Convoluzione di Immagini con Parallelizzazione CUDA

Lorenzo Mugnai

Università degli Studi di Firenze

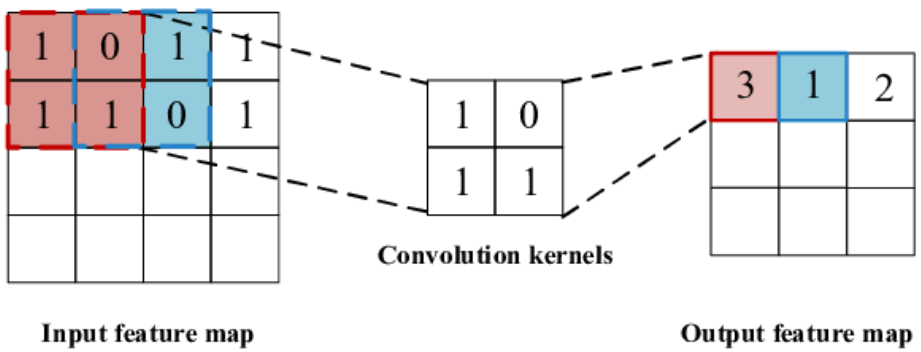
Maggio 2025

Introduzione al Progetto

- Obiettivo: accelerare la convoluzione di immagini RGB utilizzando CUDA.
- Confronto tra tre versioni:
  - **Sequenziale** su CPU (baseline).
  - **CUDA Non Vettorializzato**: 1 thread per canale.
  - **CUDA Vettorializzato**: 1 thread per pixel.
- Analisi su immagini di diverse risoluzioni (fino a 8K).
- Valutazione delle prestazioni tramite tempi di esecuzione e speedup.

Cos'è la Convoluzione

- La convoluzione è un'operazione fondamentale in elaborazione di immagini.
- Consiste nell'applicare un **kernel** (o filtro) a ogni pixel dell'immagine.
- Ogni nuovo pixel è ottenuto come media pesata dei pixel vicini.
- Utilizzata per effetti di *sfocatura*, *bordatura*, *sharpening*, ecc.



Algoritmo Sequenziale su CPU

- Ogni pixel viene elaborato singolarmente in triplo ciclo annidato.
- Per ogni posizione (x, y) e canale c, si applica il kernel.
- Nessuna parallelizzazione: ogni pixel è calcolato in serie.

Algoritmo Sequenziale su CPU

```cpp
unsigned char ResizeImage::applyKernel(int x, int y, int c) {
    float acc = 0.0f;
    for (int i = -offset; i <= offset; ++i) {
        for (int j = -offset; j <= offset; ++j) {
            int nx = x + j;
            int ny = y + i;
            if (nx < 0 || ny < 0 || nx >= w || ny >= h) continue;
            int idx = (ny * w + nx) * 3 + c;
            acc += img_in[idx] * kernel[i + offset][j + offset];
        }
    }
    return static_cast<unsigned char>(std::clamp(acc, 0.0f, 255.0f));
}

unsigned char *ResizeImage::transform() {
    unsigned char *output = new unsigned char[w * h * 3];
    std::memset(output, 0, w * h * 3);

    for (int y = 0; y < h; ++y)
        for (int x = 0; x < w; ++x)
            for (int c = 0; c < 3; ++c)
                output[(y * w + x) * 3 + c] = applyKernel(x, y, c);

    return output;
}
```

Limiti della Versione Sequenziale

- L'intera immagine viene elaborata **pixel per pixel**, canale per canale.
- L'approccio è semplice ma estremamente lento per immagini ad alta risoluzione.
- Complessità computazionale elevata: $\mathcal{O}(w \cdot h \cdot k^2 \cdot 3)$.
- Nessun utilizzo della potenza parallela offerta dalla GPU.

Parallelizzazione con CUDA

- CUDA permette di eseguire migliaia di thread in parallelo sulla GPU.
- Ogni thread elabora una porzione indipendente dell'immagine.
- I thread sono organizzati in:
  - **Blocchi** 2D o 3D.
  - **Griglie** di blocchi.
- L'accesso parallelo ai pixel consente una drastica riduzione del tempo di elaborazione.
- La parallelizzazione è spaziale: un thread $\rightarrow$ un pixel o un canale.

