ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA

Corso di Laurea in Ingegneria e Scienze Informatiche

Un Algoritmo Efficiente per il Calcolo del Filtro Mediano su Immagini ad Alta Profondità

Relatore:
Prof. Moreno Marzolla

Presentata da:
Davide Bartoli

Sessione di Ottobre
Anno accademico 2024/2025

Sommario

Questa tesi si concentra sul filtro mediano, un operatore non lineare ampiamente utilizzato nell'elaborazione di immagini per la rimozione del rumore e il miglioramento della qualità visiva. Nonostante la sua apparente semplicità, il calcolo efficiente del filtro mediano rappresenta ancora una sfida, soprattutto nel caso di immagini ad alta profondità di colore e con raggi di filtro elevati, dove le prestazioni degli algoritmi tradizionali tendono a degradarsi sensibilmente. Inizialmente viene presentata una panoramica di numerosi algoritmi e implementazioni per il calcolo del filtro mediano su CPU, con particolare attenzione alle loro limitazioni. L'obiettivo principale di questo lavoro è presentare un nuovo algoritmo per CPU, progettato per garantire efficienza anche in scenari particolarmente complessi, come quelli appena descritti. L'algoritmo viene descritto nel dettaglio, e vengono discusse diverse ottimizzazioni volte a migliorarne le prestazioni. Il metodo proposto viene successivamente confrontato con numerose implementazioni esistenti, mostrando prestazioni superiori su immagini ad alta profondità e con raggi di filtro elevati rispetto a tutte le soluzioni testate.
Infine viene presentata un'estensione dell'algoritmo per l'elaborazione di immagini tridimensionali, sempre più diffuse soprattutto in ambito medico, e ne vengono presentate le prestazioni.

Indice dei contenuti

Introduzione

Al giorno d'oggi, l'elaborazione e l'analisi delle immagini rivestono un ruolo cruciale in moltissimi ambiti, con applicazioni dalla medicina alla sorveglianza, alla guida autonoma e all'industria manifatturiera. È sempre più necessario riuscire a estrarre informazioni preziose e prendere decisioni automatiche basate sui dati visivi. Per facilitare tali processi, le immagini vengono spesso preprocessate per renderle adatte ad essere analizzate. Uno degli strumenti più utilizzati a questo scopo sono i filtri, operatori che trasformano i pixel di partenza applicando diverse funzioni locali. I filtri possono migliorare la qualità dell'immagine, rimuovendo rumore e artefatti indesiderati, o evidenziare caratteristiche specifiche, come bordi o texture. Negli ultimi decenni sono stati sviluppati numerosi filtri e algoritmi per ottimizzarne le prestazioni, rendendoli sempre più veloci ed efficienti, compito reso sempre più necessario dall'enorme quantità di dati richiesti da molte applicazioni moderne e la diffusione di dispositivi mobili e sensori che generano immagini ad alta risoluzione e profondità. Gran parte della ricerca si è concentrata su filtri lineari, che operano mediante combinazioni lineari dei valori dei pixel circostanti, come il filtro di media mobile o il filtro gaussiano. Questi filtri presentano proprietà che ne facilitano l'ottimizzazione, ad esempio la possibilità di essere separati nelle due dimensioni spaziali. Al contrario, i filtri non lineari rimangono computazionalmente costosi. Tra questi il filtro mediano è uno degli strumenti più utilizzati, ma al contempo tra i più complessi da ottimizzare, soprattutto quando si lavora con grandi finestre e immagini a elevata profondità.

1.1. Il filtro mediano

Il filtro mediano è un operatore non lineare che sostituisce il valore di un pixel con la mediana dei valori dei pixel circostanti, selezionati all'interno di una finestra locale detta kernel. Più precisamente, dato un raggio R, il filtro mediano applica una finestra quadrata di dimensione $2R + 1$ centrata su ogni pixel, calcolando la mediana dei valori contenuti in tale regione. I pixel dell'immagine originale vengono quindi sostituiti con il valore mediano calcolato. Il filtro mediano è particolarmente efficace nel rimuovere il rumore impulsivo, come il rumore sale e pepe, riuscendo a preservare i dettagli dell'immagine come possiamo vedere nella Figura 1, a

differenza di molti filtri lineari che tendono a produrre un effetto di sfocatura. Questo perché la mediana, rispetto alla media, è meno sensibile ai valori anomali, rendendo il filtro mediano più robusto in presenza di rumore. Il filtro mediano trova applicazione in numerosi campi dell'analisi di immagini, come la visione artificiale e il processing di immagini mediche. Inoltre viene utilizzato nell'elaborazione di audio e anche nell'ambito del machine learning, dove è una componente fondamentale alla base di molti algoritmi più complessi.

Figura 1: Esempio di filtro mediano applicato a un'immagine con rumore sale e pepe. A sinistra l'immagine originale, al centro l'immagine con rumore, a destra l'immagine filtrata con R=2. Osserviamo come il filtro mediano riesca a rimuovere il rumore in maniera molto efficace, preservando i bordi e i dettagli dell'immagine originale.

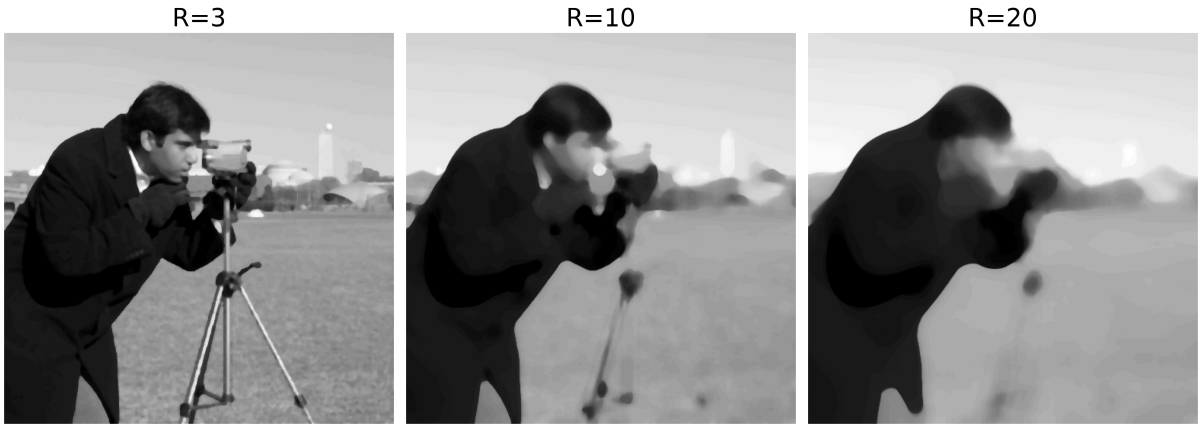

Figura 2: Esempio di filtro mediano di diversi raggi applicato all'immagine originale.

L'utilizzo di raggi diversi permette di ottenere effetti differenti, un raggio piccolo preserva i dettagli dell'immagine, mentre un raggio grande riesce a rimuovere il rumore in maniera più efficace, ma comporta un maggiore effetto di sfocatura come osserviamo nella Figura 2, che viene talvolta utilizzato per creare effetti artistici.

Nonostante la sua efficacia e il diffuso utilizzo, il filtro mediano rimane un'operazione computazionalmente costosa. Si conoscono algoritmi efficienti per immagini con bassa profondità (8 bit per pixel) o per raggi di filtro piccoli, ma l'applicazione del filtro mediano a immagini ad alta profondità usando raggi di filtro elevati rimane un problema complesso. Con la

diffusione di immagini ad alta profondità e risoluzione, come ad esempio le immagini HDR (High Dynamic Range), e la varietà delle applicazioni che possono richiedere valori di raggio diversi, è diventato sempre più importante sviluppare algoritmi efficienti per calcolare il filtro mediano in modo rapido e scalabile in tutti i contesti.

1.2. Ambiente di test

Tutti i risultati sperimentali sono stati ottenuti su Ubuntu 24.04 utilizzando una CPU AMD EPYC™ 7543P, virtualizzata a livello hardware tramite KVM (kernel virtual machine), con 8 core fisici e 16 thread, e 64 GB di RAM. Tutti i programmi sono stati compilati utilizzando il compilatore `g++` versione 13.3.0, con le flag `-O2 -fopenmp -march=native`. Dove non specificato diversamente, le misurazioni sono state effettuate utilizzando immagini generate casualmente.

Panoramica degli algoritmi esistenti

In questa tesi chiamiamo W e H la larghezza e l'altezza dell'immagine, B il numero di bit per pixel dell'immagine, detto anche profondità dell'immagine, e R il raggio del filtro. L'implementazione naive del filtro mediano consiste nel calcolare la mediana per ogni pixel dell'immagine in modo indipendente, ordinando i valori dei pixel circostanti e selezionando il valore centrale. Questo approccio richiede di ordinare i $(2R+1)^2$ valori del kernel, e ha una complessità computazionale di $O(R^2 \log R)$ per pixel. Questo algoritmo diventa rapidamente inefficiente all'aumentare del raggio, rendendolo impraticabile in molti contesti reali. Per affrontare questo problema, nel corso degli anni sono stati proposti numerosi algoritmi per calcolare il filtro mediano in modo più efficiente. Molti di questi possono essere generalmente suddivisi in due categorie: algoritmi basati sull'ordinamento e algoritmi basati sugli istogrammi.
Sono stati studiati anche approcci approssimati, che rinunciano alla precisione esatta della mediana in cambio di prestazioni superiori. In alcune applicazioni pratiche, questi metodi possono addirittura produrre risultati qualitativamente migliori rispetto agli algoritmi esatti [1]. Tuttavia questi algoritmi non vengono discussi in questa tesi, che si concentra esclusivamente su algoritmi esatti.

2.1. Algoritmi basati sull'ordinamento

Gli algoritmi basati sull'ordinamento si concentrano sull'ottimizzazione del processo di ordinamento dei pixel appartenenti al kernel per calcolare la mediana. Una strategia consiste nell'utilizzare algoritmi di ordinamento efficienti per interi, come il radix sort, che permette di ordinare n interi a b-bit con una complessità di $O(b \times n)$. Osservando che non abbiamo necessità di ordinare tutti i pixel, ma siamo solamente interessati alla mediana, possiamo utilizzare tecniche di selezione parziale, come l'algoritmo di quick selection [2], che permette di trovare il k-esimo elemento di un insieme con una complessità media di $O(n)$. Tuttavia

n rappresenta il numero di pixel all'interno della finestra, pari a $(2R+1)^2$, e quindi questi algoritmi scalano in tempo quadratico con il raggio del filtro, e sono quindi inadatti per raggi di filtro elevati.

Gli sviluppi più recenti in questo ambito si concentrano sull'utilizzo di reti di ordinamento ("sorting networks"). Questi metodi calcolano una rete di ordinamento, ovvero una sequenza fissa di confronti tra coppie di elementi, progettata per ordinare un insieme di dati indipendentemente dai valori specifici. Nella Figura 3 possiamo osservare un esempio di una rete di ordinamento per 4 elementi.

