimenti, è di fondamentale importanza per stabilire se

una data implementazione di un determinato algoritmo sia parallelizzabile in modo efficiente oppure no sulle GPU.

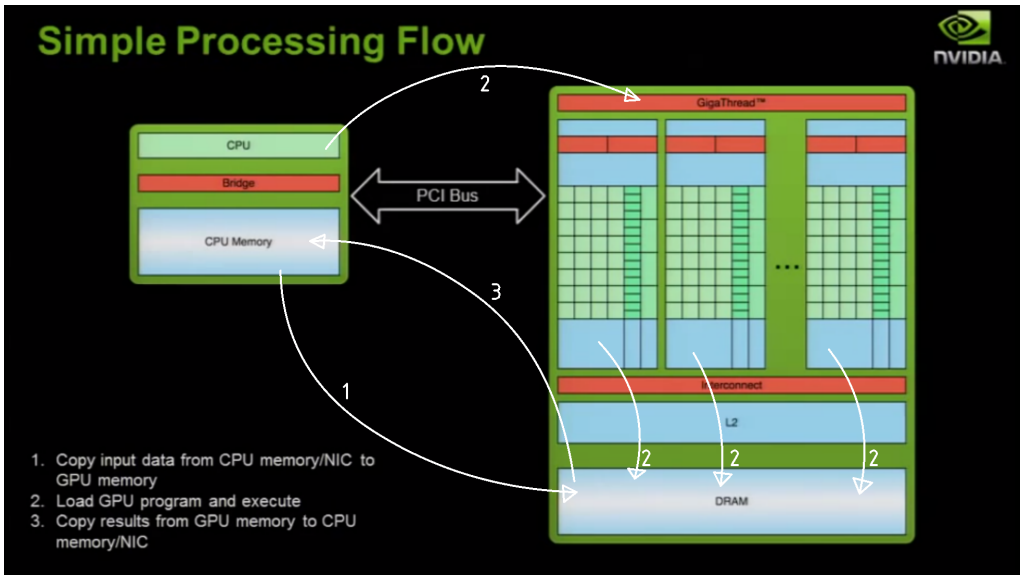

Figura 2.8: Flusso di istruzioni per eseguire un codice su una GPU [11].

2.6 GPU NVIDIA, Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) è un'estensione del linguaggio C, presente anche in altri linguaggi, come C++, Fortran e Python, che ha come obbiettivo quello di permettere di eseguire delle parti di codice sulle GPU NVIDIA in parallelo.

Il codice da portare in CUDA viene diviso in parti sequenziali, elaborate quindi dalla CPU, e parti parallele, elaborate dalla GPU, inserite all'interno di particolari funzioni dette *kernel*, che sono lanciate utilizzando particolari parametri che saranno descritti nel paragrafo successivo.

2.6.1 Organizzazione delle GPU NVIDIA e notazioni

I *thread* in una GPU NVIDIA sono raggruppati in blocchi, a loro volta raggruppati in griglie. Gli *Streaming Multiprocessors* (SM) eseguono i blocchi e ogni *core* esegue un *thread*. Ogni *kernel* quindi lancia una griglia organizzata in blocchi di *thread* che eseguiranno la stessa operazione su dati differenti del problema. Per associare l'unità elementare di parallelizzazione a un particolare dato si utilizzano gli Id identificativi della GPU, che identificano il *thread* e il blocco a cui appartiene, relativi alla griglia lanciata dal *kernel*. Questo modo di impostare il problema rispecchia in pieno il

paradigma SIMT. In estrema sintesi parallelizzare un problema in CUDA significa separare i singoli elementi del problema nei suoi costituenti fondamentali, di mapparli nello spazio della GPU, e di risolverli attraverso l'uso delle istruzioni contenute all'interno del *kernel*.

Secondo lo schema mostrato nella precedente sezione, eseguire un codice su una GPU consiste anche in operazioni di trasferimento di dati fra la CPU e la GPU. Di seguito sono riportate le funzioni CUDA che permettono questo flusso di dati:

- cudaMemcpy(*destinazione, *partenza, cudaMemcpyHostToDevice), per trasferire dati dalla CPU alla GPU
- cudaMemcpy(*destinazione, *partenza, cudaMemcpyDeviceToHost), per trasferire dati dalla GPU alla CPU

I *kernel* hanno una sintassi propria e sono dichiarati in questo modo:

- nome_kernel (argomenti) {*istruzioni*}

Quando invece vanno richiamati si utilizza questa sintasi:

- nome_kernel<<<nB, nTb>>>(argomenti)

Il numero di blocchi (nB) e di *thread* per blocco (nTb) possono assumere dei valori precisi, dipendenti dalla GPU in uso, e avere fino a tre dimensioni, utilizzabili a seconda del problema da risolvere. Il numero totale dei *thread* presenti nella griglia determina il numero complessivo di processi paralleli. L'indicizzazione di *thread* e blocchi nella griglia è possibile grazie a variabili *built-in* a tre dimensioni, accessibili attraverso l'uso di ".x", ".y" e ".z" sulla variabile stessa:

- dim3 gridDim, identifica la dimensione della griglia nelle dimensioni x e y
- dim3 blockDim, identifica la dimensione del blocco nelle dimensioni x, y, e z
- uint3 blockIdx, contiene gli indici (Id) x, y e z del blocco nella griglia
- uint3 threadIdx, contiene gli indici (Id) x, y e z del *thread* nel blocco

Per esempio nel caso di un'immagine è possibile mappare ogni pixel dell'immagine su un processo della GPU e poter quindi lavorare in parallelo su tutti i pixel (Figura 2.9).

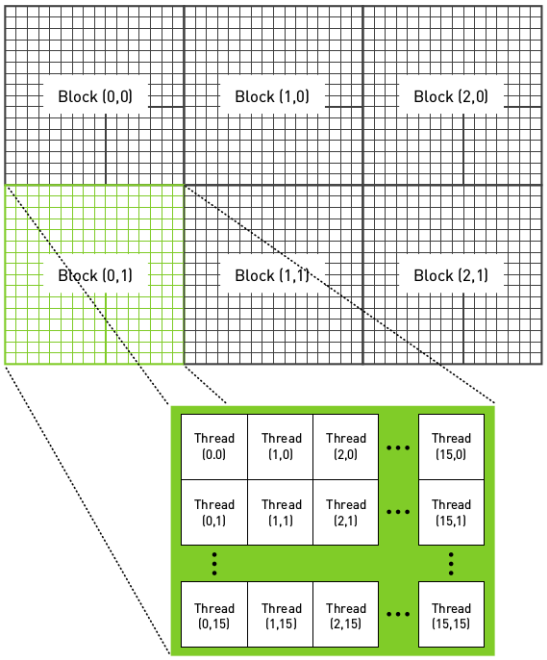

Figura 2.9: Esempio di utilizzo a due dimensioni dello spazio della GPU per un'immagine da 48x32, [12].

2.6.2 Esempio: esecuzione in parallelo della somma di uno scalare ad ogni elemento di un vettore

Ad esempio, dato un vettore di sedici elementi, si aggiunga, in parallelo, ad ogni elemento un numero reale b. Questo problema è a una dimensione, pertanto i blocchi e i *thread* per blocco saranno solo dei vettori. Per problemi a due o tre dimensioni si utilizzano delle matrici o dei cubi di processi. Supponendo di voler utilizzare quattro blocchi, allora il numero di *thread* per blocco più naturale da scegliere risulta quattro. In questo modo si hanno quattro blocchi con al loro interno quattro *thread* per un totale di quattro per quattro, cioè sedici, processi eseguibili in parallelo.

Lo schema con cui ogni elemento del vettore è mappato nella GPU è mostrato nella Figura 2.10, dove in verde si rappresentano i blocchi e in nero i *thread* per blocco. La formula generale per mappare ogni elemento del vettore sulla GPU e poter quindi eseguire la funzione "somma gpu" sarà:

$$idx = blockDim.x * blockIdx.x + threadIdx.x \tag{2.14}$$

Il *kernel* esegue la somma del numero b ad ogni elemento del vettore nel modo seguente:

```
__global__ void somma_gpu(float *vettore, float b, int N){
        int idx = blockDim.x*blockIdx.x + threadIdx.x;
        if(idx < N)
                vettore[idx] += b;
}
```

Si richiama il *kernel* in questo modo, in funzione del numero di elementi da processare:

```
int main(){
        ...
        somma_gpu<<<(4, 1, 1), (4, 1, 1)>>>(vettore, b, N);
}
```

Come detto in precedenza, poiché il vettore ha 16 elementi è sufficiente utilizzare quattro blocchi con quattro *thread* ciascuno per avere i sedici processi.

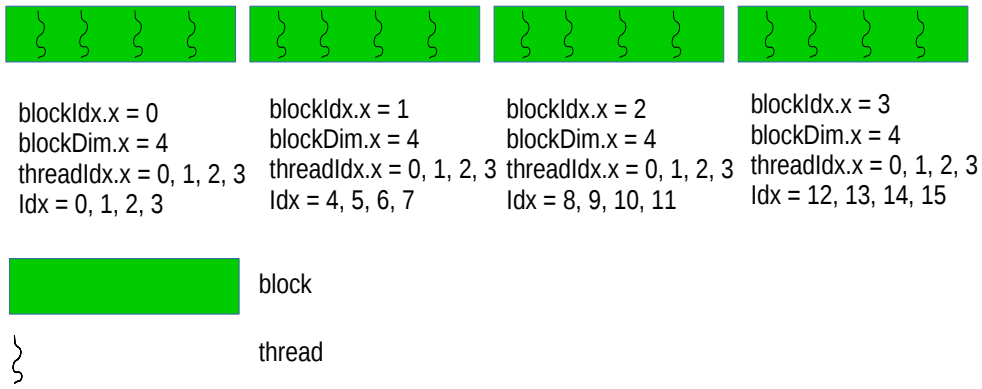

Figura 2.10: Schema della mappatura nello spazio della GPU per l'esempio dato.

Capitolo 3

Strumentazione utilizzata e parallelizzazione dell'algoritmo di Feldkamp

3.1 Descrizione dei device utilizzate

I *device* utilizzati per questo lavoro di tesi sono di due architetture differenti: un cluster *low-power*, composto da *board System-On-Chip* (SoC), e un *cluster* composto da schede con prestazioni molto elevate per calcolo scientifico (*High Performance Computing*, HPC). I primi sono *device* a basso consumo energetico, pensati per ottimizzare la potenza di calcolo in rapporto al consumo energetico, mentre le seconde sono ottimizzate per avere la massima potenza di calcolo possibile senza pensare al consumo energetico.

3.1.1 Cluster low-power, schede System-On-Chip

Le schede del *cluster low-power* sono formate da *chip* elettronici detti *System-On-Chip* (SoC), in cui sullo stesso circuito integrato sono presenti più elementi come CPU, GPU, coprocessori, e altri componenti. Data l'efficienza energetica di questi sistemi essi vengono utilizzati in tutti quei dispositivi in cui il consumo energetico è determinante per il loro funzionamento, come *smartphone* e *tablet*. Negli ultimi anni, dato il crescente aumento della potenza di calcolo, ci si è chiesti se sia possibile o meno utilizzare sistemi SoC per il calcolo scientifico. La possibilità di utilizzare questi sistemi ha l'indubbio vantaggio di risparmiare molta più energia rispetto a sistemi HPC, producendo un immediato e sensibile abbattimento dei costi energetici. Il problema però è la limitata potenza di calcolo e di memoria che questi sistemi ancora hanno rispetto a quelli tradizionali.

Le schede del *cluster low-power* utilizzate per questo lavoro di tesi sono di due tipi diversi: le NVIDIA Jetson Tegra K1 (TK1) a 32 bit, con una

GPU NVIDIA Kepler, e le NVIDIA Jetson Tegra X1 (TX1) a 64 bit, con una GPU NVIDIA Maxwell.

Le specifiche di queste schede sono presentate nell'appendice B.

3.1.2 Cluster High Performance Computing

I sistemi HPC sono progettati per avere la massima potenza di calcolo e pertanto non sono pensati per il risparmio energetico, poiché in genere vengono usati per elaborazioni in campo scientifico. Le schede HPC utilizzate sono la Tesla $K20m$ e la Tesla $K40m$, basate sull'architettura NVIDIA Kepler. Anche le specifiche di queste schede sono presentate nell'appendice B.

3.2 Versioni dei compilatori utilizzati sulle varie schede

In questa sezione si riporta la tabella in cui sono presenti le versioni dei vari compilatori utilizzati per ogni scheda, dove gcc è il compilatore C, nvcc il compilatore per utilizzare CUDA e mpcc il compilatore per utilizzare l'implementazione di MPI con OpenMPI.

Macchina	gcc	nvcc	mpicc
K1	4.8.4	6.5	1.6.5
X1	4.9.0	8.0	1.6.5
K20	4.9.0	7.0	*
K40	4.9.0	7.0	*
HPC	4.8.4	*	1.6.5

Tabella 3.1: Versioni dei vari compilatori utilizzati. "*" segnale che il compilatore non è presente.

3.3 Dataset utilizzati

I *dataset* utilizzati per il lavoro di tesi provengono dall'analisi TAC effettuate nel 2008 presso il centro di conservazione e restauro "La Venaria reale" a Torino, sul Kongo Rikishi, una statua lignea giapponese del XIII secolo [13]. Le radiografie utilizzate si riferiscono alla parte centrale della statua. Da queste sono stati estratti diversi *dataset* di immagini con dimensioni differenti.

Figura 3.1: Analisi tomografica del Kongo Rikishi, a sinistra la ricostruzione tridimensionale di due sezioni della veste del Kongo, a destra la statua con evidenziata in rosso la zona analizzata [13].

3.3.1 Dimensioni dei dataset utilizzati

Il volume da ricostruire in tre dimensioni è definito dai tre limiti spaziali *X_depth*, *Y_depth* e *Z_depth*. *X_depth* indica la dimensione orizzontale della radiografia (numero di pixel della riga del rivelatore che coincide con il numero di pixel orizzontali nella slice) mentre la *Z depth* indica quella verticale (numero di righe del rivelatore e numero di righe della slice). La terza dimensione, quella della profondità, è associata al numero di slice da ricostruire ed è identificata come *Y depth*. Questi valori sono riportati nella seguente tabella.

Dataset	X_ depth	Z_ depth
512	512	179
1024	1024	359
1330	1330	466
2048	2048	718

Tabella 3.2: Tabella delle dimensioni dei *dataset* utilizzati.

3.4 Stato dell'arte della parallelizzazione su GPU dell'algoritmo di Feldkamp-Davis-Kress

Come mostrato in [14] e [15], l'algoritmo FDK è parallelizzabile sulle GPU, ottenendo delle rese migliori che parallelizzando su CPU. L'algorit-

mo FDK è composto da due fasi distinte ed indipendenti: il filtraggio e la ricostruzione, che è la parte più onerosa dal punto di vista computazionale. Data la natura dell'algoritmo, come mostrato nel primo capitolo, la ricostruzione di ogni singolo *voxel* risulta essere indipendente da tutti gli altri. Questa proprietà è fondamentale nella possibilità di risolvere il problema in parallelo. Infatti se un sistema ha un numero enorme di processori, allora ad ognuno di essi si potrà associare la ricostruzione di un singolo *voxel*. Le CPU non hanno un numero sufficientemente alto di processori per poter parallelizzare in questi termini, ma le GPU sì, rendendole lo strumento ideale per risolvere il problema della ricostruzione con FDK.

La situazione ottimale sarebbe parallelizzare sia il filtraggio che la ricostruzione sulle GPU. Dal punto di vista matematico il filtraggio consiste in una FFT e una successiva convoluzione dell'immagine con il filtro (sezione 1.4), ma, come riportato da [16], e anche confermato durante diversi test eseguiti durante questo lavoro, parallelizzare in CUDA la FFT e la convoluzione non porta a significativi vantaggi. Pertanto si è deciso di portare su GPU solo la fase di ricostruzione, la cui parallelizzazione risulta estremamente efficiente sulle GPU. L'operazione di filtraggio invece può essere parallelizzata in modo molto efficiente sulle CPU, utilizzando il paradigma di parallelizzazione *Message Passing Interface* (MPI), come ottenuto da [13]. Di conseguenza si è pensato di unire la parallelizzazione in MPI per la fase di filtraggio con quella di ricostruzione in CUDA in modo da porsi nella condizione migliore possibile.