Calcolo delle Coordinate in CUDA

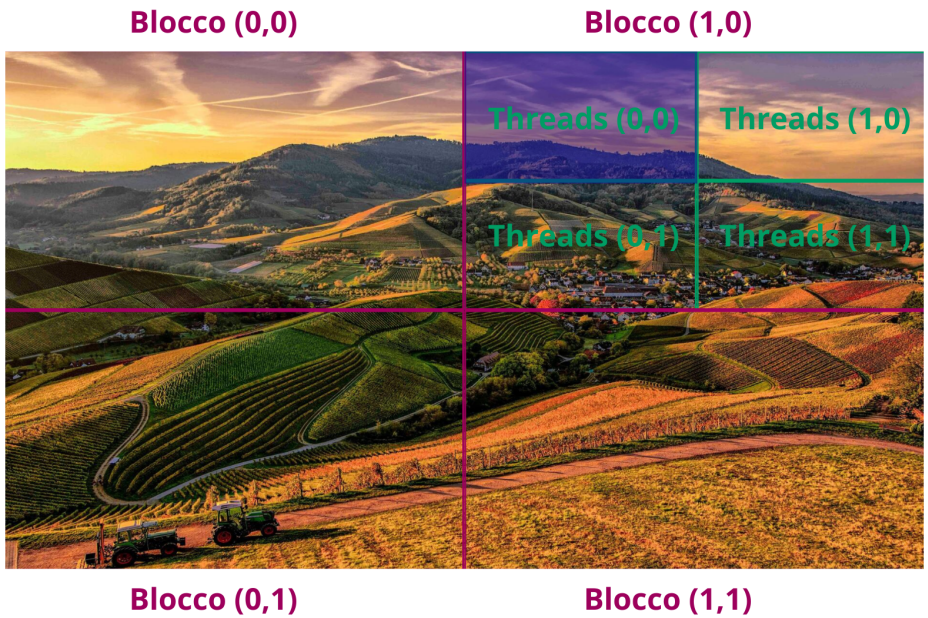
- Ogni thread identifica il pixel da elaborare tramite:

$$x = \texttt{blockIdx.x} \cdot \texttt{blockDim.x} + \texttt{threadIdx.x}$$

$$y = \texttt{blockIdx.y} \cdot \texttt{blockDim.y} + \texttt{threadIdx.y}$$

- Con thread 3D si gestisce anche il canale RGB:

$$c = \texttt{threadIdx.z}$$



Ottimizzazione con `constant memory`

- La `constant memory` è una porzione di memoria **a sola lettura**, condivisa tra tutti i thread.
- Ideale per dati condivisi come il **kernel di convoluzione**.
- Più veloce della global memory se gli accessi sono uniformi.
- Dichiarazione:

$$\_\_constant\_\_ float\ d\_kernel\_const[25];$$

- Copia da host a device:

$$\texttt{cudaMemcpyToSymbol}(d\_kernel\_const, \text{kernel}, sizeofKernel)$$

- Riduce la latenza negli accessi ripetuti durante la convoluzione.

CUDA – Versione Non Vettorializzata

- Ogni **thread elabora un singolo canale** (R, G o B) di un pixel.
- I thread sono lanciati in configurazione 3D: (x, y, c).
- L'indice globale è calcolato con `threadIdx` e `blockIdx`.
- Vantaggio: più thread $\rightarrow$ maggiore parallelismo.
- Svantaggio: maggiori accessi separati in memoria.

```
__global__ void convolveKernel_NonVec(unsigned char* input, unsigned char* output,
                                      int width, int height, int ksize) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int c = threadIdx.z;  // canale RGB

    int offset = ksize / 2;
    if (x >= width || y >= height || c >= 3) return;

    float acc = 0.0f;
    for (int i = -offset; i <= offset; ++i) {
        for (int j = -offset; j <= offset; ++j) {
            int nx = x + j;
            int ny = y + i;
            if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
                int idx = (ny * width + nx) * 3 + c;
                acc += input[idx] * d_kernel_const[(i + offset) * ksize + (j + offset)];
            }
        }
    }
    output[(y * width + x) * 3 + c] = static_cast<unsigned char>(clamp(acc));
}
```