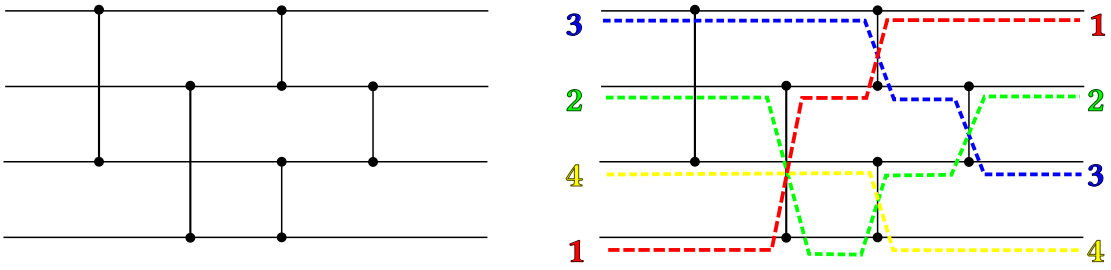

Figura 3: Esempio di rete di ordinamento per un array di 4 elementi, che richiede 5 confronti. I confronti sono rappresentati dalle barre verticali, che collegano i due elementi da comparare. Immagine tratta da [3].

I confronti vengono eseguiti in sequenza, e ognuno di essi scambia i due elementi se non si trovano nell'ordine corretto, progressivamente ordinando l'array di partenza. Nel caso del filtro mediano queste reti vengono costruite per trovare direttamente la mediana tra i valori del kernel, evitando l'ordinamento completo. Il vantaggio principale di questo approccio è che, essendo la sequenza di confronti predefinita, è possibile precalcolare la rete e ottimizzarla per riutilizzare i confronti effettuati per i pixel adiacenti a quello corrente, riducendo il lavoro ridondante. Inoltre, essendo la rete precalcolata, il codice generato può essere ottimizzato anche in fase di compilazione, tramite tecniche come loop unrolling e vettorizzazione. Questo approccio però richiede la creazione di una rete di ordinamento per ogni dimensione del filtro. Le prime reti di ordinamento sono state proposte per kernel di dimensione 5×5 [4] e successivamente 7×7 [5]. Recentemente è stato proposto un algoritmo [6] per generare programmaticamente reti di ordinamento per kernel di dimensione arbitraria. Questi algoritmi rappresentano ad oggi le soluzioni più efficienti per calcolare il filtro mediano con un raggio piccolo (tipicamente minore di 20). Tuttavia il numero di confronti richiesti cresce velocemente con l'aumentare del raggio, e le prestazioni degradano per raggi di filtro elevati.

2.2. Algoritmi basati sugli istogrammi

Un'altra classe di algoritmi per il calcolo del filtro mediano si basa sull'utilizzo di istogrammi, che possono essere usati per calcolare il valore del k-esimo elemento di un insieme. L'idea centrale consiste nel mantenere un istogramma delle frequenze dei valori dell'insieme. Per trovare la mediana di un insieme di n elementi, è sufficiente calcolare il valore del k-esimo elemento, dove $k = \lfloor \frac{n}{2} \rfloor$, scorrendo l'istogramma in ordine crescente fino al raggiungimento della somma cumulativa pari a k.
La ricerca in questo ambito si è concentrata sull'ottimizzazione del calcolo incrementale della mediana tra pixel adiacenti, sfruttando la forte sovrapposizione tra le rispettive finestre. Il primo algoritmo [7] proposto ha complessità $O(R)$ per pixel e si basa sull'osservazione che due finestre centrate su pixel adiacenti condividono la maggior parte dei valori, ad eccezione di due strisce di $2R + 1$ pixel ciascuna, come possiamo vedere nella Figura 4.

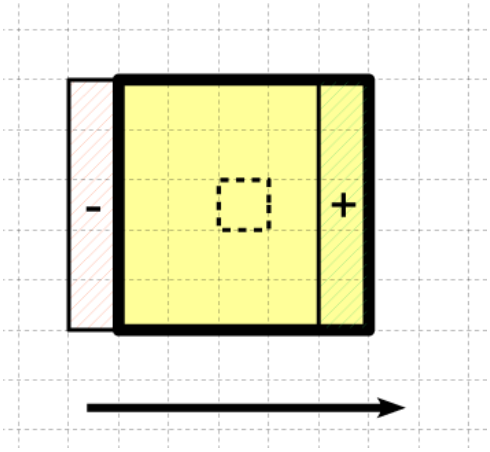

Figura 4: Rappresentazione dei pixel condivisi tra due finestre 5×5 adiacenti. Immagine tratta da [8]

Questa osservazione permette quindi di mantenere un istogramma dei valori presenti all'interno della finestra e aggiornarlo in modo incrementale, aggiungendo i nuovi pixel e rimuovendo quelli non più presenti nella finestra. Successivi miglioramenti hanno permesso di ridurre la complessità a $O(\log R)$ per pixel [9] e infine a $O(1)$ per pixel [8]. Questa soluzione è utilizzata in numerosi software e librerie di elaborazione delle immagini, tra le quali OpenCV [10].
Tuttavia questi algoritmi presentano una limitazione legata alla dimensione dell'istogramma, infatti in un'immagine con una profondità di B, ogni pixel può assumere 2^B valori distinti, e quindi è necessario mantenere un istogramma di dimensione 2^B. Inoltre questi algoritmi presentano un fattore costante che cresce esponenzialmente con la profondità dell'immagine. Sono quindi molto efficienti per immagini a 8 bit per pixel, anche con raggi elevati, ma inapplicabili a immagini con profondità maggiori. Alcune estensioni sono state proposte per immagini a 16-bit per pixel [11], ma richiedono una quantità di memoria significativa e presentano un fattore costante non trascurabile e non sono generalizzabili a immagini con profondità maggiore.

Per superare questo limite è stata descritta un'implementazione [12] che estende l'algoritmo di Huang [7], utilizzando una rappresentazione sparsa dell'istogramma tramite un albero binario di ricerca. Questa soluzione permette di supportare immagini con profondità arbitraria, ma presenta una complessità media pari a $O(R \log R)$ per pixel.

2.3. Wavelet Matrix

Un recente approccio alternativo [13] per calcolare il filtro mediano consiste nel determinare la mediana di ogni pixel un bit alla volta, utilizzando una struttura dati nota come Wavelet Matrix. Questo approccio fornisce un algoritmo in $O(B \log W)$ per pixel, dove W è la larghezza dell'immagine, e funziona per immagini a profondità arbitraria. Il tempo di esecuzione è indipendente da R, rendendo l'algoritmo particolarmente adatto per filtri di raggio elevato. Gli autori hanno inoltre sviluppato una versione accelerata tramite CUDA, recentemente integrata in OpenCV [10], dimostrando buone prestazioni su GPU. Tuttavia una limitazione di questo algoritmo è la necessità di costruire la struttura dati iniziale, che richiede $O(B \times W \times H \times \log W)$ memoria. Questo rende la versione su CPU poco performante nella pratica nonostante l'ottima complessità teorica dell'algoritmo.

Algoritmo

In questa sezione viene presentato un nuovo algoritmo per il calcolo del filtro mediano, che ha come obiettivo di superare le limitazioni degli algoritmi presentati in precedenza e ottenere buone prestazioni per immagini ad alta profondità e con raggi di filtro elevato.

3.1. Fenwick Tree

Iniziamo presentando una struttura dati chiamata Fenwick Tree [14], o Binary Indexed Tree, che viene utilizzata nell'algoritmo presentato in questa tesi. Questa struttura gestisce un array di N elementi, e permette di modificare un valore dell'array e calcolare la somma di un intervallo di elementi. Entrambe le operazioni hanno complessità $O(\log N)$. Indichiamo con $lsb(i)$ il bit meno significativo di i nella sua rappresentazione binaria. Il Fenwick Tree mantiene un array t di dimensione $N + 1$, dove per ogni posizione i memorizza la somma dei valori degli elementi da $i - lsb(i) + 1$ a i, come mostrato nella Figura 5.

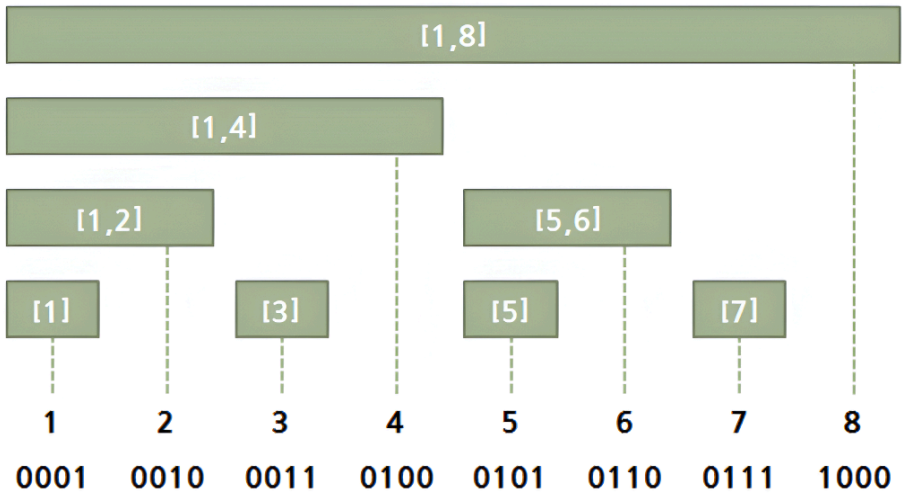

Figura 5: Rappresentazione di un Fenwick Tree, che mostra per ogni indice dell'array l'intervallo che viene sommato. Immagine tratta da [15].

Possiamo ora definire le due funzioni fondamentali del Fenwick Tree, $add(x, v)$ e $sum(x)$, che rispettivamente aggiungono v alla posizione x e restituiscono la somma dell'intervallo $[1, x]$. Entrambe le operazioni accedono a $O(\log N)$ posizioni dell'array t. Per ottenere la somma di un intervallo generico $[l, r]$ basta calcolare $sum(r) - sum(l - 1)$.

Algoritmo 1: FenwickTree(N)

```
1  t[0...(N + 1)] ← 0
2  func add(x, v):                          ▷ aggiunge v alla posizione x
3      while x ≤ N do
4          t[x] += v
5          x = x + lsb(x)
6
7  func sum(x):                             ▷ restituisce la somma da t[1] a t[x]
8      s ← 0
9      while x > 0 do
10         s += t[x]
11         x = x − lsb(x)
12     return s
13
14 func query(l, r):                        ▷ restituisce la somma da t[l] a t[r]
15     return sum(r) − sum(l − 1)
```

Algoritmo 1: Pseudocodice di un Fenwick Tree.

Questa struttura dati è stata scelta per la sua semplicità e il fattore costante molto basso, che la rende particolarmente efficiente nelle applicazioni pratiche.

3.2. Calcolo della mediana un bit alla volta

La soluzione proposta si basa su un algoritmo che calcola la mediana di un insieme di N elementi un bit alla volta, a partire dal più significativo.
Partiamo calcolando il bit più significativo della mediana, per il quale possiamo utilizzare il seguente approccio. Contiamo quanti valori dell'insieme hanno il bit più significativo uguale a 0, se questo valore è maggiore di $\lfloor \frac{N}{2} \rfloor$, allora il bit più significativo della mediana è 0, altrimenti è 1. Questo funziona perchè se ordiniamo tutti i valori dell'insieme, quelli con il bit più significativo uguale a 0 saranno i primi a comparire, e quindi se ce ne sono più di $\lfloor \frac{N}{2} \rfloor$, la mediana avrà il bit più significativo uguale a 0, altrimenti 1. Un esempio di questo processo è mostrato nella Figura 6.

