Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea Magistrale in Informatica

Analisi di immagini con trasformata Ranklet: ottimizzazioni computazionali su CPU e GPU

Tesi di Laurea in Analisi di Immagini

Relatore:
Chiar.mo Prof. Renato Campanini

Presentata da:
Pietro Ansaloni

Co-relatore:
Dott. Matteo Roffilli

Sessione III
Anno Accademico 2011/2012

Introduzione

L'analisi di immagini è il settore dell'informatica che si occupa di estrapolare ed interpretare informazioni significative dall'elaborazione di immagini digitali.
Tecniche di questo tipo hanno una gamma di applicazioni praticamente illimitata, poiché possono essere impiegate in qualunque ambito, scientifico e non, che coinvolga l'utilizzo di immagini o video. Alcune tra le applicazioni più interessanti di questi principi sono il riconoscimento facciale, la classificazione di impronte digitali, l'analisi di immagini biomediche, il tracciamento di oggetti in movimento all'interno di sequenze video e l'elaborazione di immagini satellitari.
Questo lavoro di tesi è stato realizzato in collaborazione con un gruppo di lavoro specializzato in questo settore che ha sviluppato, tra le altre cose, un sistema CAD (Computer Aided Detection) capace di individuare masse tumorali all'interno di immagini cliniche generate durante esami mammografici.
Il processo di elaborazione del lavoro presentato di seguito, è stato portato avanti con un approccio di tipo industriale, mirato all'inserimento in un flusso operativo di ricerca e alla realizzazione di un'attività di interesse pratico.
Come principale argomento di studio è stata scelta la trasformata ranklet, una particolare tecnica di analisi di immagini che gode di proprietà molto utili nel rilevamento di pattern geometrici. L'utilizzo di questa tecnica è però limitato dalle sue proprietà computazionali: la sua elaborazione richiede infatti l'esecuzione di un alto numero di operazioni di sorting, il che rende difficile realizzarne implementazioni efficienti.

Il lavoro di seguito descritto è stato sviluppato a partire da una implementazione non ottimizzata dell'algoritmo classico per il calcolo delle ranklet; questa soluzione software, seppur corretta e funzionante, non è considerata sufficientemente efficiente per l'impiego in problemi reali, caratterizzati dalla necessità di elaborare dati di grandi dimensioni.

Per questo motivo, lo scopo del lavoro di tesi è stato definito in termini di miglioramento computazionale, ponendo l'obiettivo di realizzare una implementazione che consentisse il calcolo della trasformata ranklet in maniera più efficiente. Si è quindi deciso di ottimizzare il codice per l'esecuzione in ambienti dotati di più unità di elaborazione e di sfruttare il parallelismo delle moderne piattaforme di calcolo per realizzare codice più veloce e performante.

Dopo aver considerato diverse piattaforme di esecuzione per le implementazioni svluppate, si è deciso di realizzare due versioni diverse dell'algoritmo: una per l'ambiente multicore e una per l'elaborazione con processori grafici (GPU).

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Nozioni preliminari

In questo capitolo verranno introdotti alcuni argomenti utili per comprendere il contesto nel quale è stato sviluppato l'intero lavoro di tesi.
Per prima cosa saranno trattati alcuni concetti legati alla Calcolabilità, approccio che sarà utile per mostrare l'esistenza di problemi computazionalmente difficili, valutando la situazione da un punto di vista teorico.
In seguito sarà trattato l'aspetto pratico, evidenziando come la produzione di hardware sempre più performante allarghi il range di problemi trattabili, a condizione che si riescano a sfruttare le potenzialità offerte dalle nuove tecnologie.
Infine si effettuerà una panoramica su acceleratori e coprocessori di ultima generazione.

1.1 Complessità Computazionale

La Teoria della Calcolabilità è la branca dell'Informatica Teorica che si occupa di stabilire quali problemi possano essere risolti e quali funzioni siano calcolabili con un procedimento automatico.
In questo contesto, un problema è un oggetto matematico che rappresenta un insieme di domande alle quali è possibile trovare risposta con l'esecuzione

di un algoritmo da parte di un calcolatore.
L'obiettivo primario della calcolabilità è la descrizione formale e matematicamente rigorosa dei problemi trattati e delle funzioni che servono per risolverli. Questo procedimento consente di garantire la correttezza della soluzione, dimostrarne la validità e studiare il comportamento e le proprietà degli algoritmi che la implementano.
Nella valutazione di un algoritmo, è di fondamentale importanza studiarne la computabilità, cioè analizzare la quantità di risorse utilizzate durante l'esecuzione. Lo scopo di questa analisi è stabilire se l'algoritmo può essere portato a termine correttamente e in un tempo ragionevole, o se il problema è troppo complesso e costoso per essere risolto in maniera efficiente.
Per svolgere questo genere di valutazioni in maniera corretta è necessario stabilire una serie di regole che consentano una misurazione imparziale e affidabile. Per questo motivo generalmente si fa riferimento a modelli di calcolo teorici, come ad esempio la macchina di Turing: un modello di calcolo astratto che oggi è universamente riconosciuto come la prima rappresentazione formale di un calcolatore.
Informalmente, la macchina di Turing (**MdT**) può essere descritta come un dispositivo capace di processare un insieme finito di simboli (alfabeto), dotato di uno stato interno che può essere cambiato ad ogni istante in base ad una serie di regole prestabilite, e in grado di effettuare operazioni di input/output su un nastro di lunghezza infinita. Ad ogni passo la macchina è quindi in grado di valutare il proprio stato interno e determinare quale sarà quello successivo.
Una prova della validità della MdT come strumento di calcolo teorico è la tesi di Church-Turing, che afferma che la classe delle funzioni calcolabili equivale a quella delle funzioni che è possibile risolvere con una MdT.

Nell'analisi computazionale di un problema, per misurare l'efficienza di un algoritmo, si prendono in considerazione principalmente due risorse: il tempo e lo spazio.

Per quanto riguarda la misurazione del tempo d'esecuzione, una macchina di Turing $\mathcal{M}$ opera in tempo $f(n)$ se, dato un input di lunghezza n, $\mathcal{M}$ produce il risultato in $f(n)$ passi.
Per quel che riguarda la risorsa spazio, data una MdT $\mathcal{M}$ e un input di dimensione n, la macchina opera in spazio $f(n)$ se utilizza $f(n)$ celle del nastro per effettuare la computazione, escludendo le dimensioni dell'input e dell'output.

I problemi possono quindi essere raggruppati in diverse classi di complessità, in base alle risorse di calcolo impiegate da una MdT nell'esecuzione del migliore algoritmo noto per la soluzione del problema.
Ad esempio, la classe **P** (o **PTIME**) è una delle classi di complessità fondamentali: contiene tutti i problemi decisionali che possono essere risolti in tempo polinomiale, con un algoritmo deterministico.
La classe **NEXP** (o **NEXPTIME**), invece, contiene i problemi decisionali che richiedono un tempo esponenziale ($O(2^{p(n)}$, con $p(n)$ funzione polinomiale) per essere risolti da un algoritmo deterministico.
Le due classi appena indicate rappresentano solo una parte del panorama di classi di complessità, ma sono un esempio di come problemi con caratteristiche simili vengono raggruppati in base alle loro proprietà computazionali.
All'interno della teoria della complessità è quindi possibile effettuare una distinzione tra i cosiddetti problemi facili, per i quali si conoscono algoritmi di risoluzione efficienti, e difficili, per i quali sono noti solamente algoritmi non efficienti.

Lo studio della complessità di un problema ricopre un ruolo di importanza fondamentale nella progettazione di un algoritmo, ed è una delle prime attività da svolgere se si vuole sviluppare un algoritmo efficiente.

1.2 Progressi Tecnologici

Nel 1965 Gordon Moore, cofondatore di Intel, quantificò la straordinaria crescita della tecnologia dei semiconduttori in una altrettanto straordinaria analisi. Come risultato di un approfondito studio sulla struttura dei circuiti integrati tra il 1956 e il 1965, Moore osservò che la densità dei transistor all'interno dei chip raddoppiava ad intervalli regolari, teorizzando che la crescita sarebbe continuata per lungo tempo.
Questa osservazione è nota oggi con il nome di Legge di Moore e si è dimostrata sostanzialmente valida per oltre 40 anni. Fin dalla sua formulazione è stata vista come un metodo affidabile per stimare la misura delle innovazione tecnologiche nell'ambito dei circuiti integrati, definendo le regole e la natura stessa della concorrenza nel mercato dei semiconduttori.

I fattori che hanno reso possibile l'attuazione e il mantenimento della legge di Moore sono la scoperta dei semiconduttori, l'invenzione dei transistor e dei circuiti integrati. Fin dagli anni '30 iniziarono gli studi su come sfruttare al meglio le particolari caratteristiche fisiche dei semiconduttori, ma solo nel 1947 fu realizzato il primo prototipo funzionante di transistor, ad opera di un gruppo di ricerca dei laboratori Bell Labs. I primi transistor erano lunghi circa 1 cm, ma fin da subito la ricerca indirizzò i suoi studi nella progettazione di dispositivi più piccoli.
Il secondo grande contributo al mondo dell'elettronica arrivò nel 1958, con l'invenzione del circuito integrato, grazie al quale era possibile realizzare un chip composto da diversi transistor. Una proprietà che fin da subito contraddistinse i circuiti integrati riguardava il costo di fabbricazione, che era pressoché invariato con l'aumentare della complessità di questi dispositivi. Questa caratteristica favorì ulteriormente lo sviluppo di questa tecnologia.
Si iniziarono a produrre circuiti sempre più piccoli, consentendo quindi l'integrazione di un numero sempre crescente di componenti elettronici su ogni dispositivo.

La figura 1.1 mostra l'incremento del numero di transistor nei microprocessori prodotti da Intel.

Microprocessore	**Anno**	**Transistor**
4004	1971	2 300
8008	1972	2 500
8080	1974	4 500
8086	1978	29 000
Intel286	1982	134 000
Intel386	1985	275 000
Intel486	1989	1 200 000
Intel Pentium	1993	3 100 000
Intel Pentium II	1997	7 500 000
Intel Pentium III	1999	9 500 000
Intel Pentium 4	2000	42 000 000
Intel Itanium	2001	25 000 000
Intel Itanium 2	2003	220 000 000
Intel Itanium 2 (9MB cache)	2004	592 000 000
Intel Core i7 (Quad)	2008	731 000 000
Intel Six-Core Core i7 (Gulftown)	2010	1 170 000 000
Intel 8-Core Xeon E5	2010	2 270 000 000
Intel 10-Core Xeon Westmere-EX	2011	2 600 000 000
Intel 8-Core Itanium Poulson	2012	3 100 000 000
Intel 62-Core Xeon Phi	2012	5 000 000 000

Tabella 1.1: Numero di transistor nei processori Intel.

I progressi nella miniaturizzazione dei transistor e i miglioramenti nella produzione di circuiti integrati sempre più complessi hanno permesso la realizzazione di processori sempre più veloci e potenti.
Se per decenni la legge di Moore è stata un punto fermo nel mercato dei

semiconduttori, nel corso degli ultimi anni le performance dei processori sono andate stabilizzandosi e sono emersi nuovi fattori che hanno iniziato a mettere in discussione una regola che sembrava immutabile.
La frequenza di clock, ad esempio, è rimasta praticamente invariata nel corso degli ultimi 5 anni, nonostante sia tecnicamente possibile aumentarla ancora: nel 2001 Intel stimò che la frequenza dei processori sarebbe arrivata a 30 GHz, ma nel 2005 la compagnia cambiò completamente approccio, bloccando di fatto la frequenza delle cpu appena sotto i 4 GHz.
Il principale motivo di questo stallo ha a che fare con il consumo energetico dei calcolatori e con la dissipazione del calore, fattori strettamente collegati al clock rate del processore.
Attualmente infatti, sembra più conveniente realizzare dispositivi a basso consumo, dotati di lunga autonomia, piuttosto che realizzare processori più potenti di quelli attuali.
Un altro fattore che ha avuto grande rilevanza nel mercato dei processori è l'introduzione di cpu multicore (2005), dotate di più unità di elaborazione su un singolo chip (figura 1.1).

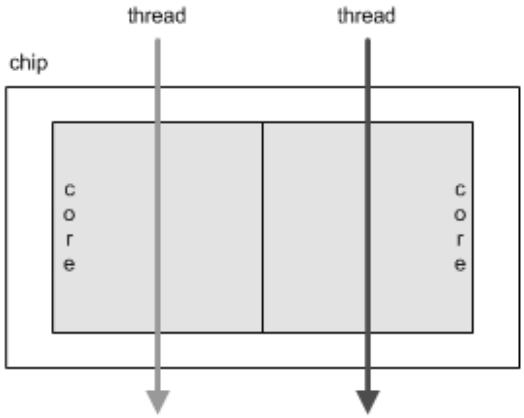


Figura 1.1: Struttura di un processore dualcore.

Questa innovazione ha permesso di incrementare notevolmente le prestazioni

dei processori senza grandi inconvenienti tecnici: ogni core dispone della propria pipeline di esecuzione ed è in grado di portare avanti in maniera autonoma l'esecuzione di un thread.
Questo tipo di processori, inoltre, sposa in maniera ottimale la filosofia multitasking dei comuni sistemi operativi che, diffondendo su più core i thread in esecuzione, sono in grado di incrementare notevolmente le performance.
I sistemi multicore portano sensibili miglioramenti alle prestazioni del software che è così in grado di sfruttare appieno il parallelismo disponibile, mentre le applicazioni sequenziali godono solo parzialmente di benefici portati da queste architetture.
Per questo motivo, nel processo di sviluppo software, è molto importante conoscere le caratteristiche e le potenzialità del calcolatore che si ha a disposizione.
Nella prossima sezione verrà effettuata una panoramica sulle architetture di ultima generazione, mostrando quali sono i principali pregi e difetti dei diversi dispositivi.

1.3 Hardware di Ultima Generazione

Il mondo del calcolo ad alte prestazioni, comunemente chiamato HPC (High Performance Computing), si evolve in maniera molto rapida, nel tentativo di ricercare continuamente soluzioni tecnologiche sempre più performanti. A partire dagli anni '90 il mercato dell'HPC si è basato sui processori tradizionali, cioè su CPU (Central Processing Unit), che garantivano un continuo miglioramento grazie all'affidabilità della legge di Moore.
Tuttavia negli ultimi anni è emersa la consapevolezza che l'impiego di processori tradizionali, basati su architetture multicore, avrebbe significato seri problemi di dissipazione del calore e un incremento inaccettabile dei consumi.
L'attenzione del mondo HPC si è quindi spostata verso architetture di tipo eterogeneo nelle quali, all'interno del nodo computazionale, viene affiancato

al tradizionale processore multicore, un coprocessore (o acceleratore) in grado di fornire, su un insieme ben definito di algoritmi, elevate prestazioni a consumi contenuti.
L'intero settore del calcolo scientifico si è quindi interessato al mondo degli acceleratori, in grado di proporre soluzioni interessanti anche per configurazioni di medie o piccole dimensioni.

In questa sezione verranno presentate alcune tra le soluzioni tecnologiche più performanti proposte negli ultimi anni da diverse aziende produttrici, molte delle quali sono state valutate nella scelta della piattaforma d'esecuzione, che deve garantire prestazioni ad alto livello per il calcolo scientifico. D'altra parte, l'aspetto commerciale dell'intero lavoro richiede che i requisiti hardware siano facilmente accessibili a potenziali clienti, quindi la piattaforma d'esecuzione deve essere relativamente economica e facilmente reperibile.

1.3.1 Field-programmable gate array

I Field Programmable Gate Array (FPGA) sono circuiti integrati digitali le cui funzionalità sono programmabili via software. Questi elementi presentano caratteristiche intermedie tra i dispositivi ASIC (Application Specific Integrated Circuit) e quelli PAL (Programmable Array Logic) e rappresentano una combinazione delle caratteristiche più interessanti di entrambi.
Gli ASIC sono circuiti special purpose, progettati e programmati per risolvere una specifica applicazione di calcolo, definita dal costruttore durante la produzione. La specificità di tali dispositivi, focalizzata sulla risoluzione di un unico problema, consente di raggiungere ottime prestazioni in termini di velocità di elaborazione e consumo elettrico, ma è anche la causa del limitato impiego di questa tecnologia.
I PAL, invece, sono i più semplici dispositivi logici programmabili, furono prodotti per la prima volta a livello industriale verso la fine degli anni '70.
La principale caratteristica di questi circuiti integrati era, ed è, la pro-

grammabilità; costruttore ed utilizzatore erano in grado di programmare ogni dispositivo per la realizzazione di una specifica funzione logica. Purtroppo però, dopo aver programmato il circuito, non era più possibile modificarlo o aggiornarlo in alcun modo.

I circuiti FPGA presentano diverse analogie con entrambi i dispositivi appena descritti. Ogni dispositivo è composto da una matrice di celle logiche che sono collegate da una rete di interconnessioni e interruttori programmabili. Solitamente la configurazione è specificata con un hardware description language (HDL), in maniera simile a quanto avviene per gli ASIC, ma le funzionalità da implementare vengono impostate dal consumatore, non dal produttore. I circuiti FPGA possono quindi essere prodotti a basso prezzo e su larga scala, per poi essere programmati dall'utente finale, che può così sfruttarne appieno le potenzialità.
Inoltre questi dispositivi, a differenza di quanto avviene per i PAL, possono essere riprogrammati in qualsiasi momento ed è quindi possibile apportare modifiche o correggere errori.

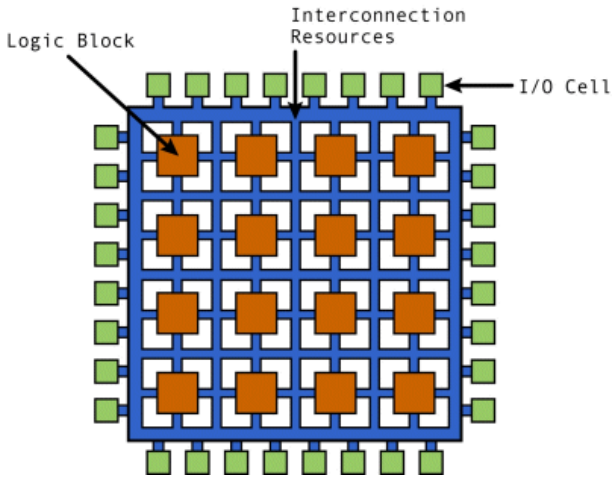


Figura 1.2: Architettura di un FPGA.

Il primo FPGA fu prodotto dalla azienda Xilinx e messo in vendita per la prima volta nel 1985, con il nome di XC2064. Questo dispositivo permette-

va di programmare sia le porte logiche, che le interconnessioni tra di esse e disponeva complessivamente di 64 blocchi logici configurabili.
Xilinx fu l'unica azienda a produrre questo tipo di dispositivi fin quasi alla metà degli anni '90, quando iniziarono a sorgere nuovi concorrenti. In questo periodo i FPGA furono impiegati in diversi settori e subirono un notevole incremento sia nel volume di produzione, sia nella qualità e sofisticazione dei dispositivi prodotti.

Recentemente è stato fatto un ulteriore passo avanti in questo settore, combinando blocchi logici e interconnessioni tipici dei FPGA con microprocessori embedded e relative periferiche, con lo scopo di sviluppare un vero e proprio sistema programmabile integrato su chip.
Nel 2010 è stato commercializzato il primo *All Programable System on a Chip*, chiamato Xilinx Zynq-7000, che include un processore dual-core ARM Cortex-A9 MPCore da 1.0 GHz integrato all'interno della struttura di un FPGA.
Attualmente, questi dispositivi sembrano la soluzione ideale da utilizzare in sistemi embedded in cui sia richiesta potenza di calcolo, ma per i quali sussistano limitazioni in termini di spazio e non sia possibile impiegare un calcolatore di dimensioni maggiori.

1.3.2 Cell Broadband Engine

Il Cell Broadband Engine (abbreviato a Cell BE) è un'architettura sviluppata dall'unione tra Sony, Toshiba e IBM, formalizzata nel 2000 con la creazione di un'alleanza nota come "STI".
La principale applicazione commerciale di questo dispositivo è stato l'utilizzo nella console PlayStation 3, presentata nel 2006.
La configurazione di base è quella di un dispositivo multicore, composta da un processore centrale, chiamato PPE (Power Processor Element) e 8 coprocessori secondari, detti SPE (Synergistic Processing Elements), specializzati

nell'esecuzione di un ristretto insieme di compiti.
Il PPE è composto da un insieme di chip, tra i quali ci sono un'unità logica di elaborazione (PXU), una serie di registri SIMD, due cache di livello L1 (una per i dati e l'altra per il codice) e una cache L2. Il suo compito è controllare e sincronizzare le unità SPE, che dovranno effettivamente eseguire la maggior parte del carico di lavoro.
Ogni SPE è composta da un processore (SXU) per l'elaborazione di istruzioni floating point, una memoria locale ad alta velocità (Local Storage, LS) e un chip per l'accesso alla memoria principale (con chiamata DMA).
I processori hanno la possibilità di comunicare tra loro e sono collegati con un bus chiamato EIB (Element Interconnect Bus).

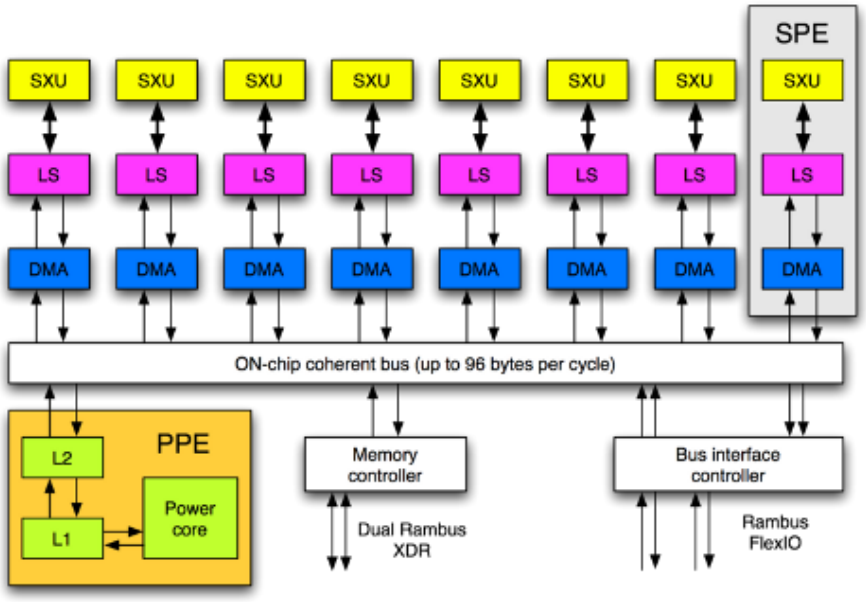


Figura 1.3: Architettura di un Cell BE.

