Convoluzione Parallela con CUDA

Lorenzo

May 1, 2025

Abstract

Questo progetto affronta il problema della convoluzione di immagini mediante tecniche di parallelizzazione su GPU, con l'obiettivo di ridurre significativamente i tempi di elaborazione rispetto a una versione sequenziale. L'implementazione è stata realizzata utilizzando CUDA, con due approcci distinti: uno non vettorializzato e uno vettorializzato.

Sono stati testati diversi kernel convolutivi (tra cui sharpening, blur e edge detection) su immagini di differenti risoluzioni, da 480x270 fino a 8k. L'uso della `constant memory` per memorizzare i kernel ha permesso di ottimizzare ulteriormente l'accesso ai coefficienti durante l'esecuzione.

I risultati sperimentali mostrano uno speedup significativo per entrambe le versioni CUDA rispetto alla CPU, con una superiorità della versione vettorializzata per immagini ad alta risoluzione. L'analisi è stata condotta attraverso un programma Python che ha prodotto grafici comparativi delle performance in funzione del numero di thread e della risoluzione dell'immagine.

Questo lavoro evidenzia i vantaggi concreti della parallelizzazione per applicazioni di image processing e mostra come tecniche di ottimizzazione mirate possano influenzare le prestazioni in base al contesto.

Contents

1 Introduzione

Il progetto presentato si concentra sull'applicazione della convoluzione di immagini attraverso tecniche di parallelizzazione utilizzando CUDA. L'obiettivo è quello di migliorare le prestazioni computazionali rispetto a un'implementazione sequenziale, sfruttando le potenzialità delle GPU.

Nel corso del progetto, è stato realizzato un confronto dettagliato tra diverse versioni di convoluzione: una sequenziale CPU, una parallela non vettorializzata CUDA e una versione vettorializzata che sfrutta ulteriormente le capacità di calcolo simultaneo.

Sono stati utilizzati diversi kernel di convoluzione, come *identity*, *sharpen*, *edge detection*, *gaussian blur*, sia 3x3 che 5x5, per testare le prestazioni e la correttezza del risultato. La gestione dei kernel è stata centralizzata in una classe che li memorizza come matrici, utilizzate in fase di convoluzione.

Nelle sezioni seguenti si descrivono le tecniche di parallelizzazione adottate, con particolare attenzione all'organizzazione della memoria, l'uso della *constant memory* e l'ottimizzazione con accessi vettoriali. Infine, vengono presentati i risultati sperimentali in termini di tempo di esecuzione, evidenziando lo speedup ottenuto con CUDA rispetto all'approccio sequenziale.

2 La convoluzione delle immagini

La convoluzione è un'operazione fondamentale nell'elaborazione delle immagini, utilizzata per applicare filtri e trasformazioni che modificano le caratteristiche visive di un'immagine. Tra gli utilizzi più comuni vi sono il rilevamento dei bordi, la sfocatura, la nitidezza e la riduzione del rumore.

Formalmente, la convoluzione bidimensionale tra un'immagine I e un kernel K (o filtro) di dimensione $n \times n$ è definita come:

$$(I * K)(x, y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} I(x+i, y+j) \cdot K(i+k, j+k)$$

dove $k = \lfloor n/2 \rfloor$, $I(x+i, y+j)$ rappresenta il valore del pixel dell'immagine in posizione traslata, e $K(i+k, j+k)$ è il valore corrispondente del kernel centrato.

Il risultato della convoluzione è una nuova immagine in cui ciascun pixel è ottenuto come combinazione lineare pesata dei pixel vicini, con i pesi forniti dal kernel. In pratica, per ogni pixel si prende una finestra centrata su di esso e si calcola la somma dei prodotti dei valori dei pixel con i valori del kernel.

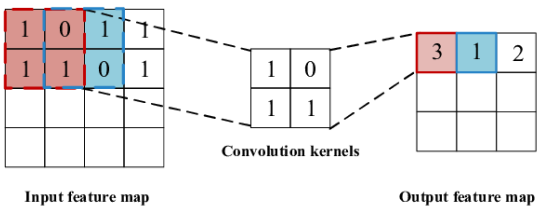

Figure 1: Illustrazione del processo di convoluzione. Il kernel viene applicato a ciascun pixel dell'immagine per produrre l'output filtrato.

2.1 Esempi di kernel comuni

Alcuni esempi di kernel utilizzati nel progetto includono:

- **Identity**: non altera l'immagine.
- **Edge detection**: evidenzia i contorni.
- **Sharpen**: aumenta la nitidezza.
- **Gaussian blur**: sfoca l'immagine in modo naturale.
- **Box blur**: sfoca uniformemente.
- **Unsharp mask**: evidenzia i dettagli mantenendo una sfocatura globale.

