



UNIVERSITÀ DEGLI STUDI DI CATANIA
DIPARTIMENTO DI MATEMATICA E INFORMATICA
CORSO DI LAUREA MAGISTRALE IN INFORMATICA

Alessio Tudisco

MultiProcess Image Processing Bench

RELAZIONE PROGETTO FINALE

Anno Accademico 2022 - 2023

Abstract

Nell'*image processing* si può fare uso della *parallelizzazione* degli algoritmi al fine di migliorare le prestazioni computazionali e accelerare l'elaborazione delle immagini.

Con il termine "parallelizzazione" si indica la capacità dei moderni processori e delle architetture di sistema di *eseguire più operazioni simultaneamente*, consentendo di sfruttare al massimo le risorse disponibili.

Tuttavia, va notato che la parallelizzazione non è sempre utile ed attuabile in tutti i casi di *image processing*. È essenziale valutare attentamente le *caratteristiche* degli algoritmi prima di decidere se la parallelizzazione sia appropriata o vantaggiosa.

Alcuni algoritmi potrebbero essere semplicemente non parallelizzabili, altri parallelizzabili poco efficacemente a causa della presenza di *dipendenze dei dati dalle precedenti fasi di elaborazione*, il che limita i vantaggi della parallelizzazione.

Infine, qualora possibile, la parallelizzazione degli algoritmi di *image processing* non è semplice, in quanto richiede una corretta *progettazione, implementazione, gestione della concorrenza e sincronizzazione delle operazioni parallele* per evitare problemi come le *race condition* o i *conflicti di memoria*.

Indice dei contenuti

1	Contesto	1
1.1	Struttura della relazione	2
2	Strumenti Utilizzati	3
2.1	Python	3
2.1.1	Ragioni di preferenza	4
2.2	Tkinter	4
2.2.1	CustomTkinter	4
2.3	Pillow	5
2.4	NumPy	5
2.4.1	Restrizioni nell'utilizzo	6
2.5	Matplotlib	6
2.6	GIL e Multiprocessing	7
3	MultiProcess Image Processing Bench	8
3.1	Uso del programma	9
3.2	Suddivisione dell'immagine	10
3.3	Algoritmi implementati	11
3.3.1	Convoluzione	12
	Identity	12
	Sobel-X e Sobel-Y	13
	Prewitt-X e Prewitt-Y	14
	Laplacian	15
	Sharpen e High-Pass	16
	Blur (Low-Pass)	17
	Blur (Gaussian)	18
3.3.2	Operatori Morfologici	18
	Erosione	19
	Dilatazione	19
3.3.3	Filtri di Noise Reduction	20
	Mean Filter	20
	Bilateral Filter	21
3.3.4	Canny	22
4	Risultati ottenuti	24

A Appendice	42
Codice Python	42
Suddivisione in linee	42
Suddivisione in quadrati	43
Convoluzione	44
Generazione Kernel di Gaussian	45
Operatore Morfologico d'erosione	45
Operatore Morfologico dilatazione	46
Filtro di Media	47
Filtro Bilaterale	48
Canny Edge Detection	49

1

Contesto

Quando attuabile, la parallelizzazione degli algoritmi di image processing può offrire diversi *vantaggi* significativi:

- *Maggior velocità di elaborazione*: la parallelizzazione consente di sfruttare simultaneamente più risorse computazionali, come processori multi-core o unità di elaborazione grafica, come le schede video. Ciò permette di ottenere un aumento significativo della velocità di elaborazione, consentendo di completare le operazioni più rapidamente;
- *Maggior capacità di elaborazione*: la parallelizzazione consente di elaborare più immagini contemporaneamente o di suddividere un'immagine in diverse regioni per l'elaborazione parallela. Ciò permette di gestire carichi di lavoro più pesanti o complessi senza sacrificare le prestazioni;
- *Miglioramento dell'efficienza energetica per dispositivi mobili o embeded*: l'utilizzo di più unità di elaborazione in parallelo può ridurre il tempo di esecuzione complessivo di un algoritmo. Di conseguenza, si riduce anche il consumo energetico complessivo, poiché le risorse computazionali sono impiegate in modo più efficiente;
- *Elaborazione in tempo reale*: la parallelizzazione può consentire l'elaborazione in tempo reale di immagini o video, consentendo di ottenere risultati immediati e reattivi;
- *Scalabilità*: la parallelizzazione offre una maggiore scalabilità, consentendo di aumentare le prestazioni dell'elaborazione delle immagini semplicemente aggiungendo più risorse computazionali;

Tuttavia, non tutti gli algoritmi possono essere parallelizzati in modo efficiente. Ci sono alcune situazioni in cui la parallelizzazione potrebbe non essere possibile o non portare a vantaggi significativi:

- *Dipendenze dei dati*: se l'algoritmo richiede l'accesso e l'utilizzo di dati da altre parti dell'immagine o richiede risultati intermedi calcolati in sequenza, potrebbe essere difficile parallelizzarlo efficacemente: tali dipendenze creano una dipendenza sequenziale tra le operazioni, limitando la capacità di eseguire in parallelo;
- *Dimensioni ridotte dell'immagine*: se si lavora con immagini di dimensioni relativamente piccole, la parallelizzazione potrebbe non essere necessaria o addirittura rallentare l'elaborazione;
- *Overhead di creazione e comunicazione dei worker*: la parallelizzazione richiede la gestione della creazione e della comunicazione tra le parti di elaborazione coinvolte, denominati *worker*. Se l'overhead diventa significativo rispetto al beneficio ottenuto dalla parallelizzazione stessa, potrebbe essere più efficiente eseguire l'algoritmo in modo sequenziale;

In generale, è essenziale valutare attentamente le caratteristiche dell'algoritmo, le dimensioni delle immagini, l'hardware disponibile e gli obiettivi prestazionali desiderati per determinare se la parallelizzazione sia adeguata o conveniente nell'ambito specifico dell'image processing.

1.1 Struttura della relazione

Di seguito vengono illustrati gli argomenti trattati in ogni capitolo presente:

- Nel *capitolo 2* vengono introdotti gli strumenti utilizzati per lo sviluppo dell'applicativo denominato *MultiProcess Image Bench*;
- Nel *capitolo 3* viene presentato il software di analisi statica automatica sviluppato;
- Nel *capitolo 4* vengono mostrati alcuni risultati ottenuti;
- Nell'*appendice* vengono riportati i codici di riferimento di alcuni algoritmi dell'image processing;

2

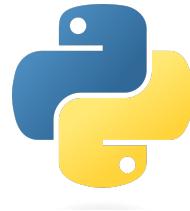
Strumenti Utilizzati

L'applicativo software sviluppato per eseguire benchmark di algoritmi per l'image processing fornisce un ambiente standardizzato per eseguire una serie di test e misurare le prestazioni in modo oggettivo.

Per lo sviluppo di tale applicativo si è fatto affidamento a svariate tecnologie e librerie esterne.

2.1 Python

Python è un *linguaggio di programmazione ad alto livello* con una sintassi semplice e intuitiva. È un linguaggio interpretato con tipizzazione dinamica debole, ovvero la tipizzazione delle variabili avviene durante l'esecuzione del codice. Questo rende Python molto flessibile e facile da utilizzare.



Lo sviluppo di Python ha avuto inizio negli anni '90 da Guido van Rossum e si è rapidamente affermato come uno dei linguaggi di programmazione più popolari, grazie soprattutto alla sua versatilità in svariate campi, tra cui web development, sviluppo di software scientifico, automazione, machine learning, statistica e molto altro ancora.

Uno dei punti di forza di Python è la sua estesa libreria standard, che include molte funzioni già implementate per molte attività comuni, come l'elaborazione di stringhe, la gestione dei file, la connessione a database, il parsing di XML e JSON. Ad affiancare la libreria standard ci sono anche numerosi sviluppatori che hanno creato molti framework e librerie esterne per molte applicazioni specifiche.

In sintesi, Python è un linguaggio di programmazione estremamente potente e versatile che rende facile sviluppare software in molti ambiti diversi. La sua sintassi semplice e intuitiva, la vasta libreria standard e la forte comunità di sviluppatori lo rendono una scelta popolare per molti programmatorei.

2.1.1 Ragioni di preferenza

Nonostante Python non sia un linguaggio orientato alle performance, come ad esempio C++, la decisione dell'utilizzo ricade sulla semplicità e flessibilità delle funzionalità del linguaggio. Ciò ha permesso uno sviluppo veloce e nei limiti tempistici dello studente.

Il software non ha applicazioni pratiche bensì puramente accademici, in quanto la maggior parte delle librerie per image processing fa uso di *binding* verso i linguaggi più performanti come C++.

2.2 Tkinter



Tkinter è una *libreria di interfaccia grafica* (GUI) per Python. Consente agli sviluppatori di creare applicazioni con una grafica interattiva utilizzando un set di *widget* basati sul toolkit *Tcl/tk*. Tkinter è inclusa nella libreria standard di Python, quindi non richiede installazioni aggiuntive, ciò lo rende altamente portabile e può essere eseguito su diverse piattaforme come Windows, macOS e Linux senza necessità di scrivere codice specifico per una piattaforma.

Tkinter offre una vasta gamma di widget predefiniti, come pulsanti, caselle di testo, etichette, menu, finestre di dialogo e molti altri, che semplificano la creazione di interfacce utente. I widget possono essere posizionati e organizzati in frame utilizzando metodi di layout come *pack()*, *grid()* o *place()*, che richiamano rispettivamente layout *assoluto*, *a griglia* o *fluido*.

Un ulteriore punto di forza di Tkinter è la sua integrazione con altri strumenti e librerie di Python, come ad esempio con la popolare libreria *Matplotlib* per visualizzare grafici o con la libreria *Pillow* (*o PIL*) per visualizzare all'interno di un'applicazione Tkinter.

In conclusione, Tkinter è una libreria versatile e potente per lo sviluppo di interfacce grafiche in Python. La sua facilità d'uso, la portabilità, la flessibilità di progettazione, l'integrazione con altri strumenti e il supporto della comunità la rendono una scelta popolare per creare applicazioni con una grafica interattiva.