Avendo principalmente il compito di gestione delle altre unità, il PPE incrementa le sue prestazioni e lo sfruttamento delle risorse con il multithreading: la grana del calcolo non è dunque particolarmente elevata ma diventa importante una buona gestione dei thread e dei processi.

Per quanto riguarda gli SPE vale invece il discorso opposto: sono unità specializzate nella computazione e hanno uno spazio di memoria relativamente grande. Il lavoro ottimale per queste unità è quindi l'esecuzione di calcoli su insiemi di dati omogenei.
Sia PPE che SPE supportano il calcolo vettoriale: in questo modo ogni singolo SPE è capace di eseguire contemporaneamente la stessa operazione su 8 valori diversi (lavorando in singola precisione).
Grazie a questo meccanismo le performance teoriche di ogni core arrivano a circa 32 GFlop/s [1].
Le performance teoriche di questo dispositivo lo rendono molto interessante per il calcolo in ambito scientifico.
Probabilmente a causa della concorrenza degli acceleratori basati su GPU, nel 2009 IBM ha annunciato l'intenzione di ridurre gli investimenti nell'evoluzione del prodotto e ne ha di fatto decretato la fine.

1.3.3 GPU computing e NVIDIA

Certamente al momento la famiglia di acceleratori più diffusa è rappresentato dalle GPU (Graphics Processor Units), i processori grafici responsabili della visualizzazione delle immagini e presenti in ogni scheda grafica.
Dalla fine degli anni '90 iniziarono ad essere programmabili in modo sempre più esteso; si incominciò quindi a considerare la possibilità di impiegarli non solo come processori grafici, ma come coprocessori dedicati all'esecuzione di calcoli di tipo scientifico, per sfruttare la loro potenza di calcolo nell'esecuzione di compiti general-purpose [2]
NVIDIA è da diversi anni l'azienda leader mondiale nella produzione di pro-

[1] Flop/s è un'abbreviazione di FLoating point Operations Per Second e indica il numero di operazioni in virgola mobile che un processore è in grado di eseguire in un secondo.

[2] Questa sezione è dedicata alla descrizione dell'hardware e delle sue prestazioni, più che alle tecniche con le quali è possibile raggiungerle. Per questo motivo la computazione su GPU (GP-GPU) viene solo accennata per ora: sarà approfondita maggiormente nel prossimo capitolo.

cessori grafici, ma recentemente si sta imponendo anche nel settore del calcolo ad alte prestazioni. Le ultime generazioni di GPU sviluppate da questa azienda, infatti, uniscono prestazioni elevate ad una migliorata efficienza energetica.
Il prodotto di punta della famiglia Kepler, presentata nel 2012, è la GPU Kepler GK110, che fornisce prestazioni di picco di oltre 1 TFlop/s in doppia precisione.

Figura 1.4: Architettura della GPU Kepler.

La GK110 contiene 15 streaming multiprocessors (SMX), dotati di 192 cores ciascuno, e 6 controller di memoria a 64 bit. Rispetto alle GPU precedenti, Kepler supporta il parallelismo dinamico che consente alla GPU di bilanciare autonomamente i thread in maniera più equilibrata, sfruttando al meglio la capacità di elaborazione parallela. Inoltre il sistema Hyper-Q consente a diversi core della CPU di condividere l'utilizzo di una singola GPU, riducendo il tempo di inattività e migliorando le prestazioni.

1.3.4 Intel Xeon Phi

L'architettura Many Integrated Core (MIC), sviluppata recentemente da Intel, combina un elevato numero di core in un unico chip, con l'obbiettivo di realizzare un coprocessore che garantisca alte prestazioni e una programmabilità simile a quella dei tradizionali processori multicore.
Nel 2012 Intel ha messo sul mercato la seconda implementazione dell'architettura MIC, il coprocessore Xeon Phi (nome in codice Knights Corner), disponibile nella versione 5110P, dotato di 60 cores, operanti a 1.053 GHz, e di 8GB di memoria GDDR5. Ogni core è multithreaded e supporta 4 hardware threads, contiene una unità vettoriale a 512 bit (utilizzabili come 8 elementi a doppia precisione o 16 elementi a precisione singola), e una memoria cache L2 di 512 KB.

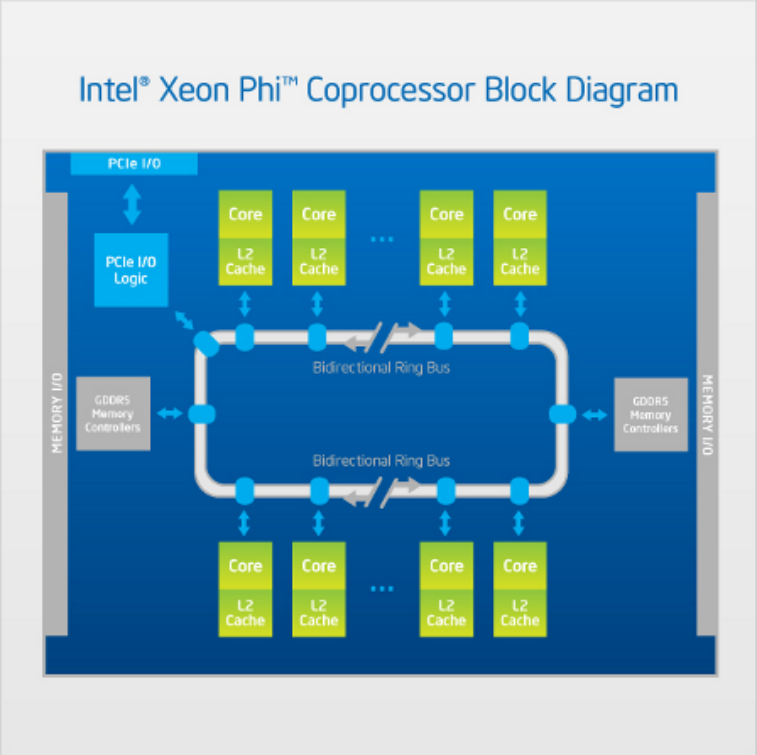


Figura 1.5: Architettura di un Intel Xeon Phi

Complessivamente il coprocessore Xeon Phi 5110P supporta 240 threads, ha una potenza di picco di circa 1 TFlop/s ed una memory bandwidth di 320 GB/s. I core sono collegati tra di loro mediante una interconnessione ad anello per massimizzare il throughput[3].

É possibile utilizzare il coprocessore Xeon Phi con diversi gradi di integrazione rispetto al processore host: i programmi possono eseguire sull'host lanciando solo alcune parti computazionali sul coprocessore (offload mode) in modo analogo a quanto avviene per le GPU, o possono eseguire completamente su Xeon Phi (autonomous mode).
I coprocessori Xeon Phi utilizzano una estensione delle istruzioni x86 e il compilatore Intel è in grado di generare codice per Xeon Phi a partire da programmi paralleli scritti con i principali linguaggi di programmazione. Questa caratteristica fornisce una forte continuità con i prodotti precedenti e consente una grande portabilità del software, rendendo il processore Xeon Phi una piattaforma di sviluppo molto interessante.

1.3.5 Tilera TILE-Gx

L'ultima generazione di processori multicore prodotti da Tilera è chiamata TILE-Gx e presenta caratteristiche simile alla famiglia Xeon Phi sviluppata da Intel. Infatti, come i dispositivi Intel, i TILE-Gx sono realizzati integrando sullo stesso chip un alto numero di core identici tra loro.
I processori Tilera sono realizzati disponendo i core (chiamati *tile*) su una struttura a mesh (matrice) e sono disponibili in 4 diverse configurazioni (16, 36, 64 o 100 core), ognuna della quali offre potenza di calcolo crescente.
Ogni tile può essere sfruttato come processore autonomo dotato di piena funzionalità; inoltre dispongono di una cache L1 da 32 KB, una L2 da 256 KB e di coprocessori dedicati all'esecuzione di compiti crittografici ad alte

[3]Il throughput di una connessione fa riferimento alla quanità di dati che è possibile trasferire da una locazione ad un'altra ed è quindi un indice di misurazione della massima portata di trasferimento di un link.

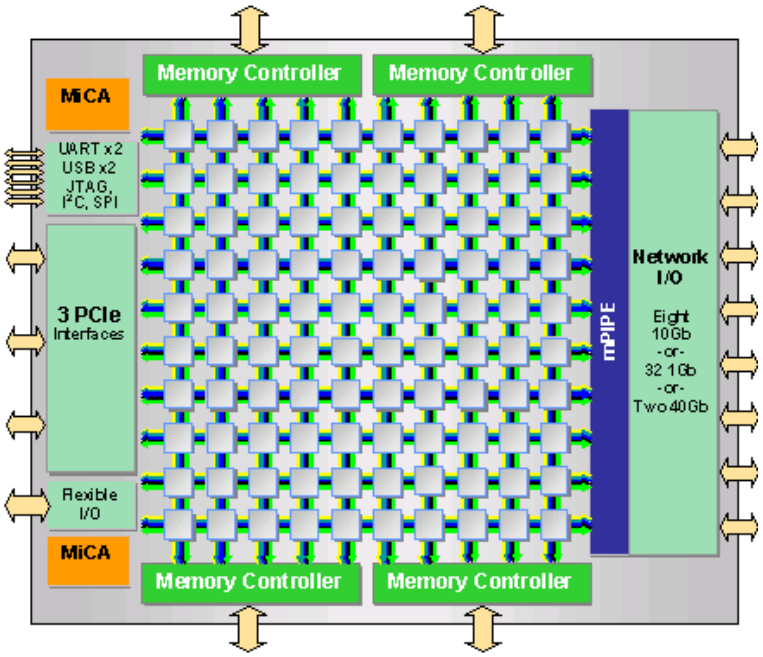


Figura 1.6: Architettura di un Tile-Gx con 100 core

prestazioni.
I tile che compongono un dispositivo sono collegati con una rete integrata sul chip, chiamata Tilera iMesh, ma sono dotati di una serie di interruttori che consentono di scollegare il core dalla mesh.
Grazie all'alto livello di parallelismo e alla potente rete di connessione interna, i dispositivi della famiglia Tile-Gx sono specializzati nell'esecuzione di applicazioni che sfruttano pesantemente la rete; inoltre sono dotati di un ricco set di porte input/output, ottimizzazioni per l'elaborazione dei pacchetti e accelerazione di rete, caratteristiche che li rendono perfetti per applicazioni di Cloud Computing.

1.3.6 AMD APU Trinity

AMD è uno dei leader mondiali sia nella produzione di microprocessori con architettura x86 (dopo Intel), sia nella produzione di chip grafici (dopo

NVIDIA).
Negli ultimi anni AMD ha portato avanti un progetto, denominato Fusion, nel quale CPU e GPU vengono progressivamente integrati all'interno di un singolo componente. Questi dispositivi ibridi sono stati battezzati dall'azienda produttrice come APU (Accelerated Processing Unit) e sono sistemi di elaborazione per il calcolo eterogeneo, che consentono di accelerare alcune operazioni al di fuori della componente CPU.
Verso la fine del 2012 sono state presentate ufficialmente le APU della serie 5000 per sistemi desktop, basate su un'architettura con il nome in codice di Trinity.
Per quanto riguarda la componente CPU, l'ultima generazione di APU presenta alcune novità a livello di architettura: la struttura principale è rimasta invariata, ma sono stati integrati vari affinamenti mirati ad incrementare le prestazioni complessive, diminuendo l'impatto in termini di consumi. Nei prodotti di punta la CPU è composta da 4 core x86 con una cache L2 di 2MB.
Una interessante novità è la tecnologia Turbo Core, grazie alla quale la frequenza di clock della CPU è gestita dinamicamente ed è possibile portarla oltre il valore predefinito, attivando un overclocking automatico in determinate condizioni di carico.
Nella logica di progettazione di AMD, la GPU svolge un ruolo decisamente importante; già nella generazione precedente, la grafica integrata era il punto di forza delle APU. Questa componente è basata su un'architettura VLIW-4 che, pur disponendo di un numero inferiore di stream processor rispetto alla versione precedente, garantisce prestazioni globalmente migliori.
I dispositivi Trinity rappresentano una valida soluzione per il calcolo ad alte prestazioni, garantendo un buon rapporto tra prezzo e prestazioni, insieme a consumi abbastanza contenuti.

1.3.7 Considerazioni conclusive

È difficile confrontare tecnologie molto diverse, come CPU e GPU, perché entrambe le soluzioni presentano sia vantaggi che svantaggi.
Per prima cosa è importante considerare il modello di programmazione: i linguaggi per programmare su GPU, se confrontati con quelli disponibili per CPU, sono ancora abbastanza restrittivi ed è richiesto l'impiego di linguaggi e tecniche di programmazione appositamente realizzate per il gpu computing.
Inoltre ogni soluzione richiede una serie di tecnologie aggiuntive come dispositivi di I/O, sistemi di raffreddamento, processore principale (per le GPU): differenze a questo livello possono influire sia sul costo che sulle prestazioni del sistema.
Inoltre bisogna considerare che le performance teoriche di picco, spesso prese come parametro per misurare le prestazioni di un sistema, non sempre si traduco nell'effettivo rendimento del lavoro, soprattutto per le GPU.
A questo si aggiunga che l'architettura delle GPU è radicalmente diversa da quella delle CPU, in particolare per l'organizzazione della memoria; per ottenere prestazioni significative è spesso necessario sfruttare caratteristiche peculiari, limitando la generalità dell'algoritmo o vincolandolo all'architettura utilizzata al momento, col rischio di dover introdurre sostanziali modifiche per poterlo eseguire in modo efficiente su architetture successive.
D'altra parte le performance offerte dalle GPU sono in crescita verticale e le piattaforme grafiche stanno occupando una fetta di mercato sempre più rilevante, anche nell'ambito del calcolo ad alte prestazioni.

La figura 1.7 mostra un confronto tra le performance di CPU Intel e GPU NVIDIA sviluppate negli ultimi anni. Il grafico mostra un trend decisamente interessante: le prestazioni dei processori grafici sono cresciute in modo esponenziale nell'ultimo decennio e questo processo non sembra destinato a fermarsi.
Allo stato dell'arte attuale, per le ragioni appena esposte, le GPU rappresentano uno scenario molto promettente per l'esecuzione di calcolo scientifico

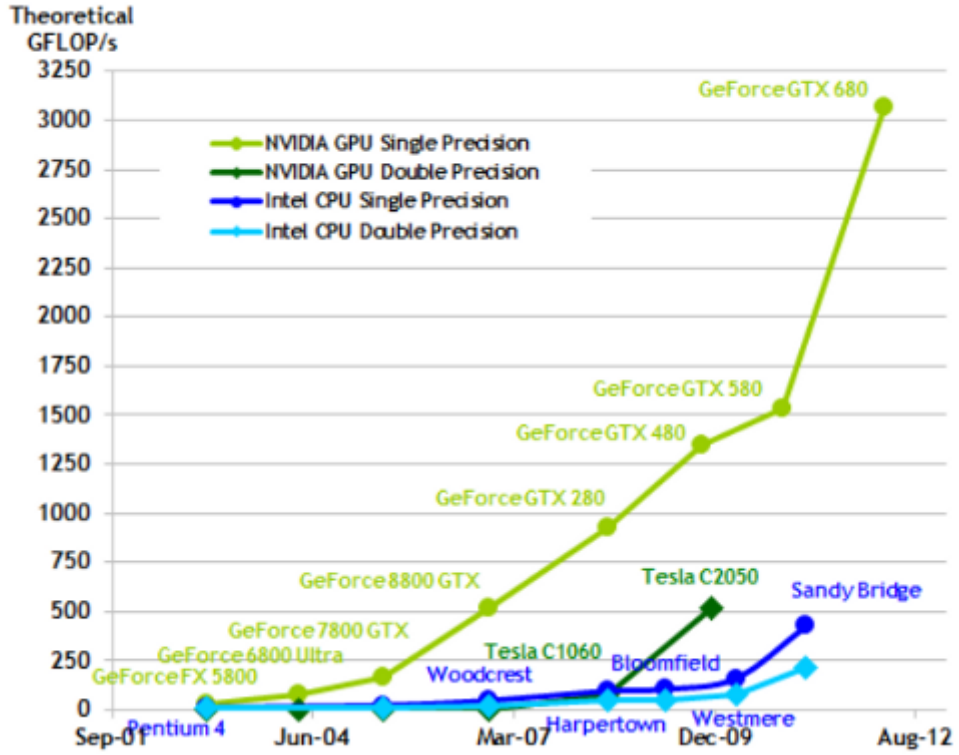


Figura 1.7: Confronto performance NVIDIA - Intel
Sorgente: CUDA C Programming Guide, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

ad alte prestazioni, ma le limitazioni in termini di programmabilità rendono difficile lo sviluppo di software completamente ottimizzato.
Lavorando con processori tradizionali, invece, è molto semplice realizzare codice parallelo che funzioni in ambiente multicore, poiché esistono interfacce di programmazione che consentono di sfruttare in maniera semplice i diversi processori che l'architettura mette a disposizione.
Le GPU sono quindi state scelte come piattaforma d'esecuzione per la versione finale del sistema software, ma si è deciso di effettuare un passo intermedio, producendo anche una versione del codice per l'esecuzione su processori multicore.
Questa scelta permette l'applicazione di diversi livelli di raffinamento e ot-

timizzazione al codice prodotto, garantisce un metro di paragone per la versione finale del software e consente di svincolarsi da una singola piattaforma di sviluppo, ampliando la gamma di tecnologie supportate.

Capitolo 2

Programmazione GP-GPU

In questo capitolo verrà introdotto il modello di calcolo GP-GPU e le principali tecniche di programmazione ad esso associate.
Per comprendere meglio questi argomenti, sarà prima descritta la struttura delle moderne GPU.

2.1 GPU

L'unità di elaborazione grafica, comunemente chiamata GPU (Graphics Processing Unit) è un circuito elettronico specializzato nel processare dati destinati ad essere visualizzati graficamente. Sebbene siano state ideate per lo svolgimento di compiti limitati, le GPU hanno continuato ad evolversi, diventando sempre più complesse e performanti.
Le GPU sono caratterizzate da una struttura altamente parallela, che consente di manipolare grandi insiemi di dati in maniera efficiente. Questa caratteristica, in combinazione con la rapida crescita prestazionale dell'hardware grafico e i suoi recenti miglioramenti in termini di programmabilità, ha portato l'attenzione del mondo scientifico sulla possibilità di utilizzare la GPU per scopi diversi da quelli tradizionali.
Il settore che si occupa di questo argomento è chiamato GP-GPU (General-

Purpose computing on Graphics Processing Units)[4] e sarà analizzato in dettaglio nel paragrafo successivo.

Per comprendere meglio la sezione seguente, verrà ora analizzata l'architettura di un generico processore grafico moderno[2], rappresentata in figura 2.1.

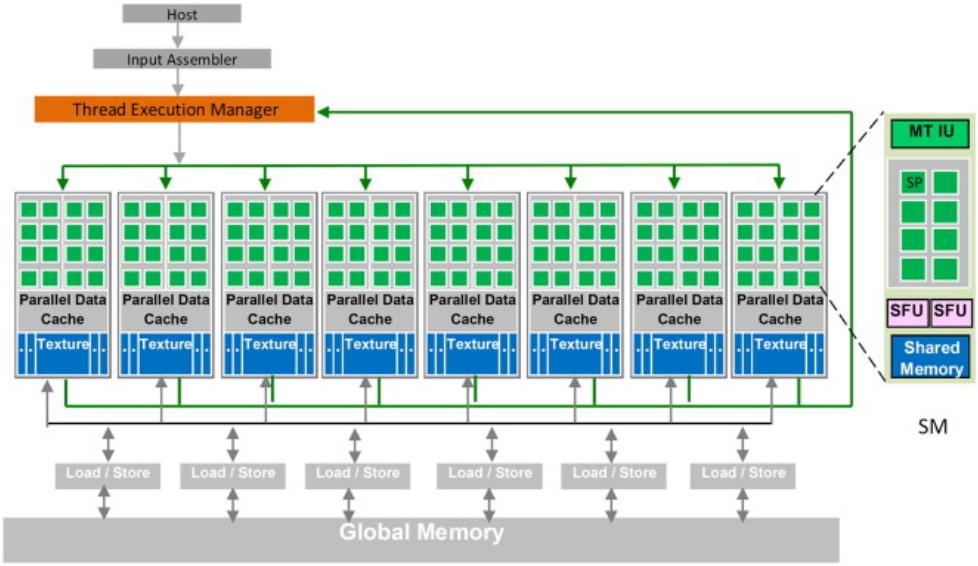


Figura 2.1: Architettura di una GPU moderna.

Le GPU sono coprocessori e in quanto tali devono essere connesse ad un processore tradizionale chiamato, in questo contesto, *host*; generalmente il collegamento tra i due dispositivi è realizzato con un bus PCI Express ad alta velocità, che consente il trasferimento di dati dalla memoria della CPU a quella della GPU e viceversa.
I processori grafici dispongono di un alto grado di parallelismo, in particolare nella struttura dei diversi dispositivi di memorizzazione e nella disposizione dei core d'esecuzione.
Ogni GPU è infatti composta da diverse unità di elaborazione, chiamate *Streaming Multiprocessor* (SM), che rappresentano il primo livello logico di

parallelismo; ogni SM infatti lavora contemporaneamente e in maniera indipendente dagli altri.
Ogni SM è a sua volta suddiviso in un gruppo di *Streaming Processor* (SP), ognuno dei quali è un core d'esecuzione reale, in grado di eseguire sequenzialmente un thread. Gli SP rappresentano la più piccola unità logica d'esecuzione e rappresentano il livello di parallelismo più fine.
La divisione in SM e SP è di natura strutturale, ma è possibile delineare una ulteriore organizzazione logica tra gli SP di una GPU, che sono infatti raggruppati in blocchi logici caratterizzati da una particolare modalità d'esecuzione: tutti i core che compongono un gruppo eseguono contemporaneamente la stessa istruzione, in maniera molto simile a quanto avviene con il modello SIMD (*Single Instruction, Multiple Data*).

