CUDA Image Processing

Gregorio Piqué

February 25, 2024

1 Introduzione

Il seguente documento costituisce il resoconto relativo al progetto finale del corso di *Parallel Programming for Machine Learning*, parte integrante del curriculum di Laurea Magistrale in Intelligenza Artificiale presso l'Università degli Studi di Firenze. La realizzazione del progetto ha coinvolto l'implementazione di un elaborato a scelta tra diverse opzioni, richiedendo anche la redazione di una relazione conclusiva che riflettesse e illustrasse i risultati ottenuti.

Il progetto selezionato ha richiesto l'implementazione di una versione parallela dell'operazione di *Image Processing*, utilizzando un *kernel* per applicare effetti sulle immagini, tra cui *blurring*, *sharpening*, *embossing*, *edge detection* e altri[1]. Tale implementazione è stata realizzata come applicazione parallela su GPU mediante l'utilizzo dell'API *CUDA*[2].

2 Convoluzione

La convoluzione è un'operazione che in questo contesto è applicata all'Image Processing, una disciplina che permette di sfruttare algoritmi per l'elaborazione e la manipolazione di immagini digitali[3]. Più nel dettaglio, la convoluzione è un'operazione tra due funzioni di una variabile che consiste nell'integrare il prodotto tra la prima e la seconda traslata di un certo valore [4]. La convoluzione delle funzioni f e g è rappresentata come $(f * g)$ e definita come:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau \tag{1}$$

Nel caso discreto, per sequenze $f[n]$ e $g[n]$, la convoluzione è espressa come:

$$(f * g)[n] = \sum_{k=-\infty}^{\infty} f[k] \cdot g[n-k] \tag{2}$$

Nel contesto dell'image processing, l'elaborazione delle immagini si ottiene eseguendo una convoluzione tra la matrice maschera, nota come *kernel*, e l'immagine stessa. Ciò comporta che ogni pixel nell'immagine di output è il risultato di un'elaborazione dei pixel circostanti a quello corrispondente nell'immagine di input, risultato ottenuto mediante la combinazione con il kernel. In forma matematica, ciò si esprime come segue:

$$\omega * I[m,n] = \sum_{i=-a}^{a} \sum_{j=-b}^{b} \omega[i,j] \cdot I[m-i, n-j] \tag{3}$$

dove ω rappresenta la matrice di kernel, e $I[m,n]$ l'immagine di input. La Figura 1 mostra intuitivamente il funzionamento della convoluzione applicata su immagini [5].

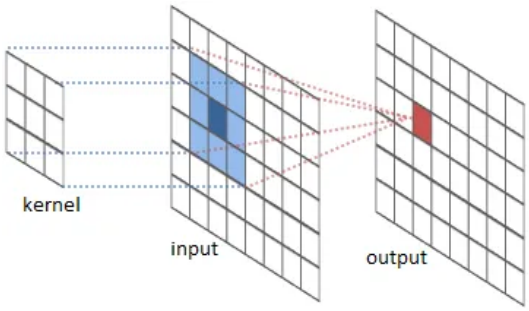

Figure 1: Funzionamento della convoluzione.

3 Metodologia

L'implementazione parallela del progetto è stata realizzata mediante l'utilizzo di CUDA. Questa tecnologia consente lo sviluppo di applicazioni eseguibili su unità di elaborazione grafica (*GPU*),