2.2.1 CustomTkinter

Nonostante Tkinter offre anche opzioni per la personalizzazione dell'aspetto dei widget con cui si possono definire stili, colori, dimensioni dei caratteri e altre proprietà per creare interfacce utente attraenti con facilità, per lo sviluppo dell'applicativo si è preferito utilizzare la libreria *CustomTkinter* sviluppato dall'utente GitHub *TomSchimansky*.

CustomTkinter è una libreria GUI per Python basata su Tkinter. Offre nuovi e moderni widget, completamente personalizzabili. Tali widget possono essere creati ed utilizzati analogamente ai widget predefiniti di Tkinter, possono anche essere utilizzati in combinazione con i widget di quest'ultimo. I widget e il *color-scheme* delle finestre si adattano all'aspetto del sistema in cui viene eseguito, ma è anche possibile impostare manualmente la modalità *chiaro* o *scuro*.

CustomTkinter supporta la scalatura HighDPI, permettendo di avere un aspetto coerente e moderno anche su sistemi ad alta risoluzione.

2.3 Pillow

Pillow è una libreria Python per l'elaborazione delle immagini, sviluppata come fork di *PIL (Python Imaging Library)*. Grazie alla sua vasta gamma di funzionalità è diventata la libreria di fatto per lavorare con immagini in Python.

La libreria fornisce un'interfaccia supporta immagini in diversi formati, tra cui JPEG, PNG, BMP, TIFF e molti altri. Può facilmente aprire un'immagine da file o sequenza binaria, modificarla, applicare filtri, ridimensionarla, ritagliarla, aggiungere testo e molto altro ancora.



Offre un'ampia gamma di metodi e funzioni per eseguire operazioni di manipolazione delle immagini: è possibile accedere ai singoli pixel dell'immagine, modificarli direttamente o utilizzare funzioni predefinite e ottimizzate per applicare effetti e trasformazioni tramite l'utilizzo dell'interfaccia *ImageFilter*.

In conclusione, Pillow è anche compatibile con NumPy, una libreria molto diffusa per il calcolo scientifico in Python. Questa integrazione consente di convertire agevolmente le immagini Pillow in array NumPy e viceversa, aprendo così la strada a numerose possibilità di elaborazione delle immagini utilizzando strumenti matematici avanzati.

2.4 NumPy



NumPy è una libreria Python ampiamente utilizzata per il *calcolo scientifico e numerico*. Offre un potente oggetto array multidimensionale chiamato *ndarray*, che consente di gestire grandi quantità di dati in modo efficiente.

Permette di eseguire *operazioni matematiche e algebriche su array*, come addizione, sottrazione, moltiplicazione e divisione, sia tra singoli elementi che tra interi array. Tali operazioni vengono eseguite in modo vettoria-

le, ovvero vengono applicate simultaneamente a tutti gli elementi dell'array, rendendo il calcolo molto efficiente.

Oltre alle operazioni di base, NumPy offre anche una vasta gamma di *funzioni matematiche avanzate*, come funzioni trigonometriche, esponenziali, logaritmiche e statistiche.

L'efficienza di NumPy è dovuta al fatto che gli array NumPy sono implementati in linguaggio C, il che consente di sfruttare al massimo la velocità di esecuzione.

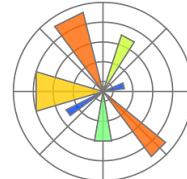
Complessivamente, NumPy è uno strumento fondamentale per il calcolo scientifico e numerico in Python, offrendo un'implementazione efficiente di array multidimensionali e una vasta gamma di funzioni matematiche.

2.4.1 Restrizioni nell'utilizzo

Per evitare uno sbilanciamento dei risultati, causato dall'implementazione in C degli ndarray, all'interno del progetto l'utilizzo della libreria NumPy è stato limitato alle *attività di minor rilievo* come ad esempio la generazione di un kernel Gaussiano o ricerca all'interno di matrici.

2.5 Matplotlib

Matplotlib è una libreria Python ampiamente utilizzata per la *visualizzazione di dati e la creazione di grafici*. Fornisce un'interfaccia intuitiva per creare una vasta gamma di grafici, inclusi *grafici a linee*, *grafici a dispersione*, *istogrammi*, *grafici a barre*, *grafici a torta* e molti altri.



Permette di personalizzare svariati aspetti dei grafici, come colori, stili di linea, etichette degli assi e titoli.

Un punto di forza della libreria è la sua integrazione con NumPy, precedentemente introdotta. È possibile utilizzare gli array NumPy come dati di input per generare grafici, semplificando notevolmente il processo di visualizzazione dei dati.

In conclusione, Matplotlib è una potente libreria di visualizzazione dei dati in Python, la cui flessibilità, facilità d'uso e integrazione con NumPy la rendono uno strumento essenziale per gli scienziati, gli ingegneri e i data scientist.

2.6 GIL e Multiprocessing



Il *GIL* (*Global Interpreter Lock*) è un meccanismo di gestione dei thread utilizzato nell'implementazione standard di CPython, l'interprete di Python di riferimento, allo scopo di consentire un'interazione più sicura con gli oggetti Python.

Il GIL è un *vincolo* che permette a solo un thread alla volta di eseguire istruzioni Python, **anche su sistemi con processori multi-core**. Questo significa che, nonostante si possano utilizzare thread multipli, essi non possono eseguire istruzioni simultaneamente: quando un thread acquisisce il GIL, gli altri thread devono attendere il loro turno per eseguire codice Python.

Rappresenta un *limite sulle prestazioni* in alcune situazioni specifiche, in particolare quando si tenta di applicare una parallelizzazione al fine di sfruttare appieno le capacità di elaborazione parallela di un sistema multi-core.

Il modulo *multiprocessing*, incluso nelle librerie standard di Python, offre una soluzione per superare le limitazioni del GIL. Fornisce un'interfaccia per la *creazione di processi multipli*, che possono eseguire codice Python **in modo indipendente** e sfruttare i vantaggi del parallelismo effettivo su sistemi multi-core.

Utilizzando multiprocessing, è possibile suddividere un compito in più processi che vengono eseguiti in parallelo. Tali processi comunicano tra loro attraverso la condivisione di dati o l'invio di messaggi tramite code o pipe.

Tuttavia, l'utilizzo di multiprocessing non è privo di svantaggi:

- Se il lavoro da eseguire in ogni processo è relativamente semplice e richiede poco tempo di calcolo, il costo aggiuntivo di creare e gestire i processi potrebbe superare i benefici del parallelismo;
- Se i processi devono comunicare o dipendere l'uno dall'altro in modo significativo durante l'esecuzione, l'utilizzo di multiprocessing può introdurre complessità aggiuntiva;
- Ogni processo creato con multiprocessing ha il proprio spazio di memoria separato. Se il processo richiede una grande quantità di dati da elaborare e mantenere in memoria, l'utilizzo di processi multipli può portare a un consumo eccessivo di memoria;
- L'invio di dati tra processi richiede la *serializzazione* e la *deserializzazione* degli oggetti utilizzando *pickle*. Questa conversione non supporta tutti i tipi di dati e introduce un certo overhead, soprattutto per oggetti complessi o di grandi dimensioni;

3

MultiProcess Image Processing Bench

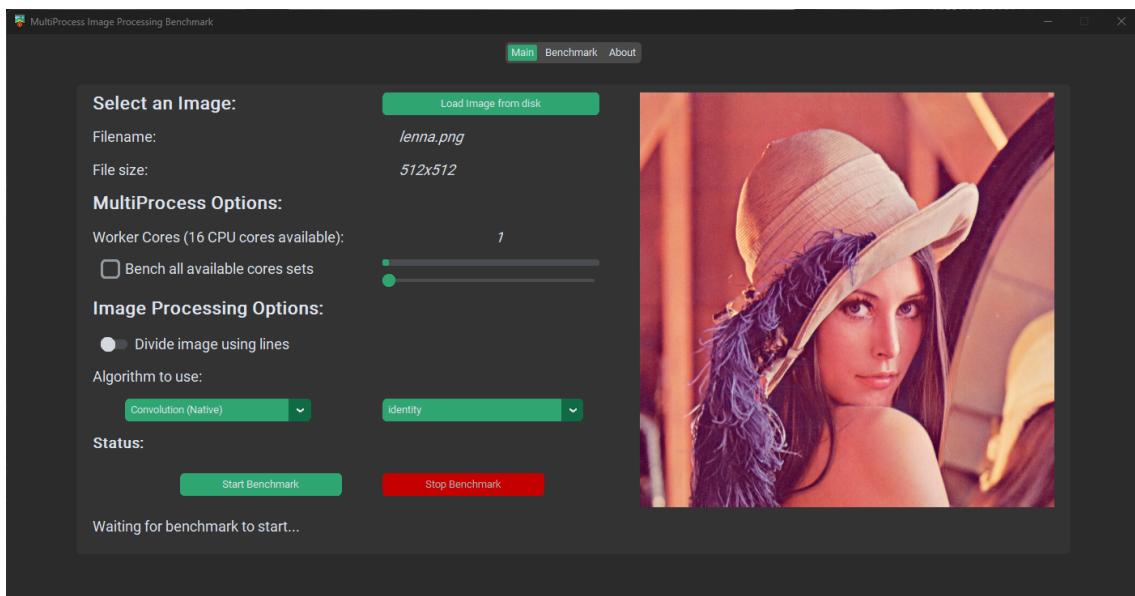


Figura 3.1: Schermata principale dell'applicativo

L'applicativo software sviluppato è denominato "*MultiProcess Image Processing Bench*", a cui, per semplicità, sarà fatto riferimento tramite l'abbreviazione "*Benchmarker*".

Benchmarker è un programma dotato di interfaccia grafica sviluppato in Python 3 con Tkinter, e quindi supportato su MacOS, Linux e Windows. L'applicativo prende in input un'immagine di dimensioni quadrate, ridimensionandola se necessario, esegue un algoritmo di image processing secondo una configurazione utente e restituisce i tempi di esecuzione sotto forma di grafico a barre laterali.