Ogni GPU, inoltre, ha a disposizione diversi tipi di memoria, ognuna delle quali è localizzata in differnti aree del dispositivo, possiede caratteristiche diverse e può essere utilizzata per compiti differenti.
L'unità di memoria più capiente è la cosiddetta memoria globale, che nelle GPU di ultima generazione raggiunge dimensioni di diversi GB, ma comporta una latenza[1] piuttosto alta. Tutti i core della GPU possono accedere a questo spazio di memoria, così come può fare anche l'host, che è collegato in maniera diretta con il dispositivo di memorizzazione. Per questo, la memoria globale è spesso utilizzata per ospitare grandi quantità di dati trasferiti dalla memoria del processore.
Non essendo particolarmente veloce, la memoria globale spesso dispone di meccanismi di ottimizzazione degli accessi o di livelli di cache, che consentono comunicazioni più efficienti con la GPU.
Un altro spazio di memoria accessibile a tutti i core è la texture memory, una regione utilizzabile dalla GPU in sola lettura, originariamente progettata per la memorizzazione di particolari immagini, chiamate texture, impiegate

[1]Con il termine latenza, in ambito informatico, si intende l'intervallo di tempo che intercorre tra il momento in cui viene inviata una richiesta e l'istante in cui si riceve la risposta. Nel caso di un dispositivo di memoria si intende quindi il tempo trascorso tra l'inizio e la fine di un'istruzione di lettura o scrittura in memoria.

nella rappresentazione grafica di elementi tridimensionali. Questo tipo di memoria dispone di specifici meccanismi di caching che la rendono efficiente nella memorizzazione di strutture bidimensionali (matrici o immagini) e nell'accesso ad elementi spazialmente vicini.
All'interno degli SM è presente inoltre uno spazio di memoria condiviso tra i core che compongono un gruppo di lavoro; questa memoria è decisamente più veloce di quella globale ma ha dimensioni molto più ridotte, dell'ordine di qualche decina di MB per ogni SM. Può essere utilizzata per memorizzare valori che devono essere utilizzati più volte, da core diversi, limitando il numero di accessi alla memoria globale.
Gli aspetti negativi della condivisione di questo spazio di memoria riguardano i noti problemi che affliggono i modelli di calcolo concorrenti: è necessario assicurare in ogni momento la coerenza dei valori memorizzati, anche a costo di sequenzializzare accessi contemporanei alla stessa locazione di memoria.
Ogni SM inoltre dispone di un certo numero di registri, che rappresentano uno spazio di memoria ad accesso rapido, temporaneo, locale (non condiviso tra i core) e di dimensioni limitate, che consente la memorizzazione di valori frequentemente utilizzati da un singolo core.

2.2 GP-GPU

Il GP-GPU[4] è il settore dedicato allo studio delle tecniche che consentono di sfruttare la potenza computazionale delle GPU per effettuare calcoli in maniera veloce, grazie all'alto livello di parallelismo interno che le contraddistingue.
Come visto nella sezione precedente, le GPU sono strutturate in maniera completamente diversa dai tradizionali processori; per questo presentano problemi di natura differente e richiedono tecniche di programmazione specifiche. La caratteristica più rilevante che contraddistingue i processori grafici è l'alto numero di core a disposizione, che permette di portare avanti molti

thread d'esecuzione concorrenti, parzialmente sincronizzati nell'esecuzione della stessa operazione.
Questa caratteristica risulta molto utile ed efficiente nelle situazioni in cui è possibile suddividere il lavoro in tante parti, effettuando le stesse operazioni su dati diversi; al contrario, è difficile utilizzare al meglio un'architettura di questo tipo quando esiste una forte sequenzialità e un ordine logico da rispettare nelle operazioni da svolgere, o il lavoro non può essere diviso equamente in tante piccole sottoparti.
Il paradigma di programmazione che caratterizza il calcolo su GPU è chiamato *Stream Processing*, perché i dati possono essere visti come un flusso omogeneo di valori ai quali vengono applicate in maniera sincrona le stesse operazioni.
Le funzioni che processano i dati nello stream e che sono eseguite sulla GPU prendono il nome di *kernel* e sono in grado di applicare solo un insieme limitato di operazioni ad ogni flusso di dati.

- Operazioni di copia: tutti gli elementi sono copiati da uno stream ad un altro.
- Operazioni di filtraggio: alcuni elementi di uno stream sono selezionati in base a determinate caratteristiche e copiati.
- Operazioni di accesso: si accede ad uno specifico elemento dello stream utilizzando un indice.
- Operazioni di calcolo: vengono effettuati calcoli sugli elementi di uno stream.

Nel modello di programmazione stream processing, un programma principale è in esecuzione sull'*host* (CPU) e gestisce il parallelismo concatenando esecuzioni di kernel su stream diversi.
La struttura di funzionamento di un kernel segue quasi sempre uno schema preciso: per prima cosa viene caricato nella memoria della GPU un flusso di dati (input), sul quale il kernel applica una serie di operazioni che producono

un flusso di dati in uscita (output). Sarà poi compito dell'host estrarre dalla memoria della GPU i dati prodotti dal kernel.

È evidente quindi che non tutte le applicazioni possono sfruttare a pieno la potenza di calcolo delle GPU perché, per ottenere un reale guadagno computazionale, deve essere possibile una completa parallelizzazione delle operazioni. L'elaborazione GP-GPU raggiunge prestazioni ottimali nella soluzione di problemi di grandi dimensioni, che possono però essere suddivisi in tanti sottoproblemi dello stesso tipo, risolubili contemporaneamente.

2.3 Approcci alla programmazione GP-GPU

Il grande aumento di performance presentato dalle GPU nell'ultimo decennio ha attirato l'interesse del mondo scientifico e degli sviluppatori verso la possibilità di impiegare i processori grafici come piattaforme per effettuare calcoli general purpose. Questo ha consentito la nascita e il rapido sviluppo di diversi linguaggi di programmazione per l'ambiente grafico.
Il GPU computing ha preso veramente il via quando, nel 2006, vennero presentati CUDA e Stream, due interfacce per la programmazione grafica progettate dai due principali sviluppatori di GPU, rispettivamente NVIDIA e AMD.
Qualche anno più tardi venne avviato il progetto OpenCL, con lo scopo di realizzare un framework d'esecuzione eterogeneo sul quale potessero lavorare sia GPU (sviluppate da diversi produttori), sia CPU.
Col passare del tempo questi linguaggi si sono evoluti enormemente, diventando sempre più simili ai comuni linguaggi di programmazione general-purpose e aggiungendo meccanismi di controllo sempre più validi ed efficaci.
Attualmente, CUDA e OpenCL rappresentano le soluzioni più efficienti per sfruttare in maniera diretta la potenza di calcolo fornita dal processore grafico e verranno presentati di seguito.

2.3.1 CUDA

CUDA (Compute Unified Device Architecture) è un modello di programmazione creato da NVIDIA come piattaforma per la computazione parallela. L'architettura CUDA include un linguaggio assembly (PTX) e un sistema di compilazione (nvcc) che rappresentano la struttura di base sulla quale è fondata l'intera computazione su GPU NVIDIA.
La piattaforma CUDA consente diversi livelli di interazione ed è accessibile agli sviluppatori in diversi modi.

- Il primo è basato su librerie accelerate dalla GPU, come cuBLAS o cuFFT; questo approccio è di semplice utilizzo e ha il vantaggio di poter essere impiegato senza la necessità di specifiche conoscenze sul funzionamento della GPU.
  Questo tipo di librerie, infatti, realizzano versioni graficamente accelerate, quindi più performanti, di alcune utili funzioni. Lo svantaggio evidente di questa soluzione è la scarsa adattabilità alle esigenze del programmatore: le ottimizzazioni computazionali sono garantite solo su un ristretto numero di operazioni.

- Un'altra soluzione consiste nell'impiego di direttive (come OpenACC). Questa tecnica prevede di inserire all'interno del codice particolari istruzioni che, interpretate dal compilatore, consentono di parallelizzarlo, utilizzando la potenza di calcolo dei processori grafici nel modo che il compilatore ritiene più efficace.
  Tale approccio è molto semplice da utilizzare ma, come il precedente, non consente un controllo completo sui meccanismi di elaborazione, delegando tutto il lavoro al compilatore.

- L'ultima soluzione consiste nell'utilizzo di un linguaggio di programmazione general-purpose esteso con istruzioni CUDA che permettono l'esecuzione su GPU. NVIDIA ha realizzato estensioni per i linguaggi C (CUDA-C/C++) e Fortran (CUDA-Fortran), ma altri partner hanno sviluppato wrapper che supportano, tra gli altri, i linguaggi Java,

Python, Perl e Ruby.
Gli sviluppatori che scelgono questa soluzione hanno quindi un accesso diretto alle caratteristiche hardware della GPU, come memoria condivisa, meccanismi di caricamento dati e core d'esecuzione. Questa tecnica è sicuramente la più complessa, ma è l'unica in grado di garantire allo sviluppatore il pieno controllo del processore grafico.

La piattaforma CUDA è progettata per essere utilizzata esclusivamente con processori grafici prodotti da NVIDIA, caratteristica che ne limita l'impiego in misura considerevole. D'altra parte questo vincolo è probabilmente il motivo per cui, sui dispositivi NVIDIA, CUDA è la soluzione che offre migliori performance: il limitato numero di dispositivi da supportare ha consentito la realizzazione di una piattaforma specializzata, mirata all'ottimizzazione di un numero ristretto di operazioni.

2.3.2 OpenCL

OpenCL (Open Computing Language) è un framework per lo sviluppo di programmi in grado di lavorare su piattaforme eterogenee, che possono essere composte indifferentemente da CPU e da GPU realizzate da diversi produttori. Questa piattaforma è stata ideata da Apple, ma è stata sviluppata e mantenuta da un consorzio no-profit chiamato Khronos Group.
OpenCL è la principale alternativa a CUDA per l'esecuzione di software su GPU, ma presenta un punto di vista diametralmente opposto; mentre CUDA fa della specializzazione il proprio punto di forza (prodotta, sviluppata e compatibile con NVIDIA), garantendo ottime prestazioni a scapito della portabilità, OpenCL propone una soluzione compatibile con la quasi totalità dei dispositivi presenti sul mercato. Il software scritto in OpenCL, infatti, può essere eseguito su processori (grafici e non) prodotti da tutte le maggiori industrie del settore, come Intel, NVIDIA, IBM, AMD.

OpenCL include un linguaggio per scrivere kernel basato su C99 (con alcune limitazioni), che consente di utilizzare in maniera diretta le potenzialità dell'hardware a disposizione, in maniera analoga a come avviene con CUDA-C o CUDA-Fortran. OpenCL mette a disposizione funzioni per l'esecuzione in ambiente altamente parallelo, primitive di sincronizzazione, qualificatori per le regioni di memoria e meccanismi di controllo per le diverse piattaforme d'esecuzione.
La portabilità dei programmi OpenCL è però limitata alla possibilità di eseguire lo stesso codice su dispositivi diversi e non garantisce che le prestazioni siano ugualmente affidabili. Per ottenere le migliori prestazioni possibili, infatti, è fondamentale fare riferimento alla piattaforma d'esecuzione, ottimizzando il codice in base alle caratteristiche del dispositivo.

2.3.3 Confronto tra le piattaforme

La necessità di una soluzione altamente performante per lo sviluppo del software oggetto di questa tesi, ha indirizzato la scelta finale del sistema hardware da utilizzare, verso la soluzione che garantiva prestazioni migliori, ovvero le GPU. La scelta della piattaforma da utilizzare ha seguito lo stesso principio: anche se le caratteristiche eterogenee di OpenCL lo rendono una soluzione molto interessante dal punto di vista scientifico, si è deciso di puntare sulla tecnologia più performante, cioè CUDA.
Allo stato attuale, infatti, non solo CUDA garantisce prestazioni nettamente migliori, ma è generalmente considerata una tecnologia più matura e aggiornata. Il rilascio di CUDA 5, nell'Ottobre 2012, ha ulteriormente contribuito ad influenzare la scelta in favore della piattaforma NVIDIA a scapito di quella OpenCL, ferma alla release 1.2 risalente al Novembre 2011.

Capitolo 3

Ranklet

Nel settore dell'analisi di immagini, le Ranklet sono una famiglia di trasformate che presentano una caratteristica interessante: sono in grado di riconoscere, all'interno delle immagini a cui sono applicate, se ci sono aree nelle quali i valori sono disposti secondo linee verticali, orizzontali o diagonali. Inoltre godono di particolari proprietà di invarianza che le rendono straordinariamente robuste, anche nel caso in cui le immagini subiscano trasformazioni di luminosità.
Le ranklet sono un argomento di studio interessante e hanno numerose applicazioni in diversi settori scientifici, ma presentano alcune difficoltà di utilizzo, principalmente di natura computazionale, che rendono complesso e costoso il loro impiego. I motivi per cui sono adeguate ad essere trattate nell'ambito del calcolo ad alte prestazioni sono quindi di diversa natura e sono elencati di seguito.

- Le loro proprietà matematiche, insieme alle relative implicazioni geometriche, sono molto utili nel riconoscimento di pattern geometrici complessi, che trova applicazione in diversi settori scientifici, come ad esempio il riconoscimento facciale [13], la classificazione di texture [14] e il rilevamento di masse in immagini mammografiche [15, 16, 17].

- Dipendono dalle operazioni di ordinamento, la cui esecuzione efficiente costituisce un problema di difficile ottimizzazione, soprattutto in un

contesto parallelo, nonostante sia un argomento noto e ampiamente studiato.

- Il loro impiego in applicazioni di interesse scientifico richiede l'elaborazione di dati di grandi dimensioni, fattore che comporta problemi di gestione e memorizzazione, oltre che di natura computazionale.

Per tali motivi la trasformata ranklet è stata considerata un valido argomento di studio ed è stata scelta come problema computazionale. In questo capitolo verrà ampiamente trattata e approfondita.

3.1 Analisi di Immagini

Un'immagine digitale è un insieme di elementi disposti secondo un ordine preciso e prestabilito, ognuno dei quali è caratterizzato da uno o più valori di intensità che ne rappresentano il colore e la luminosità. Un'immagine in bianco e nero a 8 bit, per esempio, è una matrice di numeri interi ripartiti secondo l'intervallo di valori rappresentabili con 8 bit (0 - 255), i cui estremi sono il bianco e il nero.
Con il termine pixel (contrazione della locuzione inglese *picture element*), si indicano gli elementi puntiformi che compongono la rappresentazione digitale di un'immagine; ognuno di essi contiene informazioni relative al colore che l'immagine assume in un punto ed è interpretato relativamente alla posizione che occupata nella matrice.
L'analisi di immagini digitali consiste nell'estrazione di informazioni da matrici di pixel e nella successiva elaborazione con tecniche informatiche.
In questo contesto possiamo definire una trasformazione come un processo di codifica dei dati, identificato da relazioni matematiche, che permette di cambiare la rappresentazione dell'immagine.
La trasformazione non comporta perdita d'informazione se la relazione indotta tra l'oggetto in input e quello in output è biunivoca e quindi reversibile.

I principali scopi per cui è utile applicare trasformazioni alle immagini sono fondamentalmente due:

- compressione, che consiste nella memorizzazione di un'immagine in un formato più compatto e ha lo scopo di occupare una quantità di memoria inferiore.

- Mettere in evidenza particolari di interesse che rendono più semplice l'individuazione di determinati dettagli; queste tecniche, se applicate in sequenza, possono essere utilizzate anche per semplificare, migliorare o velocizzare le elaborazioni successive.

Uno dei settori più interessanti ai quali è possibile applicare con successo metodi di analisi di immagini è il campo medico. Tecniche come radiografia digitale e tomografia computerizzata sono tra le più note in questo ambito, al quale si fa generalmente riferimento con il termine Imaging Medico.
Recentemente sono state sviluppate tecniche automatiche che consentono di localizzare regioni di interesse clinico all'interno di immagini mediche (CAD, computer-aided detection). Questo settore si è poi sviluppato, portando alla realizzazione di procedure di diagnosi assistita (CADx, computer-aided diagnosis [10]), cioè avanzate procedure computerizzate che aiutano il medico nelle fasi di interpretazione e analisi di immagini radiografiche. L'obiettivo dei sistemi CAD e CADx è accorciare i tempi di diagnosi, migliorando contemporaneamente l'accuratezza delle analisi: il personale medico può confrontare i propri risultati con quelli generati dal sistema informatico, che svolge quindi un ruolo di supporto alla diagnosi.
Attualmente, i sistemi CADx sono applicati con successo in diversi ambiti medici. Vengono utilizzati per individuare i sintomi precoci del tumore al seno nelle mammografie[11], nelle radiografie al torace per la diagnosi di tumore al polmone[12] e per individuare fratture vertebrali, aneurismi intracranici[10], tumore al colon e alla prostata.
In particolare, sono stati sviluppati sistemi CAD per l'analisi di immagini mammografiche, che sfruttano la trasformata ranklet come strumento di ri-

levamento per possibili masse tumorali e microcalcificazioni a grappolo[15, 16, 17].

3.2 Wavelet

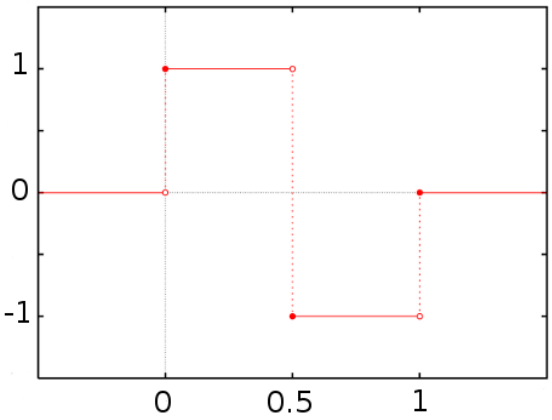
Le Ranklet possono essere considerate come una evoluzione delle Wavelet: una classe di funzioni che possono essere impiegate per rappresentarne altre. Questa idea risale al primo ventennio dell'800, quando Joseph Fourier scoprì che era possibile approssimare generiche funzioni periodiche come combinazione lineare delle funzioni sinusoidali fondamentali (serie di Fourier).
Analogamente, le wavelet possono essere usate come basi per la rappresentazione di funzioni matematiche, segnali e altri tipi di dati, come ad esempio le immagini[1]
Una caratteristica delle wavelet è che le basi sono non-lineari; per questo motivo, utilizzandole nell'approssimazione di una funzione, le basi sono scelte dinamicamente in modo da rappresentare le funzione in input nel modo più efficiente possibile. Questo consente un maggiore livello di precisione, rispetto alla serie di Fourier, soprattutto nell'approssimazione di discontinuità e punti di picco.
Nella analisi wavelet, inoltre, gioca un ruolo fondamentale il livello di specificità con cui vengono trattati i dati; una caratteristica degli algoritmi che implementano le funzioni wavelet, infatti, è l'elaborazione dei dati a diverse risoluzioni. L'aspetto interessante di questo approccio, generalmente chiamato Analisi in Multirisoluzione, riguarda la possibilità di estrarre dettagli di diversa accuratezza ad ogni livello di scala trattato.

[1] In realtà, le immagini possono essere considerate come particolari tipi di segnali digitali bidimensionali e possono essere rappresentate come funzioni $I(x, y)$, definite in termini di due coordinate spaziali x e y.

Una famiglia di wavelet può essere realizzata a partire da una funzione $\psi(x)$, generalmente chiamata *mother wavelet*, continua su un intervallo finito. Le altre funzioni della famiglia, comunemente chiamate figlie di $\psi(x)$, sono costruite per traslazione e contrazione, tramite una apposita funzione di scalatura ϕ, della funzione madre.
Nel corso della storia sono state ideate diverse famiglie di wavelet, ognuna delle quali è caratterizzata da una diversa funzione madre e presenta proprietà differenti.

Le Haar Wavelet sono la più antica famiglia di funzioni wavelet, proposta dal matematico Alfred Haar nel 1909, prima ancora che il termine wavelet venisse introdotto.
Questa famiglia è caratterizzata da funzioni di forma quadrata; lo svantaggio tecnico è che le funzioni non sono continue, e per questo non differenziabili; d'altra parte, proprio grazie a questa proprietà, sono in grado di approssimare al meglio segnali discontinui con transizioni improvvise.
La funzione madre delle Haar wavelet, $\psi(t)$, può essere definita come segue:

$$\psi(t) = \begin{cases} 1 & 0 \leq t < \frac{1}{2} \\ -1 & \frac{1}{2} \leq t < 1 \\ 0 & altrimenti \end{cases}$$



Mentre la funzione di scala $\phi(t)$ è definita in questo modo:

$$\phi(t) = \begin{cases} 1 & 0 \leq t < 1 \\ 0 & altrimenti \end{cases}$$

A partire da queste due funzioni è quindi possibile costruire tutta la famiglia di Haar wavelet.
Per l'applicazione della teoria wavelet alle immagini, invece, è necessario passare a funzioni bidimensionali che, in maniera analoga, consentono di sviluppare la trasformata wavelet su funzioni definite in due dimensioni.

3.3 Trasformata Ranklet

In statistica, le tecniche *non parametriche* sono una classe di modelli matematici che non necessitano di ipotesi a priori sulle caratteristiche della popolazione, aggirando il problema del fare assunzioni sulla distribuzione del campione di dati trattati. Tradizionalmente sono usate in relazione a metodi statistici basati sul ranking[18].
Per ranking si intende una relazione d'ordine tra gli elementi di un insieme, che assegna ad ognuno degli elementi confrontati un valore numerale indicante l'ordine o il posizionamento.
Le *rank based features* sono tecniche di grande utilità che impiegano la ranking statistic, frequentemente usate nell'analisi di immagini per risolvere problemi di corrispondenza tra immagini e localizzazione di regioni comuni.
Tra i principali vantaggi che presentano, infatti, ci sono la robustezza nel rilevare valori anormali e l'invarianza nei confronti di trasformazioni monotone, come cambiamenti di luminosità e contrasto o gamma correction.

Le ranklet possono essere definite come una famiglia di rank feature modellate sulla struttura delle wavelet: l'approccio piramidale consente di attuare i principi della multirisoluzione, mentre le caratteristiche di selettività all'orientamento dipendono dal meccanismo di suddivisione dei valori usato all'interno di operatori statistici non parametrici.
Sono quindi una famiglia di trasformate basate sul ranking, non parame-

triche, multiscala e selettive all'orientamento, caratterizzate da una forte analogia con le wavelet bidimensionali di Haar[9].
La selettività all'orientamento ereditata dalle wavelet, unita alla robustezza delle tecniche basate su ranking, fanno delle ranklet lo strumento ideale per caratterizzare pattern estesi con caratteristiche geometriche complesse, come ad esempio immagini di volti. Le ranklet sono state infatti impiegate con successo nell'ambito del riconoscimento facciale[19].