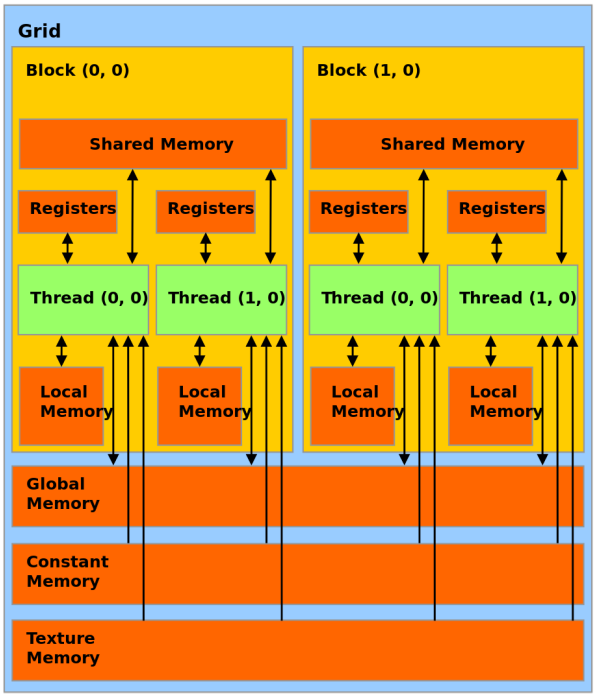

Figure 2: Struttura della memoria nella GPU.

un'architettura che si presta alla parallelizzazione di operazioni di calcolo grazie all'elevato numero di core disponibili.

Un'applicazione CUDA comprende due tipi diversi di funzionalità: codice *host* e codice *device*. Il primo è responsabile per la gestione della CPU e dell'interfaccia con il dispositivo GPU, mentre il codice device è eseguito direttamente sulla GPU ed è responsabile dell'esecuzione parallela delle operazioni specificate [6].

I *device* hanno un *memory model* composto da più tipologie di memoria, ognuna con le proprie caratteristiche; queste sono la *global*, *constant* e *shared* memory.

La *global memory* è una memoria accessibile da tutti i thread e conserva dati persistenti tra le chiamate di kernel. Ha una capacità maggiore, ma l'accesso può essere più lento a causa della latenza. La *constant memory* è una memoria di sola lettura utilizzata per conservare dati costanti durante l'esecuzione del kernel. Ha un accesso veloce, ma la sua capacità è limitata. Nell'applicazione sviluppata, è stata utilizzata per garantire un accesso rapido alla matrice del kernel, costante per tutti i thread per l'intera durata dell'esecuzione. La *shared memory* è invece una memoria condivisa tra i thread di uno stesso blocco, e offre un accesso veloce e sincronizzato. È stata impiegata nell'implementazione di una variante dell'applicazione, sfruttando l'operazione di tiling e il caricamento di una porzione dei dati nella memoria condivisa per ottimizzare la comunicazione locale all'interno del blocco.

La Figura 2 mostra la struttura e le modalità di accesso alle varie tipologie di memoria per i thread della GPU [7].

4 Implementazione

Nella soluzione implementata [8], sono state sviluppate e utilizzate classi di supporto per la gestione dei dati e delle funzionalità dell'algoritmo. Le principali classi implementate sono *Image* per la gestione dell'immagine di input, *Kernel* per la creazione e rappresentazione del filtro scelto, gli enumerativi *FilterType* e *ExecutionMode* usati per definire i tipi di filtri e modalità di esecuzione dell'algoritmo, e infine la classe *Convolution* che offre le funzionalità stesse dell'operazione di convoluzione.

4.1 Classi

Image

La classe *Image* permette la gestione dell'immagine di input poi manipolata e aggiornata con la convoluzione. La classe include campi come *width* e *height* per definire la dimensione dell'immagine, *channels* indicante il numero totale di canali per ogni pixel, *fileName* per il nome del file caricato, e *imageData* che rappresenta i dati dell'immagine stessa. Oltre a costruttori, distruttori e metodi *getter*, la classe sfrutta componenti della libreria esterna *stb*[9] per mettere a disposizione metodi pubblici per caricare e per salvare un'immagine, quali *loadImage()* e *saveImage()*. La Figura 3 presenta la struttura della classe *Image*.