Lo scopo di *Benchmarker* è rendere facile ed automatico il confronto delle prestazioni di un certo algoritmo di image processing eseguito su diverse configurazioni di parallelismo.

3.1 Uso del programma

Benchmark fa uso di librerie esterne, come ad esempio *pillow* e *matplotlib*. Per questo motivo, prima di poter utilizzare il programma è necessario installare tramite *"pip"*, il package installer di Python, le dipendenze in uno dei seguenti modi:

- *Global Package*: installare le dipendenze direttamente nel sistema, diventando di fatto globali e accessibili da qualunque script Python eseguito nel sistema;
- *Virtual Environment*: creare un ambiente virtuale in cui installare le dipendenze;

Una volta installate le dipendenze, per utilizzare Benchmark, e quindi visualizzare l’interfaccia grafica, è necessario avviare lo script *main.py*.

Una volta aperto il programma è possibile:

1. caricare l’immagine su cui eseguire gli algoritmi, qualora l’immagine non fosse di *dimensioni quadrate* verrà chiesto di ridimensionarla;
2. scegliere il *quantitativo di core* da utilizzare per l’*esecuzione parallela*, il *numero totale di core* dipende dal processore del calcolatore in cui si sta eseguendo l’applicativo;
3. selezionare la *modalità di suddivisione* in **linee o quadrati** dell’immagine originale, qualora si scelga la suddivisione in quadrati, il quantitativo di core dovrà essere un *quadrato perfetto*;
4. scegliere se provare o meno tutte le possibili configurazioni parallelo eseguibili nel calcolatore;
5. selezionare la *tipologia di algoritmo*, ed eventualmente l’algoritmo specifico, inserendo, se richiesti, i parametri dello stesso;
6. *avviare il benchmark*;

Una volta avviato il benchmark, verrà prima eseguita un’*esecuzione sequenziale* dell’algoritmo e successivamente la/le *configurazioni parallele* scelte dall’utente. I risultati verranno visualizzati nell’apposita finestra.

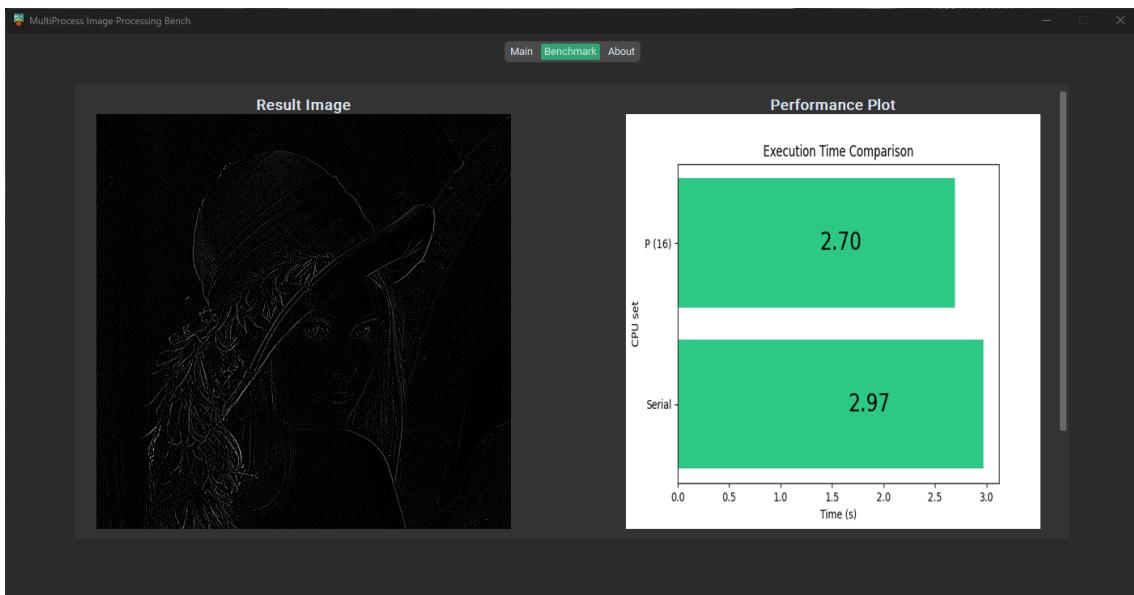


Figura 3.2: Esempio di risultati del benchmark

3.2 Suddivisione dell'immagine

Alla base del parallelismo vi è il concetto di *suddividere un grande problema in sotto problemi più piccoli della stessa natura*. Per quanto riguarda l'image processing tale concetto viene applicato suddividendo un'immagine in *sotto immagini*, ovvero immagini che se riunite ricostruiscono l'immagine originale. Per tale operazione Benchmark mette a disposizione due modalità:

- *Suddivisione in strisce, o linee*: l'immagine originale viene suddivise in N linee quanti sono i core selezionati per l'esecuzione parallela, ogni processo parallelo eseguirà l'algoritmo sull'immagine parziale assegnatogli e restituirà il risultato parziale, che verrà unito agli altri per formare il risultato completo;
 - la *lunghezza della striscia* viene calcolata semplicemente dividendo il lato del quadrato per N , numero di linee;
- *Suddivisione in quadrati*: l'immagine originale viene suddivise in $M = \sqrt{N}$ quadrati, dove N è il numero di core selezionati per l'esecuzione parallela, ogni processo parallelo eseguirà l'algoritmo sull'immagine parziale assegnatogli e restituirà il risultato parziale, che verrà unito agli altri per formare il risultato completo;
 - la *lunghezza del quadrato* viene calcolata semplicemente dividendo il lato del quadrato per M , numero di quadrati;

- il *numero di core* selezionato per l'esecuzione parallela deve essere un *quadrato perfetto*;

La logica per la suddivisione implementata in Python è presente in appendice: A.1.



Figura 3.3: Esempio di suddivisione

3.3 Algoritmi implementati

All'interno di Benchmarker sono stati implementati alcuni dei più famosi algoritmi dell'image processing.

Per scelta di stile e coerenza dei risultati, tali algoritmi, come convoluzione e operatori morfologici, sono implementati facendo uso delle funzionalità native di Python senza l'utilizzo delle funzioni avanzate offerte dalle librerie, in quanto implementate con un riguardo alle prestazioni in C.

3.3.1 Convoluzione

La *convoluzione* è un'operazione fondamentale nell'elaborazione delle immagini utilizzata per analizzare e modificare le caratteristiche di un'immagine. In sintesi, la convoluzione consiste nel sovrapporre un *kernel* su un'immagine e calcolare la somma dei prodotti degli elementi sovrapposti del kernel e dei pixel corrispondenti dell'immagine.

Il *kernel* è una piccola matrice di valori numerici, di *dimensioni generalmente dispari*, che definisce come combinare i pixel circostanti durante l'operazione di convoluzione. Ogni elemento del kernel viene moltiplicato con il valore corrispondente del pixel nell'immagine, e i risultati vengono sommati per ottenere il valore convoluto per il pixel centrale.

Questo processo viene ripetuto per ogni pixel dell'immagine.

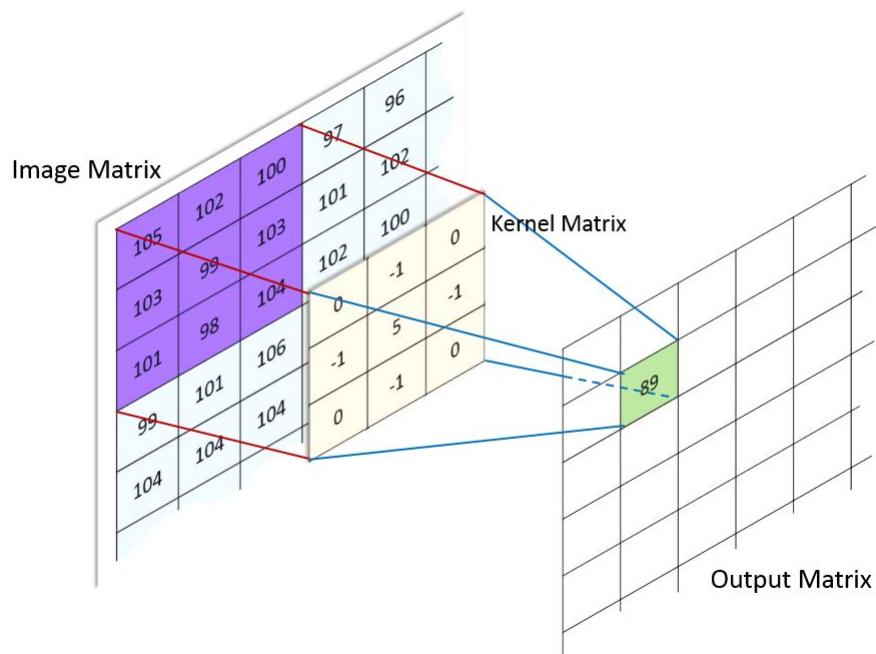


Figura 3.4: Schema della convoluzione

La convoluzione viene spesso utilizzata per applicare filtri e operazioni di modifica dell'immagine che richiedono kernel specifici, ideati per ottenere un determinato effetto.

La logica della convoluzione implementata in Python è presente in appendice: A.5

Identity

Il kernel *Identity* rappresenta il più semplice filtro convolutivo: non effettua alcuna modifica all'immagine di input.

$$\text{Identity Kernel} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Quando viene applicato a un'immagine tramite la convoluzione, mantiene inalterati i valori dei pixel. Ciò significa che il pixel di output nella posizione corrispondente al pixel di input sarà uguale al valore del pixel di input stesso. In pratica, l'immagine convoluta con il kernel identità sarà identica all'immagine di input.