Dal punto di vista statistico, calcolare il ranking di un insieme di N osservazioni, coincide con l'effettuare una permutazione π degli interi da 1 a N che esprime l'ordine relativo delle osservazioni.
Traducendo questo concetto nell'analisi di un'immagine I in scala di grigi (in bianco e nero), indicheremo con $\pi^W(x, y)$ il rank delle intensità dei pixel di I posizionati in una finestra W, centrata nel pixel (x, y).
La trasformata Ranklet sfrutta in maniera diretta la permutazione π, assegnando ad ogni pixel (i, j) il valore del suo rank: $\rho(i, j) = \pi^W(i, j)$. Questo valore corrisponde al numero di pixel all'interno di W che hanno intensità inferiore a $I(i, j)$.

Dopo aver definito formalmente la trasformata Ranklet, bisogna stabilire come vanno interpretati i valori generati, da un punto di vista statistico. Questo può essere effettuato sfruttando il test di Wilcoxon per la somma dei rank (Wilcoxon rank-sum test), relativo al confronto di due campioni di dati indipendenti[18]. Si supponga di disporre di un campione di N valori, divisi in due gruppi di osservazione secondo la terminologia standard: n valori fanno parte del gruppo di trattamento (*treatment set*), m compongono quello di controllo (*control set*). L'obiettivo è stabilire se i valori del gruppo di trattamento sono significativamente più alti di quelli del gruppo di controllo. Per questo scopo definiamo la statistica di Wilcoxon $\mathcal{W}_s$ come la somma dei rank dei valori che fanno parte del tratment set.

$$\mathcal{W}_s = \sum_{i=1}^{n} \rho(i)$$

I valori nel campione di trattamento saranno valutati maggiori di quelli di controllo se $\mathcal{W}_s > \tau$, dove il valore τ determina il livello di confidenza del test.

Per applicare questi concetti all'analisi di immagini con trasformata ranklet, si consideri una finestra W contenente N pixel e centrata in (x, y); una strategia conveniente per la costruzione dei due campioni può essere realizzata dividendo gli N pixel di W in due gruppi di dimensioni $n = m = \frac{N}{2}$.

La suddivisione dei valori introduce un notevole grado di libertà, relativa alla disposizione geometrica dei due gruppi all'interno di W. Per ognuna delle $\binom{N}{n}$ possibilità di scelta dei treatment pixel, $\mathcal{W}_s$ produrrà una diversa caratterizzazione dell'intorno considerato.

Questa interessante possibilità di scelta può essere sfruttata per realizzare le proprietà di selettività all'orientamento caratteristiche delle wavelet di Haar[21] $h_j(x)$ (per $j = 1, 2, 3$), rappresentate graficamente in figura 3.1.

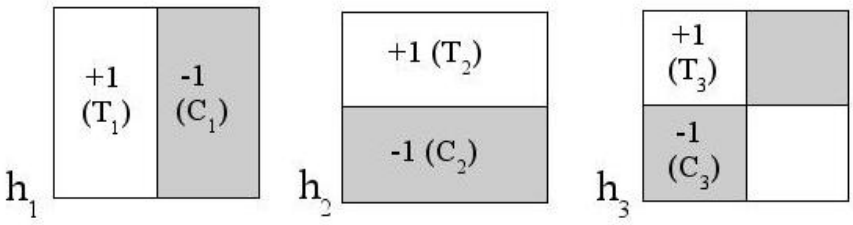


Figura 3.1: Le tre wavelet di Haar.

Il test di Wilcoxon consente quindi di stabilire se i pixel posizionati nel gruppo di trattamento sono sensibilmente più alti di quelli del gruppo di controllo. La scelta di dividere i valori di W secondo le wavelet di Haar, consente quindi di stabilire se i pixel sono disposti secondo una geometria verticale (h_1), orizzontale (h_2) o diagonale (h_3).

Possiamo poi sostituire $\mathcal{W}_s^j$ con l'equivalente statistica di Mann-Whitney $\mathcal{W}_{XY}^j$, definita come

$$\mathcal{W}_{XY}^j = \mathcal{W}_s^j - \frac{n(n+1)}{2}$$

Questa conversione consente una immediata interpretazione in termini di confronti tra pixel; infatti $\mathcal{W}_{XY}^{j}$ è equivalente al numero di coppie (x_C, x_T), con $x_C \in C_j$ e $x_T \in T_j$, per le quali vale $I(x_C) < I(x_T)$. Può quindi assumere valori compresi tra 0 e il numero di possibili coppie della forma $(x_C, x_T) \in C_j$ x T_j, che è $mn = \frac{N^2}{4}$.
Possiamo quindi definire le ranklet sfruttando la statistica di Mann-Whitney e convertendo i valori nel range $[-1, 1]$:

$$R_j = \frac{\mathcal{W}_{XY}^{j}}{mn/2} - 1$$

L'interpretazione geometrica di R_j è lineare con quanto precedentemente riportato riguardo alla statistica di Mann-Whitney e alle wavelet di Haar. Per esempio, si supponga che la finestra W si trovi esattamente sopra un margine verticale, come mostrato in figura 3.2.

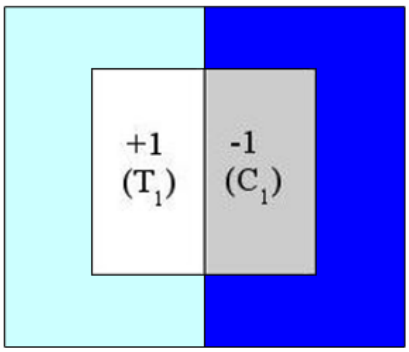


Figura 3.2: Trasformata ranklet in presenza di un margine verticale.

In questo caso i pixel del gruppo di trattamento saranno più luminosi dei pixel di controllo e presenteranno quindi valori di intensità più alti. Coerentemente con questa osservazione intuitiva, $\mathcal{W}_{XY}^{j}$ sarà molto vicino al numero di coppie C_j x T_j, cioè $\frac{N^2}{4}$; R_1 assumerà quindi un valore prossimo a $+1$.
Al contrario, nel caso in cui i due lati dell'immagine 3.2 fossero scambiati, R_1 assumerebbe un valore vicino a -1; se invece l'immagine non presentasse alcun tipo di margine verticale, il risultato sarebbe un valore intermedio,

prossimo allo 0.
In maniera analoga a quanto avviene per R_1, le ranklet R_2 e R_3 sono in grado di localizzare bordi orizzontali e diagonali.

La natura multiscala delle wavelet di Haar si estende anche alle ranklet. Per ogni traslazione x_0 e scalatura s, possiamo definire quali pixel compongono treatment e control set.

$$T_{j\ ;\ (x_0,s)} = \{x \mid h_j((x - x_0)/s) = +1\}$$

$$C_{j\ ;\ (x_0,s)} = \{x \mid h_j((x - x_0)/s) = -1\}$$

Applicando la definizione di ranklet, è quindi possibile calcolare il valore di $R_j(x_0, s)$ sull'intorno $W_{(x_0,s)}$; questo valore rappresenta il risultato della trasformata calcolato nella posizione definita da x_0, al livello di dettaglio indicato da s.
Il fattore di traslazione x_0 indica quindi il posizionamento della finestra W sull'immagine I, mentre il fattore di scala s definisce la grandezza della finestra, che a sua volta stabilisce la dimensione dei pattern geometrici da individuare nell'immagine.
Solitamente risulta interessante effettuare l'elaborazione di un livello completo di ranklet, che consiste nel calcolare la trasformata per ogni possibile posizione x_0, mantenendo costante il fattore di scalatura s; questo procedimento consente di analizzare l'intera immagine ad un preciso livello di risoluzione. La natura multiscala delle ranklet è quindi strettamente collegata al valore s: è possibile effettuare un'analisi a diversi livelli di risoluzione modificando adeguatamente il fattore di scalatura.
Questa metodologia porta alla realizzazione di diversi livelli di ranklet, che possono essere rappresentati graficamente in immagini che mostrano informazioni sui particolari geometrici rilevati dalla trasformata a diversi livelli di dettaglio.

3.4 Esempio: applicazione Ranklet

Dopo aver definito formalmente la trasformata ranklet, si è ritenuto opportuno mostrare visivamente quali risultati si ottengono applicandola ad un'immagine.

Figura 3.3: Esempio: immagine di partenza.

L'immagine di partenza, mostrata in figura 3.3, è stata elaborata a diversi livelli di risoluzione, secondo la natura piramidale tipica delle wavelet.
Per ogni livello si è calcolato il valore della trasformata ranklet in ogni possibile posizione dell'immagine, per ognuna delle tre direzioni definite precedentemente. Per analizzare gli effetti dell'applicazione delle ranklet ad un'immagine, è interessante visualizzare i valori calcolati sotto forma di nuove immagini.
La figura 3.4 mostra i risultati ottenuti elaborando l'immagine di partenza con una maschera di convoluzione di dimensione 4x4. È interessante notare come i particolari messi in evidenza siano di grana molto fine: i contorni delle figure principali possono essere localizzati in maniera abbastanza chiara.

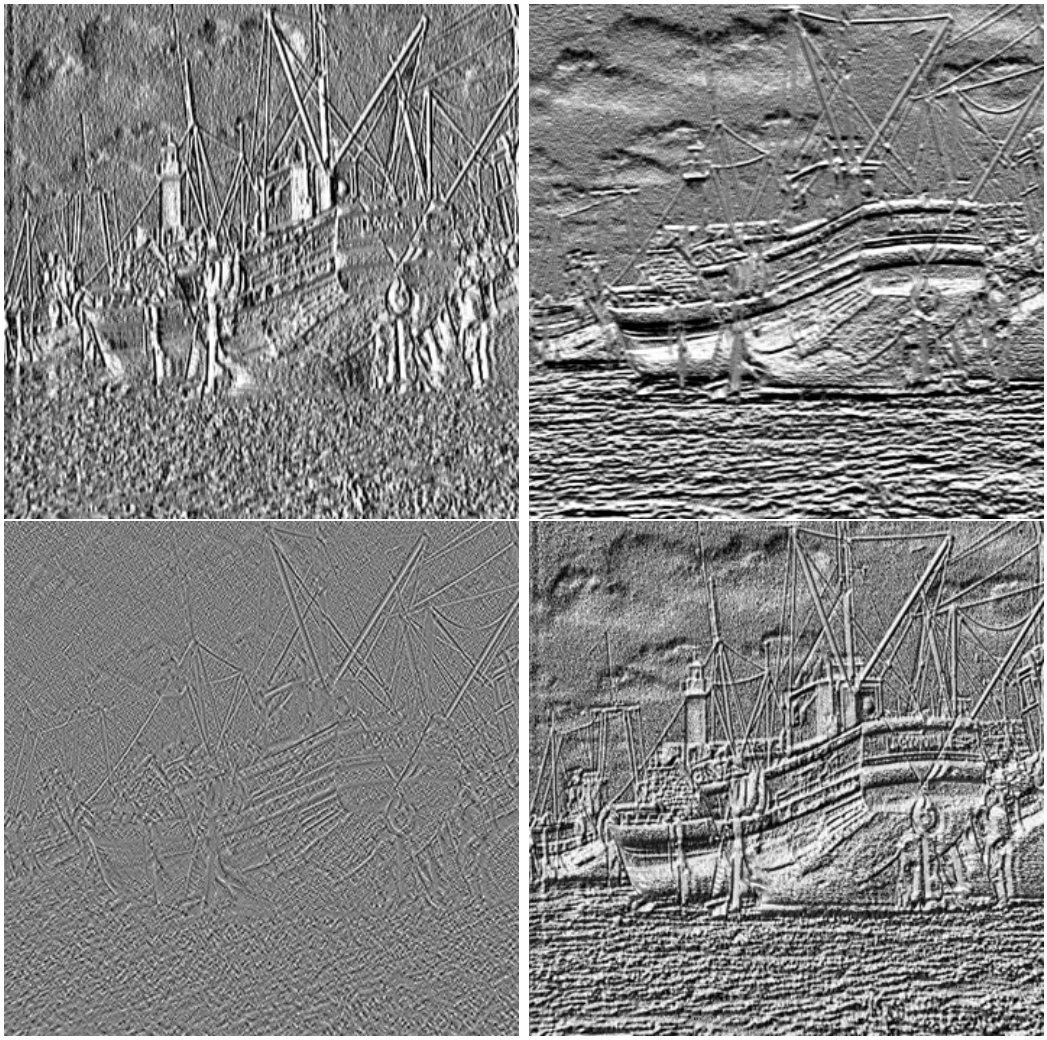


Figura 3.4: Esempio: elaborazione con maschera 4x4.

Nella prima riga sono rappresentate le componenti verticale e orizzontale della trasformata, mentre nella seconda è presentata, sulla destra, quella diagonale. La quarta immagine, in basso a sinistra, è realizzata sommando pixel per pixel il valore delle altre tre immagini e rappresenta quindi tutti i particolari messi in evidenza a questo livello di risoluzione.
È interessante notare come ogni immagine contenga particolari diversi dalle altre e sia costituita quasi esclusivamente da linee relative alla propria direzione di riferimento.

La proprietà di selettività all'orientamento, messa in luce nella definizione della trasformata, trova quindi riscontro nelle immagini che è possibile realizzare a partire dai valori calcolati.
Un altro aspetto che è interessante analizzare riguarda il livello di risoluzione: elaborando l'immagine originale con una maschera di dimensioni maggiori, i particolari messi in evidenza saranno meno definiti.

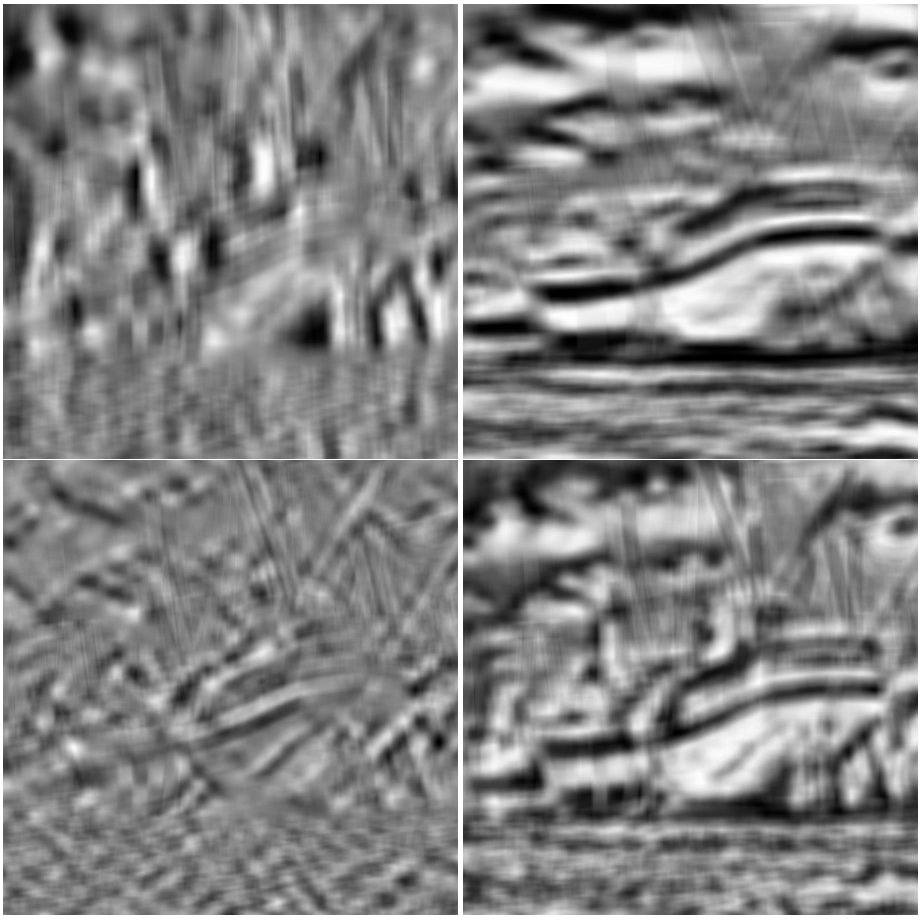


Figura 3.5: Esempio: elaborazione con maschera 32x32.

Le immagini mostrate in figura 3.5, disposte analogamente al caso precedente, rappresentano il risultato dell'elaborazione con una maschera di dimensioni 32x32. Si può notare come le geometrie riscontrate nelle diverse componenti

forniscano, anche in questo caso, informazioni direzionali relative alla trasformata. D'altra parte, è evidente che i particolari rilevati sono di grana molto meno fine: la struttura principale dell'immagine è ancora distinguibile, ma i dettagli più sottili non possono essere rilevati a questo livello di risoluzione. Ripetendo questo procedimento con maschere di dimensioni diverse è quindi possibile estrarre particolari diversi dall'immagine, mantenendo sempre una forte selettività all'orientamento.

3.5 Complessità

La procedura tradizionale per effettuare l'analisi in multirisoluzione consiste nel calcolo della piramide completa, che può essere realizzata raddoppiando progressivamente il fattore s, a partire dal valore 1, fino ad arrivare a $2^{log(N)\ 2}$.
In questo modo le dimensioni della finestra W vengono dimezzate ad ogni iterazione, il che consente di calcolare la trasformata su un insieme di valori sempre più ridotto, mettendo in evidenza particolari a risoluzione sempre più alta, come è stato mostrato nell'esempio precedente.
Supponendo che l'immagine di partenza I sia di forma quadrata e lato $N = 2^{LN}$, al j-esimo livello di scalatura il fattore s avrà valore 2^j e la finestra W, anch'essa considerata quadrata per comodità, avrà lato $\frac{N}{2^j} = 2^{LN-j}$.
Ad ogni livello di risoluzione, quindi, la trasformata ranklet dovrà essere calcolata per tutti i valori di x_0 che consentono il posizionamento della maschera interamente all'interno dell'immagine I. Al livello j questo corrisponderà a calcolare la trasformata in $(N - 2^{LN-j} + 1)^2$ posizioni.
L'elaborazione della piramide completa richiederà quindi il calcolo della trasfor-

[2]Nell'espressione $2^{log(N)}$ si fa riferimento al logaritmo in base 2, ovvero $log_2(N)$, ma la base è stata omessa per semplificare l'espressione. Da questo punto in poi, ogni volta che verrà utilizzato il logaritmo, si intenderà il log_2, senza specificare la base.

mata ranklet in un numero complessivo di posizioni pari a

$$\sum_{j=0}^{LN}(N - 2^j + 1)^2$$

Stabilita la procedura di analisi e il numero complessivo di valori da calcolare, diventa importante valutare la complessità dell'algoritmo impiegato per l'applicazione della trasformata. La soluzione più semplice segue in maniera lineare il procedimento descritto nella sezione precedente; infatti è possibile realizzare il calcolo in una singola posizione x_0 e ad un unico livello di scalatura s, in 4 passi relativamente semplici:

1. ordinamento degli n valori che si trovano all'interno della finestra $W_{(x_0,s)}$;
2. assegnazione di un valore numerale indicante il rank, ad ognuno degli n elementi;
3. sommatoria dei valori di rank relativi ai pixel che si trovano nel treatment set di W;
4. calcolo della trasformata tramite sottrazione e moltiplicazione di costanti al valore ottenuto dalla sommatoria.

Ognuna di queste operazioni può essere svolta in maniera efficiente. In particolare, l'ordinamento di n valori con un classico algoritmo di sorting, come quicksort, richiede $O(n \cdot log(n))$ passi; l'assegnazione dei rank ha complessità $O(n)$, poiché richiede una scansione completa del vettore ordinato di valori; il calcolo della sommatoria invece, richiede l'elaborazione di metà dei valori che compongono W, quindi ha complessità $O(n/2)$; l'ultima operazione, infine, consiste nell'esecuzione di un numero prestabilito di sottrazioni e moltiplicazioni, che può essere svolto in tempo costante $O(1)$. Globalmente, quindi, queste operazioni hanno complessità computazionale

$$O(n \cdot log(n) \ + \ n \ + \ n/2 \ + \ 1) \ = \ O(n \cdot log(n))$$

pari alla complessità dell'operazione più costosa delle tre: l'ordinamento.

La soluzione appena presentata è molto lineare, ma non è la più valida: è possibile migliorare la strategia di accesso all'immagine e il metodo con cui realizzare l'ordinamento. Alcuni algoritmi più efficienti sono stati proposti da F. Smeraldi nel 2009 [20].
Di seguito verrà presentato l'algoritmo *Incremental Distribution Counting* (IDC), che prevede un meccanismo di accesso efficiente nell'analisi completa di un'immagine, oltre a meccanismi di ordinamento più funzionali.
La prima caratteristica rilevante di questo algoritmo riguarda la scansione della finestra di supporto in posizioni adiacenti dell'immagine; in questi casi, infatti, le superfici occupate dalle due maschere sono quasi totalmente sovrapposte: la strategia di accesso efficiente consiste quindi nell'evitare la scansione completa della finestra, riutilizzando i valori già elaborati precedentemente.

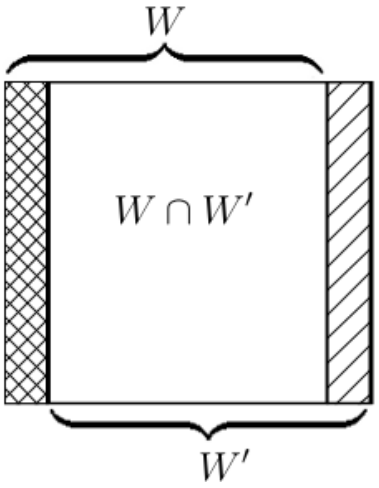


Figura 3.6: Sovrapposizione di due finestre adiacenti.