3.4.1 Parallelizzazione dell'algoritmo di Feldkamp sulle GPU

Il primo problema affrontato durante la parallelizzazione dell'algoritmo FDK su GPU, è stato quello del trasferimento dei dati dalla CPU alla GPU e viceversa. Infatti la memoria della CPU e quella della GPU non sono collegate direttamente, quindi occorre tenere conto del tempo che si impiega per trasferire le radiografie dalla CPU alla GPU e, una volta completata la ricostruzione sulla GPU, trasferire dalla GPU alla CPU il volume appena ricostruito. Questi tempi non sono affatto trascurabili e nemmeno eliminabili, perché non dipendono da fattori controllabili tramite la programmazione. Infatti data la natura dell'algoritmo FDK, volendo ricostruire anche solo una *slice* è necessario utilizzare tutte le radiografie. Questo si traduce nella necessità di caricarle una alla volta sia nella RAM che nella memoria della GPU, e poiché il numero di radiografie non varia e quindi la loro quantità in termini di dati da trasferire rimane costante, il tempo che è necessario per trasferire le radiografie è fisso. Il tempo di trasferimento del volume appena ricostruito dalla GPU alla CPU invece dipende da quante *slice* si ricostruiscono. Infatti più è alto questo valore, tanto maggiore è la quantità di dati da trasferire, con un conseguente aumento del tempo di trasferimento. L'unico modo per ridurre questi tempi è quello di trovare il giusto numero

di *slice* da ricostruire, bilanciando il fatto che è necessario caricare tutte le radiografie in memoria, indipendentemente dal numero di *slice* ricostruite, e che quindi più se ne ricostruiscono a parità di tempo di trasferimento delle radiografie, più il tempo medio di ricostruzione di una singola *slice* [1] risulta basso. Ma di contro, aumentando il numero di *slice* ricostruite aumenta di conseguenza il tempo di trasferimento del volume dalla GPU alla CPU e il relativo tempo di salvataggio. Per capire se fosse più efficiente ricostruire un numero molto alto di *slice* si è acquisito il tempo di ogni singola operazione presente nella sezione di codice in cui è presente la ricostruzione, e valutando quanto incide sul tempo medio per *slice*. Nel grafico seguente si mostra appunto l'incidenza di ogni singola operazione sul tempo medio per *slice* in funzione di *Y_depth*, ovvero il numero di *slice* ricostruite in parallelo.

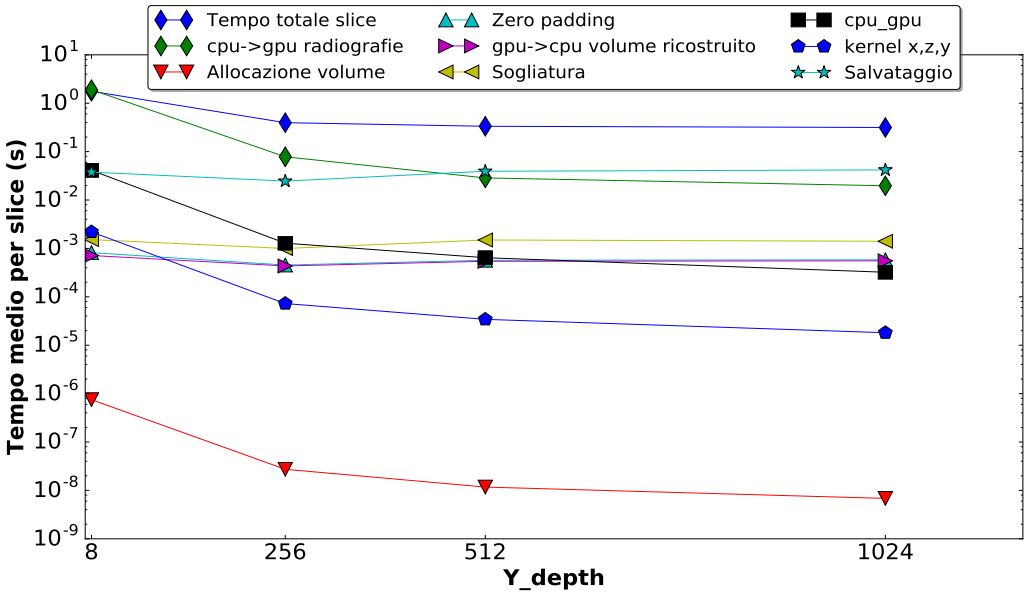

Figura 3.2: Incidenza delle varie operazioni della ricostruzione sul tempo medio per slice per il *dataset* 1330 e sul *device* K40m. I tempi sono stati acquisiti nella situazione di esecuzione più efficiente, quella con il massimo numero di *thread* per blocco.

Come si osserva dalla Figura 3.2 i tempi che incidono maggiormente sulla ricostruzione della *slice* sono il salvataggio del volume e il trasferimento delle radiografie, mentre il tempo di trasferimento del volume dalla GPU alla CPU non incide in maniera significativa. Si osserva inoltre che il tempo per ricostruire, identificato nel grafico in figura come *kernel* x, z, y sia fra i tempi che meno incidono sul tempo di ricostruzione per *slice*. Questo a

[1] Il tempo medio di ricostruzione per *slice* tiene conto di tutte le fasi, dal trasferimento dei dati all'effettiva ricostruzione.

riprova del fatto che parallelizzare l'algoritmo di FDK sulle GPU sia effettivamente la scelta migliore. Dal grafico si osserva che all'aumentare della dimensione del volume il contributo al tempo totale per trasferire le radiografie diminuisce, perché, come detto in precedenza quest'ultimo è costante, mentre il numero di *slice* è maggiore, incidendo pertanto di meno sul tempo per *slice*. Il tempo di salvataggio invece aumenta, come ci si aspettava, ma rimane superiore di poco al tempo di trasferimento delle radiografie sulla GPU quando si superano le 512 *slice* ricostruite. Quindi dal grafico si ottiene che la situazione migliore di utilizzo della parallelizzazione in CUDA sia quella in cui il numero di *slice* ricostruite in parallelo sia il massimo consentito dalla RAM del sistema. Di conseguenza il criterio seguito per scegliere il valore di *Y_depth* nella ricostruzione in parallelo è stato proprio questo.

Come detto in precedenza la fase di ricostruzione dell'algoritmo di FDK è parallelizzabile sui singoli *voxel*. Lo schema del codice della fase di ricostruzione dell'algoritmo è il seguente:

Listing 3.1: Schema del codice della ricostruzione in versione sequenziale.

```
for y=y_start to y_end do
 ConstrainXYZstartXYZendIntoSphere_r();
 for h=0 to nangles do
  Open_SDT_File(h);
  // Calcoli sugli angoli
  for x=x_start+x_shift to x_end+x_shift do
   // Calcolo del peso, delle nuove coordinate
   // t,s e controlli coordinate
   for z=z_start+z_shift to z_end+z_shift do
    // Controlli coordinate
    // Interpolazione sui pixel del detector pixelA...
    // Calcolo del valore del pixel come
    // media pesata dei pixel primi vicini
    val = wA*DataSet.Image_FL[pixelA] +
          wB*DataSet.Image_FL[pixelB] +
          wC*DataSet.Image_FL[pixelC] +
          wD*DataSet.Image_FL[pixelD];
        // Calcolo del valore del
        // pixel x, z della slice
    slice(x, z)+= peso*val;
   end for
  end for
 end for
end for
```

Nella versione sequenziale si osserva come si ricostruisca una *slice* per volta, andando a retroproiettare geometricamente a fissato y (ciclo più esterno)

tutti i pixel dell'immagine. Quando invece si parallelizza l'algoritmo, sia su CPU che su GPU, è più conveniente invertire il ciclo in y con quello in h, cioè retroproiettare geometricamente un angolo alla volta su tutto il volume. Questo da la possibilità di parallelizzare i cicli in x, y e z, cioè sul *voxel*. I tre cicli allora sono stati inseriti all'interno di un *kernel* CUDA, nel quale ogni *thread* esegue la singola iterazione, ricostruendo così l'intero volume, determinato dalle dimensioni delle immagini e da quante *slice* si decide di ricostruire. Lo schema del codice della versione parallelizzata in CUDA sul volume sarà quindi:

Listing 3.2: Schema del codice della ricostruzione in versione parallelizzata.

```
for h=0 to nangles do
 // Calcoli sugli angoli
 Open_SDT_File(h);
 kernel_CUDA<<<(X_depth, Z_depth, Y_depth), 1>>>(...);
end for
```

Il *kernel* ricostruisce tutti i *voxel* retroproiettando un angolo per volta e *X_depth, Z_depth, Y_depth* sono le dimensioni del volume da ricostruire, dove, rispettivamente, *X_depth* corrisponde alla dimensione orizzontale della *slice*, *Z_depth* a quella verticale e *Y_depth* al numero di *slice* da ricostruire. Mentre lo schema del codice per il *kernel* CUDA risulta essere:

Listing 3.3: Schema del codice del kernel CUDA.

```
__global__ kernel_CUDA(...){
    // Indici GPU.
    int ix = blockIdx.x*blockDim.x + threadIdx.x;
    int iz = blockIdx.y*blockDim.y + threadIdx.y;
    int iy = blockIdx.z*blockDim.z + threadIdx.z;
    ix = x_start + x_shift;
    iz = z_start + z_shift;
    iy = y_start;
    // Calcolo del peso, delle nuove coordinate
    // t,s e controlli coordinate
    // Controlli sulle coordinate x e z
    // Interpolazione
    val = ...;
    // Indicizzo sugli indici della GPU
    d_slice(ix, iz, iy)+=peso*val;
}
```

È necessario precisare che, poiché i *core* delle GPU sono tanti ma poco potenti rispetto a quelli delle CPU, occorre prestare molta attenzione alla complessità computazionale che il *kernel* CUDA deve svolgere. Particolarmente onerosi sono i cicli e i controlli. In questa ottica il *kernel* è stato

scritto in modo tale da mantenere solo le operazioni veramente essenziali. Infatti la funzione che determina se il *voxel* x, y, z appartiene alla sfera di ricostruzione non è stata inserita all'interno del *kernel* perché, come osservato da prove eseguite, rallentava in modo eccessivo l'esecuzione parallela del codice, ottenendo risultati perfino peggiori rispetto alla versione sequenziale. È stato quindi osservato che per ottenere un buon codice parallelo su GPU è necessario bilanciare in modo estremamente fine la complessità computazionale dell'operazione inserita all'interno del *kernel*.

3.5 Schema del numero di blocchi e threads per blocco

Come mostrato nella sezione 2.6.1, il lancio di un *kernel* viene eseguito utilizzando il numero di blocchi e il numero di *thread* per blocco. Come mostrato in 3.4.1 la parallelizzazione è stata effettuata sul singolo *voxel* pertanto è necessario che il numero di blocchi e *thread* per blocco sia almeno equivalente al numero di *voxel* da processare. Il parametro che regola l'efficienza dell'esecuzione parallela su GPU è il numero di *thread* per blocco, pertanto in funzione delle dimensioni del *dataset* si modificano il numero di blocchi e quello di *thread* per blocco in modo che il numero totale sia almeno uguale a quello dei *voxel* da processare.

3.5.1 Esempio: ottenere i parametri sul dataset 2048

Si supponga per esempio di voler ricostruire 256 *slice* del *dataset* 2048. La configurazione dei parametri con minor numero di *thread* per blocco è la seguente:

$$kernel\_feldkamp <<< (1024, 718, 256), (2, 1, 1) >>> (...)$$

Il numero di processi per ogni dimensione della GPU corrisponde a quello dell'immagine da processare, infatti:

$$\begin{aligned} x_{GPU} &= 1024 * 2 = 2048 = X\_depth \\ y_{GPU} &= 718 * 1 = 718 = Z\_depth \\ z_{GPU} &= 256 * 1 = 256 = Y\_depth \end{aligned}$$

Il limite fisico di *thread* per blocco è 1024, proprio 32*32, pertanto una delle tante configurazioni ottenibili a massimo numero di *thread* per blocco è quella in cui $nTbx$ e $nTby$, rispettivamente la dimensione x e y nello spazio dei *thread* della GPU, sono uguali 32. Quindi, per ottenere il numero di blocchi per poter utilizzare il maggior numero di *thread* per blocco, si

esegue:

$$nBx = \frac{X\_depth}{32} = \frac{2048}{32} = 64$$
$$nBy = \frac{Z\_depth}{32} = \frac{718}{32} = 22.4375 = 23$$

Si è deciso di lasciare sempre uguale ad 1 la coordinata z dei *thread* (zTh), poiché dai test svolti si è osservato che è il numero totale di *thread* per blocco a regolare l'efficienza, e nono come sono suddivisi nelle varie dimensioni. Pertanto modificare anche la terza dimensione non influisce sulle prestazioni, e, per semplicità, si è deciso di lavorare su solo due parametri.

Quindi uno dei possibili modi di richiamare il *dataset* nella configurazione di massimo numero di *thread* per blocco è il seguente:

$$kernel\_feldkamp <<< (64, 23, 256), (32, 32, 1) >>> (...)$$

Applicando questo ragionamento anche agli altri *dataset* si ottengono le combinazioni dei parametri da utilizzare per il lancio del *kernel*. Nella tabella 3.3 sono riportate le configurazioni dei blocchi (nB) lanciati a fissato numero di *thread* per blocco (nTb) e del numero di *slice* ricostruite in parallelo (Y_depth). La decisione di utilizzare i valori indicati nella tabella per il numero di *slice* ricostruite in parallelo è dovuta alla limitata memoria RAM (*Random Access Memory*) [2] che caratterizza il *device Tegra-K1*, che non consentiva, come per il caso delle macchine del *cluster* HPC, di ricostruire tutto il *dataset* in parallelo. Come detto in precedenza, se Y_depth è il massimo consentito dalla piattaforma si è nella situazione di migliore efficienza, pertanto questi valori sono stati ottenuti in questa ottica. Inoltre si è fissato il valore di Y_depth in modo che tutte le piattaforme su cui sono stati eseguiti i test avessero lo stesso valore in modo da poter fare un confronto consistente fra i risultati ottenuti. Questo però comporta che i *device Tegra-X1*, *Tesla-K20m* e *Tesla-K40m* non lavorano alla massima efficienza, perché la loro RAM è sufficientemente elevata per poter ricostruire anche tutto il volume del *dataset* a dimensione maggiore, cioè il 2048.

[2] RAM: memoria ad accesso casuale, tipologia di memoria caratterizzata dal permettere un accesso diretto e rapido ai dati in essa contenuti.

dataset	512	1024	1330	2048
Y_depth	512	512	512	256
nTB	nB	nB	nB	nB
1	(512, 180, 512)	(1024, 360, 512)	*	*
2	*	*	(1024, 466, 512)	(1024, 718, 256)
4	(256, 90, 512)	(512, 180, 512)	(665, 233, 512)	*
16	(128, 45, 512)	(256, 90, 512)	(333, 117, 512)	(256, 359, 256)
64	(63, 23, 512)	(129, 45, 512)	(167, 59, 512)	(64, 359, 256)
128	*	*	*	(32, 359, 256)
256	(32, 12, 512)	(64, 23, 512)	(84, 30, 512)	*
512	*	*	*	(16, 359, 256)
1024	(16, 6, 512)	(32, 12, 512)	(42, 15, 512)	(64, 23, 256)

Tabella 3.3: Tabella dei parametri utilizzati. "*" segnala una combinazione che si è scelto di non usare.

Capitolo 4

Risultati dei test sperimentali

Dopo le fasi di parallelizzazione dell'algoritmo FDK in CUDA, sono stati svolti una serie di test sperimentali sui *device* descritti nel capitolo 3, al fine di valutare le prestazioni in termini di tempo di esecuzione e consumo energetico

4.1 Tempi della versione sequenziale

Per valutare la bontà dell'esecuzione della versione parallela in CUDA, sono stati riportati nella seguente tabella i risultati dei tempi medi per *slice* della versione sequenziale.

Dataset	T_{K1} (s)	T_{X1} (s)	T_{HPC} (s)
512	8.40 ± 0.02	4.02 ± 0.01	2.999 ± 0.004
1024	38.42 ± 0.05	27.93 ± 0.02	11.877 ± 0.008
1330	41.62 ± 0.01	30.39 ± 0.02	12.26 ± 0.01
2048	187.7 ± 0.5	97.74 ± 0.01	46.537 ± 0.009

Tabella 4.1: Tempo medio per *slice* della versione sequenziale per le varie piattaforme.