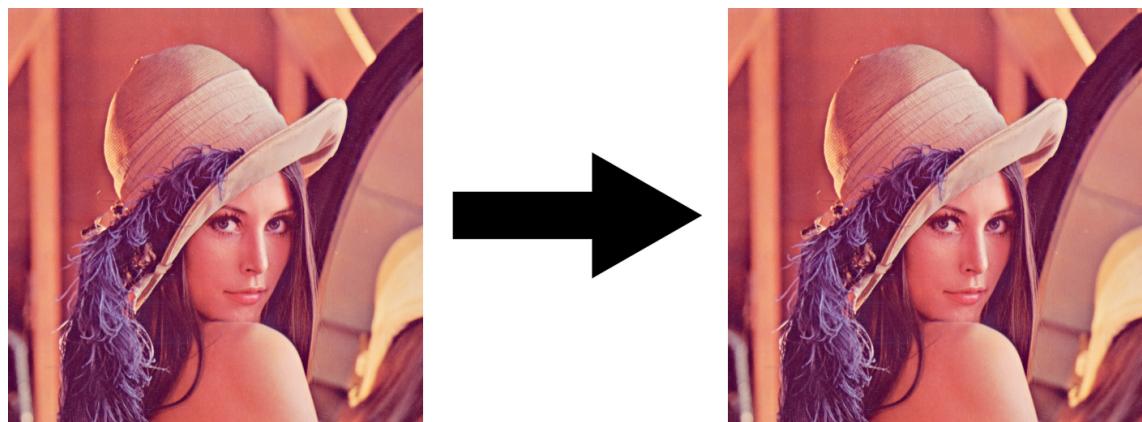


Figura 3.5: Esempio di Kernel Identity

Sobel-X e Sobel-Y

I kernel di *Sobel-X* e *Sobel-Y* sono utilizzati per il rilevamento dei bordi. In particolare, *Sobel-X* evidenzia i cambiamenti di intensità orizzontali, mentre *Sobel-Y* quelli verticali.

$$\text{Sobel-X} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 1 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad \text{Sobel-Y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix},$$

Quando vengono applicati a un'immagine tramite la convoluzione, calcolano le derivate parziali rispetto alle direzioni orizzontali e verticali, rispettivamente. Questo significa rilevano le variazioni di intensità lungo le linee orizzontali e verticali dell'immagine.



Figura 3.6: Esempio di Kernel Sobel-X



Figura 3.7: Esempio di Kernel Sobel-Y

Prewitt-X e Prewitt-Y

Simili ai kernel Sobel, i kernel di *Prewitt-X* e *Prewitt-Y* sono utilizzati per il rilevamento dei bordi. In particolare, *Prewitt-X* evidenzia i cambiamenti di intensità orizzontali, mentre *Prewitt-Y* quelli verticali.

$$\text{Prewitt-X} = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, \quad \text{Prewitt-Y} = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix},$$

Quando vengono applicati a un'immagine tramite la convoluzione, calcolano le derivate parziali rispetto alle direzioni orizzontali e verticali, rispettivamente. Questo significa rilevano le variazioni di intensità lungo le linee orizzontali e verticali dell'immagine.

Rispetto ai kernel Sobel, producono un immagine meno nitida.



Figura 3.8: Esempio di Kernel Prewitt-X



Figura 3.9: Esempio di Kernel Prewitt-Y

Laplacian

Il kernel di *Laplacian* è utilizzato per il rilevamento dei bordi o dei punti di discontinuità.

$$\text{Laplacian} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Quando viene applicato a un'immagine tramite la convoluzione, produce una mappa dei gradienti del secondo ordine, che rappresenta la variazione dell'intensità dei pixel nell'immagine, generando un'immagine convoluta che evidenzia i bordi, le linee e altre caratteristiche forti dell'immagine.

Tuttavia, può anche produrre una risposta forte al rumore presente nell'immagine, poiché amplifica le variazioni di intensità in tutte le direzioni.



Figura 3.10: Esempio di Kernel Laplacian

Sharpen e High-Pass

I kernel di *Sharpen* e *High-Pass* sono utilizzati per enfatizzare le componenti ad alta frequenza e attenuare quelle a bassa frequenza producendo un miglioramento dei dettagli di un'immagine.

$$\text{Sharpen} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}, \quad \text{High-Pass} = \begin{bmatrix} 0 & -1/4 & 0 \\ -1/4 & 2 & -1/4 \\ 0 & -1/4 & 0 \end{bmatrix}$$

Quando vengono applicati a un'immagine tramite la convoluzione, producono un'immagine convoluta in cui i bordi e i dettagli sono enfatizzati, migliorando il contrasto tra i pixel adiacenti e accentuando le transizioni di intensità.

Questo può portare ad un aspetto più definito e nitido dell'immagine.

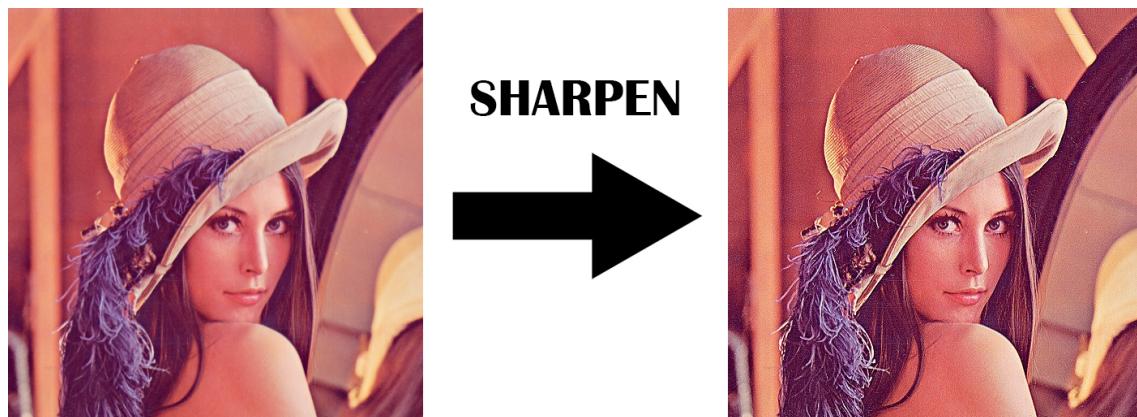


Figura 3.11: Esempio di Kernel Sharpen



Figura 3.12: Esempio di Kernel High-Pass

Blur (Low-Pass)

Il kernel di *Low-Pass* è utilizzato per attenuare le componenti ad alta frequenza e mantiene le componenti a bassa frequenza, consentendo di eliminare il rumore e ottenere un'immagine più morbida.

$$\text{Low-Pass} = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

Quando viene applicato a un'immagine tramite la convoluzione, produce un'immagine convoluta in cui le transizioni brusche dell'intensità vengono attenuate e l'immagine risulta più sfocata.

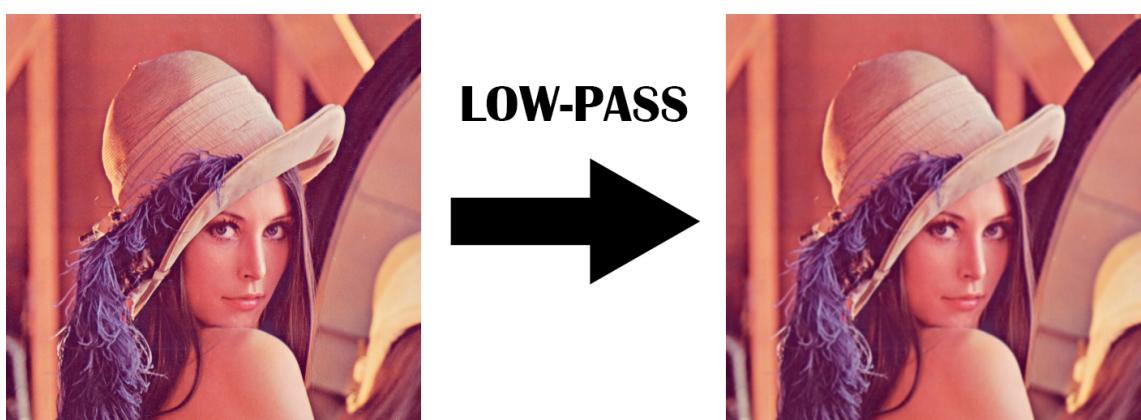


Figura 3.13: Esempio di Kernel Low-Pass

Blur (Gaussian)

Il kernel di *Gaussian* è utilizzato per eseguire una riduzione del rumore e una sfocatura dell'immagine

$$\text{Gaussian } 3 \times 3 = \begin{bmatrix} 1/16 & 2/16 & 1/16 \\ 2/16 & 4/16 & 2/16 \\ 1/16 & 2/16 & 1/16 \end{bmatrix}$$

Quando viene applicato a un'immagine tramite la convoluzione, produce un effetto di sfocatura, riducendo le variazioni di intensità tra i pixel e attenuando il rumore ad alta frequenza.

L'uso del kernel gaussiano permette di ottenere una sfocatura più naturale e graduale rispetto ad altri filtri di sfocatura più semplici, poiché tiene conto delle distribuzioni di intensità circostanti, rendendo la transizione da un pixel all'altro più morbida e realistica.

Fornendo una dimensione e un valore sigma, Benchmark è in grado di generare un kernel di Gaussian che rispetti i parametri forniti.

La logica per la generazione di un kernel gaussiano di una certa *dimensione* con un certo *sigma* implementata in Python è presente in appendice: A.6



Figura 3.14: Esempio di Kernel Gaussian

3.3.2 Operatori Morfologici

Gli *operatori morfologici* sono utilizzati per manipolare la forma, la struttura e le caratteristiche di un'immagine. Queste operazioni si basano su principi matematici che coinvolgono l'interazione tra un *elemento strutturante* e i pixel dell'immagine.

Gli *elementi strutturanti*, anche chiamati *kernel di forma*, sono matrici che definiscono la forma e la dimensione dell'area circostante a ciascun pixel durante l'applicazione

degli operatori morfologici. Questi elementi strutturanti svolgono un ruolo cruciale nel determinare l'effetto dell'operazione morfologica sull'immagine.

Benchmarker implementa e permette la selezione dei seguenti elementi strutturali:

$$\text{Square} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \text{Cross} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{Line} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{Circle} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Benchmarker implementa gli operatori morfologici di erosione e dilatazione, i cui codici sono presenti in appendice: A.7.