La figura 3.6 delinea una situazione di questo tipo: le finestre W e W' differiscono solo per una colonna di valori ed è possibile passare dall'una all'altra eliminando una colonna di valori ed aggiungendone una nuova.
Applicando questa strategia all'analisi di un livello di ranklet, è possibile ridurre in maniera determinante il tempo impiegato nell'estrazione dei valori dall'immagine. Infatti, disponendo dei valori contenuti in una finestra W di area n, è possibile elaborare i valori di una finestra W', adiacente a W, con

$O(\sqrt{n})$ operazioni.
Per sfruttare al meglio le caratteristiche dell'approccio di calcolo incrementale appena mostrato, l'IDC utilizza un algoritmo di ordinamento diverso dai tradizionalli meccanismi basati su confronto dei valori (quicksort, mergesort, ecc ...), chiamato counting sort. Sfruttando il fatto che i valori da ordinare sono numeri interi che si trovano all'interno di un range noto, l'algoritmo si occupa di contare il numero di occorrenze di ciascun valore, memorizzando queste informazioni in una struttura dati del tutto simile ad un istogramma. Il counting sort ha quindi complessità $O(N)$ e presenta interessanti caratteristiche computazionali anche per quanto riguarda le operazioni di inserimento ed eliminazione, che possono essere effettuate in tempo costante $O(1)$, caratteristica che risulta determinante per l'efficienza dell'IDC.
In conclusione, disponendo dell'istogramma H_W, relativo a W, è possibile realizzare l'istogramma $H_{W'}$ di una finestra W' adiacente a W in tempo $O(\sqrt{n})$. L'assegnazione dei rank, invece, è realizzata scorrendo tutti i valori dell'istogramma, ed è quindi effettuata con complessità $O(G)$, dove G è il numero di livelli di grigio nell'immagine.
In conclusione, l'algoritmo Incremental Distribution Counting consente di calcolare un intero livello di ranklet, con filtro di area K, con complessità pari a

$$O(\ (N - \sqrt{K} + 1)^2 \cdot (\sqrt{K} + G)\ )$$

mentre per l'algoritmo base, tale valore risulta

$$O(\ (N - \sqrt{K} + 1)^2 \cdot (K \cdot log(K))\ )$$

Il guadagno computazionale che si ottiene impiegando l'algoritmo IDC è abbastanza evidente: il valore G è praticamente una costante e $\sqrt{K}$ è molto inferiore a $K \cdot log(K)$.

Capitolo 4

Implementazione Ranklet

Dopo aver definito la trasformata ranklet e aver mostrato le caratteristiche computazionali dei principali algoritmi, in questo capitolo verrà affrontato l'aspetto implementativo.
Si è scelto di realizzare due versioni funzionanti del software, ottimizzate per l'esecuzione in due ambienti diversi: la prima è stata progettata per l'esecuzione su piattaforme multicore e multiprocessore, mentre la seconda è stata realizzata per il contesto del GPU computing.
In entrambi i casi si è deciso di implementre l'algoritmo Incremental Distribution Counting presentato nel capitolo precedente, che garantisce una notevole riduzione di complessità rispetto alla soluzione base.

4.1 Struttura condivisa

Le due soluzioni condividono la struttura principale del programma: il dato di input è costituito da un'immagine reale a K bit, di dimensioni W e H, che viene caricata e convertita in una matrice di interi, che sarà analizzata con la trasformata ranklet a diversi livelli di risoluzione. I valori calcolati vengono memorizzati in un array diviso in 3 parti, una per ogni direzione della trasformata (verticale, orizzontale e diagonale) e poi memorizzati sotto

forma di immagini o in un file di testo.
La gestione della memoria è un aspetto particolarmente delicato nell'implementazione della trasformata ranklet, poiché il numero di valori calcolati e lo spazio necessario per rappresentarli, sono in relazione esponenziale con la dimensione dell'immagine (figura 4.1).

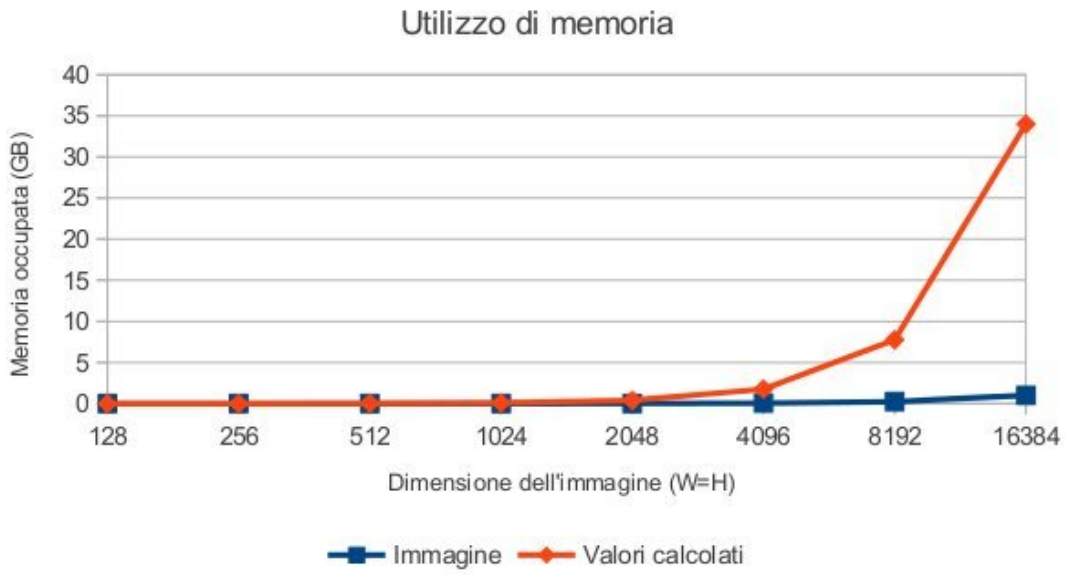


Figura 4.1: Spazio di memoria occupato nel calcolo della trasformata ranklet.

Si può notare come, nel caso di immagini 16384x16384, siano necessari 35 GB per rappresentare tutti i valori calcolati, il che comporta evidenti problemi di organizzazione della memoria. Per ovviare a questo problema si è deciso di salvare su disco i valori calcolati dopo l'elaborazione di ogni livello di piramide, in modo da occupare solo lo spazio strettamente necessario durante il calcolo. Questa soluzione consente di limitare la memoria occupata in maniera determinante, riducendo lo spazio necessario per elaborare immagini di lato 16384 a soli 3 GB.

Le parti di caricamento dei dati e memorizzazione dei risultati sono comuni a tutte le soluzioni realizzate, che differiscono esclusivamente per la proce-

dura di calcolo. Le due versioni dell'algoritmo sono quindi strutturate nello stesso modo e sono relizzate secondo una serie di loop annidati, presentati di seguito dal più esterno al più interno.

- **Loop A**: livelli di piramide. Le ranklet sono funzioni multiscala e devono quindi essere calcolate a diversi livelli di risoluzione; ad ogni iterazione di questo ciclo le dimensioni della maschera di convoluzione vengono dimezzate, aumentando così il livello di scalatura e la risoluzione dei particolari estratti con la trasformata ranklet. Inizialmente le dimensioni della maschera sono $w_0 = 2^{log_2(W)}$, $h_0 = 2^{log_2(H)}$, cioè W e H, se le dimensioni dell'immagine sono potenze di 2.

- **Loop B**: convoluzione. Ad ogni livello j una maschera di area $w_j * h_j$ viene fatta scorrere sull'immagine (sliding window) in modo da occupare solo le posizioni per le quali è interamente contenuta all'interno dell'immagine.
  In ognuna di queste posizioni viene calcolato il valore della ranklet nelle 3 direzioni della trasformata.

- **Loop C**: scansione finestra. Per ogni posizione della maschera, ad ogni livello di risoluzione, deve essere calcolato il valore della trasformata ranklet. Per compiere questa operazione è necessario elaborare interamente la maschera di convoluzione, in modo da ottenere una copia locale dei valori della matrice in ogni posizione.
  I valori dovranno poi essere ordinati ed elaborati per il calcolo della trasformata, secondo quanto visto precedentemente.

Questi tre loop annidati rappresentano la logica d'esecuzione globale e le fasi di esecuzione dell'algoritmo, che è stato poi realizzato e ottimizzato in due versioni concettualmente equivalenti ma specializzate nell'esecuzione su piattaforme diverse.

4.2 Implementazione OpenMP

La prima versione sviluppata è stata progettata per essere utilizzata in ambienti multicore o multiprocessore, dotati di molteplici unità d'esecuzione. Per sfruttare il parallelismo fornito dalle piattaforme in maniera trasparente e funzionale, si è scelto di utilizzare l'interfaccia di programmazione OpenMP[22], che fornisce un insieme di librerie, variabili d'ambiente e direttive per il compilatore, utili per specificare il parallelismo in programmi C/C++ e Fortran.
Le motivazioni che hanno portato a questa scelta riguardano principalmente la diffusione di questa soluzione e le sue caratteristiche di leggerezza ed efficienza che la rendono molto valida per la realizzazione di codice parallelo.

La prima importante decisione implementativa è stata effettuata nello scegliere il modo in cui sviluppare il parallelismo all'interno del programma. Infatti, dovendo implementare un procedimento profondamente iterativo, organizzato secondo diversi cicli innestati, le possibilità di parallelizzazione erano molteplici.
Si è scelto di applicare il parallelismo a livello del loop B, relativamente allo spostamento della maschera di convoluzione sulla superficie dell'immagine, tenendo in considerazione due aspetti ritenuti fondamentali per la realizzazione di codice efficiente:

- per utilizzare al meglio il parallelismo messo a disposizione da diverse architetture, è necessario sviluppare una soluzione che bilanci in maniera equilibrata il carico di lavoro tra i diversi thread d'esecuzione e che scali correttamente all'aumentare del numero di core;

- l'impiego dell'Incremental Distribution Counting garantisce un notevole miglioramento computazionale ed è importante impiegare questo algoritmo per sviluppare codice efficiente.

Nell'implementare la trasformata ranklet, si è quindi deciso di sviluppare l'iterazione sui diversi piani di piramide (loop A) in maniera sequenziale; in questo modo ogni livello di risoluzione è affrontato singolarmente e indipendentemente dagli altri.
Per ogni passo del loop di piramide, è quindi necessario analizzare l'intera immagine con una maschera di convoluzione di dimensioni prefissate; la parallelizzazione è stata applicata in questo punto, suddividendo la superficie dell'immagine in settori di uguale dimensione, ognuno dei quali viene assegnato ad un thread d'esecuzione.
Questo meccanismo di divisione del lavoro consente un buon bilanciamento del carico computazionale, che è distribuito equamente tra le diverse istanze d'esecuzione, che applicano lo stesso insieme di operazioni a dati diversi, in maniera autonoma e indipendente. Ogni thread implementa quindi una versione dell'algoritmo IDC, la cui esecuzione è localizzata nel proprio settore di competenza dell'immagine. Si è scelto di suddividere l'immagine in settori verticali, come mostrato in figura 4.2.

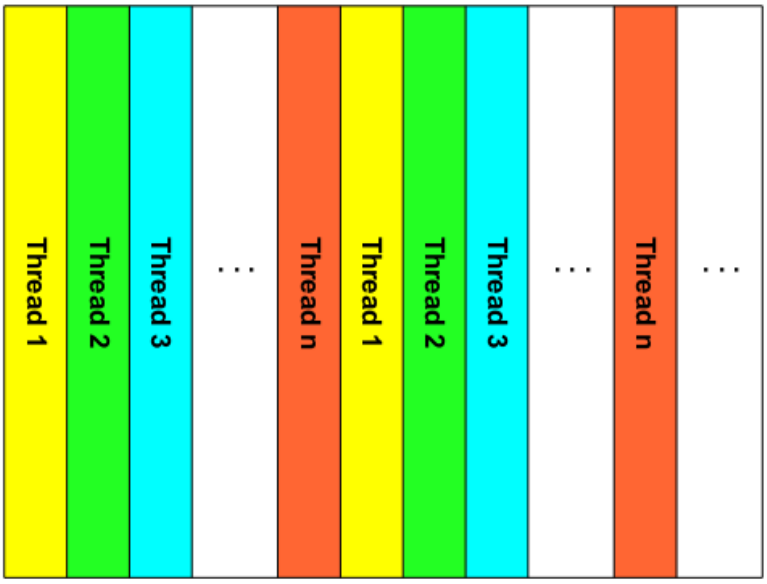


Figura 4.2: Divisione dell'immagine in settori verticali.

Al j-esimo livello di piramide quindi, analizzando un'immagine di lato N con una maschera di lato W_j, ad ognuno degli n thread d'esecuzione vengono assegnati k settori rettangolari di dimensioni W_j e $N - W_j$, dove $k \simeq \frac{N-W_j}{n}$. Il meccanismo di assegnamento dei settori ai diversi thread è realizzato in maniera implicita, utilizzando le direttive OpenMP. L'unica possiilità di intervento, da parte del programmatore, riguardo alla suddivisione del lavoro tra i thread, riguarda la scelta tra scheduling statico e dinamico.
Con la prima soluzione, il carico di lavoro è suddiviso tra i thread a tempo di compilazione, mentre con la seconda viene assegnato a run-time in base all'effettivo stato d'esecuzione; si è scelto di impiegare la seconda strategia di scheduling, perché consente un approccio maggiormente legato all'efficienza d'esecuzione, garantendo un miglior bilanciamento e una distribuzione più equilibrata del carico di lavoro.

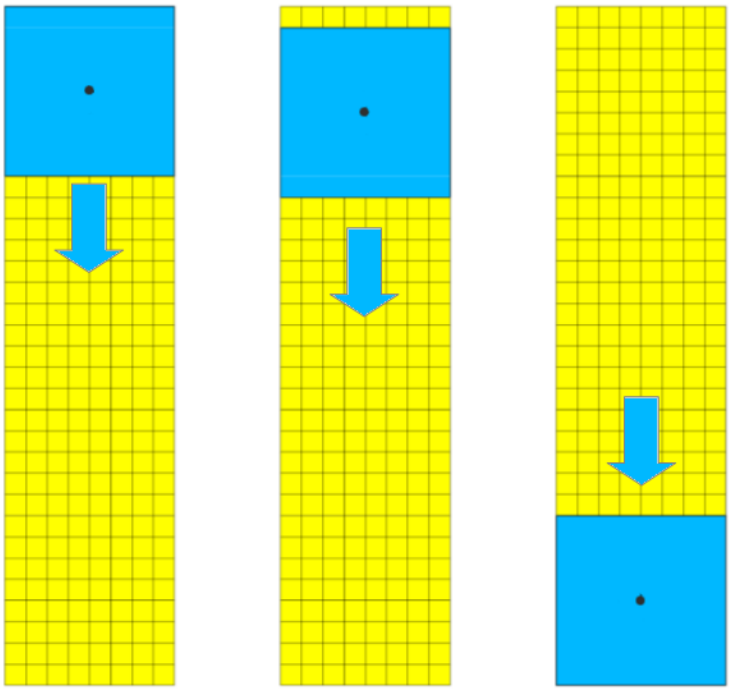


Figura 4.3: Movimento della maschera di convoluzione in un settore verticale.

L'elaborazione di un singolo settore comporta il movimento della maschera in direzione verticale, dall'alto verso il basso (figura 4.3) e realizza il calcolo di una colonna di ranklet.
Si è scelto di implementare la divisione in settori e il movimento della finestra in questo modo per realizzare una strategia di accessi efficienti in memoria. Ad ogni passo del processo di sliding window, infatti, è necessario eliminare alcuni valori dalla maschera e aggiungerne di nuovi, operazioni che comportano diversi accessi in memoria per la lettura della matrice di input.
Il linguaggio di programmazione C/C++, impiegato nella realizzazione di questo progetto, memorizza matrici e array per righe. Come conseguenza, la miglior tecnica di accesso a strutture dati di questo tipo è quella di utilizzare elementi consecutivi in ordine di posizione: in questo modo i valori sono elaborati uno dopo l'altro, in ordine di memorizzazione, senza che vengano effettuati salti tra diverse regioni di memoria.
La strategia più efficace per ottimizzare i movimenti della maschera sull'immagine è implementata tramite l'esecuzione di spostamenti verticali, che fanno in modo che i valori da aggiungere o eliminare dalla maschera siano disposti sulla stessa riga, memorizzata in maniera efficiente.
Lo scivolamento lungo le righe invece, comporta l'aggiunta e l'eliminazione di valori disposti in colonna e memorizzati in locazioni non consecutive: effettuare la convoluzione in questo modo avrebbe comportato un notevole rallentamento nell'operazione di costruzione della finestra.

Dopo aver definito come viene svolta la distribuzione del lavoro tra i thread, verrà analizzata nel dettaglio l'implementazione dell'algoritmo IDC per il calcolo di una colonna di ranklet.
Per ogni posizione della maschera, è necessario calcolare il valore delle ranklet nelle tre direzioni verticale, orizzontale e diagonale. Come descritto nella presentazione dell'algoritmo IDC, la scansione completa della maschera è effettuata solo per la posizione iniziale della colonna, mentre nei passi successivi vengono utilizzati i valori già calcolati al passaggio precedente, aggiungendo la nuova riga.

Per semplificare l'operazione di costruzione del gruppo di trattamento (T-set), i valori che compongono la finestra vengono divisi in 4 quadranti, come mostrato in figura 4.4.

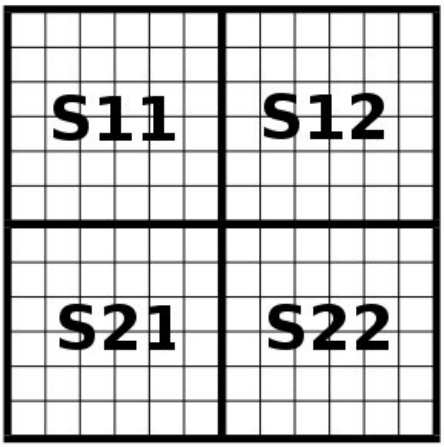


Figura 4.4: Divisione della maschera in quadranti.

Per tutta la durata del calcolo, ogni thread calcola e aggiorna cinque istogrammi, modificando i valori ad ogni movimento della finestra; il primo di questi, *hist*, conta le occorrenze dei valori nell'intera maschera, mentre ognuno degli altri fa riferimento ad un quadrante: $hist_{S11}$ per i valori interni al quadrante $S11$, $hist_{S12}$ per il quadrante $S12$ e così via.
Questa suddivisione comporta un overhead di calcolo aggiuntivo, perché richiede il mantenimento di cinque istogrammi, invece che uno solo, ma ha un grande vantaggio: consente di ricostruire in maniera immediata i T-set nelle tre direzioni di calcolo, senza la necessità di scandire nuovamente la finestra di convoluzione. Infatti ognuno dei T-set corrisponde logicamente alla somma di due quadranti:

$$T_0 = S11 + S21 \qquad \text{(verticale)}$$
$$T_1 = S11 + S12 \qquad \text{(orizzontale)}$$
$$T_2 = S11 + S22 \qquad \text{(diagonale)}$$

Dopo aver effettuato la scansione della finestra e aver aggiornato gli isto-

grammi, è necessario calcolare i rank. Ad ogni valore che compare nella maschera deve essere assegnato un numero razionale che ne rappresenta la posizione ordinata, con la proprietà che valori equivalenti devono avere lo stesso rank. Quindi, se un valore *val* compare una sola volta nell'insieme, il suo rank corrisponde alla sua posizione nel vettore ordinato; se invece ci sono più occorrenze di *val*, il rank è calcolato come la media delle posizioni che le occorrenze occupano nel vettore ordinato.

Per esempio, si supponga di dover calcolare i rank del seguente insieme di valori:

$$[\,4,\ 21,\ 9,\ 12,\ 2,\ 16,\ 12,\ 17,\ 21,\ 4,\ 18,\ 9,\ 9,\ 12,\ 27]$$

Per prima cosa è necessario ordinare i valori in maniera crescente:

$$[\,2,\ 4,\ 4,\ 9,\ 9,\ 9,\ 12,\ 12,\ 12,\ 16,\ 17,\ 18,\ 21,\ 21,\ 27]$$

A questo punto è possibile assegnare i rank agli elementi ordinati:

$$[\,1,\ 2.5,\ 2.5,\ 5,\ 5,\ 5,\ 8,\ 8,\ 8,\ 10,\ 11,\ 12,\ 13.5,\ 13.5,\ 15]$$

Si noti come ogni rank contenga informazioni riguardanti la posizione ordinata del relativo elemento e come rank uguali corrispondano a valori uguali. Nell'esempio appena mostrato compaiono 2 occorrenze dell'elemento 4 che, nel vettore ordinato, si trovano nelle posizioni 2 e 3; il rank assegnato a tutti gli elementi con valore 4 corrisponde quindi alla media delle posizioni che occupano nel vettore ordinato, cioè 2.5.
Analogamente, è possibile individuare 3 occorrenze del valore 9, in posizione 4, 5 e 6: all'elemento 9 sarà infatti assegnato rank 5.

Nel calcolo della trasformata ranklet, l'assegnazione dei rank coinvolge tutti i pixel della maschera e deve essere realizzata scandendo l'intero istogramma *hist*, dal minimo al massimo valore della scala di grigi. Per ogni posizione di *hist* viene calcolato il rank, come mostrato nell'esempio precedente.
Per calcolare il rank di elementi di pari valore è stata utilizzata una formula

che sfrutta la consecutività delle posizioni da mediare:

$$media(val_0, val_1 \; .. \; val_n) = \frac{val_0 \; + \; val_n}{2}$$

Questa operazione può essere ulteriormente ottimizzata se si considera che i rank sono numeri razionali ricavati della divisione di un numero intero per un fattore 2. La memorizzazione di questi valori come numeri floating point comporta una gestione più complessa e costosa, dal punto di vista computazionale, che può essere ottimizzata se si evita di effettuare la divisione per 2; in questo modo, infatti, i rank possono essere utilizzati come numeri interi, ottenendo notevoli miglioramenti dal punto di vista delle prestazioni. La divisione non è eliminata, perché questo comporterebbe il calcolo di valori errati, ma è rimandata alla fase successiva, nella quale, in ogni caso, è necessario passare a valori floating point.

Purtroppo questa ottimizzazione non è sempre possibile: trattare i rank come numeri interi garantisce notevoli vantaggi computazionali, ma comporta l'utilizzo di una struttura di memorizzazione più limitata. Quando la dimensione della maschera è grande, nella fase di calcolo successiva, il calcolo dei rank come numeri interi non è più possibile, perché i valori trattati diventano troppo grandi per essere rappresentati correttamente e si incorre in problemi di overflow[1].

Per garantire la correttezza dei valori calcolati, quindi, quando le dimensioni della maschera di convoluzione superano una certa soglia, è necessario ritornare al calcolo in virgola mobile, nonostante risulti meno efficiente.

Il passo successivo consiste nel calcolo della statistica di Wilcoxon che, come spiegato nel capitolo precedente, ha un significato numerico rilevante e porta al calcolo del valore della trasformata ranklet nel pixel centrale della

[1]Un overflow aritmetico è un problema relativo alle operazioni sui numeri all'interno di un computer, legato alla dimensione (in bit) della struttura di memorizzazione. Problemi di questo tipo si presentano quando si richiede di memorizzare un numero più grande del massimo valore che è possibile rappresentare con quella struttura dati.