4.2 Valutazione migliori prestazioni della versione MPI

Per valutare in che condizioni la versione MPI sia al massimo dell'efficienza si sono svolti diversi test al variare del numero di *slice* ricostruite in parallelo (Y_depth) e del numero di processi lanciati. Quello che si è ottenu-

to è che l'efficienza viene regolata dal numero di *slice* ricostruite in parallelo *Y_depth* e da quanti processi si utilizzano.

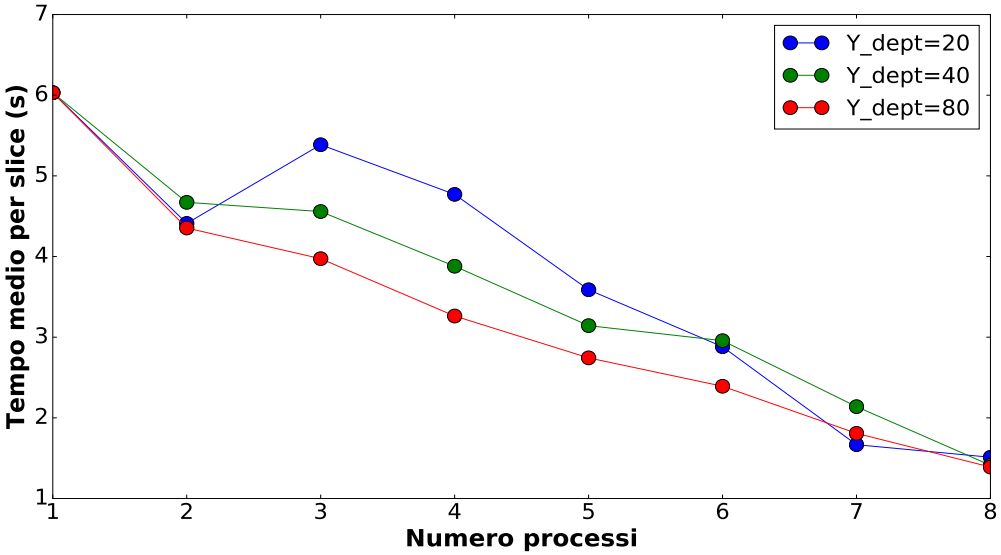

Figura 4.1: Tempo medio per *slice* in funzione del numero di processi lanciati per diverso numero di *slice* ricostruite in parallelo sul *device* XeonD.

Come si osserva dal grafico in Figura 4.1, la situazione migliore è quella in cui *Y_depth* è il massimo consentito dal *device*, e il numero di processi lanciati (in questo caso 8), sia sempre il massimo consentito. Il motivo di questo andamento decrescente all'aumentare del numero di *slice* ricostruite in parallelo è lo stesso della versione parallelizzata in CUDA, cioè che per ricostruire anche solo una *slice* occorre caricare in memoria tutte le radiografie. Pertanto più se ne ricostruiscono più il tempo per *slice* diminuisce, perché, a parità di tempo per caricare in memoria le radiografie, si ricostruiscono più *slice*.

4.3 Risultati dei test riguardanti il tempo medio per slice della versione in CUDA

In questa sezione si riportano i risultati dei test svolti sulle vari *device* della versione parallelizzata in CUDA. I grafici mostrano il tempo medio totale, in cui si considera il tempo necessario per filtrare le radiografie, i vari tempi per il trasferimento dei dati e il tempo per la ricostruzione vera e propria, impiegato per ricostruire una *slice* in funzione del numero di *thread* per blocco. Inoltre, per confronto, si inserisce anche il miglior tempo ottenuto con la versione MPI, in cui, oltre alla ricostruzione, è parallelizzato

anche il filtraggio. Si sottolinea che la versione CUDA invece ha solo la ricostruzione parallelizzata, mentre il filtraggio è nella sua forma sequenziale (sezione 3.4).

Si riportano ora i grafici dei tempi medi per *slice* in funzione del numero di *thread* per blocco di ogni *dataset*. A causa della scala gli errori sperimentali sulle misure non sono visibili perché dell'ordine di 0.01 s.

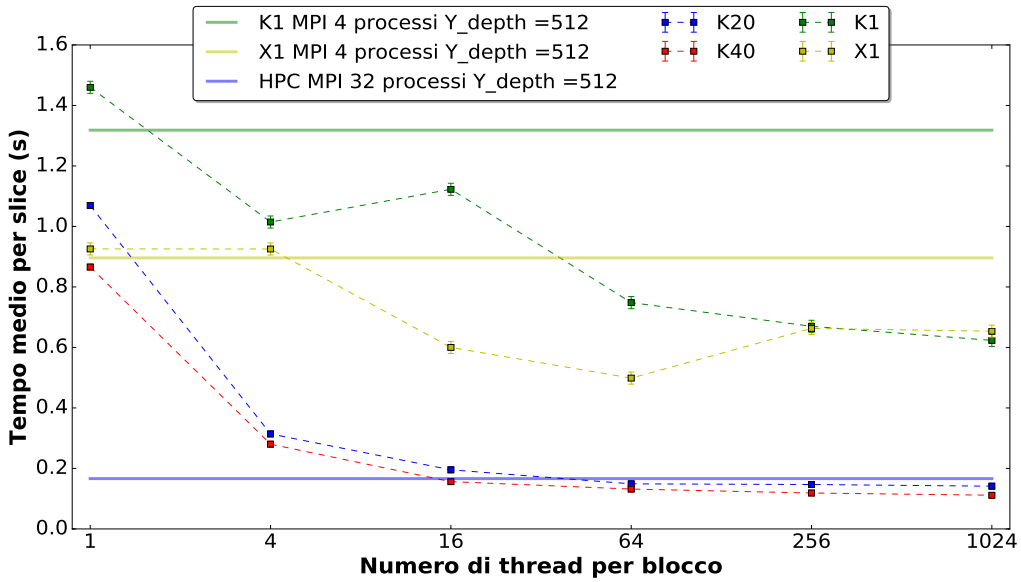

Figura 4.2: Risultati per il *dataset* 512, con $Y\_depth = 512$.

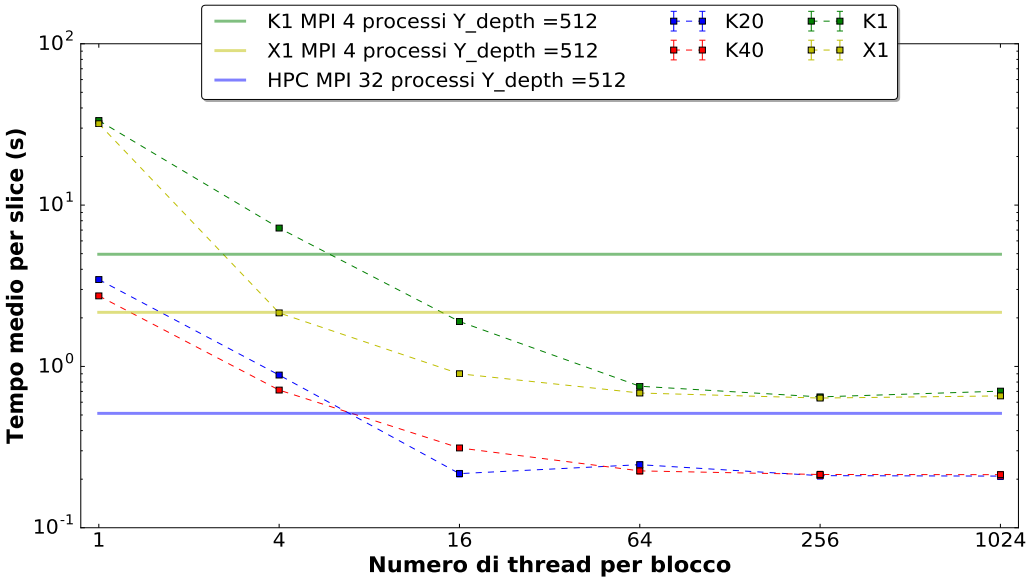

Figura 4.3: Risultati per il *dataset* 1024, con *Y_depth* = 512.

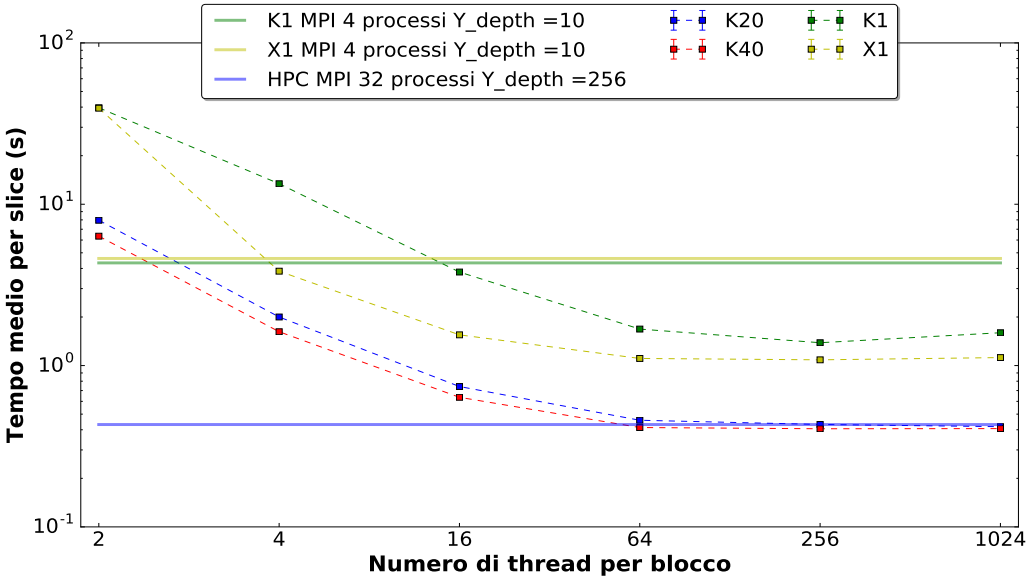

Figura 4.4: Risultati per il *dataset* 1330, con *Y_depth* = 512.

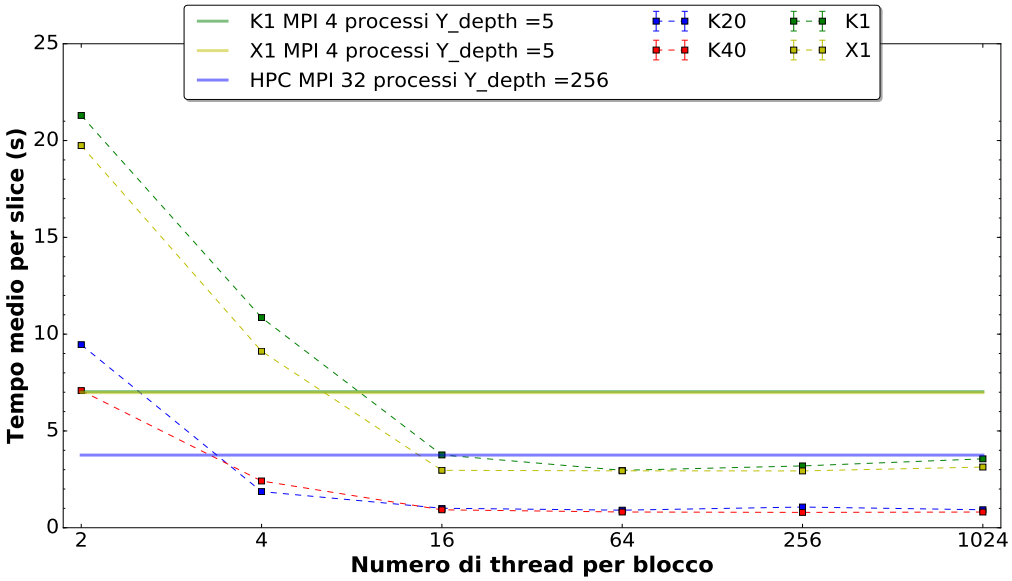

Figura 4.5: Risultati per il *dataset* 2048, con *Y_depth* = 256 sul *device* X1 e *Y_depth* = 128 sulla K1.

4.3.1 Discussione dei risultati

Dai grafici si osserva come il parametro che regola le prestazioni dell'esecuzione del codice in CUDA sia il numero di *thread* per blocco. Infatti si vede che all'aumentare di questo parametro il tempo medio per *slice* diminuisce. I tempi della versione in CUDA sono sempre inferiori rispetto a quelli ottenuti in MPI nella migliore delle situazioni, nonostante in CUDA si parallelizzi solo la ricostruzione, mentre in MPI anche il filtraggio. Questo aspetto mostra come la parallelizzazione di questo algoritmo sulle GPU sia la scelta migliore.

Inoltre si è osservato un dato molto importante, che vale la pena sottolineare. Grazie ai *tool* di monitoraggio di NVIDIA, *tegrastats*, per TK1 e TX1, e *nvidia-smi* per le GPU del *cluster* HPC, è possibile analizzare in tempo reale l'utilizzo delle varie risorse (grafici in Figura 4.6). Si osserva che durante l'esecuzione del codice nella fase di filtraggio e salvataggio, parte iniziale e finale rispettivamente (eseguite in modo sequenziale), la GPU sia giustamente allo 0%, mentre è una sola CPU per volta a lavorare al 100% (grafici centrali in Figura 4.6). Invece, nel momento in cui inizia la ricostruzione, cioè la parte di codice portata in CUDA, la GPU viene utilizzata totalmente, raggiungendo quindi il 100% (ultimo grafico in Figura 4.6). Questo andamento lo si riscontra sia sul *cluster low-power* che su quello HPC. Si ritiene che tale risultato sia significativo e da sottolineare, poiché in genere si hanno utilizzi inferiori compresi fra il 50% e il 70% [16].

Poiché l'efficienza di utilizzo delle risorse della GPU dipende fortemente da come è stato scritto il *kernel*, si ottiene quindi, oltre a un'alta efficienza di esecuzione, anche un'ottima gestione delle risorse.

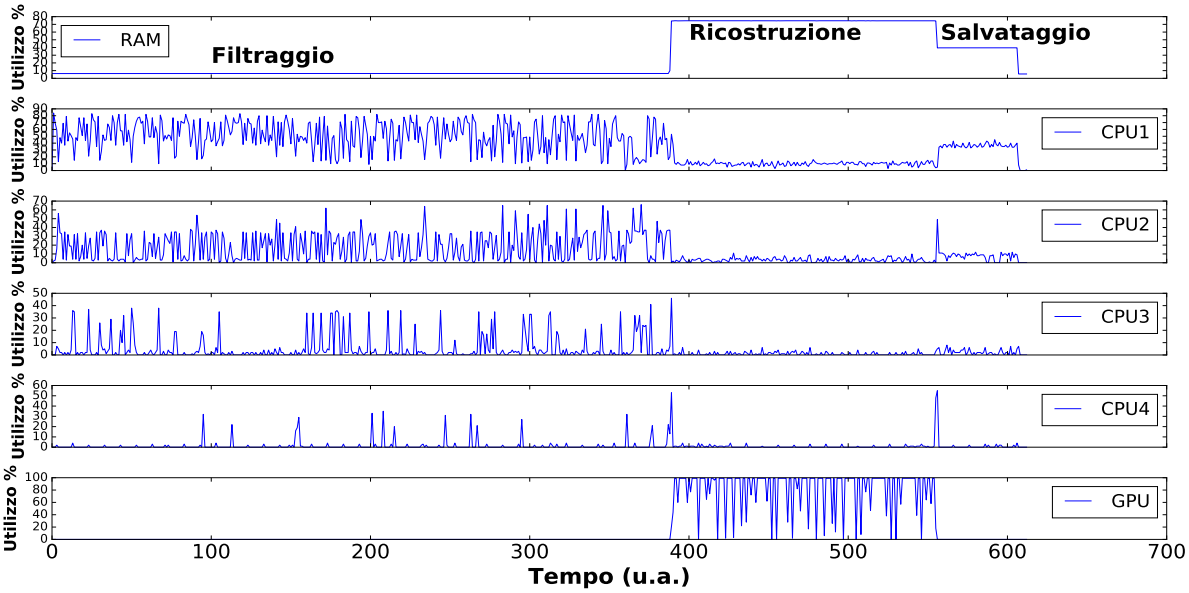

Figura 4.6: Utilizzo delle varie risorse sul *device* K1 del *dataset* 1330 con *Y_depth*=512 e 1024 *thread* per blocco. Si osserva nell'ultimo grafico l'utilizzo al 100% della GPU nella fase di ricostruzione.