Erosione

L'erosione è un'operazione morfologica che riduce le regioni o i dettagli di un'immagine. Durante l'erosione, ogni pixel nell'immagine viene confrontato con il corrispondente sotto l'elemento strutturante. Se tutti i pixel sottostanti l'elemento strutturante sono *attivi*, cioè hanno un valore di intensità alto, il pixel corrispondente nell'immagine risultante viene mantenuto attivo. In caso contrario, il pixel viene disattivato o eliminato. L'erosione tende a ridurre la dimensione degli oggetti e a rimuovere piccoli dettagli, consentendo di separare le regioni connesse.

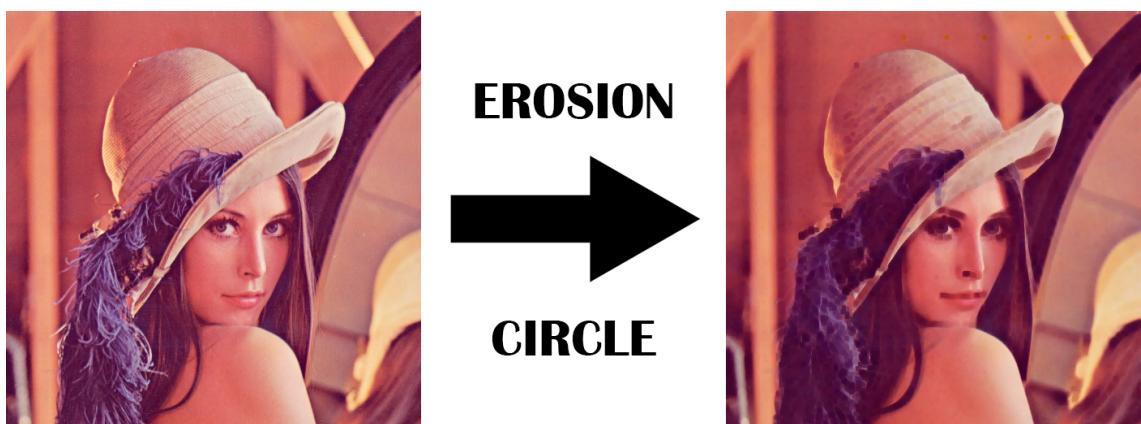


Figura 3.15: Esempio Erosione con elemento strutturante Circle

Dilatazione

La dilatazione è un'altra operazione morfologica che ha l'effetto opposto dell'erosione. Durante la dilatazione, ogni pixel nell'immagine viene confrontato con il corrispondente

sotto l'elemento strutturante. Se almeno un pixel sottostante l'elemento strutturante è *attivo*, il pixel corrispondente nell'immagine risultante viene attivato o mantenuto attivo. La dilatazione tende ad espandere le regioni e a unire gli oggetti vicini.

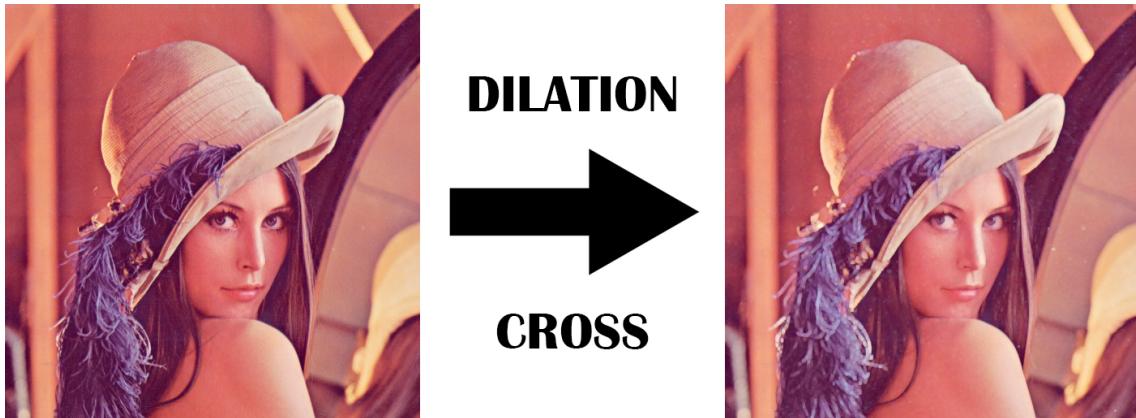


Figura 3.16: Esempio Dilatazione con elemento strutturante Cross

3.3.3 Filtri di Noise Reduction

I filtri di *noise reduction*, o *filtri di riduzione del rumore*, sono utilizzati per mitigare o eliminare il rumore presente nelle immagini digitali. Il rumore può essere causato da vari fattori, come le imperfezioni del sensore, le interferenze elettriche o il processo di acquisizione dell'immagine.

L'obiettivo principale è quello di migliorare la qualità dell'immagine, rimuovendo o riducendo il rumore senza influire eccessivamente sulla nitidezza e sui dettagli importanti dell'immagine.

Benchmark implementa il *mean filter* e il *bilateral filter*, i cui codici sono presenti in appendice: A.9.

Mean Filter

Il *filtro di media*, detto anche *mean filter*, è uno dei filtri di noise reduction più semplici ed efficaci. Sostituisce il valore di ciascun pixel nell'immagine con la media dei valori dei pixel nell'intorno del pixel stesso.

In pratica, il filtro di media calcola la media aritmetica dei valori dei pixel all'interno di una finestra quadrata e sostituisce il valore del pixel corrente con questa media.

Questo processo aiuta a ridurre il rumore, poiché il rumore casuale tende ad avere un effetto minore sulla media dei valori dei pixel.

La dimensione dell'intorno viene decisa dall'utente al momento della selezione del filtro di media.

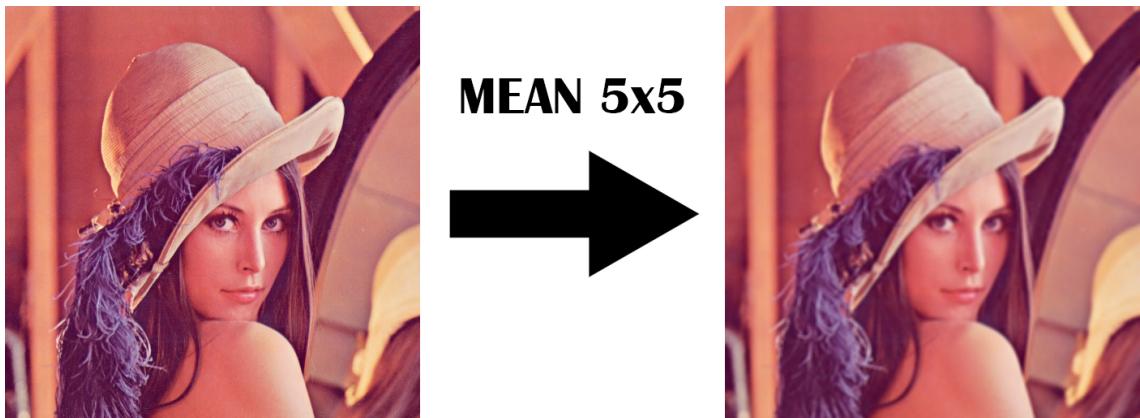


Figura 3.17: Esempio di filtro di media 5x5

Bilateral Filter

Il *filtro bilaterale*, detto anche *bilateral filter* offre una maggiore flessibilità nel mantenere i confini e i dettagli dell'immagine.

A differenza del filtro di media, il filtro bilaterale considera non solo i valori dei pixel nell'intorno, ma anche le differenze di intensità tra i pixel.

Questo filtro calcola una media ponderata dei valori dei pixel all'interno di una finestra, in base alla distanza spaziale tra i pixel e alla differenza di intensità tra i pixel.

Ciò significa che i pixel che sono simili in termini di intensità e vicini spazialmente avranno un peso maggiore nella media ponderata, mentre i pixel con differenze significative di intensità avranno un peso minore. In questo modo, il filtro bilaterale può ridurre il rumore, preservando al contempo i dettagli e i confini dell'immagine.

La dimensione dell'intorno, distanza spaziale e la differenza d'intensità vengono decisi dall'utente al momento della selezione del filtro bilaterale.

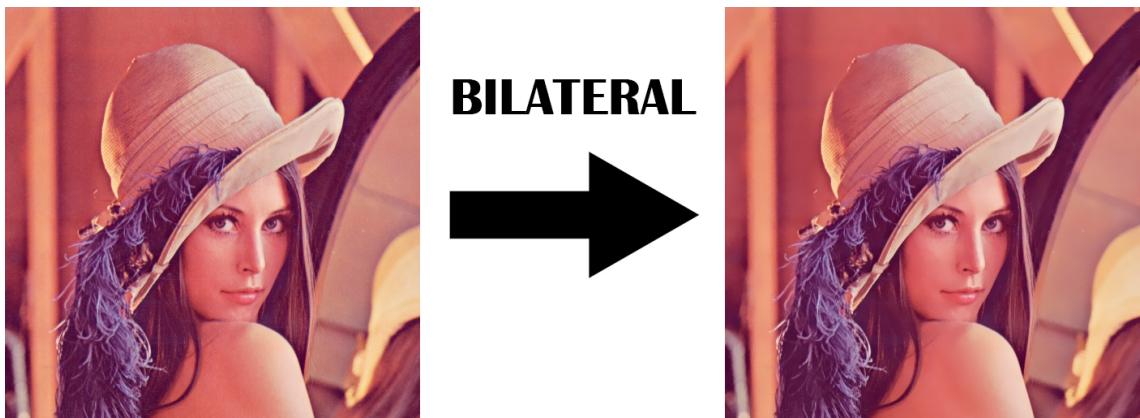


Figura 3.18: Esempio di filtro bilaterale

3.3.4 Canny

Il *filtro di Canny*, chiamato anche *algoritmo di Canny*, è un popolare filtro di *edge detection*. È un algoritmo composto da più passaggi che combinati insieme forniscono una buona precisione nel rilevamento dei bordi, riducendo al contempo il rumore.