Ad esempio, con un intero unsigned a 8 bit è possibile rappresentare tutti i valori che vanno da 0 a $2^8 - 1$, cioè 255; un overflow si presenta quando si cerca di rappresentare un numero maggiore di 255 con questo tipo di dato.

maschera.
Per ogni direzione i, questa operazione viene effettuata sommando i rank dei valori interni a T_i; grazie alla divisione in quadranti può essere effettuata in maniera molto efficiente, precalcolando un valore che corrisponde alla somma dei rank degli elementi presenti nel quadrante. Nel generico quadrante (u, v), tale valore è

$$S_{u,v} = \sum_{val=0}^{G} rank(val) * hist_{Suv}$$

Dopo questa operazione è possibile calcolare la trasformata ranklet nelle 3 direzioni:

$$\begin{aligned} R_0 &= K_{mult} * (S_{1,1} + S_{2,1} - K_{sum}) \quad \text{(verticale)} \\ R_1 &= K_{mult} * (S_{1,1} + S_{1,2} - K_{sum}) \quad \text{(orizzontale)} \\ R_2 &= K_{mult} * (S_{1,1} + S_{2,2} - K_{sum}) \quad \text{(diagonale)} \end{aligned}$$

Dove K_{mult} e K_{sum} sono i due coefficienti moltiplicativo e additivo che consentono di elaborare il risultato del test di Wilcoxon, producendo un valore compreso nell'intervallo $[-1, 1]$.

In questa sezione è stata descritta la struttura del codice relativo al calcolo della trasformata rankelt in ambiente multicore e sono stati presentati i principali miglioramenti che sono stati realizzati con lo scopo di ridurre il tempo di calcolo.
In conclusione quindi, sono state applicate ottimizzazioni di diversa specificità ai loop che compongono il calcolo della trasformata:

- nel Loop A non sono state apportate ottimizzazioni, mantenendone l'esecuzione sequenziale;
- a livello del Loop B è stata applicata la parallelizzazione, suddividendo l'operazione di sliding window tra i diversi thread d'esecuzione;
- nel Loop C sono state implementate alcune modifiche alla procedura di calcolo, ottimizzando le strutture e le operazioni da effettuare.

Le altre soluzioni prese in considerazione non garantivano la possibilità di intervenire in maniera altrettanto radicale nell'attuazione di diverse ottimizzazioni e presentavano problemi nel trade-off tra bilanciamento di carico, scalabilità e implementazione efficiente dell'algoritmo Incremental Distribution Counting.

4.3 Implementazione CUDA

In seguito all'implementazione per l'ambiente multicore e dopo aver verificato correttezza ed efficienza dell'algoritmo sviluppato, è stata realizzata una versione dell'algoritmo per l'ambiente GPU.
Poiché la versione CUDA è stata realizzata a partire da quella OpenMP, la struttura principale di questa implementazione è analoga a quella appena presentata e molte delle ottimizzazione attuate su CPU sono state mantenute nella soluzione GPU. Di seguito verrà presentata l'implementazione del codice grafico, evidenziando le modifiche realizzate rispetto alla versione multicore.
Una delle differenze più significative tra i due ambienti di sviluppo riguarda la gestione della memoria. Sui processori tradizionali la memoria in uso è relativamente efficiente, può essere considerata di un solo tipo ed è automaticamente ottimizzata con meccanismi di caching, mentre nei processori grafici sono presenti diversi dispositivi, con prestazioni, scopi e tecniche di utilizzo molto differenti tra loro. Nelle applicazioni CUDA, i meccanismi di gestione della memoria rappresentano un aspetto molto delicato: spesso risulta più costoso l'accesso ai dati che le effettive operazioni di calcolo. Per questo motivo, in questa sezione verrà data grande importanza al tipo di memoria impiegata nell'implementazione delle diverse strutture utilizzate nel codice CUDA.
La figura 4.5 mostra una rappresentazione grafica della gerarchia di memoria fornita dalla piattaforma CUDA: sarà utile, nel resto del capitolo, per spie-

gare come sono state implementate le diverse parti del codice.

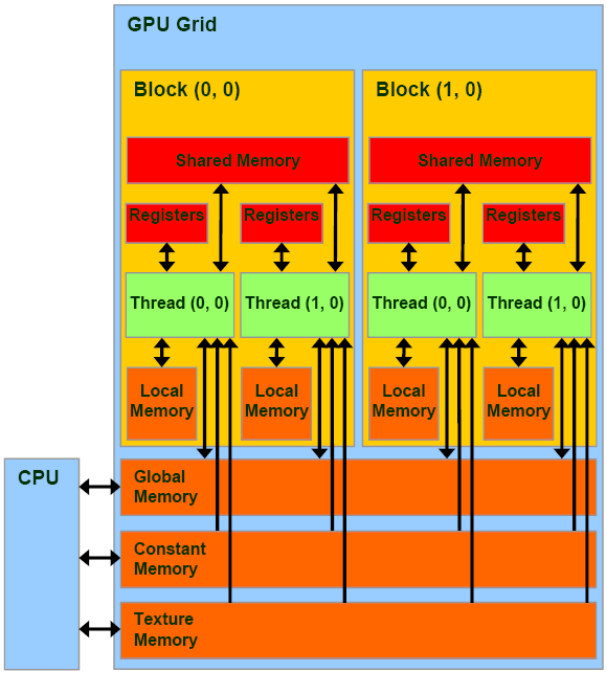


Figura 4.5: Modello della memoria CUDA.

La prima operazione da svolgere nell'implementazione per GPU è il caricamento della matrice di input dalla memoria dell'host (CPU) a quella del device (GPU). Questa operazione viene effettuata all'inizio dell'esecuzione e consiste in due passi, effettuati invocando due funzioni fornite dalla libreria CUDA:

- allocazione della memoria sul device, tramite la funzione `cudaMalloc`;
- copia della matrice dalla memoria host a quella del device, con la chiamata `cudaMemcpy`.

Queste operazioni non richiedono l'invocazione di un kernel e sono effettuate direttamente dal processore principale.
Si è scelto di memorizzare la matrice nell'area di memoria globale perché è l'unica regione che dispone di spazio sufficiente per contenere immagini

arbitrariamente grandi. Inoltre la memoria globale è accessibile in maniera diretta da tutti i core e gode della proprietà di persistenza, che consente di utilizzare le informazioni memorizzate anche tra esecuzioni di kernel diversi; è così possibile copiare la matrice nella memoria del device una sola volta.
Terminata la fase di inizializzazione, il Loop A viene eseguito in maniera sequenziale dal processore principale, in maniera analoga a quanto avveniva nel codice OpenMP. Come nel caso precedente, infatti, si è deciso di parallelizzare a livello del Loop B, dividendo la matrice in settori per consentire l'utilizzo dell'Incremental Distribution Counting all'interno del Loop C.
Si è deciso di sviluppare un modello di calcolo analogo a quello realizzato nell'implementazione OpenMP, dividendo la matrice di input in settori verticali. Questa soluzione implementativa ha il difetto di richiedere che ogni thread disponga di uno spazio di memoria privato ed efficiente, nel quale memorizzare gli istogrammi necessari per l'esecuzione dell'IDC.
Purtroppo la quantità di memoria ad alte prestazioni di cui dispongono i thread è in quantità molto ridotta ed è in forma di registri, quindi non è utilizzabile per la memorizzazione di strutture complesse, come gli istogrammi necessari per l'algoritmo IDC. La soluzione proposta da CUDA, in questi casi, consiste nell'impiego della memoria locale, che però ha il difetto di essere molto meno veloce rispetto ai registri.
Nessuno degli altri tipi di memoria rispetta i requisiti di spazio ed efficienza richiesti:

- la memoria globale ha latenza equivalente a quella locale, quindi non consente un miglioramento di prestazioni;
- la shared memory è efficiente e di dimensioni adeguate, ma è condivisa tra i thread dello stesso blocco, quindi viola le condizioni di località;
- le due regioni di texture e constant memory sono accessibili in sola lettura, quindi non sono utilizzabili per la memorizzazione di istogrammi.

La scelta migliore consiste quindi nell'utilizzo della memoria locale, nonostante le sue prestazioni non siano particolarmente buone.

Sono state prese in considerazione alcune soluzioni alternative per evitare l'utilizzo della local memory, ma nessuna di queste ha portato un miglioramento nelle prestazioni, quindi si è optato per il mantenimento del modello di calcolo sviluppato per l'ambiente multicore, basato sulla divisione della superficie dell'immagine in settori verticali.
Nonostante la struttura di funzionamento dell'algoritmo sia molto simile, il modello d'esecuzione e i meccanismi paralleli sono completamente diversi: con CUDA è possibile utilizzare centinaia di thread, che vengono divisi in blocchi d'esecuzione e portati avanti in maniera efficiente dai multiprocessori grafici. Questo modello risulta particolarmente efficiente perché i core d'esecuzione della GPU sono in grado di cambiare il thread che sta lavorando in maniera attiva, portando avanti quelli che hanno del lavoro da svolgere e sospendendo temporaneamente quelli che sono impegnati in costose operazioni di i/o.
Si è cercato quindi di adattare l'algoritmo parallelo IDC al modello d'esecuzione GPU, lanciando un CUDA-thread per ogni settore verticale della matrice e realizzando quindi un grado di parallelismo notevolente più alto.

Tutte le operazioni presentate finora sono effettuate dalla CPU; il caricamento dell'immagine in memoria, il loop A e la divisione in settori non sono propriamente operazioni di calcolo, ma possono essere considerate azioni preparatorie necessarie per organizzare la struttura dell'algoritmo. La parte di calcolo, che consiste nel movimento della maschera lungo il settore verticale e nell'esecuzione del Loop C, è realizzata dai thread CUDA che, lanciati in maniera parallela, applicano la stessa procedura di calcolo a diversi elementi dell'immagine.
Una caratteristica che contraddistingue l'elaborazione CUDA è il raggruppamento in blocchi d'esecuzione: i thread che compongono un gruppo lavorano in modo sincrono e hanno a disposizione una regione di memoria condivisa. Queste proprietà consentono di implementare alcune ottimizzazioni computazionali che non era possibile realizzare in ambiente multicore.

In particolare, l'accesso alla memoria globale è considerato relativamente costoso e viene ottimizzato in maniera automatica dalla piattaforma CUDA: quando un thread utilizza un valore memorizzato in memoria globale, un intero segmento di dati viene caricato nella cache di livello L1, dalla quale è accessibile in maniera efficiente a tutti i thread.
Questo significa che l'accesso ad elementi memorizzati sulla stessa porzione di memoria può essere ottimizzato da un gruppo di thread (warp) trasferendo dalla memoria una intera linea di cache.
Questo procedimento funziona solo se i dati richiesti sono allineati. Per esempio, che un blocco da 128 byte può essere caricato in una linea di cache solo se il suo indirizzo di partenza è multiplo di 128 (figura 4.6); in caso non sia allineato sono necessari 2 trasferimenti da 128 byte per elaborare i dati (figura 4.7).

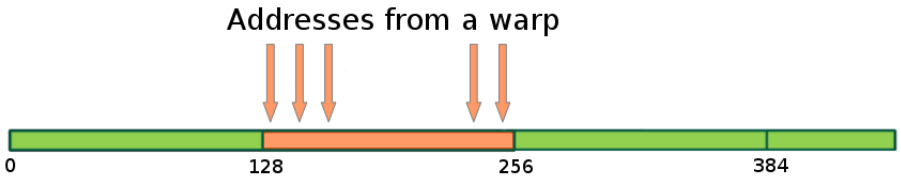


Figura 4.6: Accesso a dati allineati, memorizzati in memoria globale.

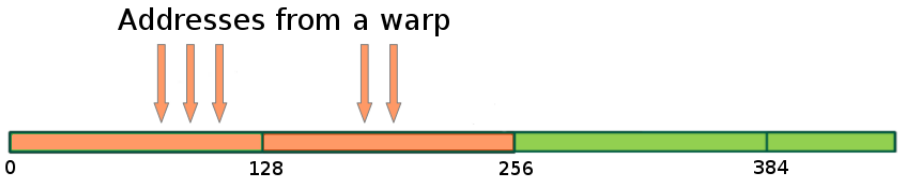


Figura 4.7: Accesso a dati non allineati, memorizzati in memoria globale.

Sfruttando questo meccanismo di accesso e la regione di memoria condivisa tra i thread di un blocco, è stata realizzata una procedura molto efficiente per la lettura dei valori dell'immagine.
L'idea alla base di questo meccanismo è che i thread che compongono un blocco occupano posizioni consecutive e hanno quindi il compito di elaborare

colonne di ranklet adiacenti, le cui finestre di calcolo si sovrappongono quasi totalmente. Questa situazione suggerisce una strategia efficiente di accesso in memoria: è possibile far collaborare un gruppo di thread in modo da trasferire un'intera linea di cache dalla memoria globale alla shared memory, effettuando una sola operazione di trasferimento (figura 4.8).

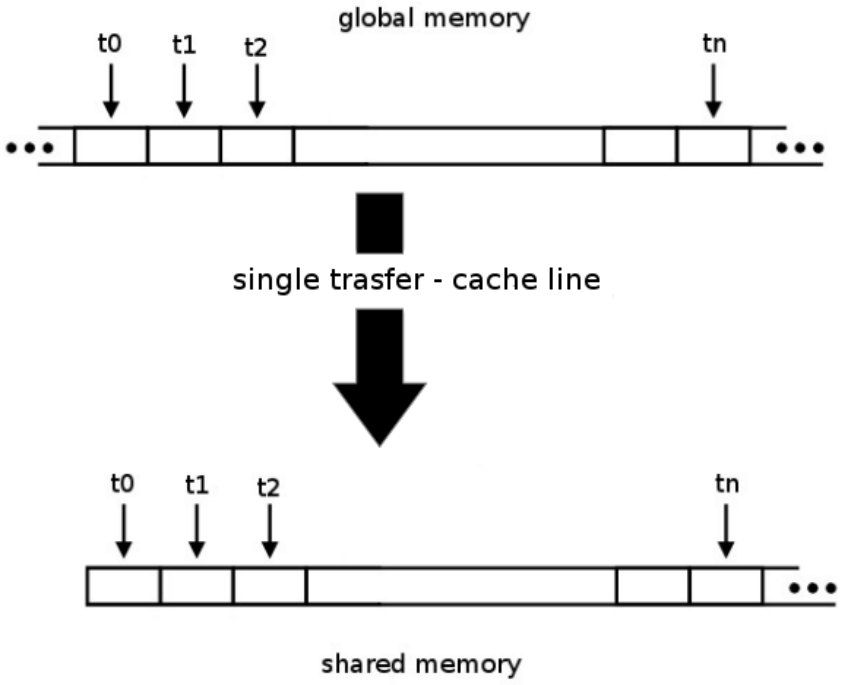


Figura 4.8: Trasferimento di dati dalla memoria globale a quella condivisa.

Le operazioni che coinvolgono la lettura di valori dalla memoria globale sono realizzate dai thread una riga alla volta, quindi, al j-esimo livello di piramide, ogni thread ha bisogno di un numero di elementi pari alla dimensione della maschera, w_j. Questo significa che un gruppo composto da n thread, ad ogni iterazione, necessita complessivamente di $n + w_j$ valori (4.9).
Tutti i valori necessari al gruppo di thread vengono quindi copiati, ad ogni iterazione, dalla memoria globale a quella condivisa, dalla quale sono accessibili in maniera molto più veloce.

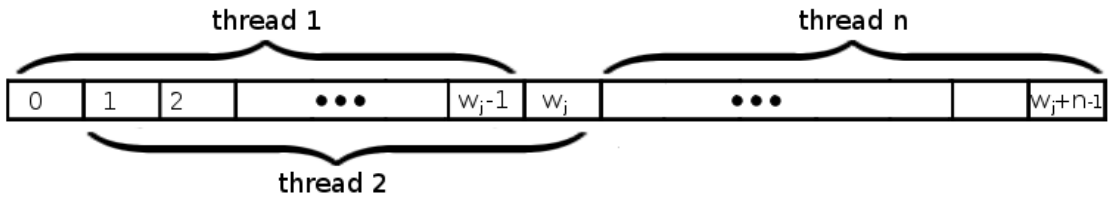


Figura 4.9: Riga di valori condivisi tra i thread di un blocco.

La tecnica più efficiente per leggere valori memorizzati nella memoria condivisa è il broadcast, che viene attuato facendo in modo che tutti i thread accedano contemporaneamente alla stessa cella di memoria. Questo genere di operazione può essere realizzata in maniera molto efficace dalla piattaforma CUDA, perché non è necessario garantire la sequenzialità degli accessi alla stessa locazione, poiché lo stesso valore viene trasferito a tutti i thread del gruppo.
Questa strategia viene attuata in tutta la procedura, ogni volta che è necessario elaborare elementi della matrice di input.
Le operazioni di calcolo, effettuate nel Loop C, sono state impelementate in maniera molto simile a quanto fatto nel codice per CPU. Le uniche differenze rilevante sono due: l'eliminazione dell'istogramma *hist* e l'impiego di valori float per il calcolo dei rank, anche quando la maschera di convoluzione è di piccole dimensioni.
La prima modifica è stata attuata per ridurre il numero di accessi alla memoria locale, che nell'implementazione CUDA risultano relativamente pesanti. Si è notato che la struttura *hist* comportava un numero molto elevato di operazioni in scrittura, ma veniva utilizzata solo nella parte di codice relativa al calcolo dei rank, nella quale è necessario accedere anche agli altri istogrammi. Si è pensato di ottimizzare la procedura calcolando, quando necessario, il valore di $hist[val]$ come somma degli istogrammi dei quadranti che compongono la finestra:

$$hist[val] = hist_{S11}[val] + hist_{S12}[val] + hist_{S21}[val] + hist_{S22}[val]$$

Questa ottimizzazione ha consentito di dimezzare il numero complessivo di accessi alla memoria locale, migliorando sensibilmente l'efficienza del programma.
La seconda modifica non ha portato alcun vantaggio computazionale, ma ha contribuito a rendere il codice più semplice e lineare, senza peggiorare le prestazioni. Si è notato infatti, che utilizzare numeri interi o floating point per memorizzare i rank non aveva alcuna influenza sul tempo di calcolo. Quindi si è deciso di impiegare la soluzione più coerente con il tipo di valori da elaborare, scegliendo di utilizzare valori razionali.

Completata l'elaborazione della trasformata nelle tre direzioni, ogni thread copia i risultati nello spazio di memoria globale dedicato ai valori calcolati e si prepara all'iterazione successiva, eliminando la prima riga di ranklet dagli istogrammi che rappresentano la posizione attuale della finestra.
Dopo aver raggiunto l'ultima posizione del settore verticale, terminate le operazioni di spostamento della maschera, la procedura termina e il controllo torna all'host, che si occupa di trasferire nel proprio spazio di memoria i valori calcolati sulla GPU. A questo punto il calcolo di un livello di ranklet è completato.

In questa sezione è stata presentata l'implementazione del codice per l'ambiente GPU, mettendo in evidenza le differenze rispetto alla versione multicore e le caratteristiche specifiche della piattaforma CUDA.
È stata dedicata un'attenzione particolare alle tipologie di memoria impiegate e alle tecniche di accesso ai dati, perché queste caratteristiche presentano notevoli differenze con l'architettura della CPU.
Non è stato possibile realizzare grandi ottimizzazioni alla parte di codice relativa al calcolo, perché tutte le possibilità di miglioramento erano già state vagliate nell'implementazione OpenMP.

4.4 Correttezza

Una delle fasi che compongono il processo di sviluppo software consiste nel verificare la correttezza del codice implementato, operazione che può essere effettuata studiando il codice in modo analitico o controllando i risultati generati durante l'esecuzione. Nella realizzazione di questo progetto, oltre alle tradizionali operazioni di verifica della correttezza, si è ritenuto opportuno effettuare specifici controlli di correttezza numerica dei valori generati dalla trasformata ranklet.

Le motivazioni che giustificano l'esecuzione di queste verifiche riguardano principalmente il modo in cui i valori calcolati vengono utilizzati nelle applicazioni che lavorano con le ranklet. Di solito infatti, i risultati della trasformata vengono confrontati con altri valori calcolati durante esecuzioni precedenti oppure analizzati con algoritmi di classificazione o reti neurali. Per garantire il corretto funzionamento di queste tecniche è necessario verificare che i valori generati dalle diverse implementazioni siano esatti dal punto di vista numerico, poiché si tratta di valori in virgola mobile.

Per queste ragioni, al termine di ogni fase implementativa, i valori prodotti dalla trasformata sono stati controllati, confrontandoli con valori di riferimento ritenuti affidabili.

Si è verificato che i valori calcolati risultano identici a quelli di riferimento fino alla quarta cifra decimale. Sono stati considerati sufficientemente precisi da non richiedere l'implementazione di modifiche alle procedure di calcolo.

Capitolo 5

Analisi delle Prestazioni

Dopo aver presentato le caratteristiche formali della trasformata ranklet e le implementazioni sviluppate nella realizzazione del progetto di tesi, in questo capitolo verranno analizzate le prestazioni ottenute nell'esecuzione del codice, confrontandole con quelle relative ad una versione di riferimento, realizzata implementando l'algoritmo base per il calcolo della trasformata ranklet.

Le proprietà di efficienza del codice realizzato vengono stabilite in base al tempo d'esecuzione necessario per portare a termine l'elaborazione della piramide completa di ranklet. Il salvataggio dei valori generati è stato escluso dal conteggio, perché non rientra nelle effettive operazioni di calcolo e dipende fortemente dai meccanismi utilizzati e dal modo in cui tali valori devono essere impiegati successivamente.

Con lo scopo di garantire una maggiore validità statistica delle misurazioni rilevate, si è stabilito, per ogni caso preso in esame, di effettuare diverse misurazioni del tempo d'esecuzione, analizzando input diversi ogni volta; dunque per ogni dimensione trattata, si è costruito un set di immagini standard, prese da database pubblici o realizzate appositamente.

Per studiare in maniera esaustiva il comportamento delle soluzioni software sviluppate, ogni implementazione è stata testata con input di diversa dimensione, ripetendo l'esecuzione per ogni immagine del campione.

Il valore finale relativo al tempo di elaborazione, per ogni caso di studio, sarà quindi calcolato come la media campionaria dei risultati delle singole prove. In aggiunta al valore medio verranno inoltre presentati il minimo e il massimo tempo d'esecuzione ottenuti nell'elaborazione delle diverse immagini del campione, valori ritenuti utili per avere un quadro più completo delle prestazioni.