Questi kernel possono avere dimensioni diverse (ad esempio 3x3 o 5x5) e sono generalmente simmetrici rispetto al centro. La scelta del kernel influisce direttamente sull'effetto ottenuto sull'immagine risultante.

2.2 Gestione dei bordi

Un aspetto critico della convoluzione è la gestione dei bordi: i pixel situati nelle vicinanze dei margini non hanno tutti i vicini richiesti per l'intera finestra del kernel. Le strategie più comuni per affrontare questo problema includono:

- Ignorare i pixel che cadono fuori dall'immagine (come fatto in questo progetto).
- Applicare padding (riempimento) dell'immagine con zeri o replicando i bordi.

Nel presente lavoro, si è adottato l'approccio di ignorare i pixel fuori dai limiti, assicurandosi che non vi siano accessi fuori memoria durante l'esecuzione, sia nella versione sequenziale che in quella parallela.

3 Algoritmo sequenziale in C++

Prima di introdurre la parallelizzazione, è stata sviluppata una versione sequenziale dell'algoritmo di convoluzione, scritta in C++. Questa versione costituisce il punto di partenza per il confronto in termini di prestazioni e correttezza dei risultati.

L'algoritmo è implementato all'interno della classe `ResizeImage`, che riceve in input:

- Un array lineare di byte rappresentante l'immagine RGB.
- La larghezza e l'altezza dell'immagine.
- Un kernel bidimensionale (matrice di float) che rappresenta il filtro da applicare.

3.1 Funzionamento dell'algoritmo

L'algoritmo si basa su tre cicli annidati:

1. Scorrimento verticale dei pixel (asse y).
2. Scorrimento orizzontale (asse x).
3. Scorrimento dei tre canali RGB (valori 0, 1, 2).

Per ciascun pixel e ciascun canale, viene eseguita una convoluzione con il kernel centrato sul pixel. Il valore convoluto viene poi scritto nell'immagine in output.

Il calcolo è effettuato dalla funzione privata `applyKernel`, che ritorna il valore risultante come `unsigned char`. Il valore finale è ottenuto sommando i contributi dei pixel vicini pesati dai coefficienti del kernel.

Algorithm 1 Funzione `transform` (versione sequenziale)

```
1: function TRANSFORM
2:     Alloca array output di dimensione w × h × 3
3:     for y ← 0 to h − 1 do
4:         for x ← 0 to w − 1 do
5:             for c ← 0 to 2 do
6:                 output[(y · w + x) · 3 + c] ← APPLYKERNEL(x, y, c)
7:             end for
8:         end for
9:     end for
10:    return output
11: end function
```

Algorithm 2 Funzione `applyKernel` (versione sequenziale)

```
1: function applyKernel(x, y, c)
2:     acc ← 0
3:     for i ← −offset to offset do
4:         for j ← −offset to offset do
5:             nx ← x + j
6:             ny ← y + i
7:             if nx < 0 or ny < 0 or nx ≥ w or ny ≥ h then
8:                 continue
9:             end if
10:            idx ← (ny · w + nx) · 3 + c
11:            acc ← acc + img_in[idx] · kernel[i + offset][j + offset]
12:        end for
13:    end for
14:    return clamp(acc, 0, 255)
15: end function
```

L'output viene poi "clippato" nel range [0, 255] per garantire che il valore finale sia valido per un canale RGB. Questo è necessario perché la convoluzione può generare valori fuori dal range ammesso per i colori.

3.2 Limiti della versione sequenziale

Sebbene semplice da implementare, la versione sequenziale presenta dei limiti significativi:

- Il tempo di esecuzione cresce linearmente con il numero di pixel e con la dimensione del kernel.
- L'algoritmo non sfrutta le potenzialità delle architetture moderne multi-core o GPU.
- Il tempo richiesto per processare immagini ad alta risoluzione diventa rapidamente elevato.

Questi limiti hanno motivato lo sviluppo di una versione parallela dell'algoritmo, descritta nelle sezioni successive.

4 Parallelizzazione con CUDA

Per migliorare le prestazioni dell'algoritmo di convoluzione, si è scelto di utilizzare CUDA (Compute Unified Device Architecture), una piattaforma di calcolo parallelo sviluppata da NVIDIA che permette di eseguire codice sulla GPU. In particolare, l'obiettivo è stato quello di sfruttare la parallelizzazione massiva offerta dai thread CUDA per velocizzare il processo di convoluzione di immagini, che risulta naturalmente adatto a una parallelizzazione spaziale.