L'algoritmo di Canny opera secondo i seguenti passaggi:

- *Riduzione del rumore*: l'immagine di input viene filtrata per ridurre il rumore. Viene applicato un filtro di *smoothing*, come ad esempio il *filtro di Gaussian* per quanto riguarda Benchmark, per sfocare leggermente l'immagine e ridurre il rumore indesiderato;
- *Calcolo del gradiente*: viene calcolato il gradiente dell'immagine utilizzando *operatori di differenza di intensità*, come ad esempio il filtro di Sobel. Il gradiente rappresenta la variazione di intensità dei pixel nell'immagine e fornisce informazioni sulla direzione e l'intensità dei cambiamenti di intensità;
- *Suppressione dei non-maxima*: viene eseguita una soppressione dei non-maxima per sottolineare solo i punti di massima intensità lungo i bordi rilevati. Questo passaggio aiuta a sottolineare i bordi sottili mantenendo solo i punti di massima intensità lungo le direzioni dei gradienti;
- *Linking dei bordi*: viene applicata una tecnica chiamata *isteresi* per collegare i bordi rimanenti. Questo passaggio coinvolge l'impostazione di due soglie, una soglia inferiore e una soglia superiore. I pixel con intensità superiore alla soglia superiore vengono considerati come *bordi forti*, mentre i pixel con intensità tra la soglia inferiore e la soglia superiore vengono considerati come *bordi deboli*. I bordi deboli vengono mantenuti solo se sono collegati a bordi forti. In questo modo, si cerca di conservare solo i bordi rilevanti e ridurre l'inclusione di rumore;

Il risultato finale dell'algoritmo di Canny è un'*immagine binaria* che rappresenta i bordi individuati nell'immagine di input. I bordi sono rappresentati come linee sottili che corrispondono ai confini tra regioni di intensità diversa.

La *dimensione del kernel* e il valore *sigma* del filtro di Gaussian e la *soglie inferiore e superiore* vengono decisi dall'utente al momento della selezione del filtro di Canny.

La logica implementativa di Canny è presente in appendice: A.11.

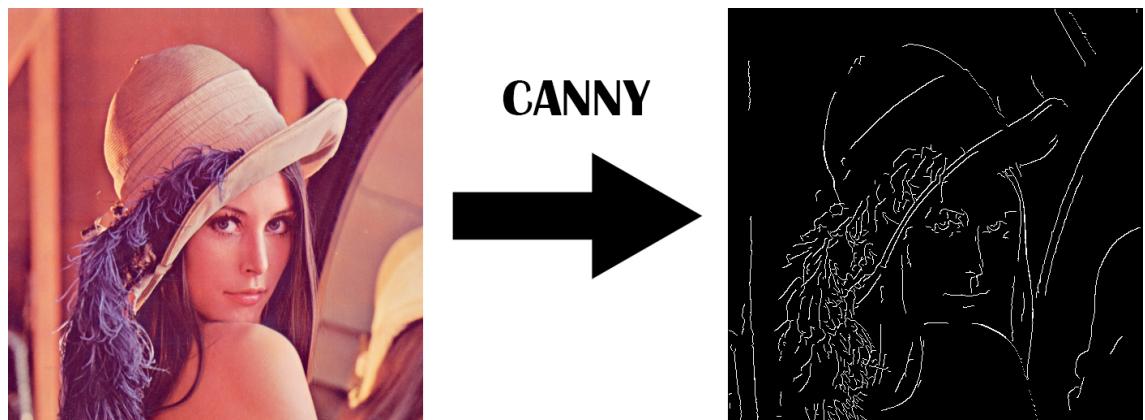


Figura 3.19: Esempio di Canny Edge Detection

4

Risultati ottenuti

Sono state eseguite numerose prove che hanno *confutato* quanto detto fino ad ora riguardo i vantaggi e problemi della parallelizzazione.

In caso di immagine di *dimensione troppo piccola*, i guadagni di performance vengono meno a causa dell'*overhead* di creazione e comunicazione dei processi. L'esecuzione sequenziale ha sempre la meglio.

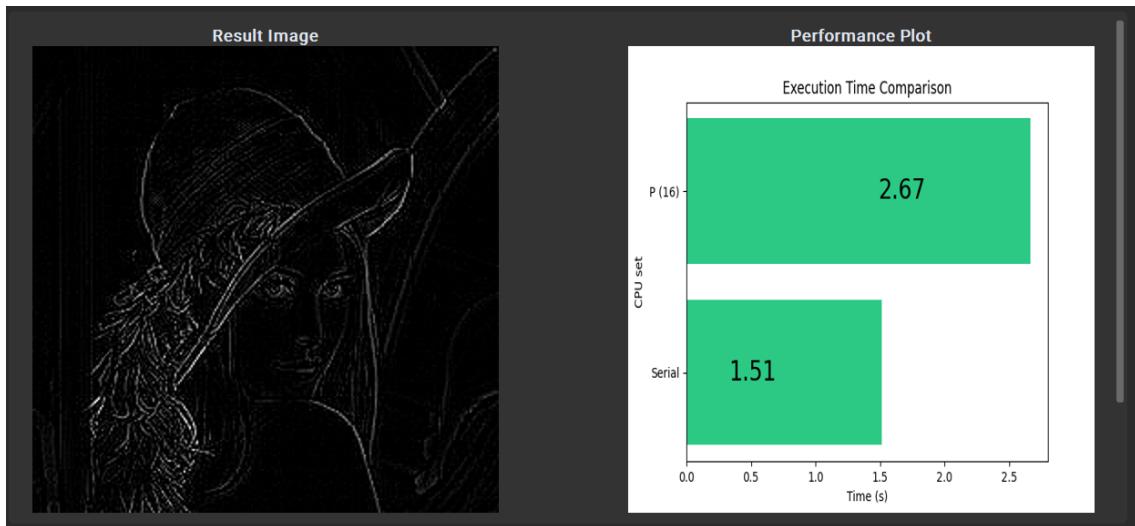


Figura 4.1: Esempio di parallelizzazione Lenna 256x256

In caso di immagine di *dimensioni medie*, si hanno guadagni di performance con suddivisioni di piccolo taglio, tali guadagni vengono prima attenuati e successivamente persi man mano che aumentano le suddivisioni, sempre a causa dell'*overhead* di creazione e comunicazione dei processi.

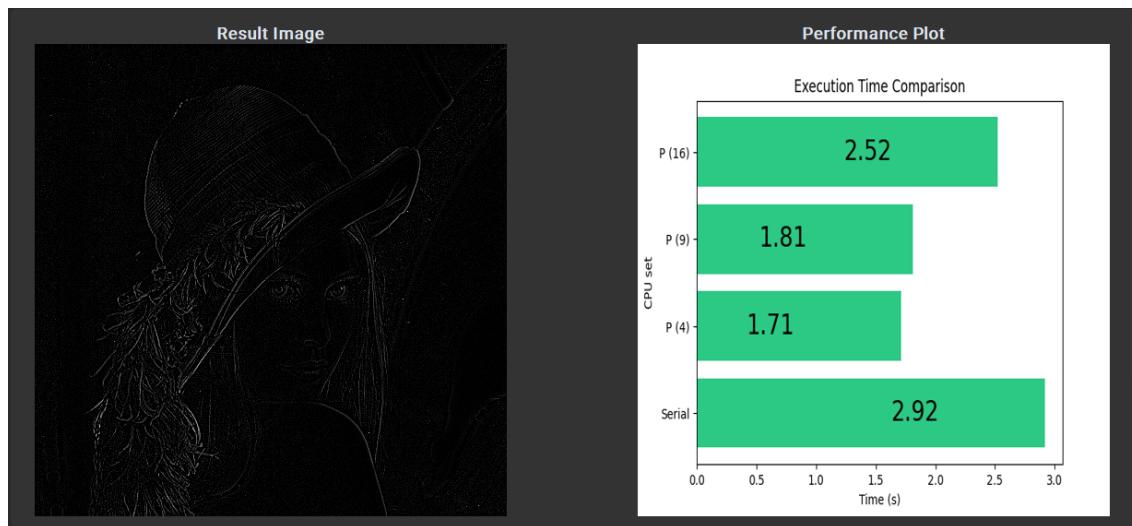


Figura 4.2: Esempio di parallelizzazione Lenna 512x512

In caso di immagine di *dimensioni grandi*, si hanno guadagni di performance evidenti. L'*overhead* di creazione e comunicazione dei processi diventa meno trascurabile all'aumentare delle suddivisioni.

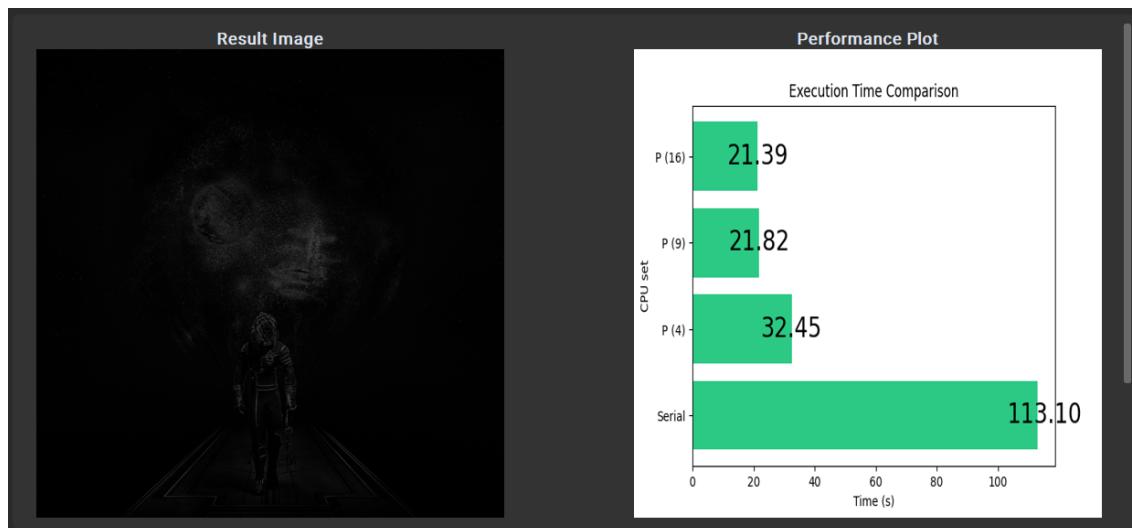


Figura 4.3: Esempio di parallelizzazione Prey 4000x4000

Infine, si è tentato un confronto fra la convoluzione implementata a mano e quella offerta dalla libreria *SciPy*, il risultato è alquanto prevedibile in quanto tale libreria fa uso di implementazioni in C. In tale configurazione il collo di bottiglia è rappresentato dalla tecnica di parallelizzazione di Python che degrada le performance offerte dal C.