5.1 Piattaforme d'esecuzione

Prima di iniziare a studiare le prestazioni degli algoritmi, è utile presentare le piattaforme d'esecuzione sulle quali sono state effettuate le prove. Le caratteristiche specifiche di questi sistemi influenzano l'elaborazione in maniera determinante e rappresentano un aspetto di primaria importanza nel valutare l'efficienza del codice sviluppato, perché lo stesso software, al variare del sistema utilizzato, può presentare prestazioni molto differenti.
Per effettuare le prove sono quindi stati impiegati due dispositivi diversi, il primo per quelle realizzate in ambiente cpu/multicore, il secondo per quelle su GPU.

- Per l'esecuzione del codice in ambiente multicore, si è utilizzato un sistema costituito da due processori Intel Xeon X5650, ognuno dei quali è composto da 6 core d'esecuzione, ha frequenza di clock pari a 2.66 GHz e architettura x86 a 64 bit.
  In questo caso il numero complessivo di unità di elaborazione è quindi 12, il che consente di realizzare un buon livello di parallelismo.
  Questa piattaforma dispone di un sistema operativo Windows Server 2008 e per questo il codice è stato compilato all'interno dell'ambiente Visual Studio Express 2012, che supporta le direttive OpenMP 2.0.

- Per testare il codice in ambiente grafico è stato impiegato un sistema composto da un processore AMD Opteron con sistema operativo GNU

Linux, frequenza di 2.15 GHz, connesso ad una GPU Tesla X2090 tramite un bus PCI Express 2.0.
Il processore grafico a disposizione appartiene alla classe Fermi e dispone di capability CUDA di livello 2.0: il dispositivo risale al 2010[1], ma gode di un buon livello di parallelismo e supporta le principali funzionalità della piattaforma CUDA.
La X2090 è dotata di 16 multiprocessori da 32 core ciascuno, per un totale di 512 unità di elaborazione ed è in grado di realizzare performance di picco pari a 665 GFlop/s in doppia precisione e oltre 1 TFlop/s con precisione singola.
Dispone inoltre di 6 GB di memoria RAM GDDR5, 64 KB di constant memory e 48 KB di memoria condivisa (per blocco).
Queste caratteristiche la rendono adeguata all'utilizzo come piattaforma d'esecuzione in ambiente grafico.
Per la compilazione è stato utilizzato gcc 4.7.2 e le librerie CUDA 5.

5.2 Prestazioni codice di riferimento

Dovendo verificare l'efficienza delle implementazioni realizzate, nella fase di analisi delle prestazioni si è ritenuto necessario utilizzare una implementazione di riferimento come misura di paragone per le versioni sviluppate.
Questa versione del codice implementa l'algoritmo base per il calcolo della trasformata ranklet in maniera sequenziale, senza alcun tipo di ottimizzazione.
Come piattaforma di test è stata utilizzata quella multicore, anche se l'esecuzione ha impegnato un solo core d'esecuzione.
La tabella 5.1 mostra le prestazioni dell'algoritmo base, espresse in secondi, per diverse dimensioni delle immagini di input.

[1] Le ultime GPU prodotte da NVIDIA appartengono alla classe Kepler, presentata nel capitolo 1.3.3, che dispone di capability 3.0.

Dimensione	Tempo Medio (s)	T. Minimo	T. Massimo
256	30,0	17,3	35,6
512	528,1	436,7	577,2
1024	8912,7	8788,0	9001,9

Tabella 5.1: Tempo d'esecuzione, algoritmo base

Nel caso di input 256x256 e 512x512, l'algoritmo è stato eseguito su un campione di 50 immagini diverse, mentre nel caso 1024 sono state effettuate solo 5 prove, a causa dell'altissimo tempo di elaborazionale richiesto.
A causa della sua inefficienza, questa versione del codice è stata testata solo su immagini di dimensioni relativamente ridotte. Si può notare la rapidità con cui il tempo d'esecuzione aumenti, al raddoppiare della dimensione del problema: sono necessari circa 30 secondi per analizzare un immagine 256x256, mentre nel caso 512x512 il tempo richiesto arriva a quasi 9 minuti, fino a raggiungere le 2 ore e mezza per immagini di dimensioni 1024x1024.
Questi valori rappresentano il punto di partenza per la valutazione delle ottimizzazioni realizzate, che ogni volta saranno confrontate con queste prestazioni di riferimento.

5.3 Prestazioni codice OpenMP (CPU)

Verrano ora analizzate da diversi punti di vista le prestazione ottenute con il codice multicore ottimizzato (di seguito: codice OpenMP), cercando di mettere in luce tutte le caratteristiche computazionali più interessanti del codice sviluppato.
Si è deciso di iniziare la procedura di analisi mostrando i risultati delle operazioni di profiling effettuate sul codice, realizzate sia dall'esterno, tramite strumenti automatici come GNU gprof[23], che dall'interno, misurando il

tempo speso all'interno del codice dalle diverse procedure.
Il modo più interessante per suddividere il lavoro svolto da ogni thread nell'elaborazione della piramide completa, consiste nel dividere la procedura di elaborazione in due parti:

- una è composta dalle operazioni di scansione e spostamento della maschera di convoluzione, mirata alla costruzione degli istogrammi (sliding window);
- l'altra fa riferimento alle effettive operazioni di calcolo della trasformata ranklet (calcolo).

La fase di profiling è stata sviluppata analizzando come cambia il rapporto tra queste due componenti in base al lato della finestra, che assume dimensioni crescenti pari alle potenze di 2.
Le misurazioni riportate di seguito fanno riferimento ad esecuzioni realizzate con un solo thread, per analizzare in modo più logico e diretto come viene speso il tempo di elaborazione all'interno della procedura.

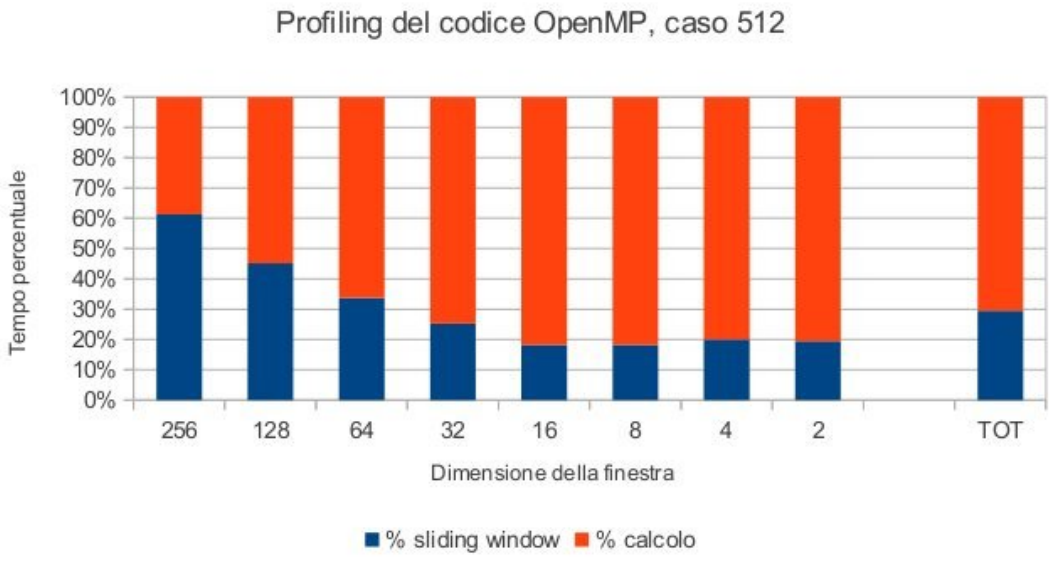


Figura 5.1: Profiling del codice OpenMP, caso 512.

I grafici 5.1 e 5.2 mostrano come viene impiegato il tempo d'esecuzione nel-

l'elaborazione di immagini di dimensione 512 e 1024.

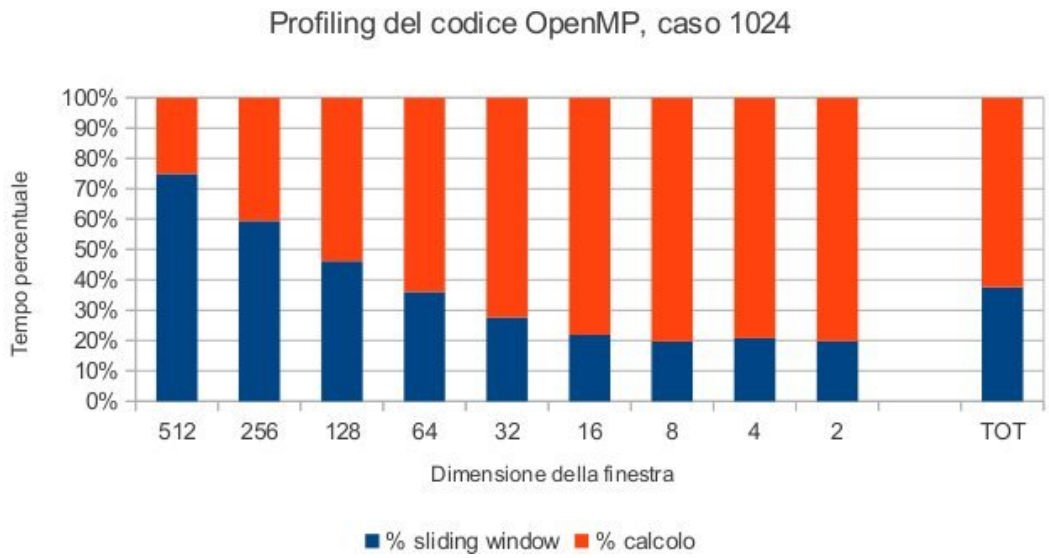


Figura 5.2: Profiling del codice OpenMP, caso 1024.

Il comportamento dell'algoritmo, nei due casi presentati e negli altri analizzati, è molto simile: nei livelli di piramide elaborati con una maschera di grandi dimensioni, il tempo viene impiegato in misura maggiore nella costruzione della finestra e nelle operazioni di sliding window, piuttosto che nel calcolo del valore della trasformata; man mano che l'area della maschera si riduce, il tempo di scansione diventa meno influente e di conseguenza aumenta il peso relativo alle operazioni di calcolo.
Negli ultimi livelli di piramide, quando la finestra assume dimensioni molto piccole, il rapporto tra le due componenti sembra stabilizzarsi intorno a valori quasi costanti: circa l'80% del tempo è impiegato in operazioni di calcolo, mentre il rimanente 20% viene speso nella gestione della finestra.
Questo comportamento è ragionevole e giustificato dalla differenza di complessità che contraddistingue le due operazioni: la parte di sliding window dipende dal lato della maschera in modo lineare, mentre la procedura di calcolo è un'operazione quasi costante, il cui costo è relativo al numero di livelli di grigio impegnati dall'immagine.

Nell'elaborazione dell'intera piramide, il tempo di calcolo risulta più influente del tempo relativo alla gestione della maschera, ma la differenza si riduce all'aumentare della dimensione dell'input.

L'analisi appena presentata mostra come l'implementazione utilizzi il tempo di calcolo, senza analizzare il modo in cui le prestazioni complessive cambino in relazione alla dimensione del problema. Questo secondo aspetto verrà trattato in questa parte del capitolo.
La tabella 5.2 contiene i tempi d'esecuzione del codice OpenMP in diversi casi di studio, ottenuti testando il codice con un solo thread, su un campione di 50 immagini.

Dimensione	**Tempo Medio (s)**	**T. Minimo**	**T. Massimo**
256	0,37	0,28	0,52
512	1,4	0,94	2,01
1024	6,4	5,5	7,7
2048	41,4	38,9	43,7
4096	280,4	269,6	295,3

Tabella 5.2: Tempi d'esecuzione del codice OpenMP (1 core)

Si può notare come il tempo aumenti molto rapidamente in relazione alla dimensione delle immagini analizzate, passando da qualche secondo (nei casi 512 e 1024) ad oltre 4 minuti con immagini 4096x4096.
È però interessante confrontare questi tempi d'esecuzione con quelli dell'implementazione base: per qualunque dimensione analizzata, le prestazioni del codice OpenMP risultano notevolmente migliori rispetto alle altre: non solo i tempi d'esecuzione sono di gran lunga inferiori, ma anche l'incremento che si ottiene raddoppiando la dimensione dell'input è più basso.
Nella tabella 5.1, infatti, si può osservare che il tempo di calcolo aumenta di circa 300 volte, passando dal caso 256 a quello 1024, mentre con l'algoritmo

ottimizzato tale fattore è circa 20.
A causa di questa differenza nel grado di incremento del tempo di calcolo, risulta difficile stabilire una misura di quanto il codice OpenMP sia più efficiente dell'altro, ma è interessante osservare come, nel caso 1024, il tempo d'esecuzione del codice base sia 1000 volte più lento di quello ottimizzato.
Il confronto appena effettuato considera le prestazioni di due algoritmi di pari livello, che utilizzano lo stesso quantitativo di risorse hardware: il primo è un'implementazione sequenziale, mentre il secondo è eseguito su un solo core. Il paragone, rappresentato in figura 5.3, mette in luce l'enorme differenza computazionale tra le due versioni dell'algoritmo per il calcolo della trasformata ranklet.

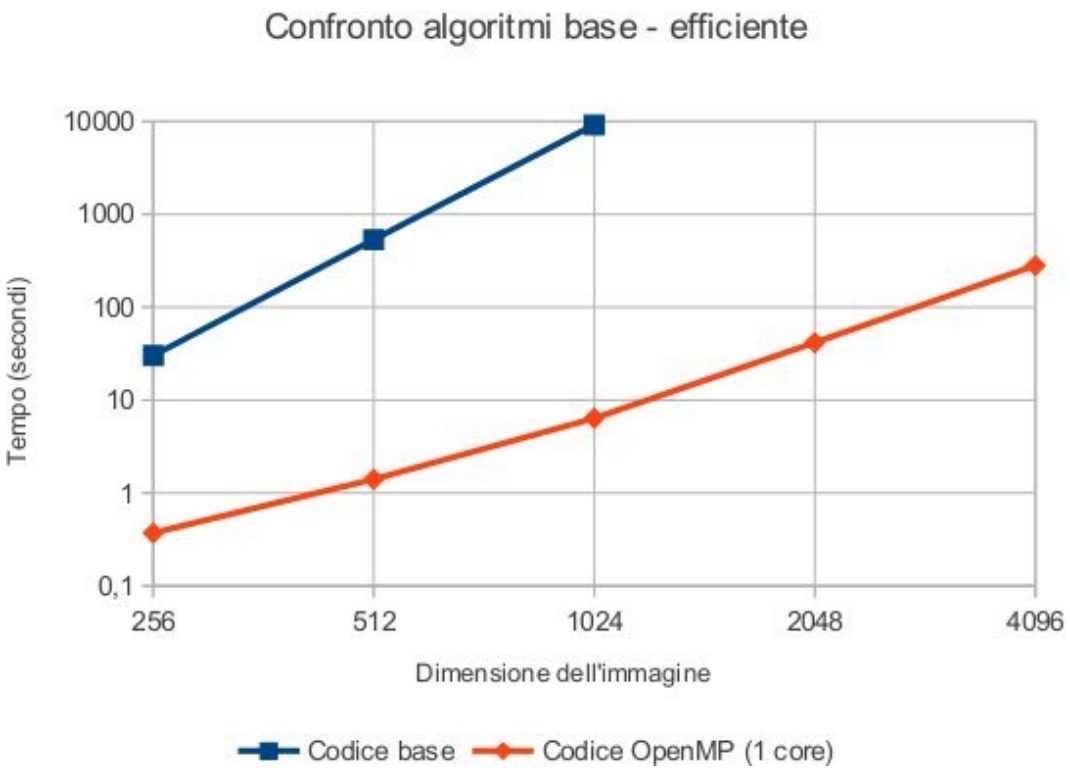


Figura 5.3: Confronto tra tempi d'esecuzione, alg. base - alg. efficiente.

Dopo aver analizzato come la procedura impiega il tempo e come questo aumenti in relazione alla dimensione dell'immagine da analizzare, si è ritenu-

to interessante mostrare come le prestazioni del codice cambino in base al numero di thread utilizzati nel calcolo.
Questa analisi valuterà quindi quanto l'algoritmo implementato sia in grado di scalare in base al numero di core messi a disposizione dall'architettura. Oltre all'esposizione dei tempi di elaborazione per alcuni casi di studio, verrà calcolato lo *speedup* che, nel settore del calcolo parallelo, rappresenta la misura di quanto un algoritmo parallelo sia migliore dell'equivalente sequenziale. Il valore di speedup $S(p)$, relativo all'esecuzione su p processori, viene infatti calcolato come

$$S(p) = \frac{T(1)}{T(p)}$$

dove i valori $T(1)$ e $T(p)$ rappresentano i tempi d'esecuzione sequenziale e parallelo (con p processori) dell'algoritmo.
Le tabelle 5.3 e 5.4 mostrano due esempi di come cambia il tempo d'esecuzione al variare del numero di thread. Considerando che i core disponibili sono 12, si è deciso di prendere in considerazione le prestazioni effettuate con 1, 2, 4, 8 e 12 thread OpenMP.

Thread	**T. Medio (s)**	**T. Minimo**	**T. Massimo**	**Speedup**
1	6,4	5,5	7,7	1,00
2	3,3	2,8	4,5	1,93
4	1,7	1,4	2,1	3,83
8	1,2	1,0	1,6	5,13
12	1,1	1,0	1,4	5,34

Tabella 5.3: Tempi d'esecuzione e speedup, codice OpenMP (caso 1024)

Nella lettura dei dati riportati, il punto di partenza è l'esecuzione effettuata con un solo thread; Le altre misurazioni dovrebbero presentare valori che scalano in relazione al numero di core impiegati nell'eleborazione.
Il valore di speedup ideale $S_{max}(p)$ equivale al numero di unità di esecuzione

Thread	T. Medio (s)	T. Minimo	T. Massimo	Speedup
1	280,4	269,6	295,3	1,00
2	141,0	136,9	147,1	1,99
4	72,6	70,9	75,8	3,86
8	44,2	41,8	47,5	6,34
12	36,7	34,5	41,8	7,65

Tabella 5.4: Tempi d'esecuzione e speedup, codice OpenMP (caso 4096)

impiegate, poiché il massimo fattore di miglioramento che è possibile teorizzare dividendo il carico di lavoro su p processori è pari a p:

$$S_{max}(p) = \frac{T(1)}{T(1)/p} = p * \frac{T(1)}{T(1)} = p$$

Lo speedup ideale rappresenta un valore limite, quasi impossibile da raggiungere in casi reali, ma è utile per analizzare la qualità dei valori ottenuti in un contesto parallelo.
Prendendo ad esempio le prestazioni ottenute con immagini 1024x1024, si può notare come siano necessari più di 6 secondi per portare a termine il calcolo nell'esecuzione con un solo core. Raddoppiando il numero di unità di elaborazione, il tempo d'esecuzione diventa circa la metà, riducendosi a poco più di 3 secondi. Utilizzando 4 thread, il calcolo richiede circa 1,6 secondi, cioè circa un quarto del tempo sequenziale. Il ripetersi di questo comportamento con diverse condizioni di parallelismo, mette in luce le proprietà di scalabilità che caratterizzano l'implementazione.
Il grafico 5.4 e la tabella 5.5 mostrano i valori di speedup ottenuti confrontando le prestazioni del codice OpenMP relativo a diverse dimensioni dell'input. Si può osservare come i valori riportati siano generalmente buoni e scalino in maniera coerente con il numero di core a disposizione.
I valori di speedup, inoltre si avvicinano a quelli ideali, con l'aumentare della dimensione dell'input, il che significa che quando il carico di lavoro è più intenso, i meccanismi di parallelizzazione funzionano meglio.

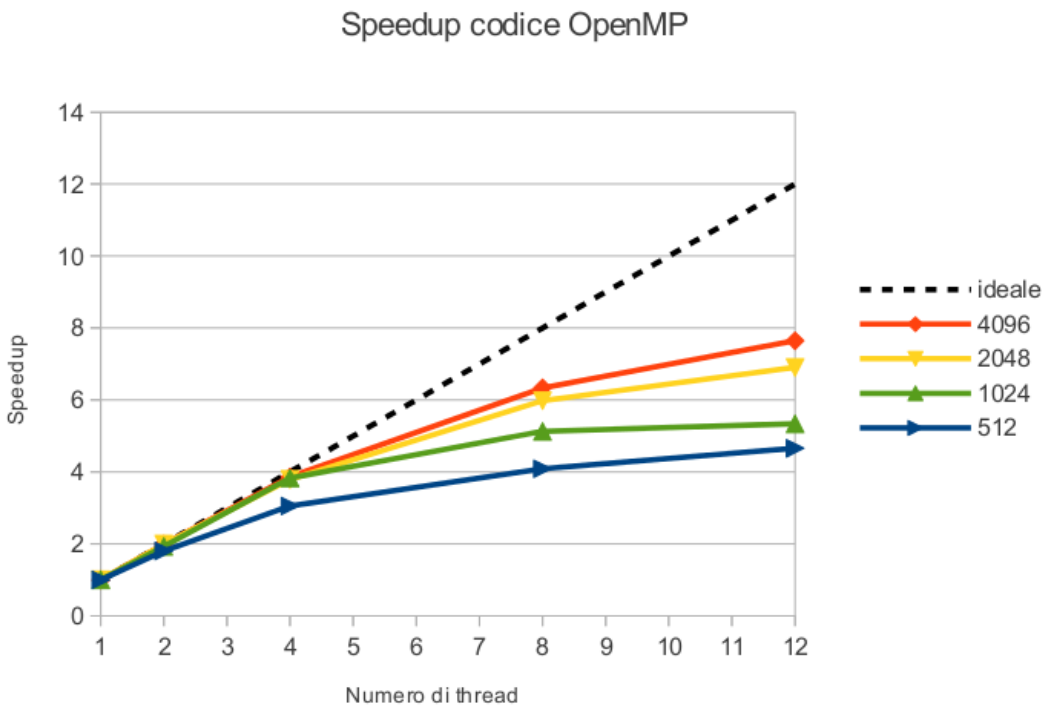


Figura 5.4: Speedup del codice OpenMP.

Thread	Ideale	512	1024	2048	4096
1	1,00	1,00	1,00	1,00	1,00
2	2,00	1,80	1,93	1,99	1,99
4	4,00	3,05	3,83	3,79	3,86
8	8,00	4,09	5,13	5,98	6,34
12	12,00	4,66	5,34	6,90	7,65

Tabella 5.5: Speedup del codice OpenMP

Per i motivi descritti precedentemente, non è stato possibile verificare le prestazione del codice con un livello di parallelismo superiore, ma l'andamento dello speedup lascia intravedere ulteriori margini di miglioramento su

piattaforme dotate di un numero maggiore di core.