4.4 Verifica della correttezza della ricostruzione delle immagini

Per valutare se l'immagine ricostruita in CUDA è corretta la si confronta con quella ricostruita in sequenziale. Per valutare se ci sono delle differenze significative fra le due immagini si calcola la deviazione *standard* dell'immagine differenza (Figura 4.7) e la si confronta con quella di una regione di interesse (ROI, *Region of Interest*) sia della versione CUDA che di quella sequenziale di una parte uniforme del fondo. Considerare la deviazione *standard* in una zona uniforme significa valutare il rumore presente nell'immagine. Quindi se il rumore dell'immagine ricostruita è maggiore di quello dell'immagine differenza fra l'immagine ricostruita in CUDA e quella in sequenziale, allora le oscillazioni presenti fra le due immagini non sono tali da poter essere osservate. Pertanto le due immagini si potranno considerare equivalenti.

Nella seguente tabella si riportano il valore medio dell'immagine differenza, il valore della deviazione *standard* dell'immagine differenza e le deviazio-

ni standard delle ROI scelte in una zona uniforme del fondo delle immagini ricostruite con CUDA e in sequenziale.

Dataset	Media differenza	$\sigma_{differenza}$	σ_{fondo_CUDA}	$\sigma_{fondo\_sequenziale}$
512	1e-07	2e-06	2e-03	2e-03
1024	1e-07	2e-07	3e-03	3e-03
1330	1e-07	1e-07	3e-03	3e-03
2048	3e-07	4e-07	6e-03	6e-03

Tabella 4.2: Risultati ottenuti dal confronto fra l'immagine ricostruita in sequenziale e quella in CUDA.

Come mostrato dai valori in tabella 4.2, la deviazione *standard* dell'immagine differenza è diversi ordini di grandezza minore rispetto a quella della regione di interesse del fondo uniforme.

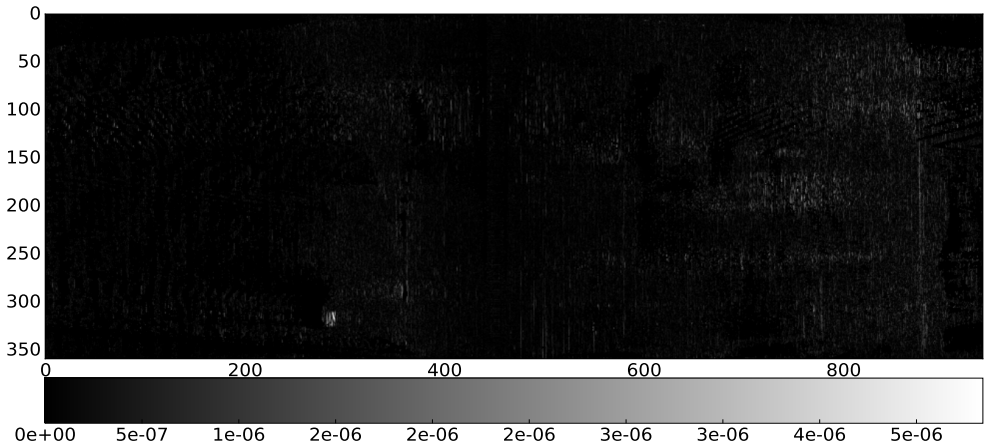

Figura 4.7: Esempio di immagine differenza per il *dataset* 1024.

4.5 Risultati dei test riguardanti i consumi energetici

In questa sezione si riportano i risultati del consumo energetico relativo all'esecuzione del codice che implementa l'algoritmo di FDK. A causa della scala le barre d'errore sull'energia non sono sempre visibili. L'errore sull'energia è dell'ordine di 0.1J. I test sono stati eseguiti sia per la versione

in CUDA che per quella in MPI. Per la stima della potenza dissipata si è dovuto procedere in due modi differenti: sul *cluster* HPC, dove non è stata possibile, durante il lavoro di tesi, un'acquisizione diretta, si è considerata una potenza costante nel tempo di 350 W. Questa stima è ottenuta considerando circa 100 W per ciascuna CPU xeon (dual xeon), 150W per le K20m e K40m e ulteriori 100 W per consumi accessori dovuti a ventole e alimentatori. Sul *cluster low-power*, invece, si è montato un multimetro in grado di misurare la corrente in funzione del tempo e la tensione di alimentazione. Da tali misure è stato possibile poi ricavare il valore della potenza della nota relazione:

$$P = VI \tag{4.1}$$

A causa delle diverse prestazioni delle versioni CUDA ed MPI, i valori del numero di *slice* ricostruibili in parallelo varia notevolmente. Nella tabella 4.3 sono riportati i valori di *Y_depth* utilizzati per ogni architettura e tipologia versione di parallelizzazione in funzione del *dataset*.

dataset	K1 C	K1 M	X1 C	X1 M	K20 C	K40 C	HPC M
512	512	512	512	512	512	512	512
1024	512	50	512	50	512	512	512
1330	512	10	512	10	512	512	266
2048	128	5	256	5	256	256	256

Tabella 4.3: *Y_depth* per la versione CUDA e MPI su tutte le schede, dove C indica la versione in CUDA e M quella in MPI.

4.5.1 Cluster HPC

Per calcolare l'energia media per slice per la versione CUDA sul *cluster* HPC si è considerata una potenza costante di 350 W. Per ricavare l'energia totale si moltiplica la potenza per i tempi delle due operazioni di filtraggio e ricostruzione:

$$E_{tot} = (T_{filt} + T_{rec})\, P \tag{4.2}$$

dove T_{filt} e T_{rec} sono rispettivamente il tempo di filtraggio e quello di ricostruzione. L'energia media per slice vale allora:

$$E_{slice} = \frac{E_{filt} + E_{rec}}{Y\_dataset} \tag{4.3}$$

dove *Y_dataset* è il numero totale delle *slice* per un dato *dataset*.

4.5.2 Cluster low-power

Ai fini del calcolo della potenza dissipata sul *cluster low-power* si è invece potuto misurare la corrente entrante nel *device* e la tensione di alimentazione, costante, in funzione del tempo, durante l'esecuzione del codice. In

questo modo è stato possibile tracciare dei grafici della corrente in funzione del tempo, ottenendo così quelli della potenza sempre in funzione del tempo, utilizzando la relazione 4.1. In questo caso l'energia totale è stata calcolata come l'integrale della potenza. Come si osserva dai grafici, la potenza assume valori ben distinti nelle fasi di filtraggio, ricostruzione e salvataggio (Figura 4.8). Per calcolare l'energia totale si è utilizzato il metodo numerico del trapezio per il calcolo degli integrali, mentre l'errore è stato stimato con la propagazione degli errori, come mostrato nell'appendice C.

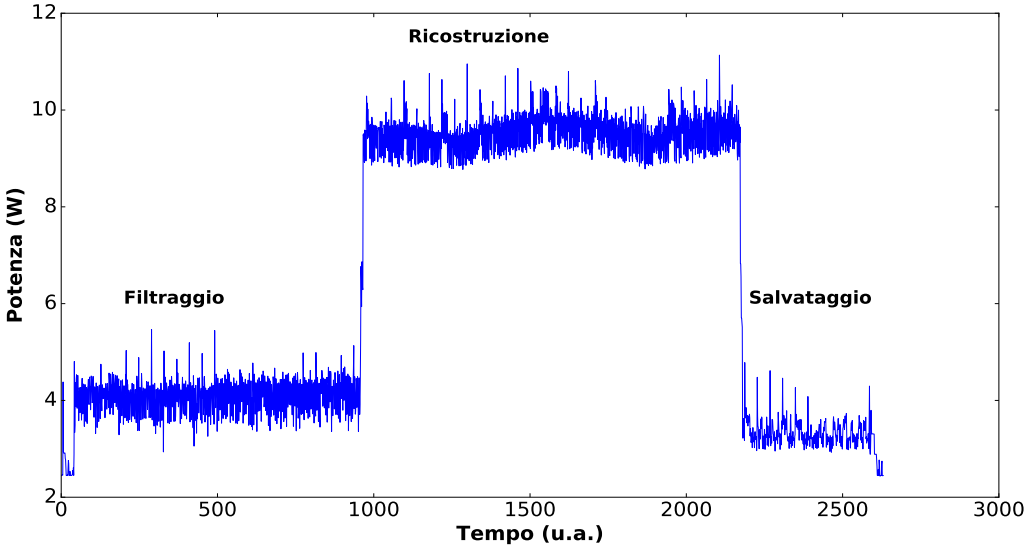

Figura 4.8: Potenza in funzione del tempo per il *dataset* 1024 sul *device* K1 con *Y_depth*=512 e 1024 *thread* per blocco.

4.5.3 Confronto energia per slice e tempo per slice

Nei seguenti grafici sono riportati i valori dell'energia media per slice e del tempo medio per slice di ogni *dataset* e su ogni architettura utilizzata. Sulla scala di sinistra è riportato il valore del tempo medio per *slice*, mentre su quella di destra quello dell'energia media sempre per *slice*. Ci si aspetta che le schede del *cluster low-power* abbiano un consumo energetico molto minore rispetto a quelle dell'HPC, ma che il tempo, invece, sia meno efficiente. Da questi grafici è possibile stabilire se convenga o meno utilizzare una architettura piuttosto che un'altra, considerando sia le prestazioni energetiche che quelle in termini di tempo di esecuzione del codice.

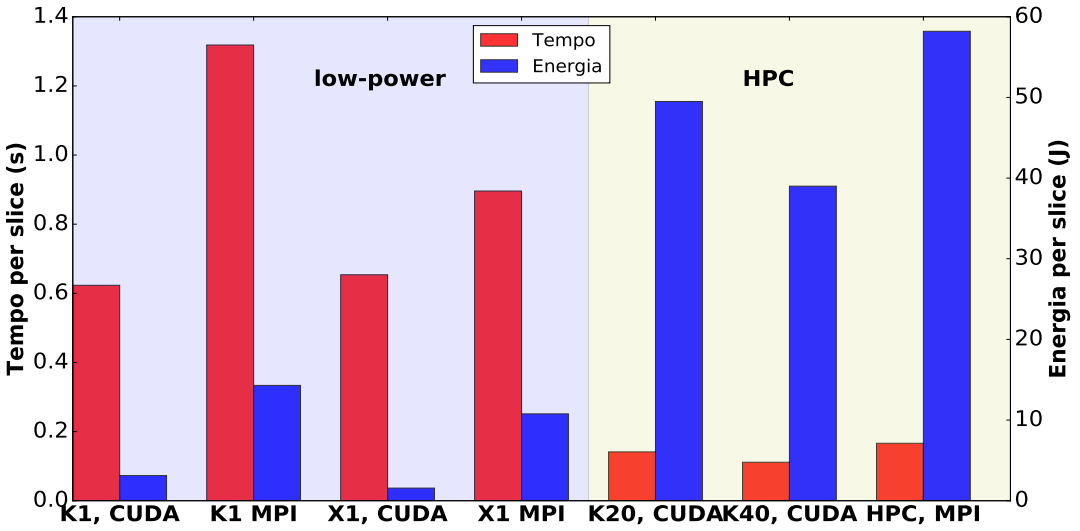

Figura 4.9: Tempo ed energia per slice per il *Dataset* 512.

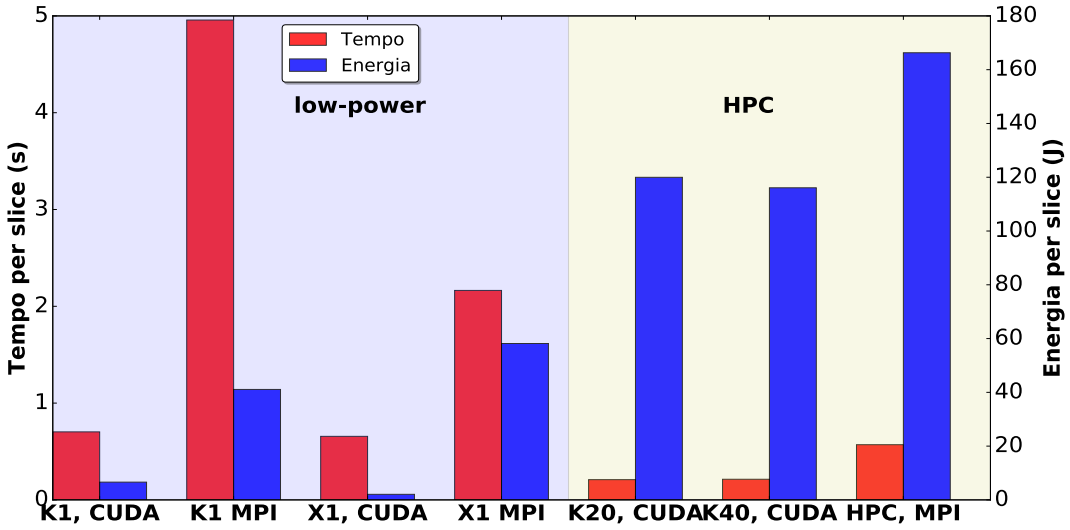

Figura 4.10: Tempo ed energia per slice per il *Dataset* 1024.

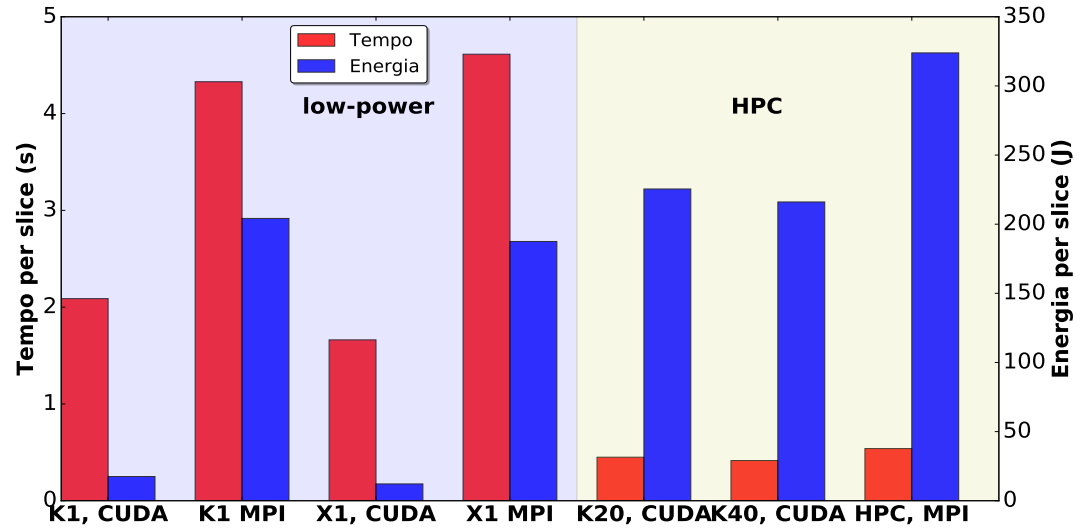

Figura 4.11: Tempo ed energia per slice per il *Dataset* 1330.