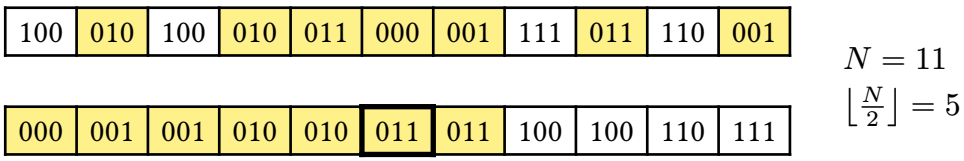

Figura 6: Insieme di $N = 11$ elementi, in alto, e versione ordinata per valore, in basso. Osserviamo che 7 elementi hanno il bit più significativo a 0. Dato che 7 è maggiore di $\lfloor \frac{N}{2} \rfloor$, il primo bit della mediana è 0.

Per generalizzare questa idea per calcolare tutti i bit della mediana, introduciamo le variabili $prefix$ e $missing$. Chiamiamo $prefix$ il prefisso della mediana corrente, ovvero i bit più significativi della mediana calcolati fino ad ora, e $missing$ l'indice della mediana corrente, inizialmente uguale a $\lfloor \frac{N}{2} \rfloor$, che indica l'indice della mediana tra i valori maggiori o uguali a $prefix$. Per calcolare il prossimo bit della mediana, contiamo quanti valori dell'insieme hanno come prefisso della mediana $prefix$ e il bit successivo uguale a 0, e chiamiamo questo valore $count$. Se $count$ è maggiore di $missing$, allora il prossimo bit della mediana è 0, altrimenti è 1 e sottraiamo $count$ a $missing$. Un esempio viene mostrato in Figura 7 e Figura 8, dove calcoliamo il secondo e il terzo bit della mediana dell'insieme visto in precedenza.

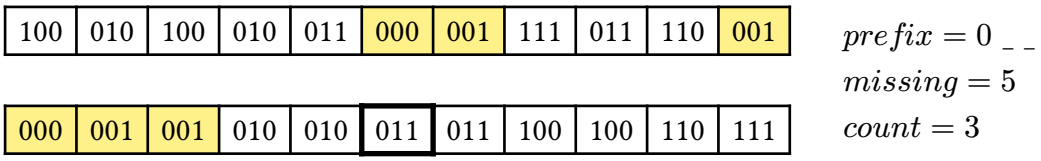

Figura 7: Seconda iterazione dell'algoritmo applicato all'insieme visto in precedenza. Dato che il conto $count$ dei valori che iniziano con 00 è minore di $missing$, il prossimo bit della mediana è 1 e sottraiamo $count$ a $missing$.

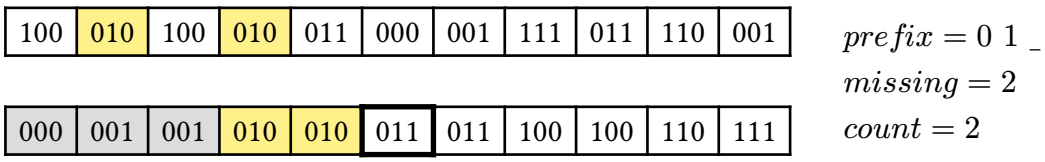

Figura 8: Ultima iterazione dell'algoritmo. Dato che il conto $count$ dei valori che iniziano con 010 non è maggiore di $missing$, il prossimo bit della mediana è 1. La mediana finale è quindi 011.

Nella Figura 8 osserviamo che ad ogni iterazione $missing$ indica l'indice della mediana tra i valori maggiori o uguali a $prefix$, ovvero escludendo i valori che iniziano con 00 visti in precedenza, colorati in grigio.

Questo metodo permette di calcolare la mediana in B iterazioni, ed è alla base dell'algoritmo di wavelet matrix [13] e dell'algoritmo presentato in questa tesi.

3.3. Descrizione dell'algoritmo

L'algoritmo proposto si basa sull'idea vista in precedenza, che permette di calcolare la mediana di un insieme un bit alla volta, applicata alla finestra di ogni pixel. Lo scopo è ridurre il lavoro ridondante, e calcolare contemporaneamente il prossimo bit della mediana per tutti i pixel che hanno lo stesso prefisso corrente.
Consideriamo l'iterazione b ($0 \leq b < B$) dell'algoritmo, per ogni prefisso $pref$ di lunghezza b vogliamo calcolare il prossimo bit della mediana per ogni pixel che ha come prefisso della mediana $pref$. Sia $S1$ l'insieme dei pixel che hanno $pref$ come prefisso della mediana e $S2$ l'insieme dei pixel che hanno come prefisso $pref$ e il bit successivo uguale a 0 come valore nell'immagine di partenza.
Per ogni posizione in $S1$ vogliamo contare quante posizioni in $S2$ sono all'interno della sua finestra di lato $2R + 1$. Notiamo che possiamo riformulare il problema come segue. Per ogni posizione (x, y) in $S2$ tracciamo un segmento verticale $(x, y - R) - (x, y + R)$, e per ogni posizione (x, y) in $S1$ consideriamo un segmento orizzontale $(x - R, y) - (x + R, y)$. La richiesta equivale a contare quanti segmenti verticali vengono intersecati da ognuno dei segmenti orizzontali. Una rappresentazione grafica di questa trasformazione è mostrata nella Figura 9.

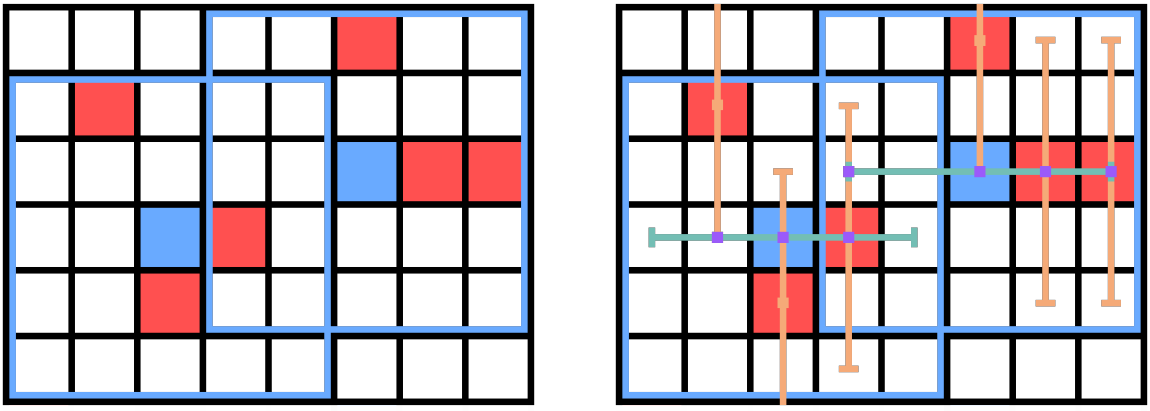

Figura 9: Rappresentazione del problema come intersezione di segmenti con $R = 2$. Le caselle blu fanno parte di $S1$ e le caselle rosse fanno parte di $S2$. Per contare quante caselle rosse sono all'interno della finestra possiamo contare le intersezioni tra i segmenti orizzontali e verticali, marcate in viola.

Questo problema può essere risolto utilizzando un algoritmo di sweepline. Manteniamo un vettore, inizialmente riempito di 0, che indica il numero di segmenti verticali attualmente «attivi». Spostandoci dall'alto verso il basso (y crescenti) ogni volta che incontriamo l'inizio di un segmento verticale aggiungiamo 1 alla posizione x, e ogni volta che incontriamo la fine di un segmento verticale sottraiamo 1 alla posizione x. Quando incontriamo un segmento orizzontale, la risposta è data dalla somma dei valori delle posizioni tra $x - R$ e $x + R$.

Per implementare questi update e query possiamo utilizzare un Fenwick Tree, che permette di effettuare entrambe le operazioni in $O(\log W)$.

Algoritmo 2: MEDIAN-FILTER-SWEEPLINE-2D(*in, out, W, H, R*)

```
missing[0...(W,H)] ← ⌊((2×R+1)^2)/2⌋                    ▷ indice della mediana
B ← bit per pixel
prefix[0...(W,H)] ← 0                                    ▷ prefisso della mediana corrente
                                  ▷ per ogni posizione in S1 voglio contare quante posizioni
                                          ▷ in S2 sono all'interno della finestra
func Sweepline(S1, S2):
    ft ← FenwickTree(W + 2R)
    op ← [ ]
    for x, y in S2 do
        op.append(y − R, 1, x)
        op.append(y + R + 1, 0, x)
    for x, y in S1 do
        op.append(y, 2, x)
    sort(op)                                             ▷ ordino le operazioni in base alla y
    for (y, k, x) in op do
        if k == 0 then
            ft.add(x, −1)
        else if k == 1 then
            ft.add(x, 1)
        else
            count ← ft.query(x − R, x + R)
            if count ≤ missing[x, y] then
                missing[x, y] − = count
                setta il prossimo bit di prefix[x, y] a 1

for i ← 0 to B do
    for distinct pref in prefix do
        S1 ← posizioni where prefix == pref
        S2 ← posizioni where in inizia con pref e il bit successivo è 0
        Sweepline(S1, S2)
out ← prefix                                             ▷ prefix ora contiene la mediana
```

Algoritmo 2: Pseudocodice dell'algoritmo proposto.

Per calcolare la complessità di questo algoritmo, osserviamo che durante ogni iterazione ogni posizione fa parte esattamente una volta di $S1$ e al più una volta $S2$, quindi la complessità

finale dell'algoritmo è $O(B \log W)$ per pixel. Osserviamo che la complessità è indipendente dal raggio del filtro. Questo algoritmo ha la stessa complessità computazionale di [13], ma utilizza memoria addizionale lineare e ha un fattore costante molto basso.

3.4. Implementazione pratica

Nell'implementazione pratica si è scelto di suddividere l'immagine iniziale in blocchi di dimensione $(2R) \times (2R)$, poi estesi con una ghost area di dimensione R. Questo abbassa la complessità finale a $O(B \log R)$ per pixel, e permette di parallelizzare facilmente l'algoritmo, in quanto ogni blocco può essere calcolato in modo indipendente. L'implementazione è stata resa parallela utilizzando OpenMP.

Per calcolare velocemente l'insieme $S2$ viene inizialmente creato un array con $(valore, x, y)$ per ogni pixel dell'immagine, ordinato in base ai valori dei pixel. L'insieme $S2$ è quindi costituito da un sottoarray contiguo di questo array, che può essere facilmente individuato tramite una ricerca binaria.

Per l'insieme $S1$ viene mantenuto un array P con le coordinate di ogni pixel dell'immagine, ordinato in base alla mediana corrente. Ogni volta che viene aggiornata la mediana di un valore, la sua posizione viene aggiunta a un array P_0 o P_1, a seconda che il bit aggiunto sia 0 o 1. Alla fine dell'iterazione $P \leftarrow P_0 + P_1$. In questo modo tutte le posizioni che hanno lo stesso prefisso si trovano contigue in P, e gli insiemi $S1$ rappresentano quindi un sottoarray contiguo di P.

Tutte le operazioni di ordinamento sono state implementate utilizzando il radix sort, dato che tutti i dati sono interi di dimensione fissa.