I migliori tempi d'esecuzione ottenuti sono quindi stati realizzati impiegando tutti e 12 i core d'esecuzione disponibili e sono mostrati in tabella 5.6. Le prestazioni fanno riferimento, come negli altri casi, a campioni composti da 50 immagini diverse.

Dimensione	**Tempo Medio (s)**	**T. Minimo**	**T. Massimo**
256	0,051	0,015	0,058
512	0,30	0,290	0,38
1024	1,1	1,0	1,4
2048	5,9	5,7	6,3
4096	36,7	34,5	41,8
8192	249,2	225,7	267,0
16384	1660,6	1620,4	1715,7

Tabella 5.6: Tempi d'esecuzione codice OpenMP (12 core)

In questa sezione si è cercato di analizzare tutti gli aspetti che contraddistinguono e influenzano le prestazioni dell'implementazione multicore. In particolare sono state presentate due caratteristiche di grande importanza:

- l'esecuzione sequenziale dell'algoritmo ottimizzato consente un miglioramento considerevole rispetto all'algoritmo base;

- il codice OpenMP scala in maniera molto buona, soprattutto nei casi con intenso carico di lavoro (input di grandi dimensioni).

Questi due aspetti, in combinazione, garantiscono un enorme miglioramento computazionale, che è un indice affidabile della qualità delle ottimizzazioni sviluppate nell'implementazione del codice OpenMP.

5.4 Prestazioni codice CUDA (GPU)

In maniera analoga a quanto effettuato per il codice OpenMP, verranno ora analizzate in dettaglio le prestazioni del codice CUDA, confrontandole con quelle relative alle altre versioni dell'algoritmo.
La prima operazione, analogamente al caso precedente, sarà quella di effettuare un profiling dell'algoritmo, suddividendo il tempo d'esecuzione in 3 categorie:

- trasferimento di dati dalla memoria del device a quella dell'host;
- scansione e movimento della maschera di convoluzione;
- calcolo della trasformata.

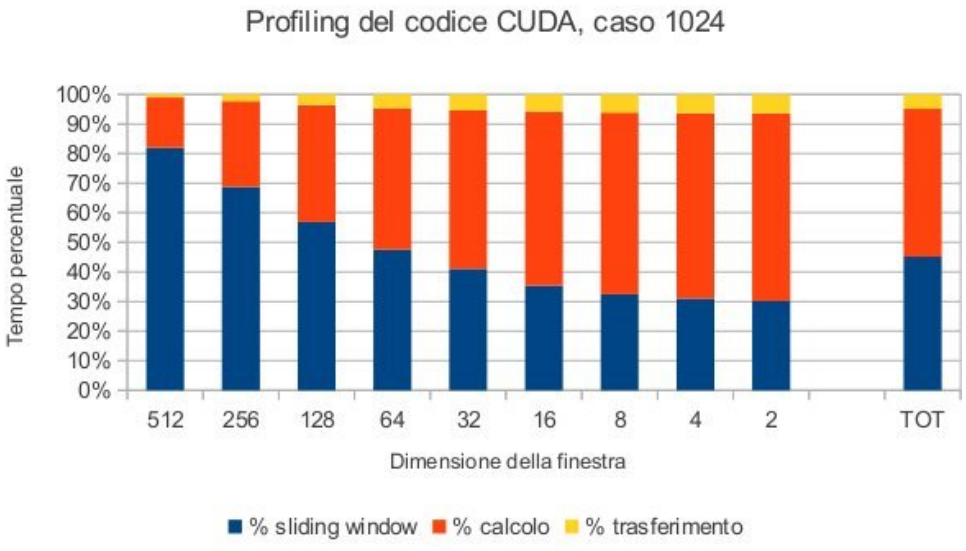


Figura 5.5: Profiling del codice CUDA, caso 1024.

La figura 5.5 mostra i risultati del profiling nell'elaborazione di immagini di lato 1024, mentre la 5.6 per il caso 2048.
In entrambi i casi si può notare un andamento simile a quello osservato per il codice OpenMP: nei livelli di piramide con finestra di grandi dimensioni,

il tempo viene impiegato prevalentemente nell'accesso all'immagine e nella gestione degli istogrammi. I valori tendono ad equilibrarsi man mano che la dimensione della maschera si riduce.

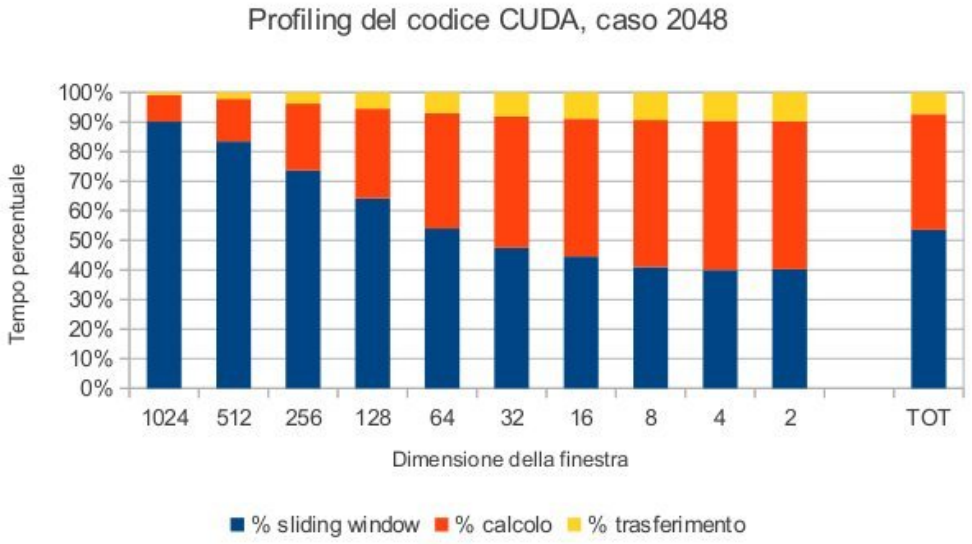


Figura 5.6: Profiling del codice CUDA, caso 2048.

Confrontando i valori appena presentati con quelli relativi al codice multicore, si nota che il tempo d'esecuzione relativo alle operazioni di sliding window occupa una percentuale maggiore, rispetto a quanto non fosse nel caso precedente, arrivando ad occupare il 45% del totale con input 1024x1024 e superando il 50% nel caso 2048.
Il tempo impiegato nelle operazioni di calcolo della trasformata, di conseguenza, influisce in maniera minore sul totale, arrivando sotto il 40% del tempo complessivo per input 2048.
Rispetto a quanto fatto per il codice OpenMP, si è ritenuto utile aggiungere, tra le tipologie di operazioni analizzate, il tempo necessario per copiare i risultati dalla memoria della GPU a quella dell'host. L'importanza specifica di questa categoria di operazioni è strettamente collegata con il numero di valori calcolati ad ogni livello e, come si può notare dai grafici, dipende sia dal livello di piramide, sia dalla dimensione dell'input.

In conclusione, quindi, si può affermare che il tempo impiegato nelle operazioni di calcolo rimane quasi costante al variare della dimensione della maschera, mentre quello relativo alle altre due categorie di operazioni cambia in maniera opposta: all'aumentare della dimensione della finestra, cresce il tempo relativo alla sua gestione, mentre diminuiscono i valori da calcolare e, di conseguenza, il relativo tempo di trasferimento.

Nell'analisi delle prestazioni del codice CUDA non è stato possibile verificarne la scalabilità, poiché tale misura dipende dal numero di unità di elaborazione utilizzate e le GPU, per loro natura, tendono a impiegare tutta la potenza di calcolo che hanno a disposizione.
Si è quindi passati direttamente all'analisi del tempo di elaborazione in funzione della dimensione dell'immagine in input; la tabella 5.7 riporta i valori rilevati, ottenuti ripetendo ogni esecuzione su un campione di 50 immagini diverse.

Dimensione	**Tempo Medio (s)**	**T. Minimo**	**T. Massimo**
512	1,7	1,6	1,8
1024	4,0	3,6	4,6
2048	11,9	10,6	12,9
4096	42,6	38,7	44,7
8192	177,6	160,9	186,3
16384	1130,6	1084,3	1204,7

Tabella 5.7: Tempi d'esecuzione del codice CUDA

È interessante notare come le prestazioni rilevate abbiano ordine di grandezza molto simile a quelle dell'implementazione OpenMP e, come in quel caso, il tempo d'esecuzione aumenti in maniera non lineare.
Infatti, per input di media dimensione, come 512 e 1024, il calcolo della piramide completa richiede pochi secondi, mentre per immagini più grandi il

tempo di elaborazione aumenta velocemente, arrivando a quasi a 3 minuti nel caso 8192 e superando i 15 minuti nell'analisi di immagini con lato 16384.

5.5 Confronto prestazioni

Per concludere il discorso relativo all'analisi delle prestazioni si è deciso di confrontare i tempi d'esecuzione delle tre implementazioni, paragonando le migliori performance che è possibile ottenere con ogni soluzione. Per questo motivo, i tempi relativi all'esecuzione OpenMP fanno riferimento a test effettuati con tutti e 12 i core a disposizione.

Dimensione	**Tempo Base**	**Tempo OpenMP (12)**	**Tempo CUDA**
256	30,0	0,051	1,2
512	528,1	0,30	1,7
1024	8912,7	1,1	4,0

Tabella 5.8: Confronto prestazioni base - CUDA - OpenMP

Entrambe le versioni ottimizzate, realizzate nel corso di questo lavoro di tesi, risultano notevolmente più efficienti del codice di riferimento, come si può notare nella tabella 5.8.
Il paragone tra le prestazioni può essere portato avanti confrontando i singoli tempi d'esecuzione oppure analizzando il modo in cui aumentano in relazione all'immagine di input. È evidente che il tempo d'esecuzione dell'algoritmo base risulti peggiore degli altri, in ogni modo lo si consideri: è più alto e aumenta più rapidamente.
Il confronto completo è stato realizzato solo per immagini di dimensioni ridotte e non è stato sviluppato ulteriormente per mancanza di dati relativi all'esecuzione dell'algoritmo base. D'altra parte la differenza tra i tempi è

talmente alta che l'analisi di casi aggiuntivi non è stata ritenuta significativa. È invece interessante notare come le prestazioni del codice CUDA e di quello OpenMP aumentino in modo diverso. Il tempo d'esecuzione della solzione CPU è nettamente migliore di quello GPU, per quanto concerne input di piccole dimensioni, ma i valori tendono ad avvicinarsi quando si elaborano immagini più grandi.
Per approfondire questa analisi si è deciso di estendere il confronto a input di maggiori dimensioni; i valori presi in considerazione sono presentati nella tabella 5.9.

Dimensione	**Tempo OpenMP (12)**	**Tempo CUDA**
512	0,3	1,7
1024	1,1	4,0
2048	5,9	11,9
4096	36,7	42,6
8192	249,2	177,6
16384	1660,6	1130,6

Tabella 5.9: Confronto prestazioni CUDA - OpenMP

Come si era notato paragonando il tempo d'esecuzione nei casi 256, 512 e 1024, le prestazioni del codice CUDA risultano peggiori di quelle OpenMP, ma la differenza va diminuendo, quando si aumenta la dimensione dell'input. L'inversione di tendenza avviene nell'analisi di immagini 8192x8192: in questo caso il codice CUDA riesce a terminare impiegado un tempo nettamente inferiore rispetto a quello OpenMP. Nel caso 16384, il trend viene ulteriormente confermato, in quanto l'esecuzione su CPU richiede quasi 10 minuti in più di quella GPU.
La figura 5.7 rappresenta le prestazioni delle diverse implementazioni considerate, riassumendo i concetti espressi nel corso di questo capitolo.
La visualizzazione in scala logaritmica consente di analizzare in maniera ap-

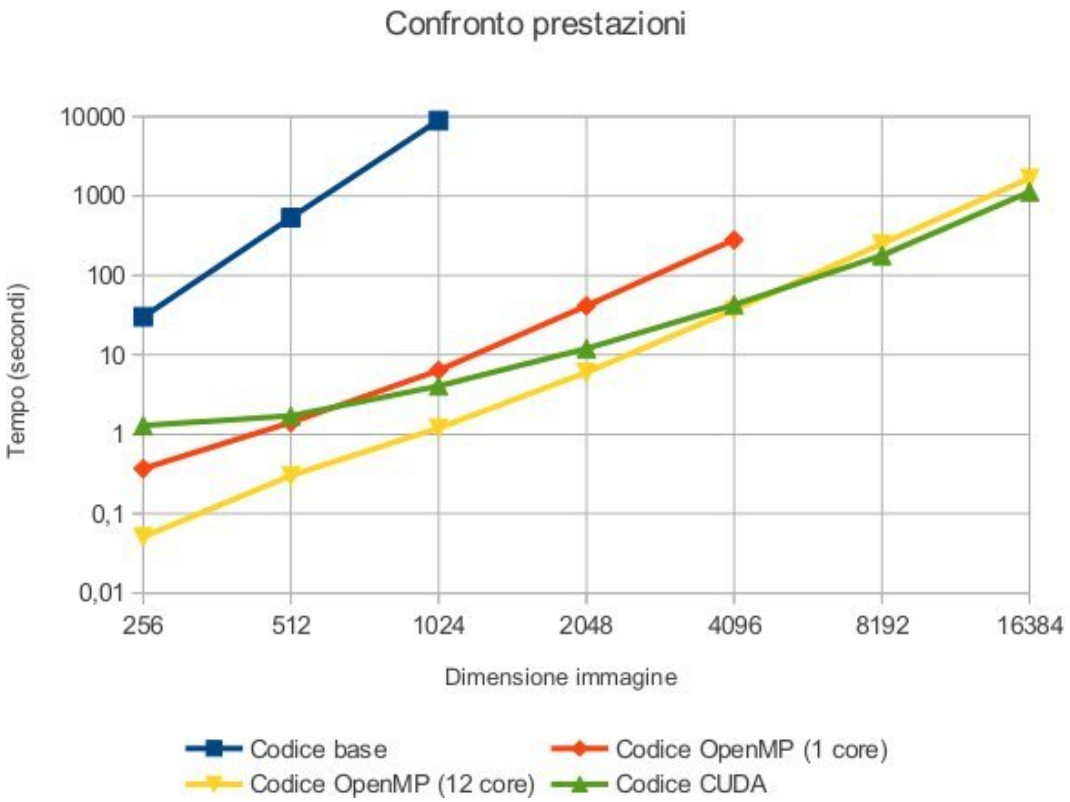


Figura 5.7: Confronto tra le prestazioni.

propriata l'evoluzione delle prestazioni: in tutte le implementazioni prese in esame il tempo d'esecuzione aumenta in maniera esponenziale, ma è chiaramente visibile come la linea relativa al codice CUDA sia meno ripida delle altre.

Questa proprietà è stata confermata nei diversi casi di studio presi in esame ed è ragionevole assumere che tale comportamento si mantenga anche per input di dimensioni maggiori, determinando così la maggiore efficienza dell'implementazione per GPU.

Capitolo 6

Conclusioni

In questo lavoro di tesi sono state presentate diverse strategie di ottimizzazione, applicate al miglioramento delle proprietà computazionali di una specifica tecnica per l'analisi delle immagini, ovvero la trasformata ranklet. L'implementazione di tali ottimizzazioni ha consentito il raggiungimento di prestazioni nettamente migliori di quelle originali, per ragioni di diversa natura:

- la scelta di un algoritmo più efficiente di quello base ha abbassato l'ordine di complessità in maniera determinante, garantendo automaticamente un incremento di prestazioni;
- la progettazione in ambiente parallelo ha consentito di suddividere il carico di lavoro tra più unità di elaborazione, riducendo il numero di operazioni da svolgere sequenzialmente e quindi il tempo di calcolo complessivo;
- sono state implementate diverse ottimizzazioni ad hoc, che hanno permesso di ridurre ulteriormente il tempo necessario per portare a termine alcune operazioni.

Nel confronto tra le due implementazioni si è osservato come il codice OpenMP risulti più veloce nell'elaborazione di immagini di piccole dimensioni, mentre

quello CUDA garantisca prestazioni lievemente migliori con input di maggiori entità. A questo proposito è importante considerare che l'obiettivo del lavoro svolto è consentire l'analisi di immagini reali, come ad esempio quelle mediche, che proprio per le loro dimensioni rendono il processo di elaborazione difficile da attuare. Per questo motivo risultano più interessanti le caratteristiche computazionali dell'implementazione CUDA, che può quindi essere considerata la più efficiente delle due, anche se la versione OpenMP rappresenta una valida alternativa.
Le implementazioni realizzate rappresentano quindi un importante passo in avanti nell'impiego della trasformata ranklet all'interno di procedure reali, poiché consentono di elaborare in un tempo ragionevole immagini di dimensioni simili a quelle impiegate in progetti biomedici di interesse scientifico, che solitamente sono dell'ordine delle decine di migliaia di pixel per lato.

Un altro aspetto da considerare è relativo ai dispositivi impiegati nello svolgimento delle prove finali: nonostante le piattaforme d'esecuzione fossero assolutamente adeguate alle esigenze di sviluppo, non possono comunque essere considerate le migliori soluzioni hardware disponibili sul mercato. Con questa osservazione si vuole mettere in evidenza la possibilità di ottenere prestazioni ancora migliori, senza modificare il codice implementato, semplicemente utilizzando soluzioni tencologiche più recenti e innovative. La scelta di adeguarsi al contesto parallelo, infatti, consente di sfruttare il trend architetturale attualmente in corso, indirizzato verso la realizzazione di dispositivi many-core altamente paralleli.

Bibliografia

[1] Gordon E. Moore. *Cramming more components onto integrated circuits*. Electronics Magazine. 1965.

[2] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., 2010.

[3] Alan Watt. *3d Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., 1993.

[4] GPGPU.org: General-Purpose computation on Graphics Processing Units `http://gpgpu.org/`.

[5] CUBLAS — NVIDIA Developer Zone `http://developer.nvidia.com/cublas/`

[6] OpenACC Home — openacc.org `http://www.openacc-standard.org/`

[7] What is CUDA — NVIDIA Developer Zone `http://developer.nvidia.com/what-cuda`

[8] OpenCL `http://www.khronos.org/opencl/`

[9] Fabrizio Smeraldi. *Ranklets: a Complete Family of Multiscale, Orientation Selective Rank Features*. Research Report RR0309-01, Department of Computer Science, Queen Mary, University of London. 2003.

[10] Kunio Doi. *Computer-Aided Diagnosis in Medical Imaging: Historical Review, Current Status and Future Potential*. Comput Med Imaging Graph. 2007.

[11] Rangaraj M. Rangayyana, Fábio J. Ayresa and J. E. Leo Desautels. *A review of computer-aided diagnosis of breast cancer: Toward the detection of subtle signs*. Medical Applications of Signal Processing, 2007.

[12] Bram van Ginneken, Bart M. ter Haar Romeny and Max A. Viergever. *Computer-Aided Diagnosis in Chest Radiography: A Survey*. IEEE Transactions on Medical Imaging, 2001.

[13] E. Franceschi, F. Odone, F. Smeraldi, A. Verri. *Feature selection with nonparametric statistics*. ICIP, 2005.

[14] M. Masotti, R. Campanini. *Texture classification using invariant ranklet features*. PRL, 2008

[15] R. Campanini, E. Angelini, E. Iampieri, N. Lanconelli, M. Masotti, M. Roffilli. *A ranklet-based CAD for digital mammography*. Digital Mammography, 8th International Workshop. Lecture Notes in Computer Science. Springer, 2006.

[16] M. Masotti, R. Campanini, E. Angelini, E. Iampieri, N. Lanconelli, M. Roffilli. *Ranklet texture features for false positive reduction in computer-aided detection of breast masses*. Computer Assisted Radiology and Surgery - 22nd International Congress and Exhibition. Barcelona, 2008.

[17] M. Roffilli. *Advanced Machine Learning Techniques for Digital Mammography*, PhD thesis, University of Bologna, Department of Computer Science, 2006.

[18] Erich L. Lehmann. *Nonparametrics: Statistical Methods Based on Ranks*. Holden-Day. 1975.

[19] Fabrizio Smeraldi. *Ranklets: orientation selective non-parametric features applied to face detection*. Proceedings of the 16th International Conference on Pattern Recognition. 2002.

[20] Fabrizio Smeraldi. *Fast algorithms for the computation of Ranklets*. Proceedings of ICIP, Cairo. 2009.

[21] Ingrid Daubechies. *Ten Lectures on Wavelets*. Society for industrial and applied mathematics (SIAM). Philadelphia, 1992.

[22] OpenMP.org `http://openmp.org/wp/`.

[23] GNU gprof `http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`.

Ringraziamenti

In conclusione a questo lavoro, mi sembra doveroso ringraziare tutte le persone che in un modo o nell'altro mi sono state vicine e mi hanno aiutato a portare a termine questo percorso di studi, e non solo.

In primo luogo ringrazio il prof. Renato Campanini, per la grande disponibilità dimostrata e per avermi dato la possibilità di partecipare a questo progetto.

Ringrazio inoltre Bioretics srl e in particolare il dott. Matteo Roffilli, per aver messo a mia disposizione il suo tempo, la sua esperienza e le risorse di cui ho avuto bisogno, per l'aiuto pratico, l'assistenza e i consigli che mi ha dispensato durante il corso di questi mesi di collaborazione.

Un ringraziamento speciale va ai miei genitori: grazie per avermi dato il buon esempio, per avermi supportato e spronato a dare sempre il massimo durante tutta la mia vita da studente e per avermi sostenuto economicamente in tutti questi anni.
Un ringraziamento anche a mio fratello, per aver condiviso con me tante esperienze, per essere stato il mio primo amico e per continuare ad esserlo in ogni situazione.

Prima di concludere questa parte, non posso non ringraziare i miei amici: grazie a tutti i ragazzi e le ragazze di Riale con cui ho passato tanti bei

momenti. Grazie per avermi fatto sentire parte di qualcosa, per avermi fatto ridere e divertire in tutte le situazioni e i momenti in cui ne ho avuto bisogno.

E infine grazie a te, Erika, per avermi accompagnato in questo viaggio, per ogni istante, bello o brutto, che abbiamo passato insieme, grazie per tutto quello che hai fatto e che continui a fare per me ogni giorno. Grazie perché riesci sempre a capirmi, a sostenermi e ad aiutarmi in qualunque situazione e per aver contirbuito a farmi diventare quello che sono.