L'algoritmo si basa sull'idea che ogni thread della GPU possa occuparsi del calcolo di un singolo pixel dell'immagine in output. Poiché l'operazione di convoluzione per un pixel è indipendente da quella degli altri (ad eccezione della lettura dei valori vicini nell'immagine originale), il problema si presta bene alla decomposizione in task paralleli.

4.1 Struttura generale del kernel CUDA

Nel file `ResizeImageParallel.cu` è stato definito un kernel CUDA che esegue la convoluzione. In questo contesto, ogni thread della GPU ha il compito di elaborare un singolo pixel dell'immagine di output. Affinché ciò avvenga correttamente, è necessario che ciascun thread conosca le proprie coordinate globali (x, y) all'interno dell'immagine.

In CUDA, i thread sono organizzati in **blocchi bidimensionali**, e questi blocchi sono a loro volta organizzati in una **griglia bidimensionale**. Ogni thread ha un identificativo locale all'interno del proprio blocco (`threadIdx`) e ogni blocco ha un identificativo globale nella griglia (`blockIdx`). Le dimensioni dei blocchi sono specificate dal parametro `blockDim`.

Per determinare le coordinate (x, y) di un thread rispetto all'intera immagine (coordinate globali), si utilizzano le seguenti formule:

Algorithm 3 Calcolo delle coordinate nel kernel CUDA

1: $x \leftarrow \texttt{blockIdx.x} \cdot \texttt{blockDim.x} + \texttt{threadIdx.x}$
2: $y \leftarrow \texttt{blockIdx.y} \cdot \texttt{blockDim.y} + \texttt{threadIdx.y}$

Il valore di x rappresenta la colonna dell'immagine su cui il thread sta lavorando, mentre y rappresenta la riga. L'idea chiave è che si moltiplica la posizione del blocco per la dimensione del blocco stesso per ottenere un offset globale, a cui si somma la posizione locale del thread all'interno del blocco.

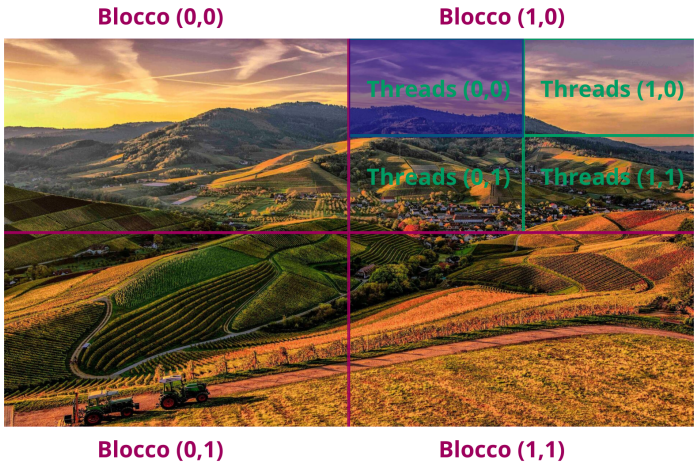

Figure 2: Calcolo delle coordinate globali a partire da blocchi e thread (esempio con blocchi 2×2)

Nell'esempio illustrato, ogni blocco contiene 2×2 thread. Il thread evidenziato in blu si trova nella prima riga e prima colonna del `Block (1,0)`, con `threadIdx = (0,0)`. Applicando le formule sopra, si ottiene:

$$x = 1 \cdot 2 + 0 = 2 \qquad y = 0 \cdot 2 + 0 = 0$$

Il thread lavorerà quindi sul pixel $(2, 0)$ dell'immagine. Questo calcolo viene ripetuto per ogni thread attivo nella griglia, garantendo che tutti i pixel dell'immagine siano processati in parallelo.

La corretta comprensione del calcolo delle coordinate è fondamentale per evitare accessi fuori dai limiti dell'immagine e per garantire la coerenza del risultato nel processo di convoluzione.

4.2 Utilizzo della `constant memory`

Per ottimizzare le performance del kernel CUDA, è stato scelto di memorizzare la matrice di convoluzione (kernel) all'interno della `constant memory`. La constant memory è una particolare area della memoria della GPU, caratterizzata dal fatto che:

- è a **sola lettura** da parte dei thread del device;
- viene caricata in una **cache L1 condivisa** tra tutti i multiprocessori;
- è particolarmente efficiente quando tutti i thread accedono agli **stessi indirizzi di memoria** simultaneamente.