3.5. Analisi delle prestazioni

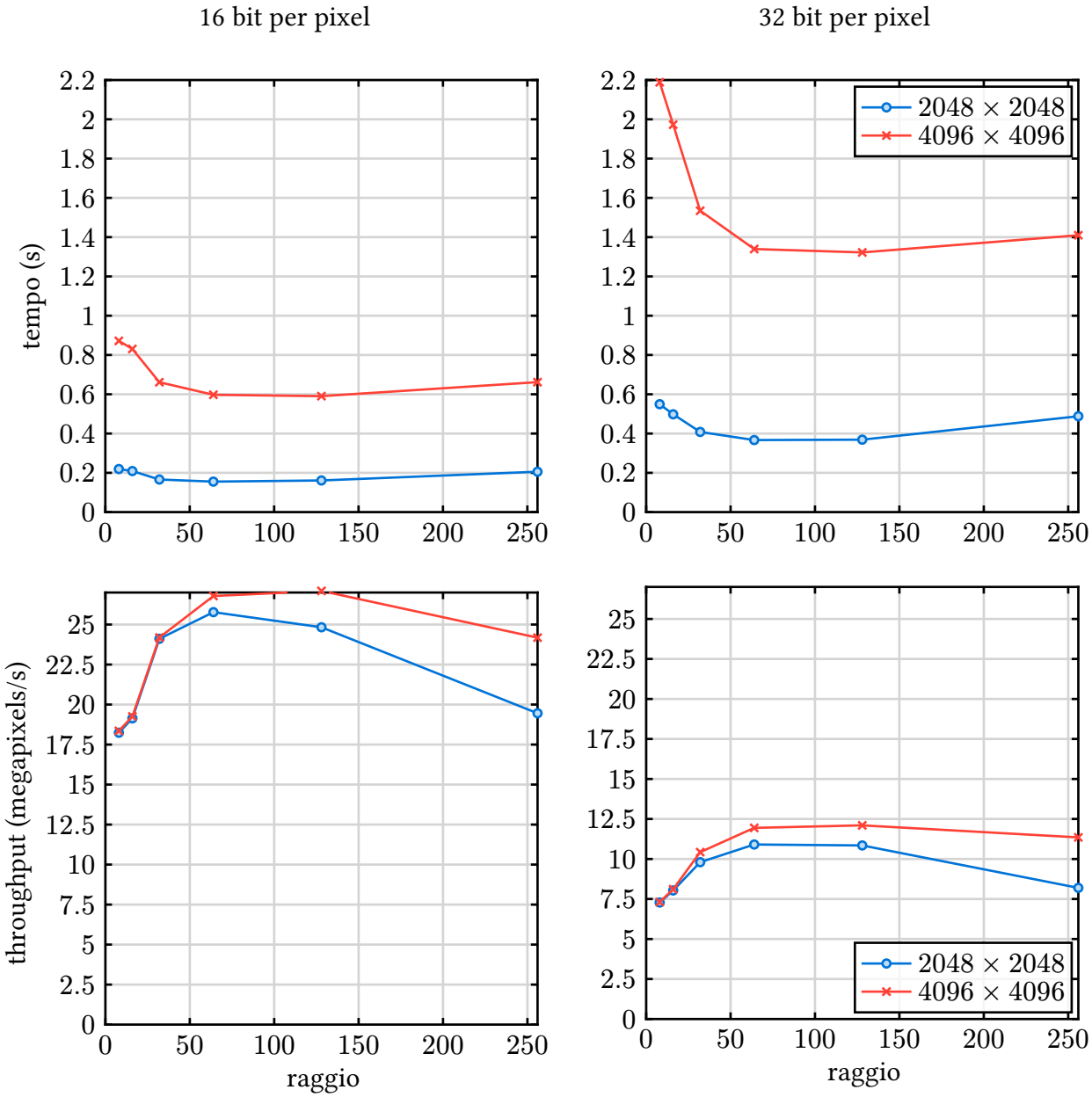

Figura 10: Prestazioni dell'algoritmo per immagini a 16 e 32 bit di diverse dimensioni e utilizzando filtri con diversi raggi

Nella Figura 10 possiamo osservare le prestazioni dell'algoritmo descritto, applicato a immagini di dimensioni 2048×2048 e 4096×4096. Notiamo che il throughput è maggiore per immagini di grandi dimensioni, questo è parzialmente dovuto all'overhead di OpenMP, che è più evidente per immagini piccole, ma soprattutto a causa della ghost area, che per raggi elevati occupa una porzione più significativa nelle immagini più piccole. In generale osserviamo prestazioni buone, con throughput che si aggirano intorno ai 20 megapixel al secondo per immagini a 16 bit per pixel e 10 megapixel al secondo per immagini a 32 bit per pixel. Osserviamo inoltre che il tempo di esecuzione rimane molto simile al variare del raggio, e che quindi come ricercato questo algoritmo è adatto anche ad essere utilizzato per raggi elevati.

Ottimizzazioni

Durante lo sviluppo dell'algoritmo sono state implementate e testate numerose ottimizzazioni, volte a migliorarne le prestazioni pratiche. In questa sezione vengono presentate le ottimizzazioni più significative, che hanno portato a una riduzione notevole dei tempi di esecuzione.

4.1. Ottimizzazione per insiemi piccoli

Quando uno tra gli insiemi $S1$ e $S2$ è piccolo, possiamo calcolare la risposta in modo brute-force, provando tutte le coppie, come mostrato nell' Algoritmo 3. Questo migliora le prestazioni in particolare delle ultime iterazioni dell'algoritmo, quando il numero di pixel che hanno lo stesso prefisso è molto ridotto.

Algoritmo 3: MEDIAN-FILTER-SWEEPLINE-2D(*in, out, W, H, R*)

```
1                                  ▷ per ogni posizione in S1 voglio contare quante posizioni
2  func Bruteforce(S1, S2):                ▷ in S2 sono all'interno della finestra
3      for x1, y1 in S1 do
4          count ← 0
5          for x2, y2 in S2 do
6              if |x1 − x2| ≤ R and |y1 − y2| ≤ R then
7                  count + = 1
8          if count ≤ missing[x1, y1] then
9              missing[x1, y1] − = count
10             setta il prossimo bit di prefix[x1, y1] a 1
```

Algoritmo 3: Pseudocodice dell'ottimizzazione brute-force.

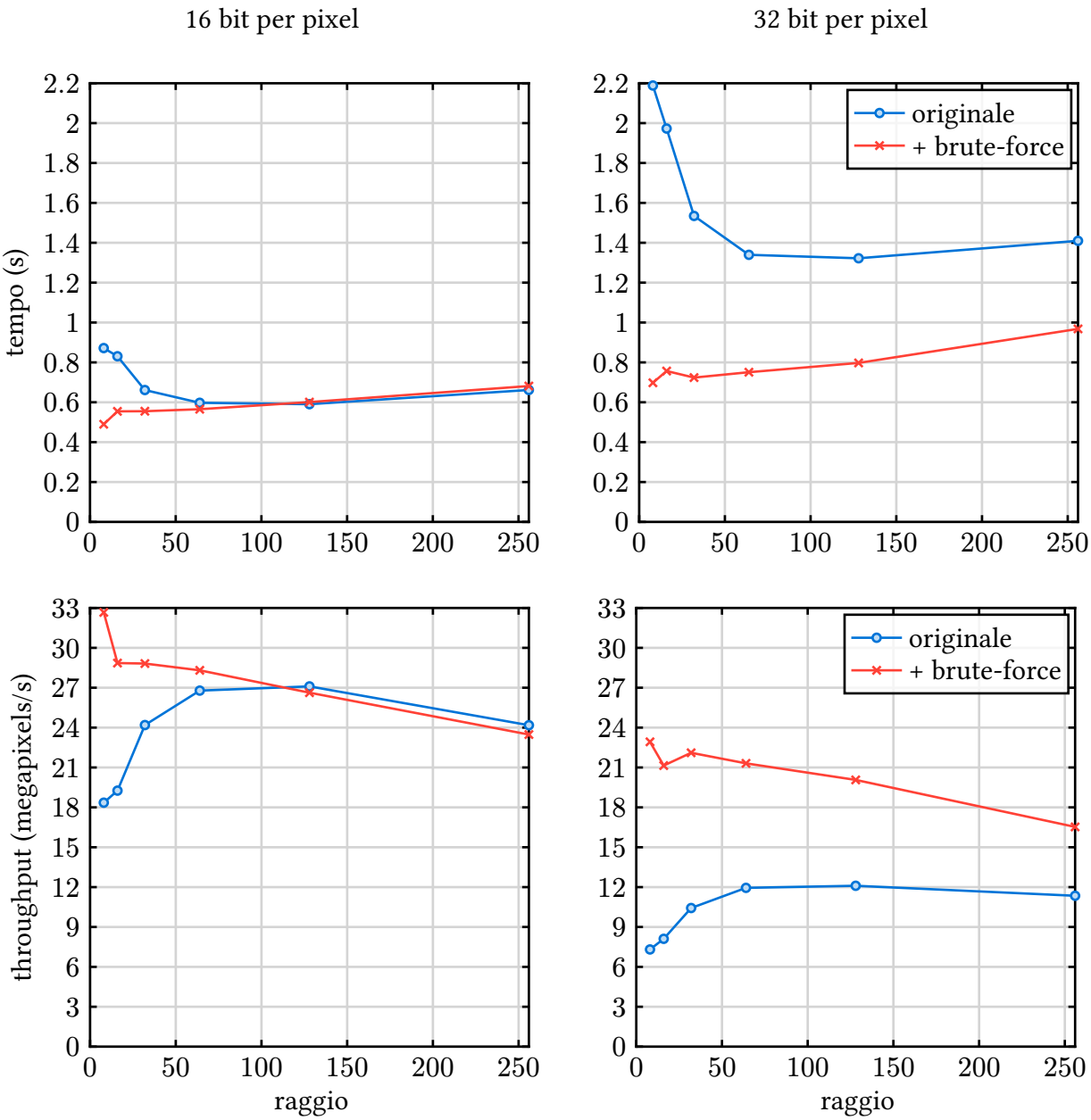

Figura 11: Confronto tra la versione originale dell'algoritmo e la versione ottimizzata utilizzando l'algoritmo brute-force, testate utilizzando immagini di dimensione 4096×4096 a 16 e 32 bit per pixel.

Nella Figura 11 notiamo che questa ottimizzazione risulta particolarmente efficace per immagini a 32 bit per pixel. Infatti nelle immagini ad alta profondità abbiamo molte iterazioni, ma il numero di pixel che hanno lo stesso prefisso diventa rapidamente molto piccolo, e quindi questa ottimizzazione riesce a migliorare le prestazioni delle ultime iterazioni in modo significativo.

4.2. Mediana ordinale

Un'ottimizzazione utilizzata in alcuni algoritmi [9] consiste nell'applicare una trasformazione ordinale all'immagine, ovvero ordinare i pixel in base al loro valore e sostituirli con la loro posizione nell'ordinamento Figura 12. Al contrario dei filtri lineari, il filtro mediano è invariante rispetto a questa trasformazione, quindi possiamo applicarla senza modificare il significato del risultato finale. In questo modo al posto di calcolare il valore del pixel, calcoliamo il suo indice nel vettore di valori ordinati.