CUDA – Versione Vettorializzata

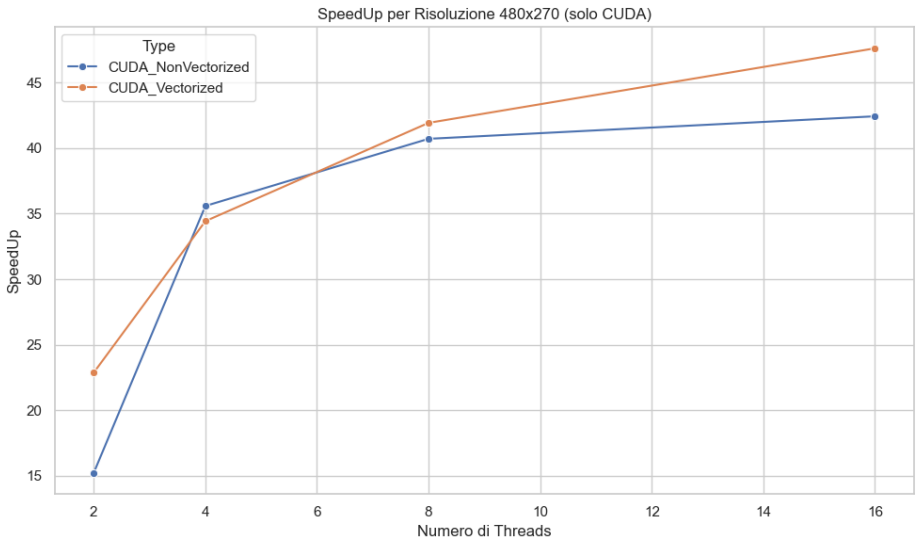
- Ogni **thread elabora un intero pixel** (x, y), gestendo i tre canali RGB in sequenza.
- Riduce il numero totale di thread lanciati.
- I canali vengono processati all'interno di un ciclo `for (c = 0; c < 3)`.
- Più semplice da implementare, ma:
  - meno parallelismo,
  - e più carico per thread singolo.

CUDA – Versione Vettorializzata

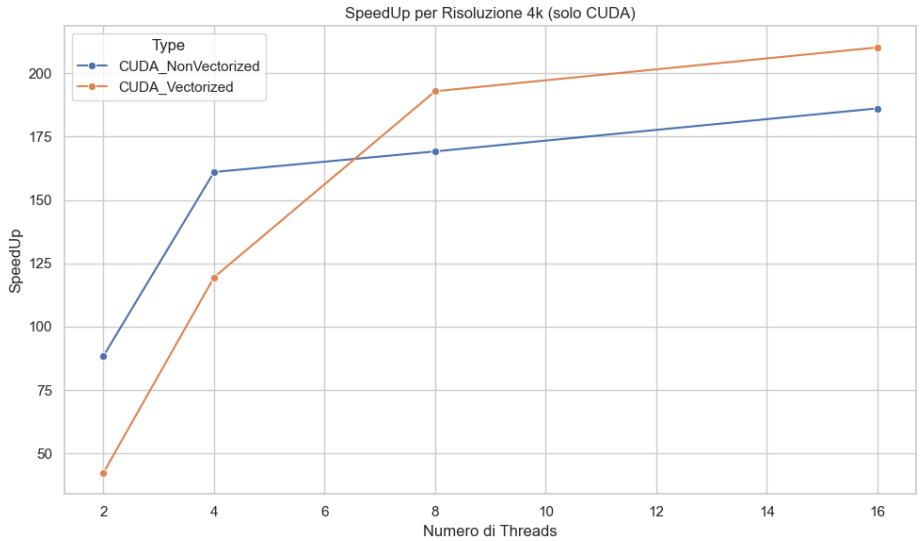
```
__global__ void convolveKernel_Vec(unsigned char* input, unsigned char* output,
                                   int width, int height, int ksize) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    int offset = ksize / 2;
    if (x >= width || y >= height) return;

    for (int c = 0; c < 3; ++c) {
        float acc = 0.0f;
        for (int i = -offset; i <= offset; ++i) {
            for (int j = -offset; j <= offset; ++j) {
                int nx = x + j;
                int ny = y + i;
                if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
                    int idx = (ny * width + nx) * 3 + c;
                    acc += input[idx] * d_kernel_const[(i + offset) * ksize + (j + offset)];
                }
            }
        }
        output[(y * width + x) * 3 + c] = static_cast<unsigned char>(clamp(acc));
    }
}
```

SpeedUp su Immagini di Varie Risoluzioni

- Lo speedup varia al variare della risoluzione e del numero di thread.
- La versione **non vettorializzata** è mediamente più veloce.
- La versione **vettorializzata** scala meglio solo con immagini molto grandi.