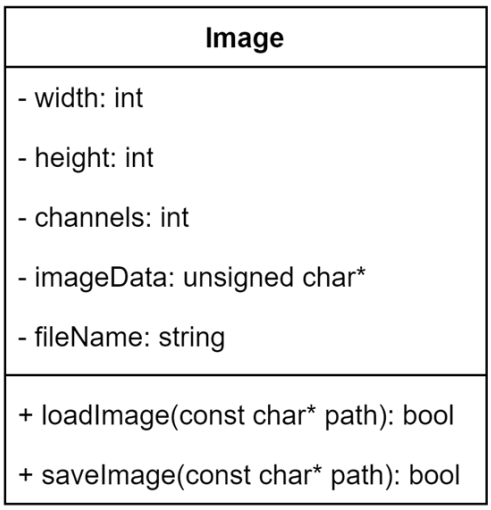

Figure 3: Classe Image.

Kernel

La classe *Kernel* permette di gestire la maschera da usare nella convoluzione. Include campi quali *kernelMatrix* che rappresenta la matrice del filtro, *kernelWidth* e *kernelHeight* che ne indicano la dimensione, *filterType* che ne identifica la tipologia e *filterNormalizationValue*, un valore di normalizzazione usato soprattutto con particolari tipi di filtri (come il filtro *emboss* [10]). Oltre a costruttori, distruttori e metodi *getter*, la classe mette a disposizione il metodo *setFilter()* per impostare e definire quale filtro usare nella convoluzione. La Figura 4 presenta la struttura della classe *Kernel*.

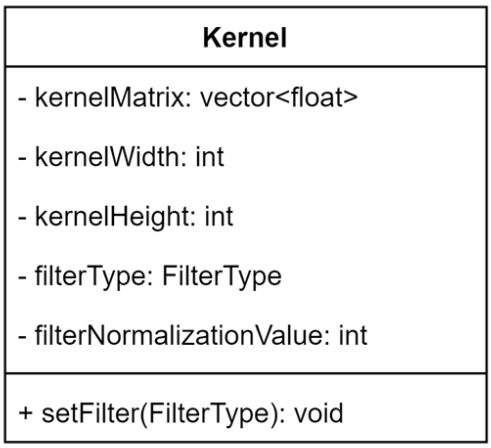

Figure 4: Classe Kernel.

FilterType

L'enumerativo *FilterType* rappresenta l'elenco dei possibili filtri applicabili nella convoluzione. Questi includono *box blur* e *gaussian blur* per creare effetti di sfumatura, *edge detection* per individuare i bordi degli oggetti, *horizontal emboss* e *vertical emboss* per evidenziare aree con forti cambiamenti cromatici, *sharpen* per accentuare i contrasti e simulare una maggiore nitidezza, e *identity* che lascia l'immagine inalterata. La Figura 5 presenta l'enumerativo *FilterType*.

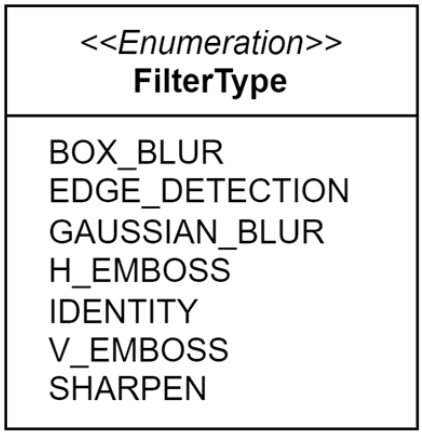

Figure 5: FilterType.

ExecutionMode

L'enumerativo *ExecutionMode* rappresenta le modalità di esecuzione del programma. Permette di definire quale tipologia di memoria della GPU utilizzare: *GLOBAL* porta a basarsi esclusivamente sulla memoria globale, *CONSTANT* sulla memoria costante per la matrice del filtro, mentre *SHARED* porta a sfruttare la shared memory con l'operazione di tiling effettuata sull'immagine e a usare inoltre la constant memory per contenere il kernel. È inoltre possibile eseguire una versione sequenziale dell'algoritmo con la modalità *SEQUENTIAL*. La Figura 6 presenta l'enumerativo *Kernel*.