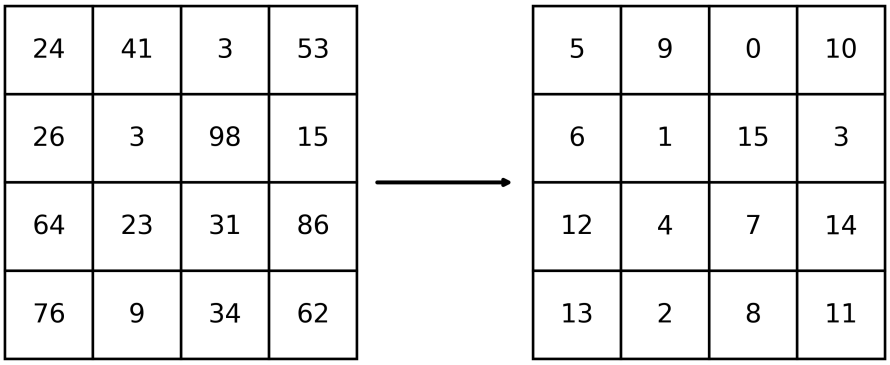

Figura 12: Esempio di trasformazione ordinale applicata a una griglia di valori. I pixel sono ordinati in base al loro valore, e sostituiti con il loro indice nell'ordinamento. Valori duplicati sono rappresentati con indici consecutivi.

La mediana ordinale viene applicata ad ogni blocco in modo indipendente, comprendendo anche la ghost area, e permette di ridurre il numero di bit necessari per rappresentare i pixel da B a $\log(dimensione\ blocco)$. Dato che l'algoritmo effettua un numero di iterazioni uguale al numero di bit necessario per rappresentare i valori, questo permette di ridurre il numero di iterazioni nel caso in cui $\log(dimensione\ blocco)$ è minore di B. Questo è particolarmente utile per immagini a profondità elevata come mostrato nella Figura 13, dove B è grande, e la mediana ordinale permette di ridurre di molto il numero di iterazioni effettuate.

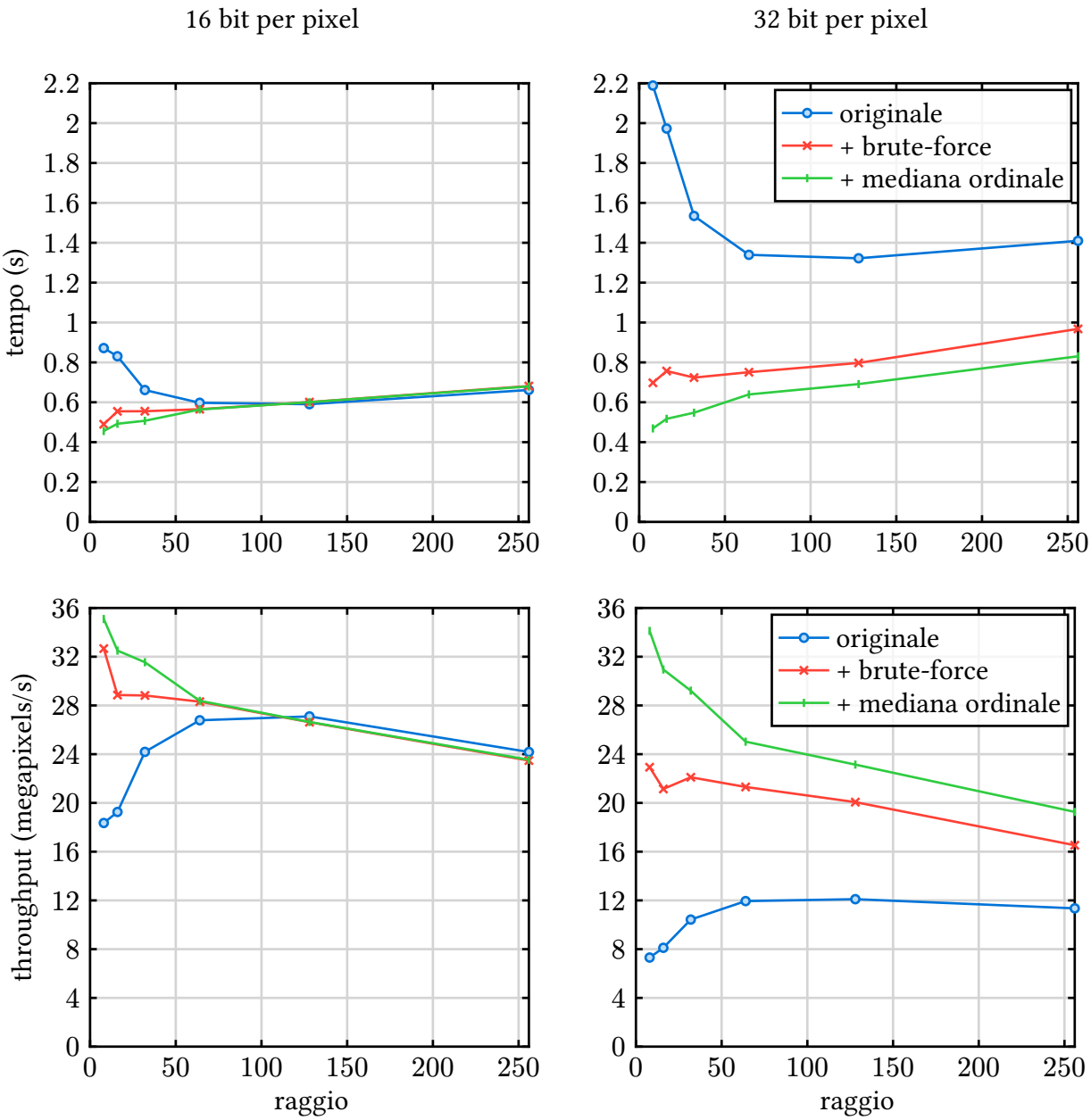

Figura 13: Confronto tra la versione originale dell'algoritmo, la versione ottimizzata utilizzando l'algoritmo brute-force e l'implementazione che utilizza sia l'ottimizzazione brute-force che della mediana ordinale. I test sono stati effettuati su immagini di dimensione 4096×4096.

Osserviamo che come atteso questa ottimizzazione si rivela efficace per immagini a 32 bit per pixel, nelle quali viene sempre effettuata la mediana ordinale, che ora mostrano prestazioni molto simili a quelle delle immagini a 16 bit per pixel. Notiamo che il miglioramento è più evidente per raggi piccoli, infatti all'aumentare di R aumenta anche la dimensione dei blocchi ed è necessario un maggior numero di bit per rappresentare gli indici, rendendo questa ottimizzazione meno efficace.

4.3. Primi 8 bit in tempo costante

Come descritto precedentemente, esistono algoritmi basati sugli istogrammi molto veloci per calcolare il filtro mediano di immagini a bassa profondità. Possiamo quindi utilizzare uno di questi algoritmi per calcolare i primi 8 bit della mediana e successivamente calcolare i restanti $B - 8$ bit utilizzando l'algoritmo presentato. Questo necessita di modificare l'algoritmo scelto per farci restituire l'array *missing*, necessario a proseguire il calcolo della mediana, ma migliora molto le prestazioni, riducendo di 8 il numero di iterazioni necessarie. È stata scelta l'implementazione realizzata da Adams [6] dell'algoritmo di Perreault [8].

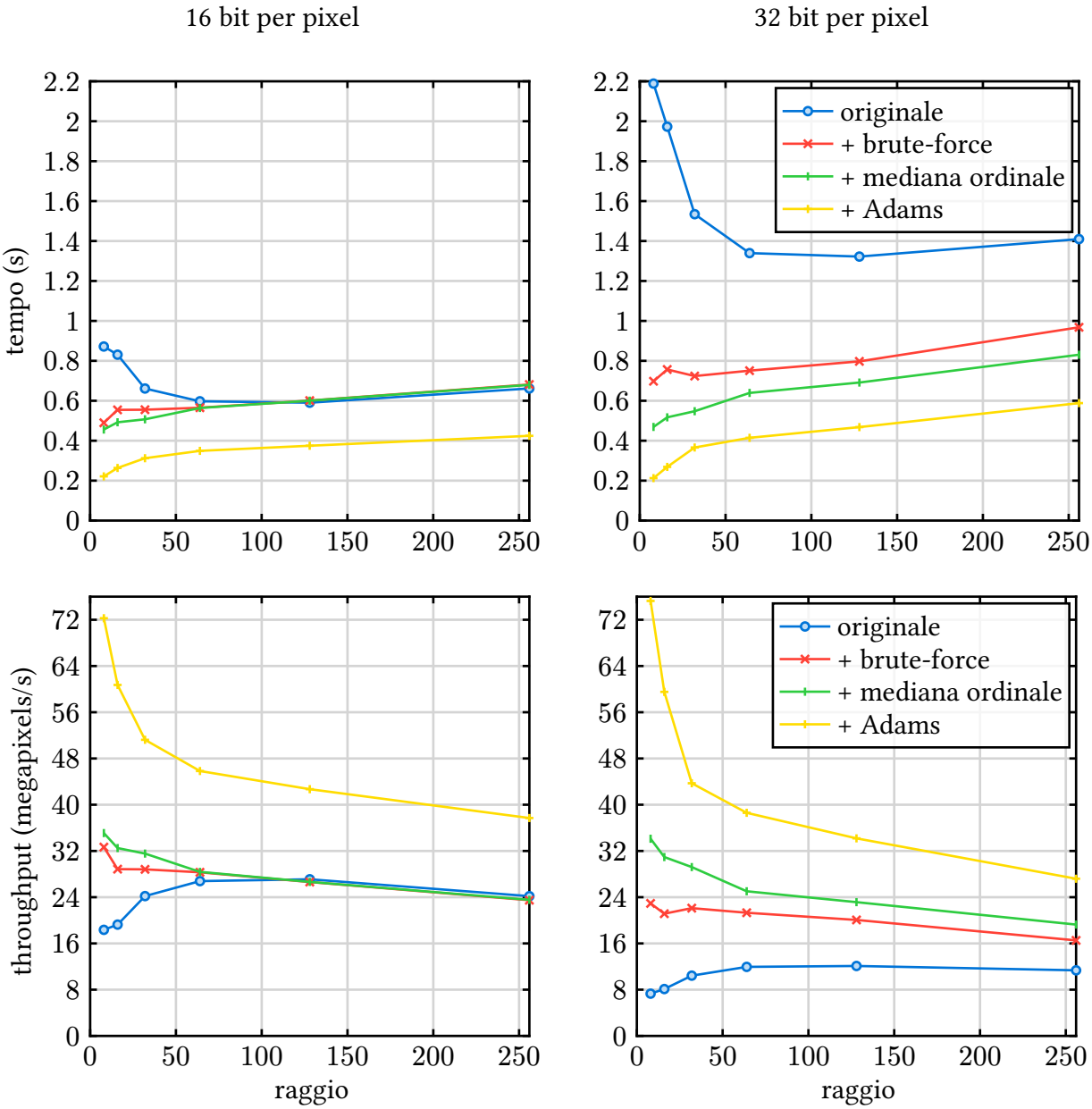

Figura 14: Confronto tra le tre precedenti versioni dell'algoritmo e la versione che combina tutte le ottimizzazioni descritte: l'approccio brute-force, l'utilizzo della mediana ordinale e l'implementazione di Adams per calcolare i primi 8 bit della mediana. I test sono stati effettuati su immagini di dimensione 4096×4096.