Queste proprietà la rendono ideale per contenere strutture dati statiche condivise tra tutti i thread, come appunto le matrici dei kernel di convoluzione. In questo progetto, il kernel è un array bidimensionale i cui valori sono identici per ogni thread che lo applica a una diversa zona dell'immagine.

Definizione nella constant memory

La dichiarazione del kernel nella constant memory avviene con la seguente istruzione, posta all'inizio del file `ResizeImageParallel.cu`:

```
__constant__ float const_kernel[25];
```

Questo array ha dimensione massima 5×5, sufficiente per tutti i kernel utilizzati nel progetto (inclusi quelli Gaussiani e Unsharp). Poiché le dimensioni massime della constant memory sono limitate (di solito 64KB), è importante mantenere il kernel compatto.

Copia del kernel dal lato host

Poiché la constant memory è accessibile solo dal device, ma viene inizializzata dal codice host (CPU), si utilizza la funzione `cudaMemcpyToSymbol` per trasferire il kernel dalla RAM alla constant memory della GPU:

```
cudaMemcpyToSymbol(const_kernel, kernel_host, size * sizeof(float));
```

Dove:

- `const_kernel` è il nome del simbolo dichiarato nel device.
- `kernel_host` è il puntatore all'array contenente il kernel lato host.
- `size` è il numero di elementi (tipicamente $ksize \times ksize$).

Questa operazione deve essere effettuata **prima del lancio del kernel CUDA**, affinché i dati siano già residenti nella GPU al momento dell'esecuzione.

Benefici osservati

L'utilizzo della constant memory ha permesso di:

- Ridurre la latenza di accesso ai coefficienti del kernel.
- Migliorare la coalescenza dei dati tra i thread.
- Semplificare la logica del codice CUDA (i coefficienti sono ora globali e non passati come argomenti).

Questa ottimizzazione, sebbene invisibile nel comportamento funzionale del programma, ha contribuito in modo significativo alle performance ottenute nei test sperimentali, soprattutto con kernel di dimensioni maggiori e immagini ad alta risoluzione.

4.3 Gestione delle dimensioni e casi bordo

La gestione dei casi al bordo dell'immagine è un aspetto cruciale. I pixel che si trovano vicino ai bordi non hanno tutti i vicini necessari per l'intera finestra del kernel. Per gestire questi casi, l'implementazione verifica che le coordinate `nx` e `ny` siano valide prima di accedere all'immagine di input. I pixel non validi vengono ignorati nel calcolo della somma convolutiva.

Questa gestione consente di evitare accessi illegali alla memoria e preserva la correttezza dell'elaborazione.

5 Descrizione degli algoritmi

In questa sezione vengono descritte in dettaglio le implementazioni degli algoritmi di convoluzione per immagini nella versione parallela con CUDA. Entrambe le versioni parallele condividono una logica comune: per ogni pixel dell'immagine, viene calcolata la somma pesata dei pixel circostanti secondo una matrice di convoluzione (kernel). Tuttavia, il modo in cui questa logica è eseguita differisce radicalmente.

5.1 Versione CUDA non vettorializzata

Nella versione CUDA non vettorializzata, ogni thread CUDA è responsabile del calcolo della convoluzione di un singolo pixel e di un singolo canale RGB. Rispetto alla versione sequenziale, questa soluzione è altamente parallelizzata: ogni thread lavora in modo indipendente su una componente colore di un pixel.
L'algoritmo si basa sui seguenti passaggi:

- Ogni thread calcola le proprie coordinate globali (x, y) e il canale c.
- Il kernel viene applicato centrato su (x, y) solo per il canale c.
- Il valore convoluto viene scritto direttamente in memoria globale nell'array di output.

Il calcolo delle coordinate nel kernel CUDA non vettorializzato serve ad assegnare a ciascun thread la porzione specifica di immagine (e il canale colore) su cui lavorare. In particolare:

- La variabile x rappresenta la **colonna dell'immagine** su cui il thread sta operando.
- La variabile y rappresenta la **riga dell'immagine** corrispondente.
- La variabile c identifica il **canale colore** (0 per R, 1 per G, 2 per B).

Ogni thread CUDA ha un identificativo all'interno del proprio blocco tridimensionale:

- `threadIdx.x`, `threadIdx.y` e `threadIdx.z`

Questi identificativi, combinati con la posizione globale del blocco (`blockIdx`) e le dimensioni dei blocchi (`blockDim`), permettono di calcolare le **coordinate assolute** del thread rispetto all'intera griglia CUDA.
Nel nostro caso:

- Ogni thread elabora un singolo pixel (x, y).
- Ogni thread è assegnato anche a un solo canale c (Red, Green o Blue), gestito tramite `threadIdx.z`.