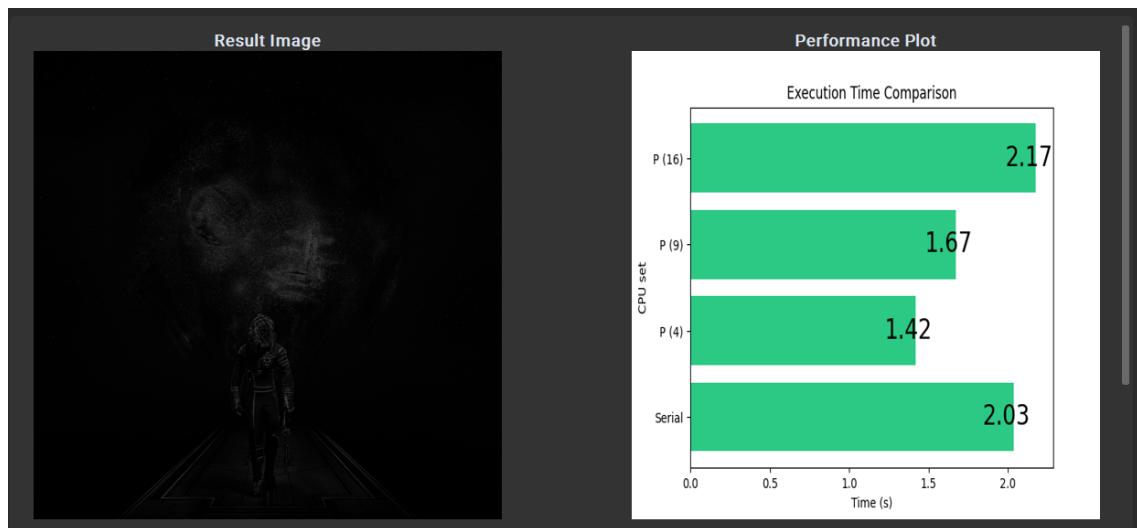


Figura 4.4: Esempio di parallelizzazione SciPy Prey 4000x4000

A

Appendice

Codice Python

Suddivisione in linee

```
1 def _line_image_sub_divider(self) -> list[Image.Image]:
2 """
3 Divide the image using lines.
4 :return: A list of the lines that the image is divided into.
5 """
6     tmp_image: Image.Image = self._target_image.copy()
7     if self._target_cpu_core_set == 1:
8         return [tmp_image]
9     sub_images: list[Image.Image] = []
10    line_width = tmp_image.width // self._target_cpu_core_set
11    for i in range(self._target_cpu_core_set):
12        left = i * line_width
13        right = left + line_width
14        sub_images.append(tmp_image.crop((left, 0, right, tmp_image.height)))
15    return sub_images
```

Codice A.1: Codice per la suddivisione in linee

```
1 def _merge_image_lines(self, lines: list[Image.Image]) -> Image.Image:
2 """
3 Merge the lines of an image into one image.
4 :param lines: The lines to merge.
5 :return: The merged image.
6 """
7     if len(lines) == 1:
8         return lines[0]
9     tmp_image = Image.new("RGBA", self._target_image.size)
10    line_width = tmp_image.width // self._target_cpu_core_set
11    for i in range(self._target_cpu_core_set):
12        left = i * line_width
13        right = left + line_width
14        tmp_image.paste(lines[i], (left, 0, right, tmp_image.height))
15    return tmp_image
```

Codice A.2: Codice per l'unione delle linee

Suddivisione in quadrati

```

1 def _square_image_sub_divider(self) -> list[Image.Image]:
2 """
3 Divide the image using squares.
4 :return: A list of the squares that the image is divided into.
5 """
6     tmp_image: Image.Image = self._target_image.copy()
7     if self._target_cpu_core_set == 1:
8         return [tmp_image]
9     else:
10        num_parts: int = int(self._target_cpu_core_set ** 0.5)
11        square_size: int = tmp_image.width // num_parts
12        sub_images: list[Image.Image] = []
13        for row in range(num_parts):
14            for column in range(num_parts):
15                x_start: int = column * square_size
16                y_start: int = row * square_size
17                x_end: int = x_start + square_size
18                y_end: int = y_start + square_size
19                sub_images.append(tmp_image.crop((x_start, y_start, x_end, y_end)))
20
return sub_images

```

Codice A.3: Codice per la suddivisione in quadrati

```

1 def _merge_image_squares(self, squares: list[Image.Image]) -> Image.Image:
2 """
3 Merge the squares of an image into one image.
4 :param squares: The squares to merge.
5 :return: The merged image.
6 """
7     if len(squares) == 1:
8         return squares[0]
9     tmp_image = Image.new("RGBA", self._target_image.size)
10    square_size: int = squares[0].width
11    iterator: int = int(len(squares) ** 0.5)
12    for row in range(iterator):
13        for column in range(iterator):
14            x_start: int = column * square_size
15            y_start: int = row * square_size
16            x_end: int = x_start + square_size
17            y_end: int = y_start + square_size
18            tmp_image.paste(squares[row * iterator + column], (x_start, y_start, x_end, y_end))
19
return tmp_image

```

Codice A.4: Codice per l'unione dei quadrati

Convoluzione

```

1 def convolve_native(target_image: Image.Image, kernel: list[list[float]], gray_scale:
2     bool = False) -> Image.Image:
3 """
4     Convolves the target image with the given kernel.
5     :param target_image: The image to convolve.
6     :param kernel: The kernel to convolve with.
7     :param gray_scale: Whether to convert the image to grayscale before convolving.
8     :return: The convolved image.
9 """
10    img_type: str = "L" if gray_scale else "RGB"
11    image_width, image_height = target_image.size
12    kernel_size: int = len(kernel)
13    padding: int = int(kernel_size / 2)
14    result_image: Image.Image = Image.new(img_type, (image_width, image_height))
15    target_image = target_image.convert(img_type)
16    target_pixels = target_image.load()
17    result_pixels = result_image.load()
18    for y in range(image_height):
19        for x in range(image_width):
20            x_start: int = x - padding
21            x_end: int = x + padding + 1
22            y_start: int = y - padding
23            y_end: int = y + padding + 1
24            conv_pixel: list[float, float, float] | float = 0.0 if gray_scale else [0, 0, 0]
25            iterator: int = 0
26            for ky in range(y_start, y_end):
27                for kx in range(x_start, x_end):
28                    px = max(0, min(kx, image_width - 1))
29                    py = max(0, min(ky, image_height - 1))
30                    pixel = target_pixels[px, py]
31                    kernel_value = kernel[int(iterator / kernel_size)][iterator % kernel_size]
32                    if gray_scale:
33                        conv_pixel += pixel * kernel_value
34                    else:
35                        conv_pixel[0] += pixel[0] * kernel_value
36                        conv_pixel[1] += pixel[1] * kernel_value
37                        conv_pixel[2] += pixel[2] * kernel_value
38                    iterator += 1
39            normalized: tuple[int, int, int] | int
40            if gray_scale:
41                normalized: int = max(0, min(255, int(conv_pixel)))
42            else:
43                normalized: tuple[int, int, int] = (
44                    max(0, min(255, int(conv_pixel[0]))),
45                    max(0, min(255, int(conv_pixel[1]))),
46                    max(0, min(255, int(conv_pixel[2]))))
47            result_pixels[x, y] = normalized
48    return result_image

```

Codice A.5: Codice per la convoluzione

Generazione Kernel di Gaussian

```

1 def generate_gauss_kernel(kernel_size: int, sigma: int) -> list[list[float]]:
2 """
3 Generates a Gaussian kernel of the given size and sigma.
4 :param kernel_size: The size of the kernel.
5 :param sigma: The sigma value.
6 :return: The generated kernel.
7 """
8     kernel = np.fromfunction(lambda x, y: (1 / (2 * np.pi * sigma ** 2)) * np.exp(
9         -(x - (kernel_size - 1) / 2) ** 2 + (y - (kernel_size - 1) / 2) ** 2) / (2 * sigma
10            ** 2)), (kernel_size, kernel_size))
11     return (kernel / np.sum(kernel)).tolist()

```

Codice A.6: Codice per la generazione di un kernel di Gauss

Operatore Morfologico d'erosione

```

1 def _erosion(target_image: Image.Image, structural_element: list[list[int]]) -> Image.
2     Image:
3 """
4 Applies the erosion operator to an RGB image.
5 :param target_image: The image to apply the operator to.
6 :param structural_element: The structural element to use.
7 :return: The eroded image.
8 """
9     width, height = target_image.size
10    kernel_size: int = len(structural_element)
11    target_image = target_image.convert("RGB")
12    target_pixels = target_image.load()
13    result_image = Image.new("RGB", (width, height))
14    result_pixels = result_image.load()
15    for y in range(height):
16        for x in range(width):
17            pixel_chs: list[int] = [255, 255, 255]
18            for ky in range(kernel_size):
19                for kx in range(kernel_size):
20                    px = x + kx - int(kernel_size / 2)
21                    py = y + ky - int(kernel_size / 2)
22                    if 0 <= px < width and 0 <= py < height:
23                        pixel_r, pixel_g, pixel_b = target_pixels[px, py]
24                        kernel_value = structural_element[ky][kx]
25                        pixel_chs[0] = min(pixel_chs[0], pixel_r - kernel_value)
26                        pixel_chs[1] = min(pixel_chs[1], pixel_g - kernel_value)
27                        pixel_chs[2] = min(pixel_chs[2], pixel_b - kernel_value)
28                    result_pixels[x, y] = (pixel_chs[0], pixel_chs[1], pixel_chs[2])