Come mostrato nella Figura 14, questa ottimizzazione risulta molto efficace, e i miglioramenti sono particolarmente evidenti per raggi piccoli, dove il numero di iterazioni era limitato grazie alla mediana ordinale e quindi la rimozione di 8 iterazioni è molto significativa.

4.4. Prestazioni dell'algoritmo finale

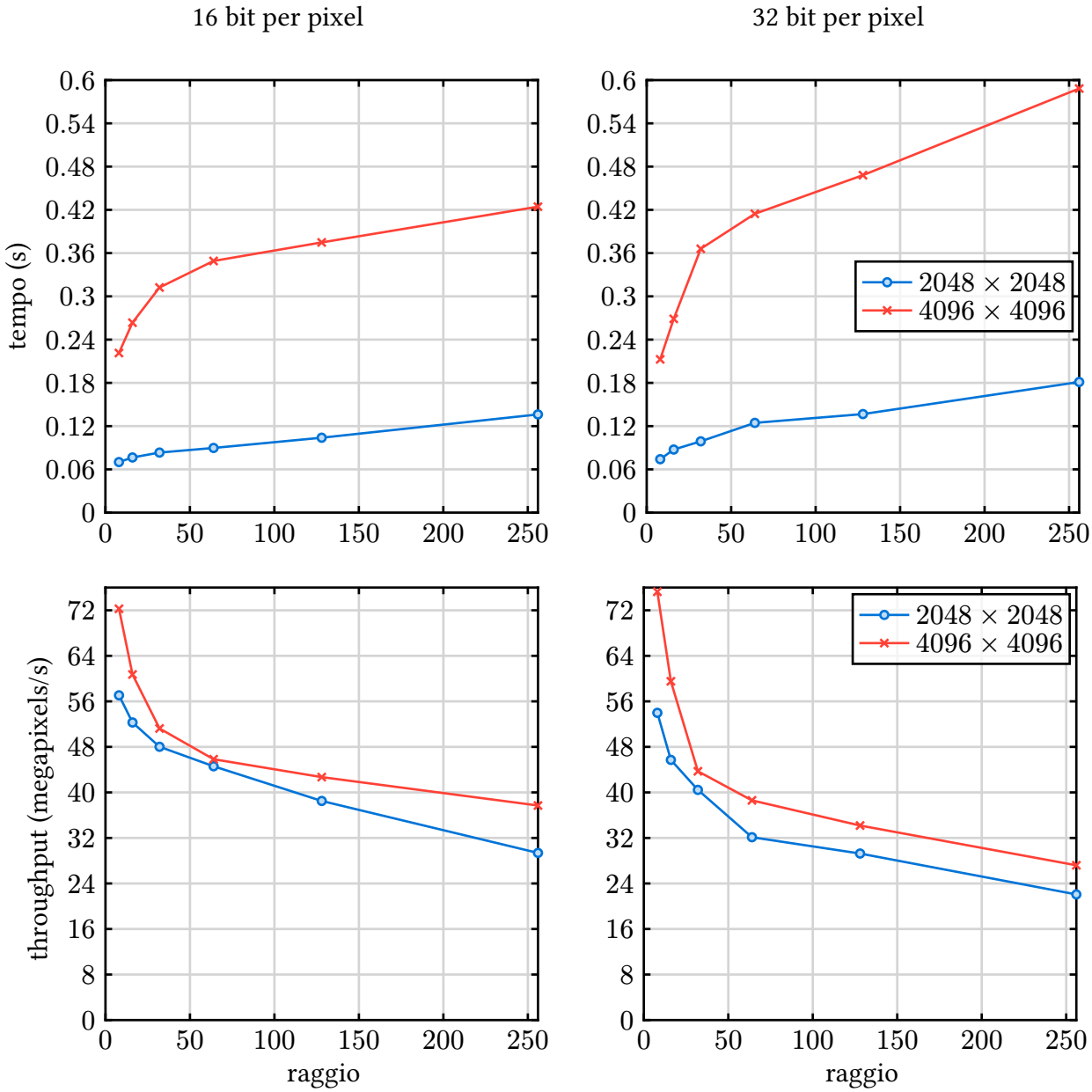

Figura 15: Prestazioni dell'algoritmo finale, che combina tutte le ottimizzazioni descritte: l'approccio brute-force per insiemi piccoli, la mediana ordinale e il calcolo dei primi 8 bit in tempo costante. I test sono stati effettuati su immagini di dimensione 2048 × 2048 e 4096 × 4096.

Nella Figura 15 osserviamo che le ottimizzazioni implementate hanno portato a un netto miglioramento delle prestazioni, per immagini di dimensioni 4096×4096 e 16 bit per pixel l'algoritmo finale è 3.9 volte più veloce della versione iniziale con $R = 8$ e 1.6 volte con $R = 256$. Per immagini a 32 bit per pixel il miglioramento è ancora più evidente, 10.3x con $R = 8$ e 2.4x con $R = 256$.

4.4.1. Speedup e strong scaling efficiency

Per verificare l'efficienza della parallelizzazione dell'algoritmo possiamo calcolare lo speedup e la strong scaling efficiency. Lo speedup è definito come $\frac{t(p)}{t(1)}$, dove $t(p)$ è il tempo di esecuzione dell'algoritmo utilizzando p core, e $t(1)$ è il tempo di esecuzione utilizzando un solo core, e indica quante volte l'algoritmo è più veloce quando parallelizzato su p unità di calcolo.
La strong scaling efficiency è definita come $\frac{speedup}{p}$, e misura quanto l'algoritmo si avvicina allo speedup ottimale pari a p.

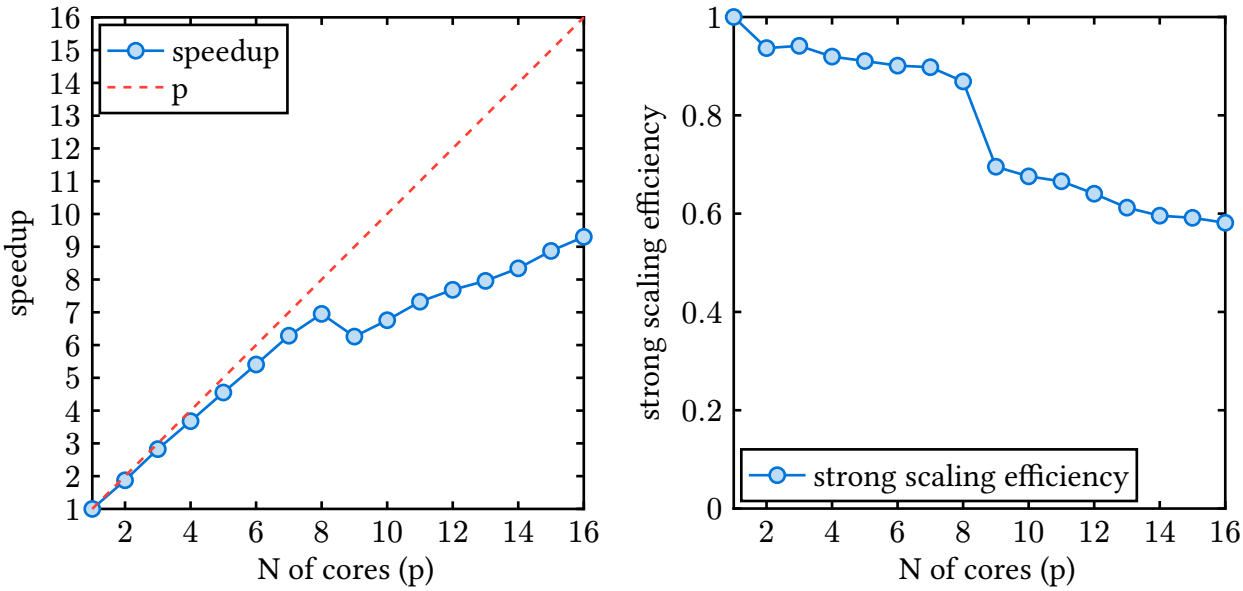

Figura 16: Speedup e strong scaling efficiency calcolati su un'immagine 4096×4096 e raggio 64

Osserviamo che lo speedup e la strong scaling efficiency sono molto buoni fino a $p = 8$, il numero di core fisici della macchina utilizzata. Questo dimostra che la parallelizzazione dell'algoritmo è efficace, e riusciamo a sfruttare tutte le risorse a nostra disposizione. Per p maggiori di 8 facciamo uso dell'hyperthreading, che permette di continuare a migliorare le prestazioni all'aumentare del numero di core logici, ma come atteso siamo lontani dallo speedup ottimale.

Confronto con altri algoritmi

In questa sezione viene confrontato l'algoritmo presentato con altri algoritmi esistenti discussi in precedenza, includendo implementazioni di diverse librerie. Questi benchmark sono stati realizzati con il programma pubblicato da Adams [6], esteso con l'algoritmo di Wavelet Matrix [13], l'algoritmo di sparse_hist [12] e l'algoritmo presentato in questa tesi. Il programma utilizza i benchmark adattivi di Halide con il parametro accuracy settato a 0.0001, e i codici sono stati testati sul canale del rosso della Figura 17, modificata per ottenere diverse profondità.

Figura 17: Immagine 5800×3240 della rocca sforzesca di Imola, utilizzata per i benchmark. Immagine tratta da [16].

Gli algoritmi testati insieme a quello presentato sono i seguenti:

1. *Intel performance primitives* (IPP) [17] versione 2022.1.0.649, libreria proprietaria di Intel contenente l'implementazione di molti algoritmi. L'implementazione del filtro mediano è stata resa parallela dividendo l'immagine in strisce orizzontali dagli autori del benchmark.
2. Adams [6], implementazione dell'algoritmo che utilizza reti di ordinamento, sia la versione dinamica, che interpreta la rete di ordinamento a runtime, sia la versione statica, che

genera il codice in fase di compilazione. La versione statica richiede molto tempo per la compilazione e funziona solo per raggi piccoli, ma ha prestazioni molto buone in quanto il codice generato può essere ottimizzato dal compilatore.

3. Wavelet Matrix [13], implementazione dell'algoritmo che utilizza la Wavelet Matrix. Si nota che gli autori dell'algoritmo si sono concentrati principalmente sull'implementazione CUDA, e sostengono che la loro implementazione CPU sia ottimizzabile ulteriormente.
4. ctmf (constant time median filter), algoritmo in tempo costante di Perreault [8], generalizzato per immagini a 16 bit per pixel nell'implementazione proposta in [11], modernizzata da Adams [6]. Non supporta immagini a 32 bit per pixel. La versione a 8 bit è la stessa che viene utilizzata nell'ultima ottimizzazione dell'algoritmo presentato per calcolare i primi 8 bit della mediana.
5. sparse_hist, implementazione dell'algoritmo descritto da Marzolla [12], che utilizza un albero binario di ricerca per rappresentare l'istogramma.

I risultati sono mostrati in Figura 18 e Figura 19, dove sono stati testati raggi da 1 a 176 e immagini a 16 e 32 bit per pixel. I grafici mostrano i throughput ottenuti dai diversi algoritmi in una scala logaritmica.