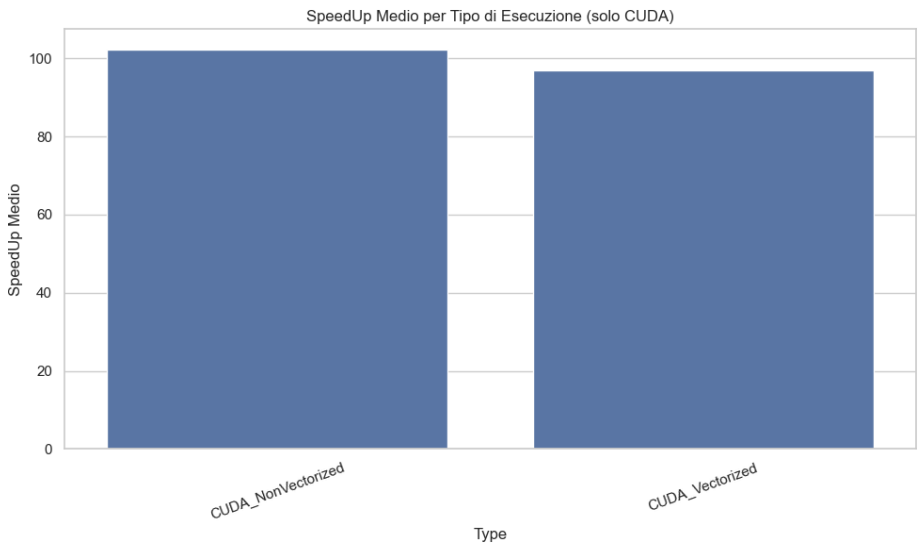
480p



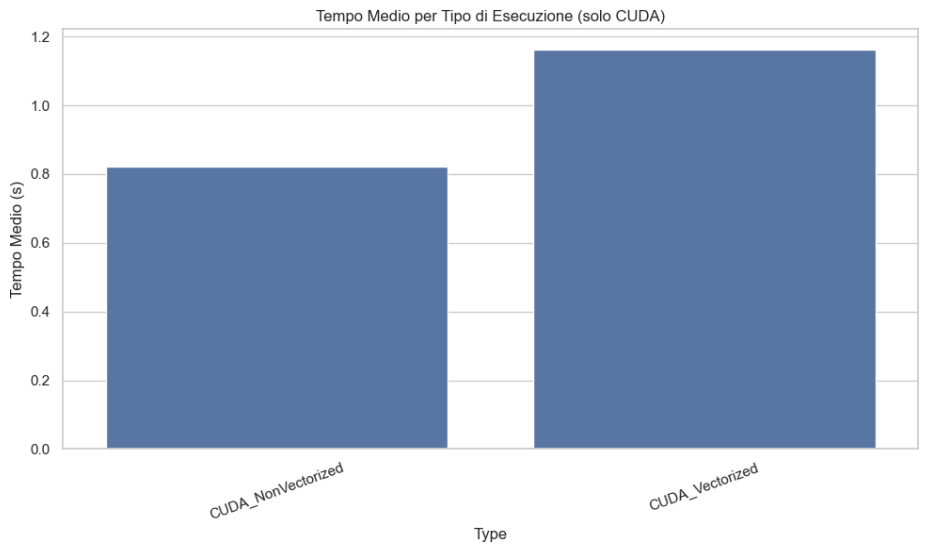
4K

SpeedUp e Tempo Medio

- I grafici mostrano il comportamento medio complessivo dei due approcci.
- La versione **non vettorializzata** è più veloce in media.
- La versione **vettorializzata** introduce overhead per thread singolo.



SpeedUp medio



Tempo medio

Conclusioni

- Entrambe le versioni CUDA hanno mostrato un significativo speedup rispetto alla versione sequenziale.
- La **versione non vettorializzata** si è rivelata più efficiente in quasi tutti i casi:
  - sfrutta un numero maggiore di thread attivi;
  - ottimale su immagini piccole e medie.
- La **versione vettorializzata** è scalabile, ma efficace solo su immagini molto grandi e con molti thread.