Questo schema consente di parallelizzare non solo sui pixel, ma anche sui canali, ottenendo un parallelismo completo sull'intera immagine RGB.

Algorithm 4 Calcolo delle coordinate nel kernel CUDA non vettorializzato

1: $x \leftarrow$ `blockIdx.x` $\times$ `blockDim.x` + `threadIdx.x`
2: $y \leftarrow$ `blockIdx.y` $\times$ `blockDim.y` + `threadIdx.y`
3: $c \leftarrow$ `threadIdx.z`

La logica completa del kernel è la seguente:

Algorithm 5 Kernel CUDA non vettorializzato per convoluzione RGB

function convolveKernel_NonVec($input$, $output$, $width$, $height$, $ksize$)
  Calcola x, y, c come in Alg. 4
  $offset \leftarrow \lfloor ksize/2 \rfloor$
  if $x \geq width$ or $y \geq height$ or $c \geq 3$ **then**
    return
  end if
  $acc \leftarrow 0$
  for $i \leftarrow -offset$ to $offset$ **do**
    for $j \leftarrow -offset$ to $offset$ **do**
      $nx \leftarrow x + j$
      $ny \leftarrow y + i$
      if $nx \in [0, width)$ and $ny \in [0, height)$ **then**
        $idx \leftarrow (ny \cdot width + nx) \cdot 3 + c$
        $kidx \leftarrow (i + offset) \cdot ksize + (j + offset)$
        $acc \leftarrow acc + input[idx] \cdot d\_kernel\_const[kidx]$
      end if
    end for
  end for
  $output[(y \cdot width + x) \cdot 3 + c] \leftarrow$ `clamp`(acc)
end function

Dopo aver calcolato le coordinate globali del thread, viene verificato che siano all'interno dei limiti validi dell'immagine. L'operazione di convoluzione si basa su un doppio ciclo annidato che attraversa l'area locale definita dal kernel centrato su (x, y). Per ogni posizione (i, j) attorno al centro:

- Viene calcolato l'indice idx per accedere al pixel dell'immagine.
- Viene calcolato l'indice $kidx$ per accedere al valore corrispondente nel kernel (in forma lineare).
- Il valore del pixel viene moltiplicato per il peso del kernel e accumulato in `acc`.

Alla fine del ciclo, il valore accumulato viene `clippato` nel range $[0, 255]$ e scritto nella posizione corrispondente nell'array di output.

Questa implementazione permette di sfruttare al massimo la parallelizzazione tridimensionale, assegnando a ciascun thread un lavoro specifico su un solo canale di un singolo pixel.

5.2 Versione CUDA vettorializzata

La versione vettorializzata del kernel CUDA è strutturata in modo che ciascun thread elabora un intero pixel, ovvero tutti e tre i canali RGB (Red, Green, Blue) associati alla coordinata (x, y). Questo approccio è più efficiente rispetto alla versione non vettorializzata, in cui ogni thread gestisce un singolo canale, perché riduce il numero totale di thread lanciati e migliora l'accesso coalescente alla memoria globale.
In questa versione:

- Ogni thread elabora un singolo pixel dell'immagine.
- Un ciclo interno sul canale c ($c = 0, 1, 2$) consente di processare R, G e B in sequenza.
- Il kernel viene applicato indipendentemente a ciascun canale.

Il calcolo delle coordinate globali è:

Algorithm 6 Calcolo delle coordinate nel kernel CUDA vettorializzato

$x \leftarrow \texttt{blockIdx.x} \times \texttt{blockDim.x} + \texttt{threadIdx.x}$
$y \leftarrow \texttt{blockIdx.y} \times \texttt{blockDim.y} + \texttt{threadIdx.y}$

La logica completa del kernel è la seguente:

Algorithm 7 Kernel CUDA vettorializzato per convoluzione RGB

function convolveKernel_Vec($input$, $output$, $width$, $height$, $ksize$)
    Calcola x, y come in Alg. 6
    $offset \leftarrow \lfloor ksize/2 \rfloor$
    if $x \geq width$ or $y \geq height$ **then**
        return
    end if
    for $c \leftarrow 0$ to 2 **do**
        $acc \leftarrow 0$
        for $i \leftarrow -offset$ to $offset$ **do**
            for $j \leftarrow -offset$ to $offset$ **do**
                $nx \leftarrow x + j$
                $ny \leftarrow y + i$
                if $nx \in [0, width)$ and $ny \in [0, height)$ **then**
                    $idx \leftarrow (ny \cdot width + nx) \cdot 3 + c$
                    $kidx \leftarrow (i + offset) \cdot ksize + (j + offset)$
                    $acc \leftarrow acc + input[idx] \cdot d\_kernel\_const[kidx]$
                end if
            end for
        end for
        $output[(y \cdot width + x) \cdot 3 + c] \leftarrow \texttt{clamp}(acc)$
    end for
end function