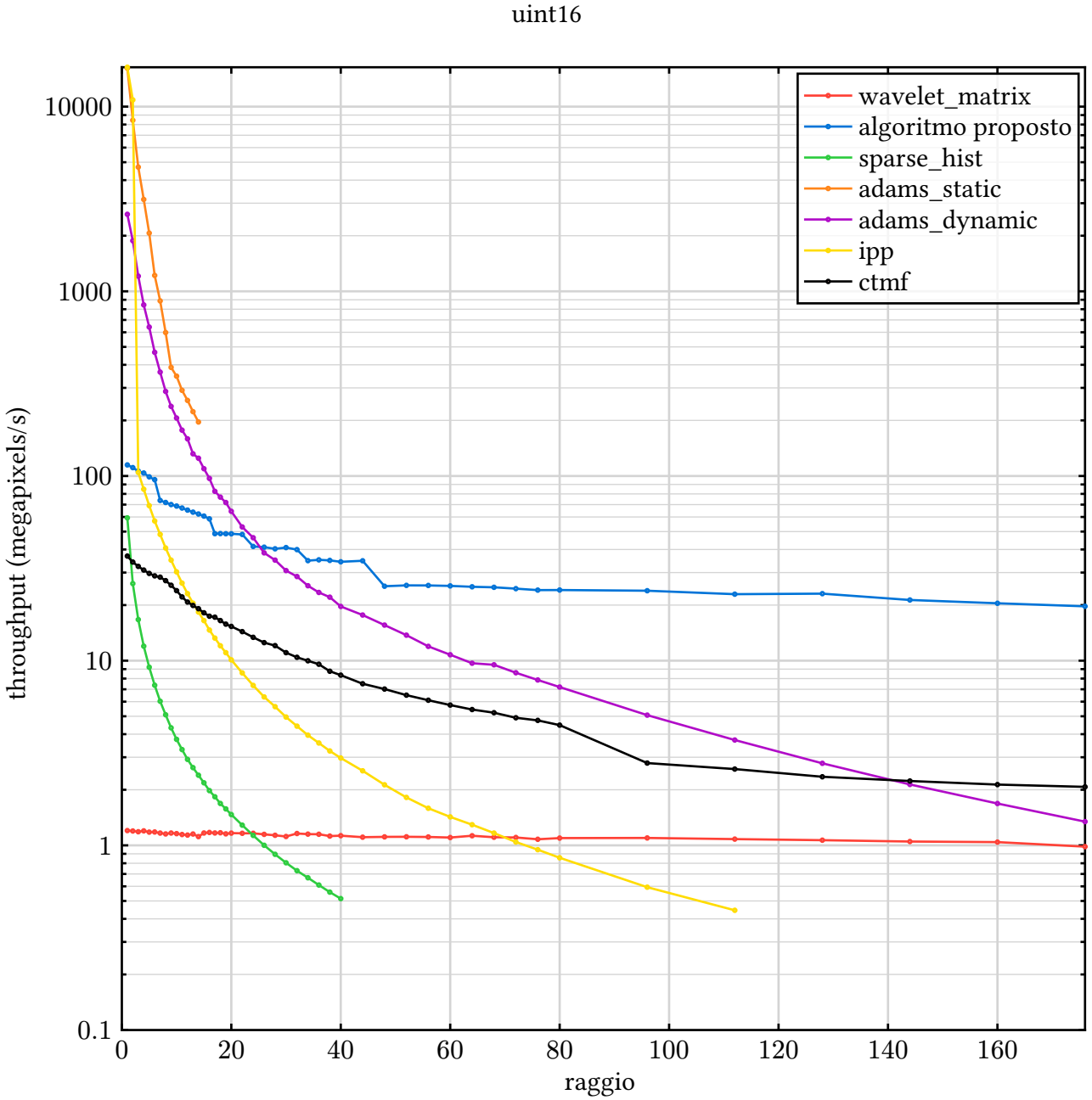

Figura 18: Confronto tra gli algoritmi utilizzando l'immagine a 16 bit per pixel.

Possiamo notare che l'algoritmo presentato ha prestazioni molto buone, e risulta essere l'algoritmo più veloce per raggi maggiori di 26. Utilizzando un raggio di 40 l'algoritmo presentato è 1.7 volte più veloce della seconda miglior implementazione, e la differenza aumenta all'aumentare del raggio, raggiungendo prestazioni quasi 10 superiori per $R = 160$. Inoltre osserviamo che l'algoritmo descritto è il migliore che non utilizza reti di ordinamento per tutti i raggi testati.

Analizzando il grafico possiamo osservare degli scalini nel throughput dell'algoritmo presentato, dovuti all'aumento della dimensione dei blocchi, che richiede un numero maggiore di bit per rappresentare gli indici dei pixel. Una volta raggiunto un certo raggio il throughput si stabilizza, in quanto i blocchi sono diventati abbastanza grandi da rendere inefficace l'uso della mediana ordinale.

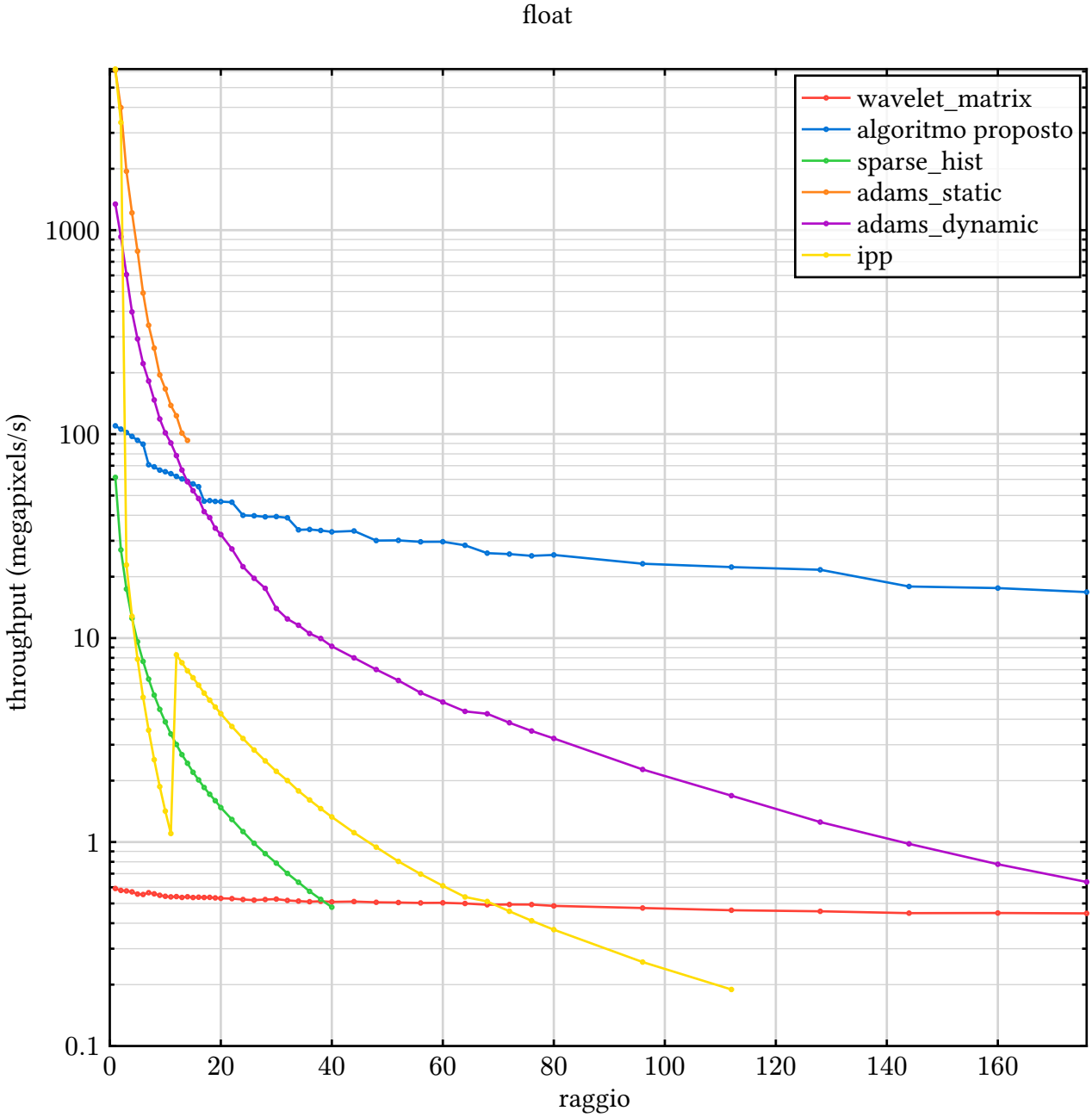

Figura 19: Confronto tra gli algoritmi utilizzando un'immagine con un float (32 bit) per pixel.

I risultati sono ancora più evidenti per immagini a 32 bit, non supportate da alcuni algoritmi tra cui ctmf, dove l'algoritmo presentato è il più veloce a partire da un raggio di 15. Per $R = 40$ l'algoritmo è 3.6 volte più veloce della seconda miglior implementazione, differenza che aumenta e raggiunge 22.5 volte per $R = 160$. Anche in questo caso l'algoritmo presentato è il miglior algoritmo che non utilizza reti di ordinamento per tutti i raggi testati. Per estendere l'algoritmo a immagini con un float per pixel viene sempre utilizzata la mediana ordinale, che permette di lavorare con gli indici dei pixel che sono interi.

I grafici mostrano chiaramente i risultati positivi ottenuti dall'algoritmo presentato, che si dimostra essere il migliore per il calcolo del filtro mediano con raggi elevati, in particolare per immagini a 32 bit per pixel.

Algoritmo 3D

Negli ultimi anni si è assistito a un aumento dell'uso di immagini tridimensionali, in particolare in ambito medico, dove volumi tridimensionali come le scansioni TAC e RMN sono comunemente analizzati per fini diagnostici e di ricerca. In questo contesto, è spesso necessario applicare filtri spaziali, come il filtro mediano, direttamente sull'intero volume 3D per ridurre il rumore preservando al contempo i contorni delle strutture anatomiche.

L'algoritmo presentato può essere facilmente esteso per lavorare su immagini tridimensionali. La logica generale rimane invariata: si continua a calcolare la mediana un bit alla volta utilizzando le stesse idee, tuttavia invece di contare i punti all'interno di una finestra di un piano bidimensionale, occorre contare i punti all'interno di finestre tridimensionali. Applicando lo stesso approccio geometrico visto in precedenza, possiamo riformulare il problema in termini di intersezioni tra segmenti e rettangoli. Applicando l'algoritmo di sweepline questo può essere risolto utilizzando un Fenwick Tree in due dimensioni, mostrato nell'Algoritmo 4, che permette di modificare un valore di una matrice e calcolare la somma di una regione rettangolare con complessità $O(\log W \times \log H)$.

Algoritmo 4: FenwickTree2D(N, M)

```
1  t[0...(N + 1, M + 1)] ← 0
2  func add(x, y, v):                          ▷ aggiunge v alla posizione (x, y)
3      while x ≤ N do
4          j ← y
5          while j ≤ M do
6              t[x, y] += v
7              j = j + lsb(j)
8          x = x + lsb(x)
9
10 func sum(x):                                ▷ restituisce la somma da t[1,1] a t[x,y]
11     s ← 0
12     while x > 0 do
13         j ← y
14         while j > 0 do
15             s += t[x, j]
16             j = j − lsb(j)
17         x = x − lsb(x)
18     return s
19
20 func query(x_0, y_0, x_1, y_1):             ▷ restituisce la somma da t[x_0, y_0] a t[x_1, y_1]
21     return sum(x_1, y_1)
22         − sum(x_0 − 1, y_1) − sum(x_1, y_0 − 1)
23         + sum(x_0 − 1, y_0 − 1)
```

Algoritmo 4: Pseudocodice di un Fenwick Tree in due dimensioni.