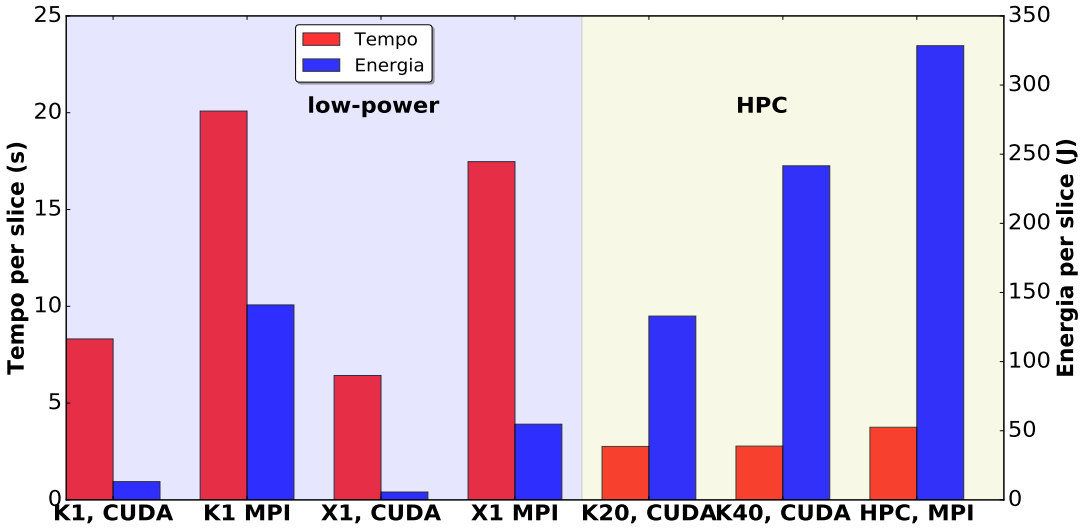

Figura 4.12: Tempo ed energia per slice per il *Dataset* 2048.

4.5.4 Confronto dell'energia per slice in funzione della dimensione del dataset

Nel seguente grafico si riportano l'energia media per *slice* e il tempo medio per *slice* in funzione della dimensione del *dataset*. È interessante vedere come all'aumentare della dimensione del *dataset* i tempi medi per *slice* e l'energia media per *slice* aumentino. Questo perché all'aumentare del tempo totale il *device* deve lavorare per più tempo, dissipando una quantità di energia maggiore.

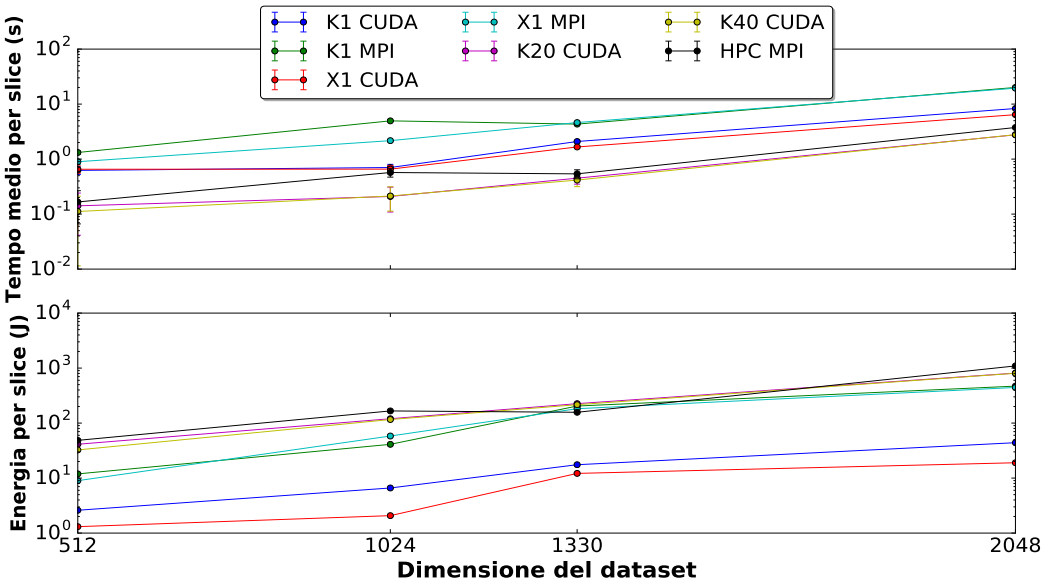

Figura 4.13: Energia e tempo medio per slice in funzione della dimensione del *dataset*.

4.5.5 Discussione dei risultati

Come è ben visibile dai grafici a barre riportati dalle figure 4.9 a 4.12, nei sistemi HPC il tempo di esecuzione è molto più basso che in quelli *low-power*, ma l'energia spesa per ricostruire una *slice* è molto più alta. Inoltre si osserva come, all'aumentare della dimensione del *dataset*, l'energia media spesa per *slice* e il tempo medio per *slice* aumentino di conseguenza. Il motivo dell'aumento dell'energia spesa è legato a quanto tempo il *device* deve lavorare, e quindi, aumentando la dimensione delle immagini aumenta il tempo di esecuzione. Quindi se da un lato le architetture HPC sono molto più efficienti in termini di tempo medio per *slice* che quelle *low-power*, dall'altro consumano molta più energia.

Per valutare quale sia l'architettura più efficiente dal punto di vista energetico si calcolano quante slice sono ricostruite per unità di tempo e di ener-

gia e successivamente si calcola il parametro di efficienza come il prodotto del numero di slice ricostruite per unità di tempo con quelle per unità di energia. Di seguito si riportano per ogni *dataset* i grafici delle *slice* ricostruite al per unità di tempo ed energia, calcolate su ogni *device* su cui sono state eseguite delle prove.

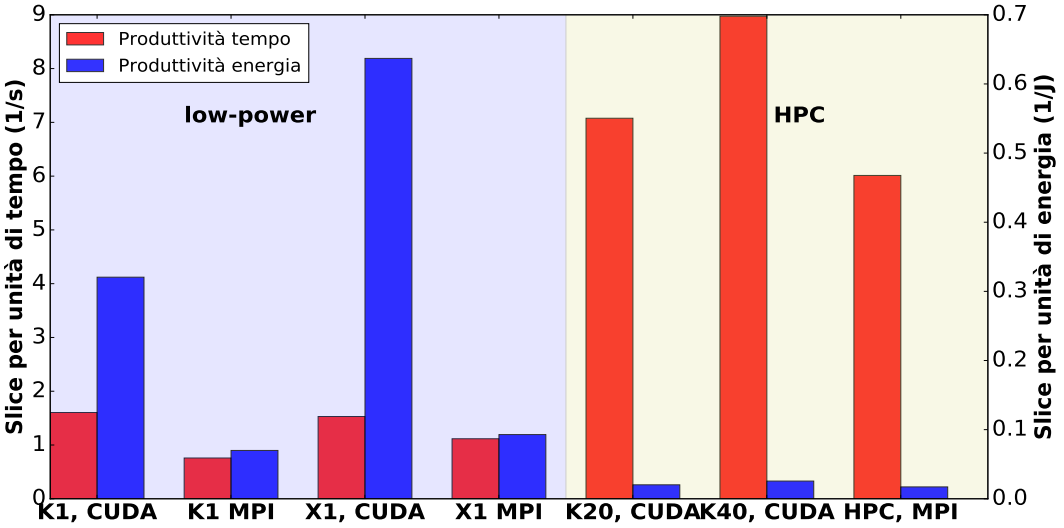

Figura 4.14: *Slice* per unità di tempo ed energia per il *dataset* 512.

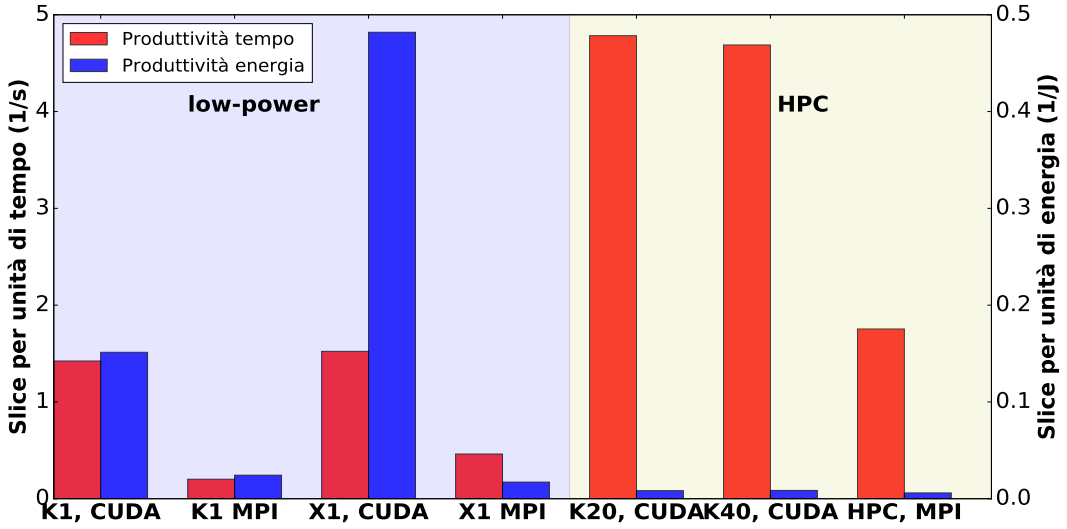

Figura 4.15: *Slice* per unità di tempo ed energia per il *dataset* 1024.

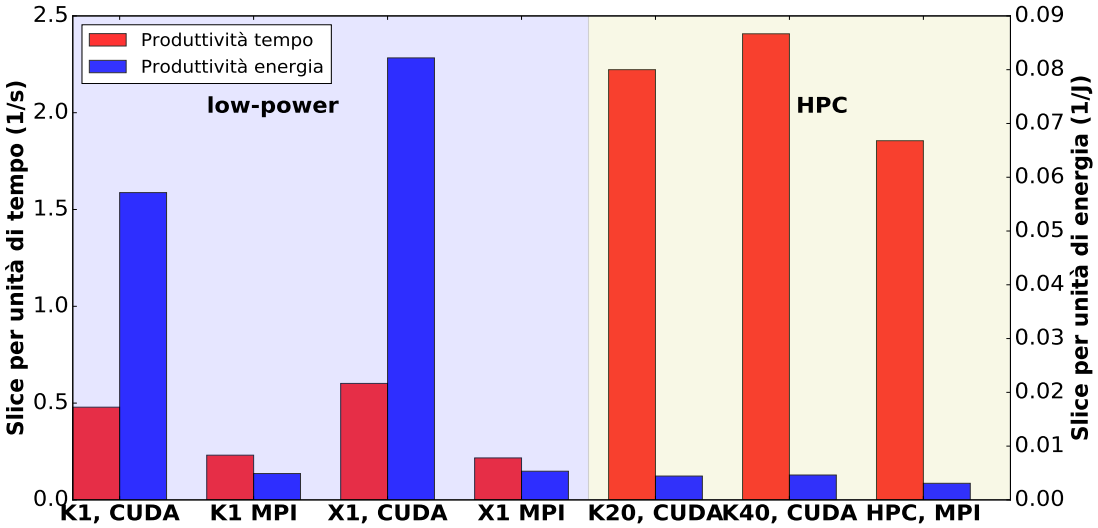

Figura 4.16: *Slice* per unità di tempo ed energia per il *dataset* 1330.

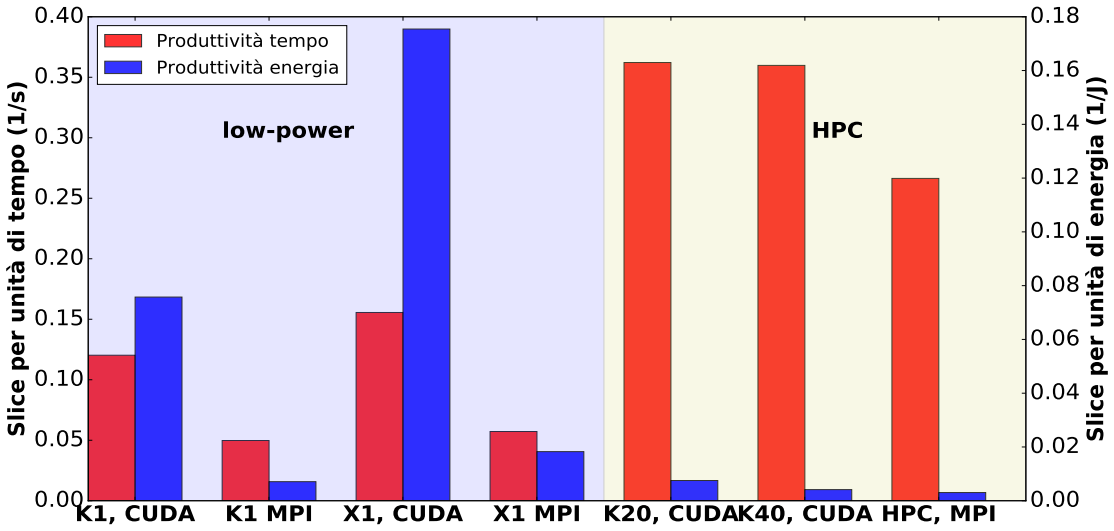

Figura 4.17: *Slice* per unità di tempo ed energia per il *dataset* 2048.

Come si osserva dai grafici, il numero di slice ricostruite per unità di tempo è molto basso per architetture SoC *low-power* e molto alto per quelle HPC, mentre per unità di energia si è nella situazione opposta.

Nei seguenti grafici si riporta il parametro di efficienza per ogni *dataset*.

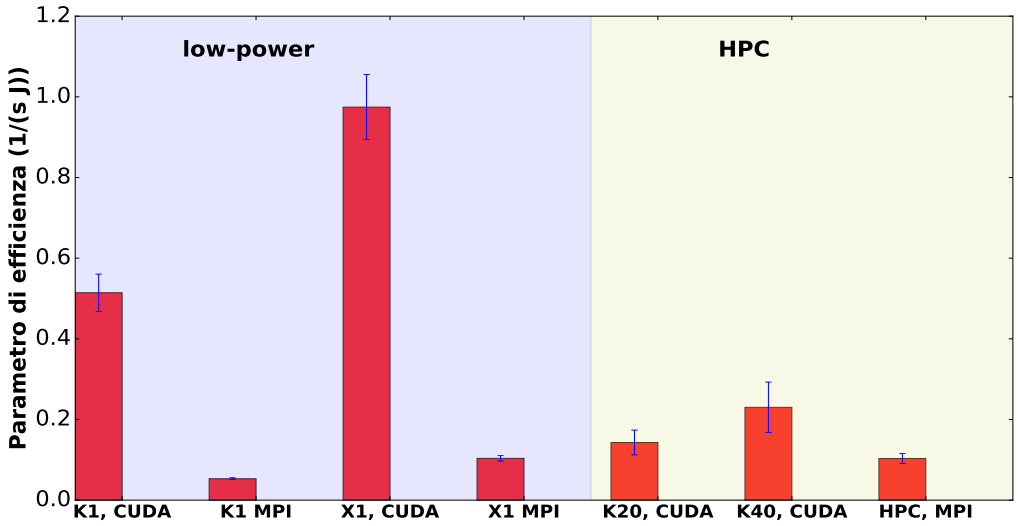

Figura 4.18: Parametro di efficienza per il *dataset* 512.

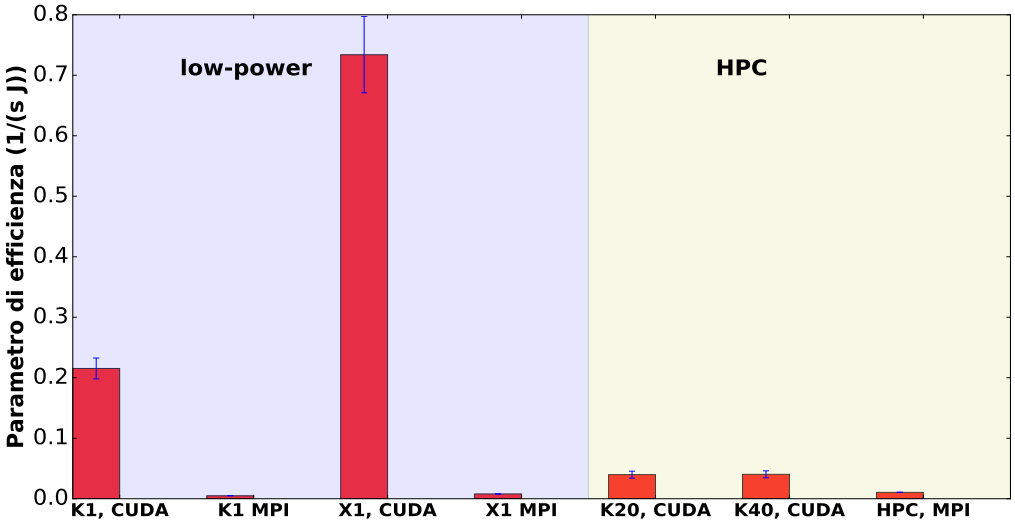

Figura 4.19: Parametro di efficienza per il *dataset* 1024.