Il kernel `convolveKernel_Vec` implementa una versione vettorializzata della convoluzione, in cui ciascun thread elabora l'intero pixel (x, y), ossia tutti e tre i canali R, G e B. Il thread calcola le proprie coordinate globali x e y, verifica che siano all'interno dei limiti dell'immagine e procede al calcolo della convoluzione per ciascun canale $c = 0, 1, 2$.
Per ogni canale:

- Viene inizializzato un accumulatore `acc` a zero.
- Si scorre l'area del kernel centrata su (x, y).
- Per ogni posizione locale (i, j) si calcolano:
  - le coordinate del pixel da leggere: $(nx, ny) = (x + j, y + i)$;
  - l'indice nell'array lineare dell'immagine: `idx`;
  - l'indice nell'array lineare del kernel: `kidx`.
- Se (nx, ny) è valido, si somma il prodotto tra il valore del pixel e il corrispondente coefficiente del kernel.

Al termine dei cicli, il valore convoluto è clippato nel range $[0, 255]$ e scritto nel pixel (x, y) per il canale c nell'array di output. Questo approccio migliora l'efficienza rispetto alla versione non vettorializzata, riducendo il numero di thread e sfruttando accessi coalescenti alla memoria globale.
Rispetto alla versione non vettorializzata, questa implementazione utilizza un numero di thread tre volte inferiore (uno per pixel invece di uno per canale), riducendo il numero di lanci kernel e migliorando l'efficienza globale del calcolo.

6 Risultati sperimentali

Per valutare l'efficacia della parallelizzazione con CUDA, sono stati eseguiti esperimenti su immagini di diverse risoluzioni, da 480×270 fino a 7680×4320 (8k). I tempi di esecuzione per ciascuna configurazione (non vettorializzata e vettorializzata) sono stati salvati in un file `times.csv` e analizzati tramite uno script Python, che ha generato i grafici qui riportati.

6.1 Confronto tra immagini piccole e immagini 8k

La seguente figura mostra lo speedup per la risoluzione 480×270:

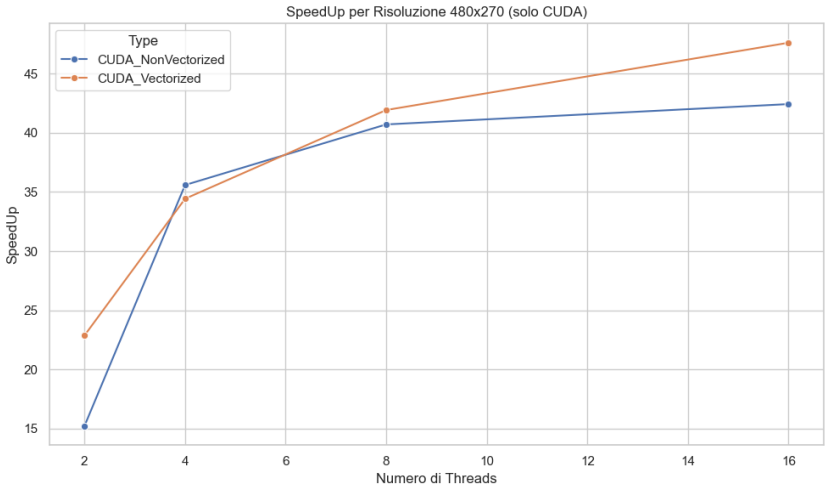

Figure 3: Speedup CUDA per risoluzione 480x270

In questo caso, la differenza tra la versione non vettorializzata e quella vettorializzata è ridotta. La versione vettorializzata ottiene performance leggermente migliori per un numero elevato di thread.