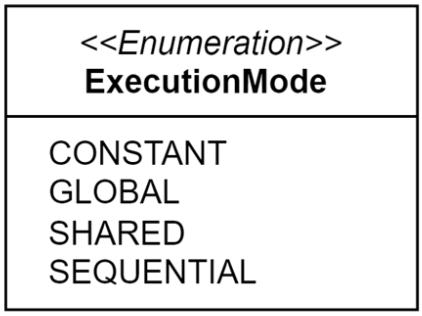

Figure 6: ExecutionMode.

Convolution

La classe *Convolution* fornisce le funzionalità dell'algoritmo. Oltre al costruttore e distruttore, l'unico metodo pubblicamente disponibile è *apply()*, che permette di avviare il processo con la configurazione definita dai parametri specificati; in base a questa, vengono chiamati altri metodi di supporto privati che implementano più nel dettaglio la convoluzione. Ognuno di questi permette infatti di eseguire una propria versione del processo, come sequenziale o parallela, usando *constant*, *global* o *shared memory*. La Figura 7 presenta la struttura della classe *Convolution*.

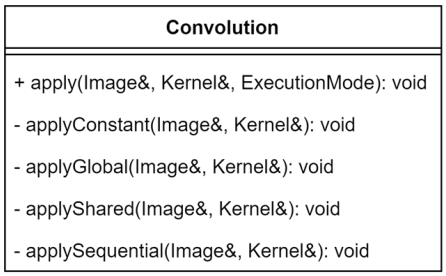

Figure 7: Classe Convolution.

4.2 Codice Sorgente

Questa sezione presenta il codice sorgente delle parti principali del processo. In particolare, vengono riportati e commentati i metodi *applyConstant()* e *constantKernel* della classe *Convolution* per l'esecuzione stessa dell'algoritmo. L'intero codice sorgente sviluppato (con le altre modalità di esecuzione) è pubblicamente disponibile al repository https://github.com/Pikerozzo/cuda-kernel-img-proc.

applyConstant()