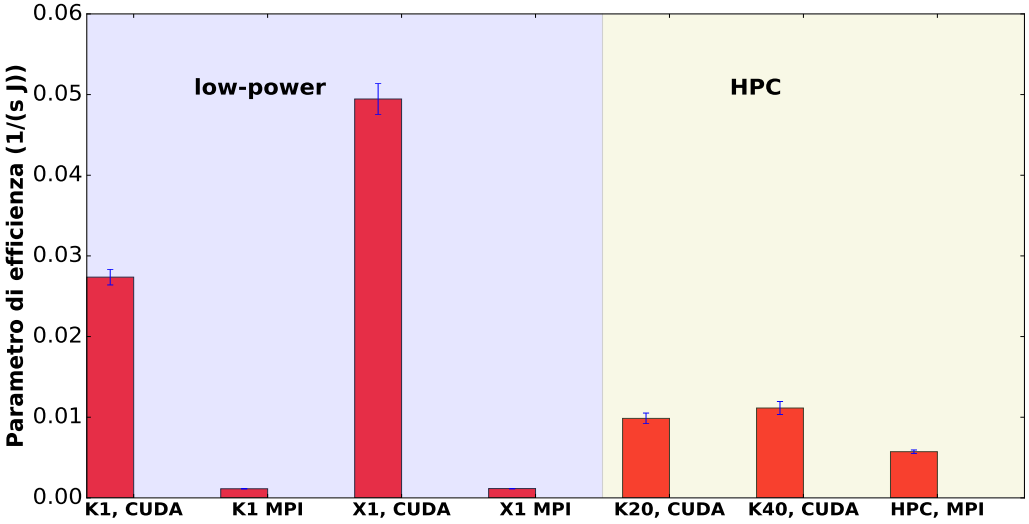

Figura 4.20: Parametro di efficienza per il *dataset* 1330.

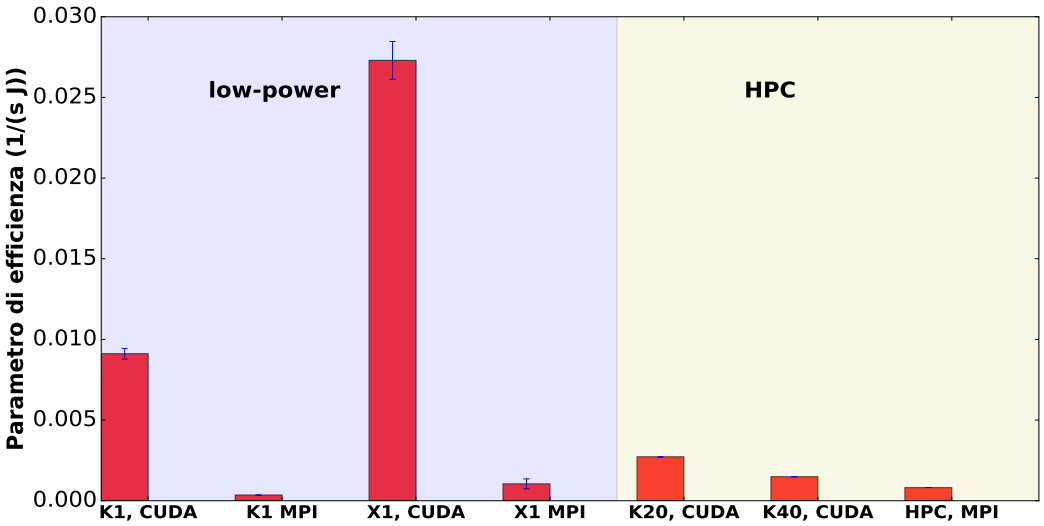

Figura 4.21: Parametro di efficienza per il *dataset* 2048.

Dai grafici appena mostrati si osserva come le architetture SoC *low-power* siano molto più efficienti che quelle HPC in termini di *slice* ricostruite per unità di tempo e di energia.

Infine si mostra l'andamento del parametro di efficienza in funzione della dimensione del *dataset*.

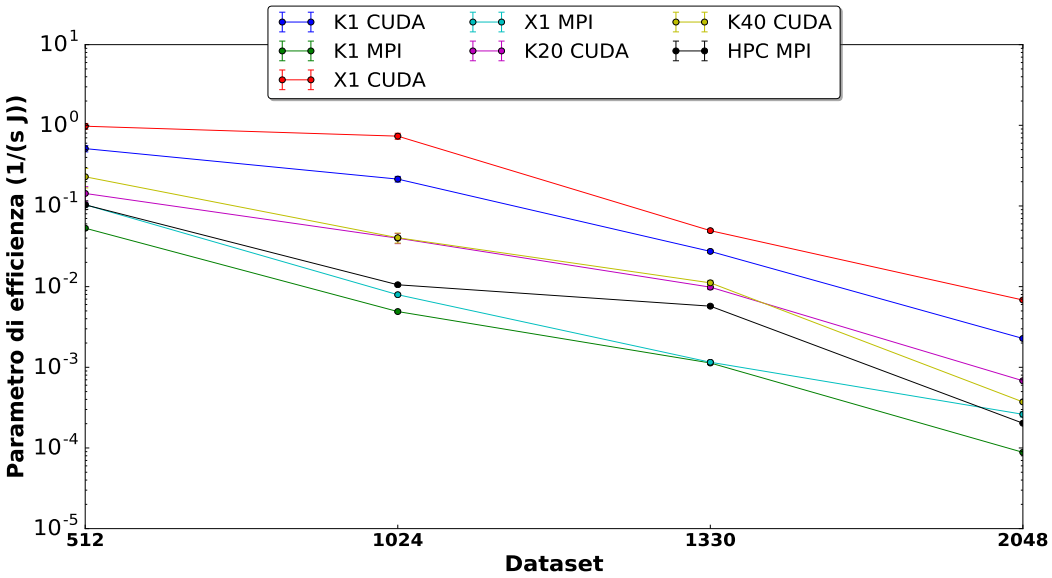

Figura 4.22: Parametro di efficienza in funzione della dimensione del *dataset*.

Come si osserva dai grafici si può dire che le architetture *low-power* sono le più efficienti, perché hanno il parametro di efficienza maggiore, in particolare la X1 risulta essere la migliore. Tuttavia si osserva che la dimensione del *dataset* è determinante sul parametro di efficienza (Figura 4.22). Questo perché all'aumentare della dimensione del *dataset* il *device* deve lavorare per molto più tempo e questo porta ad una diminuzione della produttività. Inoltre si osserva come la versione MPI sul *cluster low-power* sia estremamente poco efficiente, anche meno del *cluster* HPC. Questo perché le piattaforme TEGRA prodotte da NVIDIA sono progettate per avere la massima efficienza sia energetica che di esecuzione quando si eseguono dei codici parallelizzati in CUDA [17] e [18].

4.6 Versione ibrida MPI più CUDA

Come riportato nella sezione 3.4, parallelizzare in CUDA l'operazione di filtraggio non è conveniente, pertanto si è deciso di utilizzare la versione sequenziale. Dai test eseguiti sulla versione in MPI però si osserva che risolvere in parallelo il filtraggio porta ad un considerevole abbattimento del tempo di esecuzione. Ci si ritrova ad avere quindi un filtraggio ottimizzato nella versione in MPI, ma con una ricostruzione relativamente lenta rispetto a quella fatta con CUDA, il cui filtraggio, però, è sequenziale. Poiché l'operazione di filtraggio e quella di ricostruzione sono indipendenti fra loro, e possono pertanto essere eseguite in modo autonomo, si è pensato di unire le due parallelizzazioni in modo da considerare la tipologia di parallelizzazione più efficiente in funzione del tipo di operazione da eseguire. Nei grafici che seguono si mostra proprio questo, cioè che il filtraggio in MPI è più efficiente che nella versione in CUDA, mentre, per quanto riguarda la ricostruzione, ci si trova nella situazione opposta (CUDA più efficiente di MPI). Da qui l'idea di filtrare in MPI e di ricostruire in CUDA, ottenendo un tempo per *slice* ancora più efficiente.

Nei grafici che seguono si è abbreviato MPI con M, CUDA con C e MPI+CUDA con M+C.

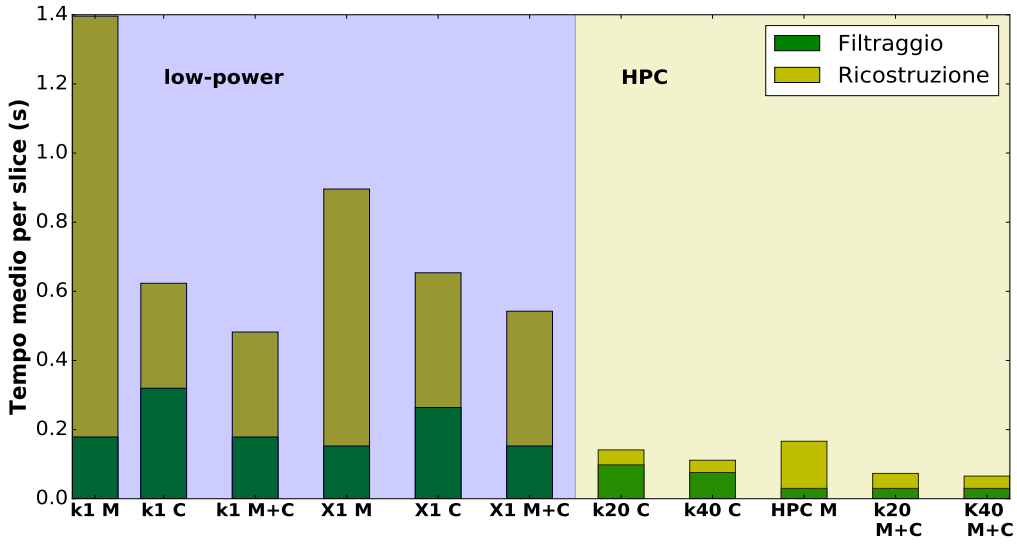

Figura 4.23: Confronto e unione dei tempi di esecuzione delle varie operazioni per il *dataset* 512.

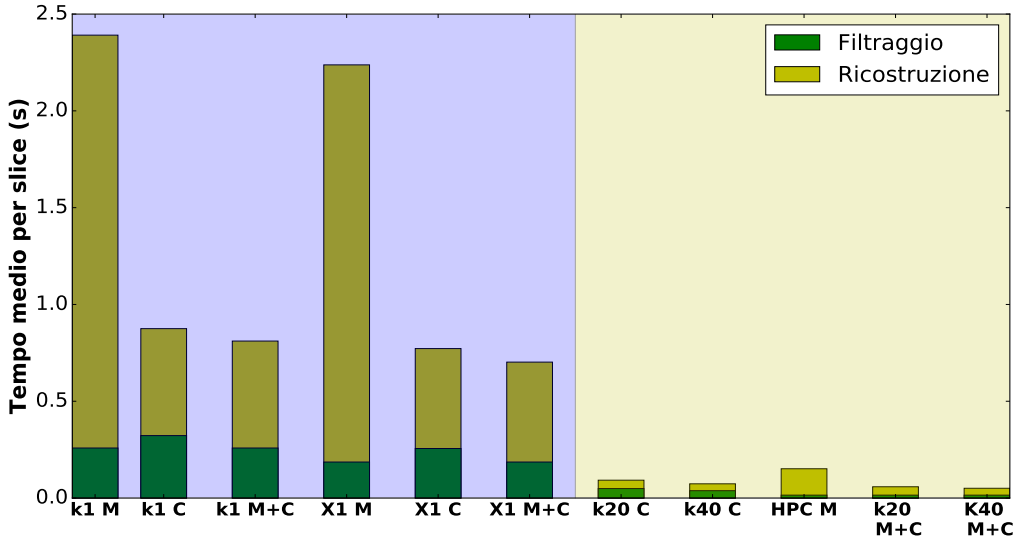

Figura 4.24: Confronto e unione dei tempi di esecuzione delle varie operazioni per il *dataset* 1024.

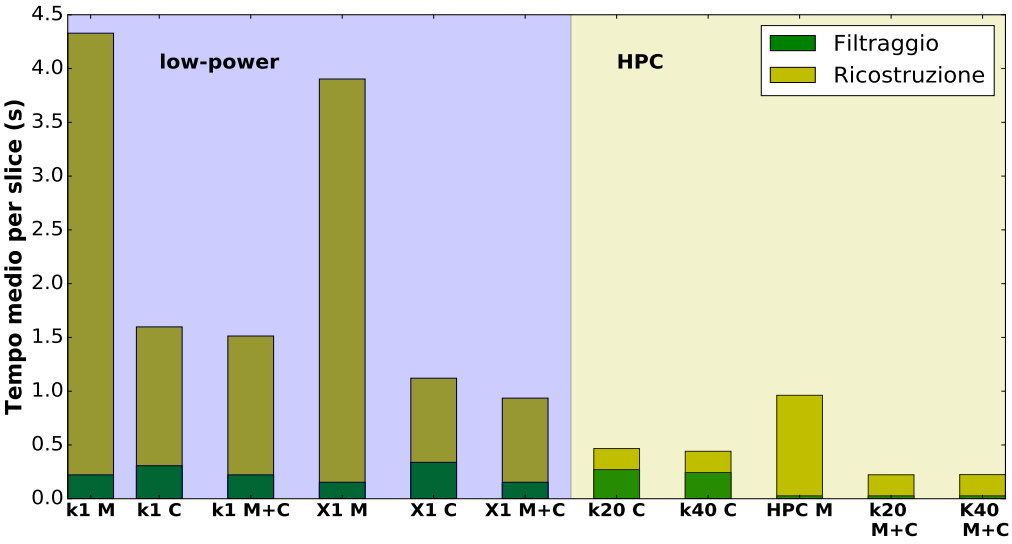

Figura 4.25: Confronto e unione dei tempi di esecuzione delle varie operazioni per il *dataset* 1330.

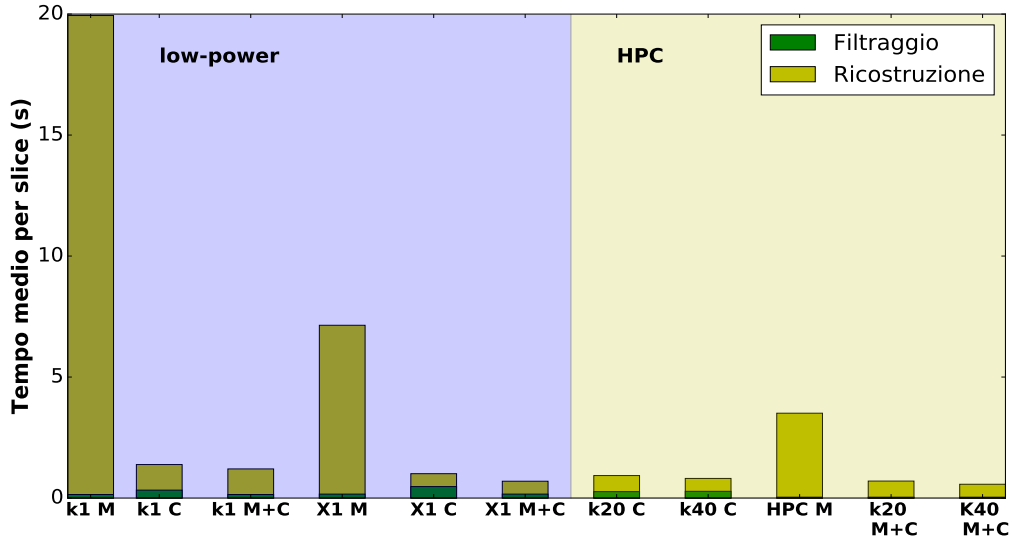

Figura 4.26: Confronto e unione dei tempi di esecuzione delle varie operazioni per il *dataset* 2048.

Come si osserva dai grafici appena mostrati il tempo di filtraggio in CUDA, che non è stato parallelizzato (paragrafo 2.6.1), risulta l'operazione

meno efficiente. In MPI invece è la ricostruzione ad essere la meno efficiente, quindi unendo il filtraggio in MPI con la ricostruzione in CUDA il tempo che si ottiene risulta essere migliore rispetto alle parallelizzazioni separate, per tutti i *dataset* testati e per tutti i *device*.