La complessità finale diventa quindi $O(B \log W \log H)$, anche in questo caso riducibile a $O(B \log^2 R)$ dividendo l'immagine in blocchi. Nell'implementazione pratica è stato scelto di dividere l'immagine in blocchi di dimensione $4R \times 4R \times 4R$.

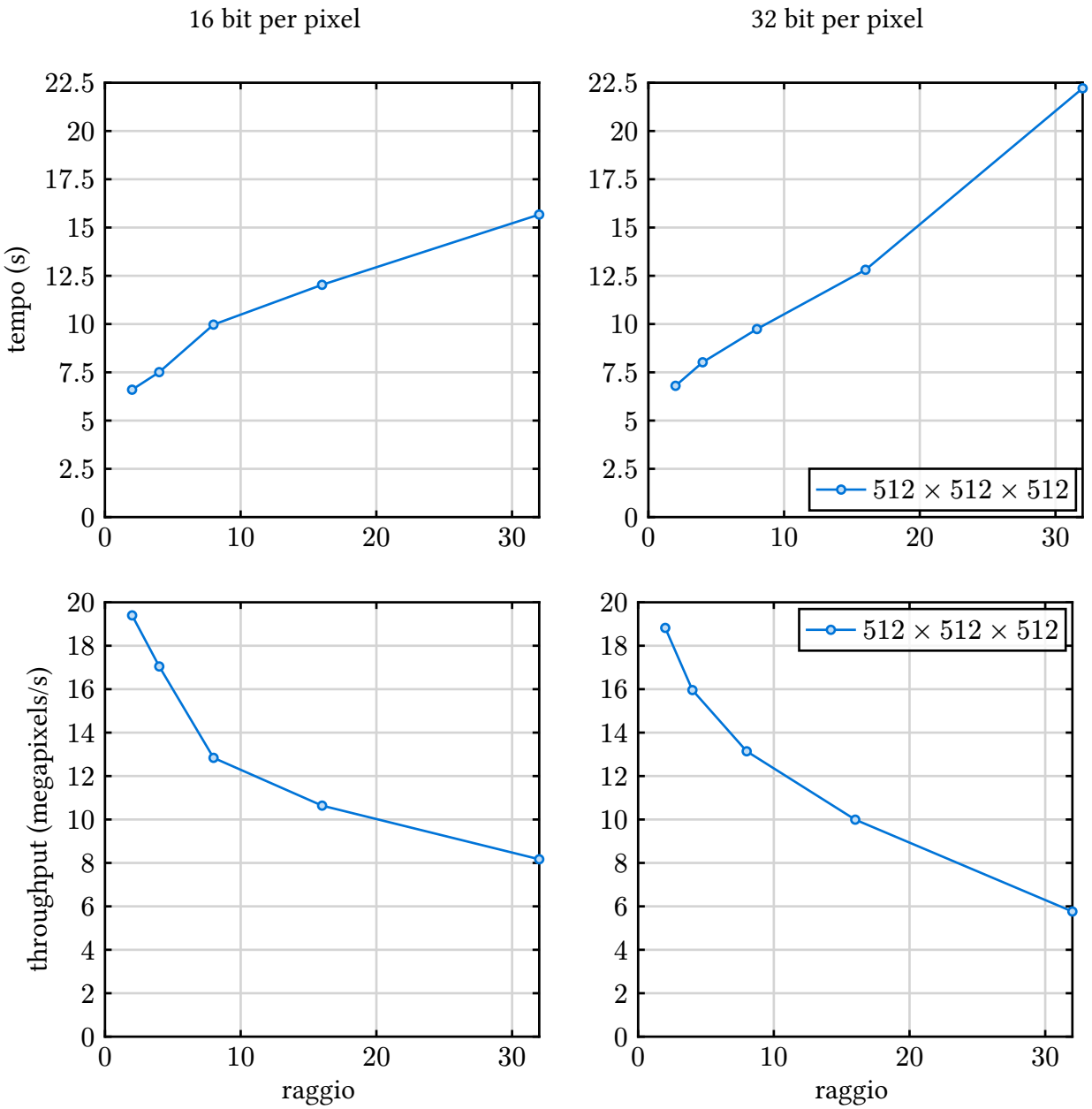

Figura 20: Prestazioni dell'algoritmo 3D, testato su immagini di dimensione $512 \times 512 \times 512$. L'implementazione utilizza le ottimizzazioni dell'approccio brute-force quando almeno uno tra $S1$ e $S2$ è piccolo e della mediana ordinale.

Nella Figura 20 osserviamo che le prestazioni dell'algoritmo diminuiscono velocemente all'aumentare del raggio, a causa della dimensione della ghost area, che in 3 dimensioni cresce molto rapidamente e occupa una porzione significativa del blocco. Nonostante ciò, l'algoritmo mostra buone prestazioni, con un throughput di circa 10 megapixel al secondo per $R = 16$.

Conclusioni

Il filtro mediano è uno degli strumenti più efficaci e diffusi per la rimozione del rumore nelle immagini, in particolare per il rumore impulsivo, grazie alla sua capacità di preservare i contorni e i dettagli rilevanti. Nonostante il suo diffuso utilizzo, il suo calcolo in modo efficiente rappresenta ancora oggi una sfida significativa, soprattutto quando si opera su immagini ad alta profondità e con raggi di filtro elevati.
In questa tesi sono stati analizzati diversi algoritmi e implementazioni esistenti per il calcolo del filtro mediano, mettendone in evidenza punti di forza e limitazioni. È stato quindi proposto un nuovo algoritmo per CPU, progettato per superare le principali limitazioni riscontrate, in particolare per quanto riguarda le prestazioni in presenza di immagini ad alta profondità e raggi elevati. L'algoritmo è stato descritto in dettaglio e sono state presentate le ottimizzazioni utilizzate per migliorare le prestazioni pratiche. L'implementazione è stata testata su immagini di diverse dimensioni e con diverse profondità, e ne sono stati analizzati i risultati. Il confronto con altre soluzioni esistenti ha evidenziato la superiorità dell'approccio rispetto a tutte le implementazioni confrontate, migliorando lo stato dell'arte per il calcolo del filtro mediano con immagini ad alta profondità e raggi elevati.
È stata inoltre presentata un'estensione dell'algoritmo adattata per immagini tridimensionali, sempre più diffuse ad esempio in ambito medico, e ne sono state valutate le prestazioni. Possibili sviluppi futuri includono l'adattamento dell'algoritmo per lavorare su finestre non rettangolari, che stanno diventando sempre più studiate e utilizzate. Rimangono aperti anche possibili miglioramenti dell'implementazione, ad esempio attraverso l'utilizzo di tecniche SIMD, ovvero di parallelismo a livello di istruzioni, o la possibilità di adattare l'algoritmo per un utilizzo su GPU, compito reso però complesso dalla struttura dell'algoritmo, che richiede l'accesso a dati non contigui e la gestione di strutture dati complesse come il Fenwick Tree.

Bibliografia

[1] G. Perrot, S. Domas, e R. Couturier, «How separable median filters can get better results than full 2D versions: Theoretical approach, experimental study and GPU-optimized implementation», *J. Supercomput.*, vol. 78, fasc. 7, pp. 10118–10148, mag. 2022, doi: 10.1007/s11227-021-04233-1.

[2] C. A. R. Hoare, «Algorithm 65: find», *Commun. ACM*, vol. 4, fasc. 7, pp. 321–322, lug. 1961, doi: 10.1145/366622.366647.

[3] O. Sigvardsson, «Simple Sorting Network Full Operation». [Online]. Disponibile su: https://commons.wikimedia.org/wiki/File:SimpleSortingNetworkFullOperation.svg

[4] M. Kim, D. Kim, M. Sung, e W. W. Ro, «An accelerated separable median filter with sorting networks», in *2015 IEEE International Conference on Image Processing (ICIP)*, 2015, pp. 803–807. doi: 10.1109/ICIP.2015.7350910.

[5] G. Perrot, S. Domas, e R. Couturier, «Fine-tuned High-speed Implementation of a GPU-based Median Filter», *Journal of Signal Processing Systems*, vol. 75, fasc. 3, pp. 185–190, giu. 2014, doi: 10.1007/s11265-013-0799-2.

[6] A. Adams, «Fast median filters using separable sorting networks», *ACM Trans. Graph.*, vol. 40, fasc. 4, lug. 2021, doi: 10.1145/3450626.3459773.

[7] T. Huang, G. Yang, e G. Tang, «A fast two-dimensional median filtering algorithm», *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 27, fasc. 1, pp. 13–18, 1979, doi: 10.1109/TASSP.1979.1163188.

[8] S. Perreault e P. Hebert, «Median Filtering in Constant Time», *IEEE Transactions on Image Processing*, vol. 16, fasc. 9, pp. 2389–2394, 2007, doi: 10.1109/TIP.2007.902329.

[9] B. Weiss, «Fast median and bilateral filtering», *ACM Trans. Graph.*, vol. 25, fasc. 3, pp. 519–526, lug. 2006, doi: 10.1145/1141911.1141918.

[10] G. Bradski, «The OpenCV Library», *Dr. Dobb's Journal of Software Tools*, 2000.

[11] G. Pau, F. Fuchs, O. Sklyar, M. Boutros, e W. Huber, «EBImage—an R package for image processing with applications to cellular phenotypes», *Bioinformatics*, vol. 26, fasc. 7, pp. 979–981, 2010, doi: 10.1093/bioinformatics/btq046.

[12] M. Marzolla, M. Ravaioli, e E. L. Piccolomini, « Parallel Median Filter with Arbitrary Window Size and Image Depth », in *2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Los Alamitos, CA, USA: IEEE Computer Society, mar. 2025, pp. 9–12. doi: 10.1109/PDP66500.2025.00010.

[13] Y. Moroto e N. Umetani, «Constant Time Median Filter Using 2D Wavelet Matrix», *ACM Trans. Graph.*, vol. 41, fasc. 6, nov. 2022, doi: 10.1145/3550454.3555512.

[14] P. M. Fenwick, «A new data structure for cumulative frequency tables», *Software: Practice and Experience*, vol. 24, fasc. 3, pp. 327–336, 1994, doi: 10.1002/spe.4380240306.

[15] kimkoungho, «Fenwick Tree». [Online]. Disponibile su: https://github.com/kimkoungho/kimkoungho.github.io/blob/master/assets/images/posts/20181020/fenwick_tree_01.jpeg

[16] TeKappa, «Imola - Rocca sforzesca di Imola». [Online]. Disponibile su: https://commons.wikimedia.org/wiki/File:Imola_-_Rocca_sforzesca_di_Imola_-_2024-09-22_17-43-13_001.jpg

[17] Intel Corporation, «Intel® Integrated Performance Primitives». 2022. [Online]. Disponibile su: https://www.intel.com/content/www/us/en/developer/tools/oneapi/ipp.html