Al contrario, per immagini in alta risoluzione come la $8k$ si osserva una dinamica interessante:

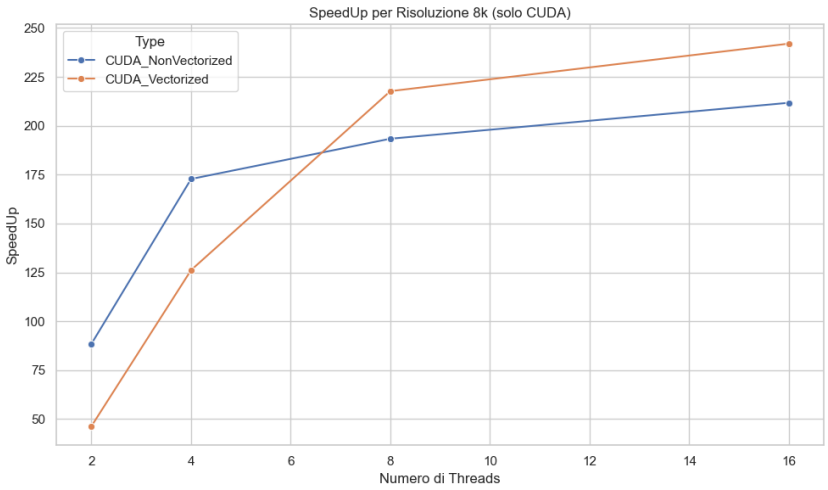

Figure 4: Speedup CUDA per risoluzione 8k

Anche in questo caso, la versione non vettorializzata mantiene prestazioni superiori per valori bassi di thread. Tuttavia, al crescere del numero di thread, la versione vettorializzata riesce a superarla, ottenendo lo speedup massimo.

6.2 Comportamento intermedio su altre risoluzioni

Per completezza, riportiamo anche i grafici relativi a risoluzioni intermedie:

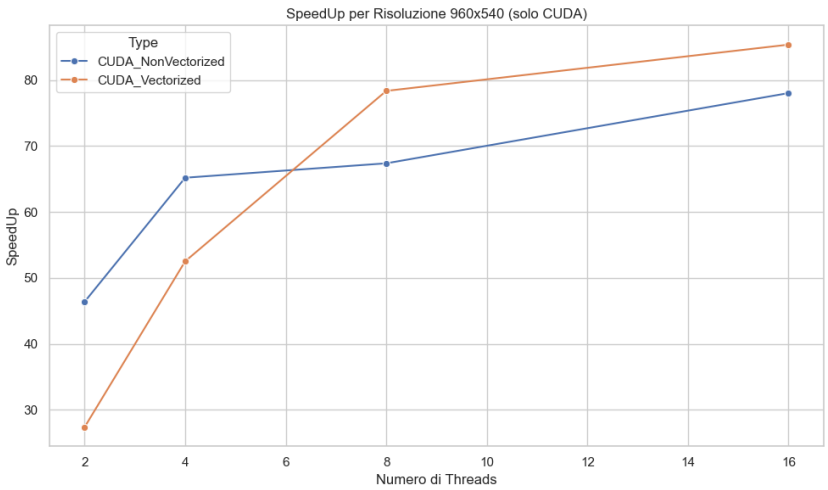

Figure 5: Speedup CUDA per risoluzione 960x540

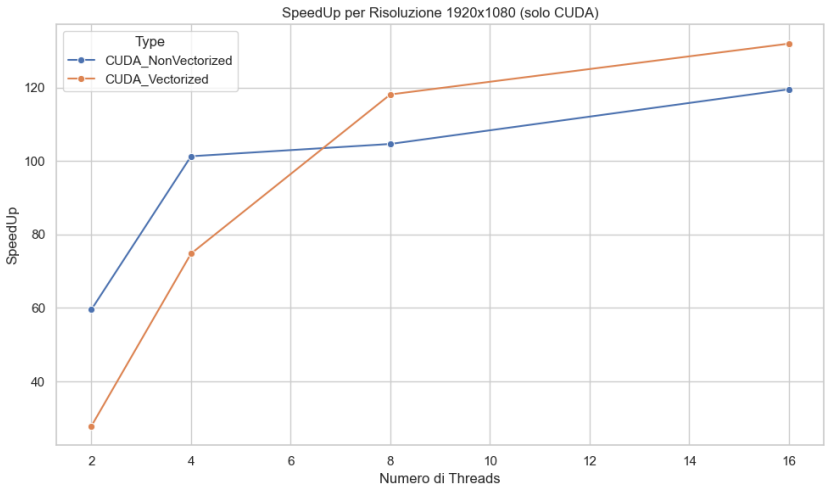

Figure 6: Speedup CUDA per risoluzione 1920x1080

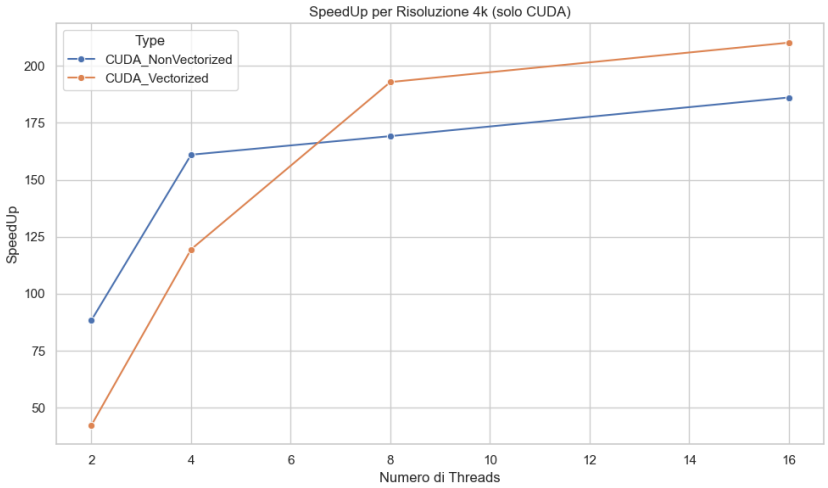

Figure 7: Speedup CUDA per risoluzione 4k

In tutte queste risoluzioni, la versione non vettorializzata mostra migliori prestazioni per un basso numero di thread, mentre la versione vettorializzata migliora man mano che il carico di lavoro per thread aumenta. Tuttavia, raramente riesce a superare completamente la versione non vettorializzata.

6.3 Speedup medio per tipo di esecuzione

La media dello speedup conferma che la versione non vettorializzata è, in media, più veloce:

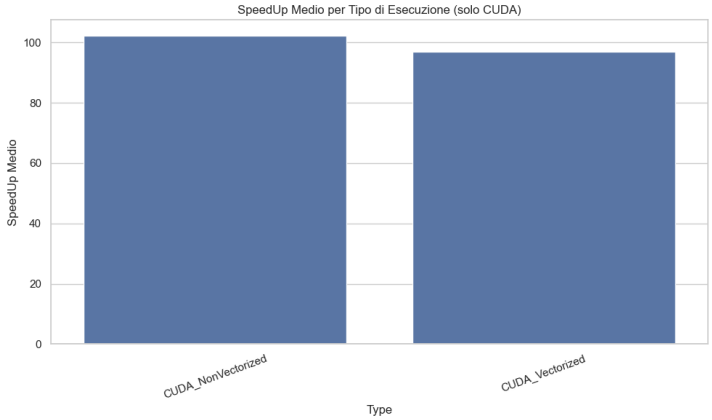

Figure 8: Speedup medio per tipo di esecuzione

Questo è dovuto al fatto che la versione non vettorializzata riesce a parallelizzare meglio sui canali, sfruttando un maggior numero di thread attivi simultaneamente, soprattutto su immagini di piccole e medie dimensioni.

6.4 Tempo medio per tipo di esecuzione

In termini di tempo assoluto, si osserva lo stesso comportamento:

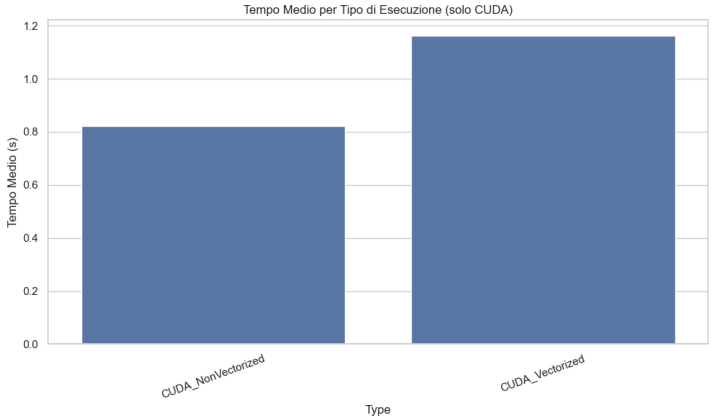

Figure 9: Tempo medio di esecuzione

La versione vettorializzata risulta più lenta, soprattutto per immagini non particolarmente grandi. Questo è dovuto al fatto che ogni thread elabora tutti e tre i canali RGB in sequenza, con minore parallelismo globale rispetto alla versione non vettorializzata.

6.5 Conclusioni sui risultati

Nel complesso, entrambe le versioni parallele mostrano uno speedup elevato rispetto alla versione sequenziale. Tuttavia, i risultati sperimentali confermano che:

- La versione **non vettorializzata** è mediamente più veloce e scalabile su immagini piccole e medie.
- La versione **vettorializzata** può superare l'altra solo su immagini molto grandi e con un numero elevato di thread.

Pertanto, per applicazioni pratiche su GPU di fascia media (come la GTX 1050 Ti), la versione non vettorializzata risulta preferibile.