4.7 Risultati esecuzione versione MPI migliorata

Alla fine del lavoro di tesi è stato possibile eseguire dei test su una versione MPI migliorata che permette di avere un valore di *Y_depth* equivalente a quello dell'implementazione in CUDA, situazione non possibile in precedenza. Per mancanza di tempo non si è riusciti a rieseguire tutti i test, si è scelto quindi di lavorare solo sul *dataset* più significativo, il 2048 (con $Y\_depth = 256$). I risultati che sono stati ottenuti sono in linea cone quelli precedenti, confermando e sostenendo le osservazioni svolte fino ad ora. Infatti, svolgendo alcuni test, in particolare sull'HPC, dove l'esecuzione del codice parallelo implementato con MPI risulta essere più efficiente rispetto a quelli sul *cluster low-power*, il tempo totale di filtraggio che si ottiene risulta $76.91 \pm 0.01s$, mentre quello di ricostruzione $318.58 \pm 0.01s$. Si ricorda invece che in CUDA, sulla K40m, si otteneva $541.42 \pm 0.01s$, per il filtraggio sequenziale, e $136.51 \pm 0.01s$, per la ricostruzione. Questi risultati confermano e sostengono tutto ciò che più volte è stato affermato durante il corso di questa tesi, ovvero che: l'implementazione più efficiente dell'algoritmo FDK sia quella in CUDA rispetto a quella in MPI. E che inoltre, unendo i due approcci si raggiunge la massima efficienza, riuscendo a sfruttare in modo completo l'*hardware* a disposizione.

Capitolo 5

Conclusioni

Il lavoro di tesi è stato svolto svolto presso i laboratori del CNAF, in collaborazione con il gruppo *X-ray Imaging Group* del Dipartimento di Fisica e Astronomia di Bologna, che ha sviluppato le versioni sia sequenziale che parallela MPI dell'algoritmo FDK. In accordo con gli obbiettivi previsti per il lavoro di tesi, sono stati presentati i risultati ottenuti dalla parallelizzazione di questo algoritmo su GPU NVIDIA, valutando sia l'efficienza in termini di esecuzione che quella in termini di consumo energetico per architetture di tipo SoC *low-power* e HPC. Valutare oltre all'efficienza di esecuzione anche quella energetica permette di caratterizzare in modo competo un'architettura di calcolo. Infatti le architetture HPC sono pensate per il calcolo scientifico, massimizzando le prestazioni. Invece le piattaforme SoC *low-power*, pensate principalmente per *device* con batteria, tipo *smartphone* o *tablet*, massimizzano le prestazioni in funzione del consumo energetico. Valutare allora se sia più conveniente utilizzare anche per il calcolo scientifico architetture SoC *low-power* è attualmente di grande interesse nel campo scientifico, in quanto i sistemi HPC consumano quantità di energia enormi e sono molto complicati da gestire, al contrario delle architetture SoC *low-power* che consumano poca energia e hanno una complessità di gestione molto minore.

In principio sono stati eseguiti dei test della versione MPI per valutare quale numero di *slice* ricostruite in parallelo (parametro *Y_depth*) e di processi, fossero necessari per raggiungere la massima efficienza. Si è ottenuto che l'efficienza migliore si raggiunge utilizzando il numero massimo di processi disponibili e con il massimo numero permesso di *slice* ricostruite in parallelo. La dipendenza delle *performance* di esecuzione da *Y_depth* è legata all'algoritmo FDK e non al modo in cui è implementato, perché, anche per ricostruire una sola *slice* è necessario caricare tutte le radiografie in memoria, e poiché il tempo necessario per compiere questa operazione è costante, ricostruire il massimo delle *slice* possibile in parallelo porta ad una riduzione significativa del tempo medio per *slice*.

Si è quindi parallelizzato l'algoritmo FDK sulle GPU e si sono eseguiti dei test per studiarne le *performance*. È stato osservato che parallelizzare il filtraggio non porta a miglioramenti delle prestazioni, e pertanto si è deciso di parallelizzare solo la fase di ricostruzione, lasciando il filtraggio in sequenziale. I test effettuati hanno messo in evidenza che i parametri che regolano l'efficienza di esecuzione sono il numero di *thread* per blocco e il numero di *slice* ricostruite in parallelo. All'aumentare del numero di *thread* per blocco il tempo cala drasticamente. Rispetto alla versione MPI il numero di *slice* ricostruite incide in modo maggiore, in quanto le memorie della CPU e quella della GPU non sono in collegamento diretto fra loro, quindi è necessario trasferire le radiografie dalla CPU alla GPU e il volume ricostruito dalla GPU alla CPU. Nonostante questi tempi aggiuntivi, rispetto alla versione MPI, e nonostante la versione in CUDA non abbia il filtraggio parallelizzato come quella in MPI, la versione in CUDA risulta essere sempre, e su ogni architettura, più performante della versione parallelizzata in MPI, sia in termini di tempo speso che energetici, della versione parallelizzata in MPI. Inoltre la versione in CUDA ottenuta da questo lavoro di tesi è più efficiente nella gestione della memoria perché permette di ricostruire in parallelo molte più immagini che la versione MPI. Alla fine del periodo di tesi è stato possibile utilizzare una versione MPI successiva a quella utilizzata per i test effettuati durante questo lavoro di tesi, in cui il valore di *Y_depth* è lo stesso sia per CUDA che per MPI. Per motivi di tempo non è stato possibile svolgere dei test completi su quest'ultima versione; tuttavia valutando le prestazioni sul *dataset* 2048, quello in cui l'esecuzione risulta più onerosa, è ancora possibile affermare che la parallelizzazione in CUDA risulta la più efficiente, nonostante in MPI si parallelizzi sia il filtraggio che la ricostruzione, mentre in CUDA solo la ricostruzione. Inoltre si sottolinea un dato molto importante, ovvero che la versione parallelizzata in CUDA ha la GPU che lavora al massimo delle sue capacità, cioè al 100%, sia sui SoC *low-power* che sull'HPC. Questo è un ulteriore riprova della bontà della versione CUDA.

Eseguendo la differenza fra le immagini ricostruite con la versione sequenziale e quelle ricostruite con la versione in CUDA, si è osservato che le due immagini risultano equivalenti.

Successivamente alla creazione della versione CUDA e all'esecuzione di svariati test per valutarne la migliore efficienza, si sono acquisiti i consumi energetici sulle architetture utilizzate. Sulle SoC *low-power* è stato possibile misurare la corrente e la tensione di alimentazione con un multimetro, e questo ha permesso di stimare l'energia spesa come l'integrale della potenza in funzione del tempo, calcolato numericamente con il metodo dei trapezi. Sulle architetture HPC invece non è stato possibile, durante il lavoro di tesi, misurare direttamente la corrente e la tensione, e pertanto si è considerato un valore costante della potenza a 350 W. Tale stima ottenuta considerando circa 100 W per ciascuna CPU xeon (dual xeon), 150W per le K20m e K40m

e ulteriori 100 W per consumi accessori dovuti a ventole e alimentatori. I risultati ottenuti sono in accordo con quelli attesi e cioè che le architetture SoC *low-power* sono più efficienti dal punto di vista energetico che quelle HPC ma che in termini di tempo la situazione risulta invertita. Valutando un parametro di efficienza definito come il numero di *slice* ricostruite per unità di tempo e di energia, si ottiene, per tutti i *datast* di prova, che la TEGRA TX1 è la più efficiente.

Infine si propone un metodo ibrido MPI unito a CUDA per implementare l'algoritmo FDK. Infatti dai test svolti sia in MPI che in CUDA si osserva che, per quanto riguarda l'operazione di filtraggio, la versione MPI, che è parallelizzata, è molto più efficiente, come ci si aspetta, che quella in CUDA, dove si è tenuta la versione sequenziale. Poiché il filtraggio e la ricostruzione sono due operazioni indipendenti si è pensato quindi di utilizzare la parallelizzazione più efficiente per il tipo di operazione da svolgere. Da qui allora l'idea di utilizzare MPI per il filtraggio e CUDA per la ricostruzione. I risultati ottenuti con questa versione ibrida risultano migliori in termini di efficienza a quelli ottenuti dalle singole parallelizzazioni.

In conclusione di questo lavoro di tesi è quindi possibile affermare che l'algoritmo di ricostruzione tomografica FDK può essere eseguito con successo, con tempi efficienti, con l'utilizzo completo di tutte le risorse *hardware* (la GPU lavora al 100%) e con consumi ridotti su architetture SoC *low-power*, nonostante sia un problema dal punto di vista computazionale particolarmente oneroso. Questo aspetto risulta essere molto importante perché contribuisce a mostrare come sia possibile utilizzare *device* a ridotta potenza di calcolo e basso consumo energetico rispetto ai sistemi tradizionali di calcolo.

Come sviluppo futuro di questa tesi è sicuramente interessante valutare le prestazioni nel caso di utilizzo di più GPU per ricostruire un dato volume, aggiungendo un ulteriore livello di parallelizzazione. Questo potrebbe permettere di superare il limite fisico di *thread* per blocco utilizzabili su una singola GPU. Infatti si è osservato che se il *thread* per blocco è massimo l'efficienza di esecuzione del codice in CUDA migliora notevolmente. Però per *dataset* superiori al 1024 non è possibile utilizzare solo un blocco con 1024 *thread* perché non ci sarebbero sufficienti numeri di processi quanti sono i *voxel* da processare. Utilizzare più GPU invece potrebbe risolvere questo problema, associando ad ognuna di esse una porzione di volume da ricostruire pari a 1024 *voxel*, potendo così lanciare un solo blocco con 1024 *thread*, rimanendo nella situazione più efficiente. Si potrebbe anche valutare la versione MPI unita a CUDA su un *cluster* di GPU SoC *low-power*, in cui il filtraggio viene parallelizzato su CPU con MPI sui nodi del *cluster* e la ricostruzione con CUDA sempre suddivisa sui vari nodi del *cluster* SoC *low-power*.

Oltre a test svolti su GPU NVIDIA dei modelli utilizzati si potrebbero svolgere test su modelli più recenti e molto più potenti, come la NVIDIA Vol-

ta. Si potrebbero inoltre testare le prestazioni su altre marche di GPU, come quelle di AMD, utilizzando però openCL (*Open Computing Language*) come strumento di parallelizzazione dell'algoritmo. Sarebbe anche interessante eseguire test anche su macchine a *many-core* come xeonPhi.

Infine si potrebbe valutare come parallelizzare in CUDA la fase di filtraggio, modificano l'algoritmo utilizzato e riscrivendolo in una implementazione favorevole a una parallelizzazione su GPU.

Appendice A

Fan beam per detector equispaziati

In questa appendice si riposta la risoluzione del problema della ricostruzione da geometria *fan beam* per detector equispaziati, come in [1][Capitolo 3]. Si considera $R_\beta(s)$ come la proiezione acquisita all'angolo β nel punto s come mostrato in figura A.1. L'algoritmo FBP in questo caso consiste nel filtrare la proiezione e retroproiettarla successivamente. È conveniente eseguire i calcoli rispetto al piano immaginario $D_1^{'}D_2^{'}$ in cui i fasci sono nella condizione di geometria a fasci paralleli (A.1). Eseguire questa trasformazione applicando gli opportuni pesi permette di applicare il teorema della *slice* di Fourier e quindi l'algoritmo FBP. Si utilizza quindi un nuovo sistema di riferimento descritto dalla retta t, ottenuta in modo che il fascio considerato vi sia perpendicolare, come mostrato in figura A.2. Per passare dal sistema s del *detector* a quello t si utilizzano le seguenti relazioni:

$$\begin{aligned} t &= s\cos\gamma, & \theta &= \beta + \gamma \\ t &= \frac{sD}{\sqrt{D^2+s^2}}, & \theta &= \beta + \arctan\left(\frac{s}{D}\right) \end{aligned}$$

Per l'algoritmo FBP si sa che il valore della funzione $f(x,y)$, che in questo caso in coordinate polari $r,\phi)$ per comodità, vale come:

$$f(r,\phi) = \int_0^{2\pi} \left[\int_{-t_m}^{t_m} P_\theta(t) h(r\cos(\theta-\phi)-t)dt \right] d\theta \qquad \text{(A.1)}$$

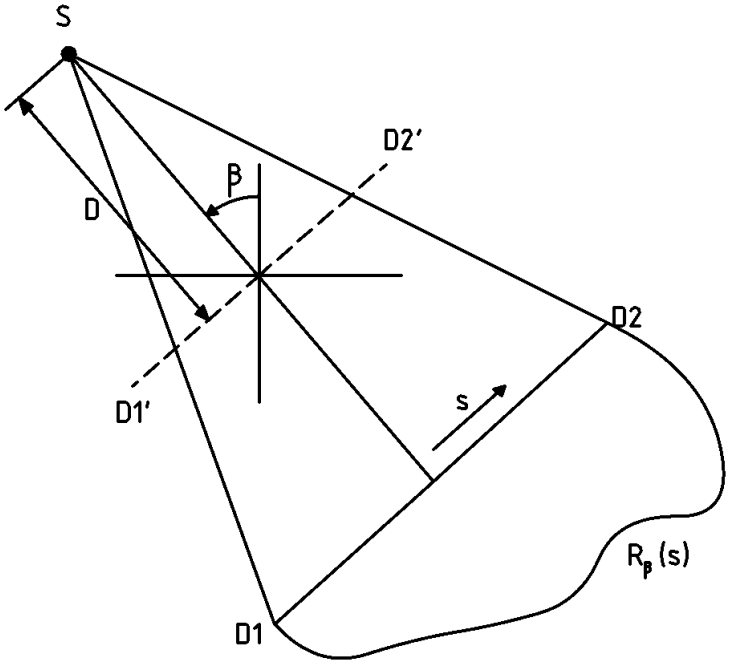

Figura A.1: Geometria fan beam.

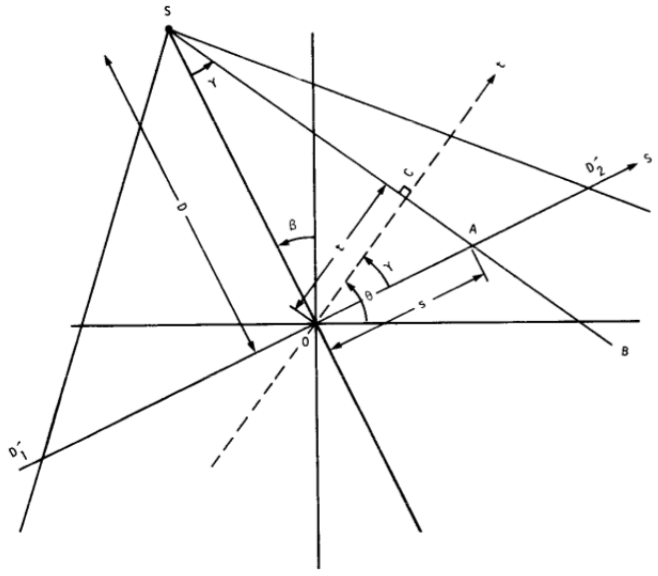

Figura A.2: Nuovo sistema di riferimento, [1].