```

Codice A.7: Codice dell'operatore morfologico erosione

Operatore Morfologico dilatazione

```
1 def _dilation(target_image: Image.Image, structural_element: list[list[int]]) -> Image.  
    Image:  
2 """  
3 Applies the dilation operator to an RGB image.  
4 :param target_image: The image to apply the operator to.  
5 :param structural_element: The structural element to use.  
6 :return: The dilated image.  
7 """  
8     width, height = target_image.size  
9     kernel_size: int = len(structural_element)  
10    target_image = target_image.convert("RGB")  
11    target_pixels = target_image.load()  
12    result_image = Image.new("RGB", (width, height))  
13    result_pixels = result_image.load()  
14    for y in range(height):  
15        for x in range(width):  
16            pixel_chs: list[int] = [0, 0, 0]  
17            for ky in range(kernel_size):  
18                for kx in range(kernel_size):  
19                    px = x + kx - int(kernel_size / 2)  
20                    py = y + ky - int(kernel_size / 2)  
21                    if 0 <= px < width and 0 <= py < height:  
22                        pixel_r, pixel_g, pixel_b = target_pixels[px, py]  
23                        kernel_value = structural_element[ky][kx]  
24                        pixel_chs[0] = max(pixel_chs[0], pixel_r + kernel_value)  
25                        pixel_chs[1] = max(pixel_chs[1], pixel_g + kernel_value)  
26                        pixel_chs[2] = max(pixel_chs[2], pixel_b + kernel_value)  
27            result_pixels[x, y] = (pixel_chs[0], pixel_chs[1], pixel_chs[2])  
28    return result_image
```

Codice A.8: Codice dell'operatore morfologico dilatazione

Filtro di Media

```
1 def mean_filter(target_image: Image.Image, filter_size: int) -> Image.Image:
2 """
3 Applies the mean filter to an RGB image.
4 :param target_image: The image to apply the operator to.
5 :param filter_size: The size of the filter. By default it is 3.
6 :return: The filtered image.
7 """
8     width, height = target_image.size
9     half_size: int = int(filter_size / 2)
10    target_image = target_image.convert("RGB")
11    target_pixels = target_image.load()
12    result_image = Image.new("RGB", (width, height))
13    result_pixels = result_image.load()
14    for x in range(width):
15        for y in range(height):
16            pixel_ch: list[int] = [0, 0, 0]
17            count: int = 0
18            for i in range(-half_size, half_size + 1):
19                for j in range(-half_size, half_size + 1):
20                    nx = min(max(x + i, 0), width - 1)
21                    ny = min(max(y + j, 0), height - 1)
22                    pixel = target_pixels[nx, ny]
23                    pixel_ch[0] += pixel[0]
24                    pixel_ch[1] += pixel[1]
25                    pixel_ch[2] += pixel[2]
26                    count += 1
27            result_pixels[x, y] = (int(pixel_ch[0] / count), int(pixel_ch[1] / count), int(pixel_ch[2] / count))
28    return result_image
```

Codice A.9: Codice del filtro di media

Filtro Bilaterale

```

1 def bilateral_filter(target_image: Image.Image, diameter: int, sigma_color: int,
                      sigma_space: int) -> Image.Image:
2 """
3 Applies the bilateral filter to an RGB image.
4 :param target_image: The image to apply the operator to.
5 :param diameter: The diameter of the neighborhood.
6 :param sigma_color: The sigma color value. The greater the value, the colors farther to
                      each other will start to get mixed
7 :param sigma_space: The sigma space value. The greater its value, the further pixels
                      will mix together, given that their colors lie within the sigmaColor range.
8 :return: The filtered image.
9 """
10    def bilateral_filter_gaussian(dist: float, sigma_value: int) -> float:
11        return (1 / (2 * math.pi * sigma_value ** 2)) * math.exp(-(dist ** 2) / (2 *
12            sigma_value ** 2))
13    width, height = target_image.size
14    target_image = target_image.convert("RGB")
15    target_pixels = target_image.load()
16    result_image = Image.new("RGB", (width, height))
17    result_pixels = result_image.load()
18    for x in range(width):
19        for y in range(height):
20            r_acc, g_acc, b_acc = 0.0, 0.0, 0.0
21            w_acc = 0.0
22            for i in range(-diameter, diameter + 1):
23                for j in range(-diameter, diameter + 1):
24                    nx = min(max(x + i, 0), width - 1)
25                    ny = min(max(y + j, 0), height - 1)
26                    pixel = target_pixels[nx, ny]
27                    spatial_dist = math.sqrt(i ** 2 + j ** 2)
28                    range_dist = math.sqrt((pixel[0] - target_pixels[x, y][0]) ** 2 +
29                        (pixel[1] - target_pixels[x, y][1]) ** 2 +
29                        (pixel[2] - target_pixels[x, y][2]) ** 2)
30
31                    weight = bilateral_filter_gaussian(spatial_dist, sigma_space) *
32                    bilateral_filter_gaussian(range_dist, sigma_color)
33                    r_acc += pixel[0] * weight
34                    g_acc += pixel[1] * weight
35                    b_acc += pixel[2] * weight
36                    w_acc += weight
37    result_pixels[x, y] = (int(r_acc / w_acc), int(g_acc / w_acc), int(b_acc / w_acc))
38
39    return result_image

```

Codice A.10: Codice del filtro bilaterale

Canny Edge Detection

```

1 def _non_maximal_supress(gradient_magnitude: np.ndarray, gradient_direction: np.ndarray)
   -> np.ndarray:
2 """
3 Suppresses the non-maximal values in the given gradient magnitude array.
4 :param gradient_magnitude: The gradient magnitude array.
5 :param gradient_direction: The gradient direction array.
6 :return: The suppressed gradient magnitude array.
7 """
8 suppressed_magnitude = np.zeros_like(gradient_magnitude)
9 angles = np.rad2deg(gradient_direction)
10 angles[angles < 0] += 180
11 for i in range(1, gradient_magnitude.shape[0] - 1):
12     for j in range(1, gradient_magnitude.shape[1] - 1):
13         direction = angles[i, j]
14         if (0 <= direction < 22.5) or (157.5 <= direction <= 180): # 0 degrees
15             neighbors = [gradient_magnitude[i, j - 1], gradient_magnitude[i, j + 1]]
16         elif 22.5 <= direction < 67.5: # 45 degrees
17             neighbors = [gradient_magnitude[i - 1, j - 1], gradient_magnitude[i + 1, j + 1]]
18         elif 67.5 <= direction < 112.5: # 90 degrees
19             neighbors = [gradient_magnitude[i - 1, j], gradient_magnitude[i + 1, j]]
20         else: # 135 degrees
21             neighbors = [gradient_magnitude[i - 1, j + 1], gradient_magnitude[i + 1, j - 1]]
22         if gradient_magnitude[i, j] >= max(neighbors):
23             suppressed_magnitude[i, j] = gradient_magnitude[i, j]
24     return suppressed_magnitude
25
26 def _double_threshold(image_array: np.ndarray, low_threshold: int, high_threshold: int)
   -> np.ndarray:
27 """
28 Filters the given image array with the given thresholds.
29 :param image_array: The image array.
30 :param low_threshold:
31 :param high_threshold:
32 :return: The filtered image array.
33 """
34 edge_map = np.zeros_like(image_array)
35 edge_map[(image_array >= high_threshold)] = 255
36 edge_map[(image_array >= low_threshold) & (image_array < high_threshold)] = 127
37 return edge_map
38
39 def _link_edges(image_array: np.ndarray) -> np.ndarray:
40 """
41 Links the edges of the given image array
42 :param image_array: The image array.
43 :return: The image array with the edges linked.
44 """
45 for i in range(1, image_array.shape[0] - 1):
46     for j in range(1, image_array.shape[1] - 1):
47         if image_array[i, j] == 127:
48             if np.max(image_array[i - 1:i + 2, j - 1:j + 2]) == 255:
49                 image_array[i, j] = 255
50             else:
51                 image_array[i, j] = 0
52     return image_array
53
54
55 def canny_edge_detector(target_image: Image.Image, gauss_size: int, sigma: int,
   low_threshold: int,
   high_threshold: int) -> Image.Image:
56 """
57 Applies the canny edge detection algorithm to the given image.
58 :param target_image: The image to apply the canny algorithm to.
59 """

```

```
60 :param gauss_size: The size of the gaussian kernel.
61 :param sigma: The sigma value for the gaussian kernel.
62 :param low_threshold: The low threshold for the double thresholding.
63 :param high_threshold: The high threshold for the double thresholding.
64 :return: The image with the canny algorithm applied.
65 """
66 target_image = target_image.convert("L")
67 # Phase 1 Smoothing applying gaussian kernel
68 gauss_kernel = generate_gauss_kernel(gauss_size, sigma)
69 gauss_image = convolve_native(target_image, gauss_kernel)
70 # Phase 2 Finding the gradients
71 sobel_x_array = np.array(convolve_native(gauss_image, convolution_kernels["edge_detection (sobel-x)"], True))
72 sobel_y_array = np.array(convolve_native(gauss_image, convolution_kernels["edge_detection (sobel-y)"], True))
73 gradient_magnitude = np.hypot(sobel_x_array, sobel_y_array)
74 gradient_direction = np.arctan2(sobel_y_array, sobel_x_array)
75 # Phase 3 Non-maximal suppression
76 suppressed = _non_maximal_supress(gradient_magnitude, gradient_direction)
77 # Phase 4 Double thresholding
78 thresholded = _double_threshold(suppressed, low_threshold, high_threshold)
79 # Phase 5 Edge tracking by linking edges
80 result_array = _link_edges(thresholded)
81 return Image.fromarray(result_array.astype(np.uint8))
```

Codice A.11: Codice del filtro di Canny