Dove t_m è il più alto valore di t. Utilizzando le relazioni precedenti su t, γ, β è possibile riscrivere il precedente integrale come:

$$f(r,\phi) = \frac{1}{2}\int_{-\arctan\left(\frac{s_m}{D}\right)}^{\arctan\left(\frac{s_m}{D}\right)} \left[\int_{-s_m}^{s_m} P_{\beta+\gamma}\left(\frac{sD}{\sqrt{D^2+s^2}}\right) h\left[r\cos\left(\beta - \arctan\left(\frac{s}{D}\right) - \phi\right) - \frac{sD}{\sqrt{D^2+s^2}}\right] \frac{D^3}{(D^2+s^2)^{\frac{3}{2}}} ds \right] d\beta \tag{A.2}$$

Dove s_m sarà il valore massimo lungo la retta s e $\frac{D^3}{(D^2+s^2)^{\frac{3}{2}}} ds\ d\beta = dt\ d\theta$. Sulla retta SA si ha che:

$$P_{\beta+\gamma}\left(\frac{sD}{\sqrt{D^2+s^2}}\right) = P_\theta(t) = R_\beta(s) \tag{A.3}$$

Dove $R_\beta(s)$ è il valore della proiezione dell'angolo beta nel punto s. Quindi la ricostruzione vale:

$$f(r,\phi) = \frac{1}{2}\int_0^{2\pi} \left[\int_{-s_m}^{s_m} R_\beta(s) h\left[r\cos\left(\beta - \arctan\left(\frac{s}{D}\right) - \phi\right) - \frac{sD}{\sqrt{D^2+s^2}}\right] \frac{D^3}{(D^2+s^2)^{\frac{3}{2}}} ds \right] d\beta \tag{A.4}$$

Riscrivendo ora l'argomento di h come:

$$r\cos\left(\beta + \arctan\left(\frac{s}{D}\right) - \phi\right) - \frac{sD}{\sqrt{D^2+s^2}} = r\cos(\beta-\phi)\frac{D}{\sqrt{D^2+s^2}} - (D + r\sin(\beta-\phi))\frac{s}{\sqrt{D^2+s^2}} \tag{A.5}$$

Si introducono due nuove variabili U e $\frac{s'}{\overline{SO}}$, dove la prima corrisponde al rapporto fra la distanza della proiezione di E sulla retta passante per l'origine e la distanza del rivelatore da quest'ultimo, mentre il secondo ha per valore di s quello che ha il raggio passante per il pixel (r,ϕ), chiamato con s' (A.3).

$$U(r,\phi,\beta) = \frac{\overline{SO} + \overline{OP}}{D} = \frac{D + r\sin(\beta-\phi)}{D} \tag{A.6}$$

$$\frac{s'}{\overline{SO}} = \frac{\overline{EP}}{\overline{SP}} \tag{A.7}$$

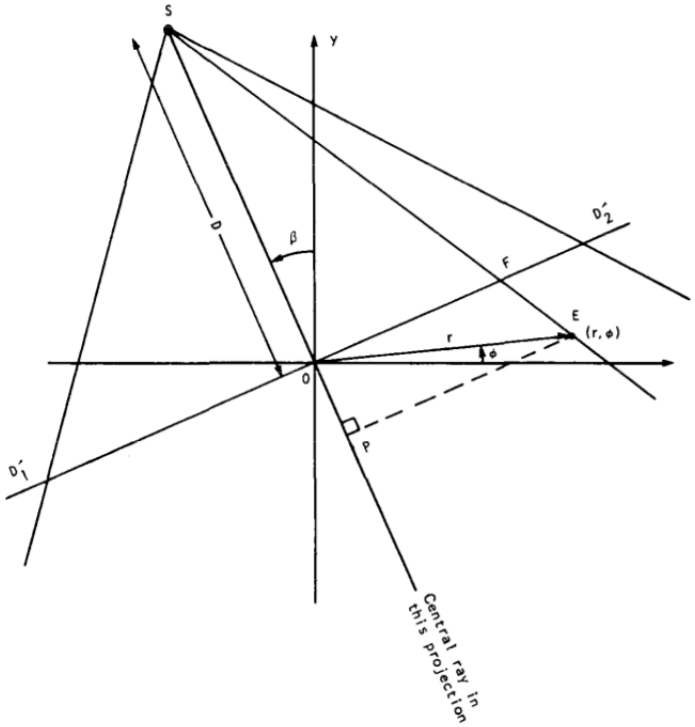

Figura A.3: Nuove variabili, [1]

Ricavando s' dalla precedente relazione si ottiene:

$$s' = D\frac{r\cos(\beta-\phi)}{D + r\sin(\beta-\phi)} \tag{A.8}$$

Quindi riscrivendo il termine di h in funzione di U e s' si ottiene:

$$r\cos\left(\beta + \arctan\left(\frac{s}{D}\right) - \phi\right) - \frac{Ds}{\sqrt{D^2+s^2}} = \frac{s'UD}{\sqrt{D^2+s^2}} - \frac{sUD}{\sqrt{D^2+s^2}} \tag{A.9}$$

L'integrale A.1 nelle nuove coordinate diventa:

$$f(r,\phi) = \frac{1}{2}\int_0^{2\pi}\left[\int_{-s_m}^{s_m} R_\beta(s)h\left[(s'-s)\frac{UD}{\sqrt{D^2+s^2}}\right]\frac{D^3}{(D^3+s^2)^{\frac{3}{2}}}ds\right]d\beta \tag{A.10}$$

Riscrivendo nei nuovi termini delle variabili U ed s' si ottengono dei fattori moltiplicativi detti pesi, propri della particolare geometria dell'apparato di acquisizione, in questo caso per fasci equispaziati. Riscrivendo il termine $h(t)$ in funzione di s si ottiene:

$$h\left[(s'-s)\frac{UD}{\sqrt{D^2+s^2}}\right] = \frac{D^2+s^2}{U^2D^2}\int_{-\infty}^{+\infty} \mid w' \mid e^{2\pi i(s'-s)w'}dw' = \frac{D^2+s^2}{\sqrt{D^2+s^2}}h(s'-s) \tag{A.11}$$

Dove w' vale $w\frac{UD}{\sqrt{D^2+s^2}}$. Quindi l'integrale A.1 in funzione di s vale:

$$f(r,\phi) = \int_0^{2\pi}\frac{1}{U^2}\left[\int_{-\infty}^{+\infty} R_\beta(s)g(s'-s)\frac{D}{\sqrt{D^2+s^s}}ds\right]d\beta \tag{A.12}$$

Dove $g(s'-s) = \frac{1}{2}\ h(s)$. Indicando con $R'_\beta(s)$ la proiezione pesata, equivalente a $R_\beta(s)\frac{D}{\sqrt{D^2+s^2}}$, la proiezione pesata e filtrata risulta uguale a:

$$Q_\beta(s') = \int_{-\infty}^{+\infty} R_\beta(s)'g(s'-s)ds \tag{A.13}$$

Integrando su tutti gli angoli la proiezione pesata e filtrata e pesandola ancora per il fattore U si ottiene l'integrale definitivo di ricostruzione come:

$$f(r,\phi) = \int_0^{2\pi}\frac{1}{U^2}Q_\beta(s')d\beta \tag{A.14}$$

Dalla formula precedente si ottiene il valore della funzione nel punto del pino rappresentato da r,ϕ) per una geometria del detector a raggi equispaziati. Da questa forma si parte per ricavare quella per l'algoritmo FDK poiché i pesi utilizzati sono gli stessi nel caso di un *detector* equispaziato.

Appendice B

Specifiche tecniche delle macchine utilizzate

B.1 Cluster low-power

B.1.1 NVIDIA Tegra K1 SoC

- NVIDIA Kepler GPU with 192 CUDA cores
- NVIDIA 4-Plus-1 quad-core ARM Cortex A15 CPU
- 2 GB x16 memory with 64 bit width
- 16 GB 4.51 eMMC memory
- 1 Half mini-PCIE slot 1
- Full size SD/MMC connector
- 1 USB 2.0 port, micro AB 1
- USB 3.0 port, A
- 1 Full-size HDMI port
- 1 RS232 serial port
- 1 ALC5639 Realtek Audio codec with Mic in and Line out
- 1 RTL8111GS Realtek GigE LAN
- 1 SATA data port
- SPI 4MByte boot flash

B.1.2 NVIDIA Tegra X1 SoC

- GPU 1024 GFLOPS 256-core Maxwell
- CPU 4x 64-bit ARM A57 CPUs — 1.6 GHz
- Memory 4 GB LPDDR4 — 25.6 GB/s
- Video decode 4K 60Hz H.264 / H.265
- Video encode 4K 30Hz H.264 / H.265
- CSI Up to 6 cameras — 1400 Mpix/s
- Display 2x DSI, 1x eDP 1.4, 1x DP 1.2/HDMI
- Wi-Fi 802.11 2x2 ac
- Networking 1 Gigabit Ethernet
- PCI-E Gen 2 1x1 + 1x4
- Storage 16 GB eMMC, SDIO, SATA
- Other 3x UART, 3x SPI, 4x I2C, 4x I2S, GPIOs
- Power 10-15W, 6.6V-19.5VDC
- Size 50mm x 87mm

B.2 Cluster HPC

B.2.1 NVIDIA Tesla K20m

- CUDA PARALLEL PROCESSING CORES 2496
- PEAK DP IN TFLOPS ¿1 TFlop
- INTERFACE 320-bit
- PROCESSOR CORE CLOCK 706 MHz
- FRAME BUFFER MEMORY 5 GB GDDR5
- MEMORY BANDWIDTH 208 GB/s
- DISPLAY CONNECTORS None
- MAX POWER CONSUMPTION 225 W
- POWER CONNECTORS (2) x 6-pin PCI Express power connectors
- GRAPHICS BUS PCI Express 2.0 x16
- FORM FACTOR 110 mm (H) x 265 mm (L)
- Dual Slot
- THERMAL SOLUTION Active

B.2.2 NVIDIA Tesla K40m

- CUDA PARALLEL PROCESSING CORES 2880
- FRAME BUFFER MEMORY 12 GB GDDR5
- PEAK DOUBLE PRECISION FLOATING POINT PERFORMANCE 1.43 Tflops
- PEAK SINGLE PRECISION FLOATING POINT PERFORMANCE 4.29 Tflops
- INTERFACE 384-bit
- MEMORY BANDWIDTH 288 GB/s
- DISPLAY CONNECTORS None
- MAX POWER CONSUMPTION 235 W
- PROCESSOR CORE CLOCK 745 MHz
- POWER CONNECTORS (2) x 6-pin PCI Express power connectors
- GRAPHICS BUS PCI Express 2.0 x16
- FORM FACTOR 110 mm (H) x 265 mm (L)
- Dual Slot
- THERMAL SOLUTION Active

B.2.3 Xeon

- Number of Cores 8
- Number of Threads 16
- Processor Base Frequency 2.00 GHz
- Max Turbo Frequency 2.50 GHz
- Cache 20 MB SmartCache
- Bus Speed 7.2 GT/s QPI
- Number of QPI Links 2
- TDP 95 W
- VID Voltage Range 0.65–1.30V
- RAM 128 GB

Appendice C

Calcolo dell'errore dell'energia dissipata

Data la funzione che lega la potenza P al tempo è possibile calcolare l'energia ad un dato tempo t_x come segue:

$$E(t_x) = \int_0^{t_x} P(t)dt \tag{C.1}$$

Poiché l'integrale è stato calcolato numericamente con il metodo dei trapezi allora vale:

$$E_{t_x} = \int_0^{t_x} P(t)dt \approx \frac{\Delta t}{2} \sum_{i=1}^{N_{t_x}} P(t_{i+1}) + P(t_i) \tag{C.2}$$

Dove Δt è il passo di integrazione, e dipende dal numero di campionamenti al secondo impostato sul multimetro e N il numero di intervalli. Poiché quello che si misura è la corrente funzione del tempo è necessario scrivere l'energia in funzione della corrente con:

$$P(t) = I(t)V \tag{C.3}$$

Quindi l'energia diventa:

$$E_{t_x} \approx \frac{\Delta t}{2} V \sum_{i=1}^{N_{t_x}} I(t_{i+1}) + I(t_i) \tag{C.4}$$

Le incertezze sulla corrente, sulla tensione e sul tempo sono note. Definendo I'_{t_x} come:

$$I'_{t_x} = \sum_{i=1}^{N_{t_x}} I(t_{i+1}) + I(t_i) \tag{C.5}$$

L'energia diventa:

$$E_{t_x} \approx \frac{\Delta t}{2} V I'_{t_x} \tag{C.6}$$

Riscrivendo nei termini della potenza si ha:

$$E_{t_x} \approx \frac{\Delta t}{2} P_{t_x} \tag{C.7}$$

Dove,

$$P_{t_x} = V I'_{t_x} \tag{C.8}$$

Quindi l'errore sull'energia sarà:

$$\delta E_{t_x} = E_{t_x} \sqrt{\left(\frac{\delta \Delta t}{\Delta t}\right)^2 + \left(\frac{\delta P_{t_x}}{P_{t_x}}\right)^2} \tag{C.9}$$

Di conseguenza l'errore sulla potenza risulta:

$$\delta P_{t_x} = P_{t_x} \sqrt{\left(\frac{\delta V}{V}\right)^2 + \left(\frac{\delta I'_{t_x}}{I'_{t_x}}\right)^2} \tag{C.10}$$

Con l'errore di I'_{t_x} pari a:

$$\delta I'_{t_x} = \sum_{i=1}^{N_{t_x}} \delta I(t_{i+1}) + \delta I(t_i) \tag{C.11}$$

Bibliografia

[1] A. C. Kak, M. Slaney, I. E. in Medicine, and B. Society, "Principles of computerized tomographic imaging," 1988. Published under the sponsorship of the IEEE Engineering in Medicine and Biology Society.

[2] J. A. Brink, J. P. Heiken, G. Wang, K. W. McEnery, F. J. Schlueter, and M. Vannier, "Helical ct: principles and technical considerations.," *Radiographics*, vol. 14, no. 4, pp. 887–893, 1994.

[3] J. H. Hubbell and S. M. Seltzer, "Tables of x-ray mass attenuation coefficients and mass energy-absorption coefficients 1 kev to 20 mev for elements z= 1 to 92 and 48 additional substances of dosimetric interest," tech. rep., National Inst. of Standards and Technology-PL, Gaithersburg, MD (United States). Ionizing Radiation Div., 1995.

[4] L. Feldkamp, L. Davis, and J. Kress, "Practical cone-beam algorithm," *JOSA A*, vol. 1, no. 6, pp. 612–619, 1984.

[5] J. M. Rabaey, A. P. Chandrakasan, and B. Nikolic, *Digital integrated circuits*, vol. 2. Prentice hall Englewood Cliffs, 2002.

[6] T. G. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*. Pearson Education, 2004.

[7] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[8] J. W. McCormick, F. Singhoff, and J. Hugues, *Building parallel, embedded, and real-time applications with Ada*. Cambridge University Press, 2011.

[9] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, ACM, 1967.

[10] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, pp. 532–533, 1988.

[11] P. Messmer, "Gpu architecture overview and cuda basics," *NVIDIA Corp., Zurich, Switzerland*, 2013.

[12] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional, 2010.

[13] R. Brancaccio, M. Bettuzzi, F. Casali, M. Morigi, G. Levi, A. Gallo, G. Marchetti, and D. Schneberk, "Real-time reconstruction for 3-d ct applied to large objects of cultural heritage," *IEEE Transactions on Nuclear Science*, vol. 58, no. 4, pp. 1864–1871, 2011.

[14] Y. Okitsu, F. Ino, and K. Hagihara, "High-performance cone beam reconstruction using cuda compatible gpus," *Parallel Computing*, vol. 36, no. 2, pp. 129–141, 2010.

[15] G. Yan, J. Tian, S. Zhu, Y. Dai, and C. Qin, "Fast cone-beam ct image reconstruction using gpu hardware," *Journal of X-ray Science and Technology*, vol. 16, no. 4, pp. 225–234, 2008.

[16] E. Corni, "Implementazione dell'algoritmo filtered back-projection (fbp) per architetture low-power di tipo systems-on-chip."

[17] L. Morganti, D. Cesini, and A. Ferraro, "Evaluating systems on chip through hpc bioinformatic and astrophysic applications," in *Parallel, Distributed, and Network-Based Processing (PDP), 2016 24th Euromicro International Conference on*, pp. 541–544, IEEE, 2016.

[18] L. Morganti, E. Corni, A. Ferraro, D. Cesini, D. D'Agostino, and I. Merelli, "Implementing a space-aware stochastic simulator on low-power architectures: A systems biology case study," in *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on*, pp. 303–308, IEEE, 2017.

Ringraziamenti

Vorrei ringraziare la Prof.ssa Maria Pia Morigi e la Dott.ssa Rosa Brancaccio per avermi dato la possibilità di svolgere la tesi nell'abito del calcolo parallelo, fornendomi tutti gli strumenti e il supporto necessario per svolgere al meglio questo lavoro.

Desidero ringraziare anche il CNAF per avermi dato la possibilità di utilizzare i loro laboratori per svolgere tutte le prove eseguite durante questi mesi. In particolare ringrazio la Dott.ssa Elena Corni, che per tutta la durata di questo lungo periodo è sempre stata molto presente e disponibile nell'aiutarmi. Ringrazio inoltre il Dott. Daniele Cesini, la Dott.ssa Lucia Morganti e il Dott. Andrea Ferraro per avermi fornito preziosissimi consigli durante tutto il lavoro di tesi.

Infine ringrazio la mia famiglia, per avermi fornito tutto l'aiuto necessario ad affrontare gli studi universitari.