Il metodo *applyConstant()* viene chiamato dal metodo di classe pubblico *apply()*, e consente di avviare l'esecuzione della versione parallelizzata dell'algoritmo che fa uso della *constant memory* per memorizzare la matrice del filtro. Una delle prime operazioni del metodo è quella di caricare nella *constant memory* il kernel; questo rappresenta un ottimo esempio di uso di questo tipo di memoria, in quanto il filtro è un dato comune a tutti i thread e resta costante per l'intera esecuzione del processo. Vengono successivamente allocate variabili sulla *device memory* per le immagini di input e di output e copiati i dati. Prima dell'esecuzione del kernel CUDA, viene definita la dimensione del *block* e della *grid*, indicanti rispettivamente il numero di thread in un blocco, e il numero di blocchi in una griglia: idealmente, il numero totale di thread dovrebbe corrispondere alla quantità di dati da processare, che in questo caso sono i pixel dell'immagine. Viene poi lanciato il kernel CUDA, passando come parametri i puntatori delle immagini di input e di output, la dimensione e il numero di canali dell'immagine, e un valore di normalizzazione del filtro. Terminata l'esecuzione parallela su GPU, viene copiata l'immagine di output sull'host, e infine liberate le risorse allocate.

```
1  void Convolution::applyConstant(Image& image, Kernel& kernel) {
2
3    // get the total size of the image and kernel mask
4    size_t imageSize = image.getTotalSize(true);
5    size_t kernelSize = kernel.getTotalSize(true);
6
7    // copy the kernel mask to the constant memory
8    cudaError_t cudaError = cudaMemcpyToSymbol(KERNEL_MASK, kernel.getKernelFilter(),
         kernelSize);
9    // check for CUDA errors
10   if (cudaError != cudaSuccess) {
11     fprintf(stderr, "cudaMemcpyToSymbol failed: %s\n", cudaGetErrorString(cudaError));
12     return;
13   }
14
15   // allocate device memory for input and output images and copy input image data to
         the device
16   unsigned char* dev_imgIn;
17   unsigned char* dev_imgOut;
18   unsigned char* imgOut = (unsigned char*)malloc(imageSize);
19   cudaMalloc((void**)&dev_imgIn, imageSize);
20   cudaMalloc((void**)&dev_imgOut, imageSize);
21   cudaMemcpy(dev_imgIn, image.getImageData(), imageSize, cudaMemcpyHostToDevice);
22   cudaMemcpy(dev_imgOut, imgOut, imageSize, cudaMemcpyHostToDevice);
23
24   // define the grid and block dimensions
25   dim3 dimGrid(ceil((float)image.getWidth() / (float)BLOCK_SIZE), ceil((float)image.
```

```
        getHeight() / (float)BLOCK_SIZE));
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

    // launch the CUDA kernel for constant memory convolution
    constantKernel <<<dimGrid, dimBlock>>>(dev_imgIn, dev_imgOut, image.getWidth(),
        image.getHeight(), image.getChannels(), kernel.getFilterNormalizationValue());

    // check for CUDA errors
    cudaError = cudaGetLastError();
    if (cudaError != cudaSuccess) {
      printf("CUDA error: %s", cudaGetErrorString(cudaError));
    }

    // synchronize device to ensure completion of the kernel
    cudaDeviceSynchronize();

    // copy the image result back to the host and update the image data
    cudaMemcpy(imgOut, dev_imgOut, imageSize, cudaMemcpyDeviceToHost);
    image.setImageData(imgOut);

    // free allocated device memory
    cudaFree(dev_imgIn);
    cudaFree(dev_imgOut);
}
```

constantKernel()

Viene riportata la funzione kernel *constantKernel()*. Come prima cosa, identifica quale pixel il thread dovrà processare con la convoluzione; il calcolo della posizione del pixel avviene con la formula $blockIdx.x \cdot blockDim.x + threadIdx.x$. Questa gestione della posizione permette di effettuare accessi in memoria in modo efficiente, con il pattern di *coalesced memory access*. Vi è poi una condizione che verifica che il thread sia associato a una posizione valida e che si trovi all'interno dell'immagine; questa condizione introduce un punto di divergenza necessario, per garantire il corretto funzionamento dell'algoritmo. Viene poi identificato il pixel di partenza per il calcolo dell'output corrispondente e effettuata l'operazione di convoluzione, usando la matrice *KERNEL_MASK* in *constant memory*. Viene infine aggiornata l'immagine di output con il risultato ottenuto normalizzato.

```
__global__ void constantKernel(unsigned char* in, unsigned char* out, int width, int
    height, int channels, int pixelNormValue)
{
    // get pixel coordinates for the current thread
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // check if the thread is within the valid image range
    if (row < height && col < width)
    {
      // define the starting position of the convolution mask
      int maskStartRow = row - (KERNEL_SIZE / 2);
      int maskStartCol = col - (KERNEL_SIZE / 2);

      // iterate over color channels
      for (int c = 0; c < channels; c++)
      {
        float pixelVal = 0;

        // iterate over the elements of the convolution mask
        for (int i = 0; i < KERNEL_SIZE; ++i) {
          for (int j = 0; j < KERNEL_SIZE; ++j)
          {
            // calculate the current position in the input image
            int curRow = min(height - 1, max(maskStartRow + i, 0));
            int curCol = min(width - 1, max(maskStartCol + j, 0));

            // perform convolution by multiplying the pixel value with the corresponding
                kernel value
            pixelVal += in[(curRow * width + curCol) * channels + c] * KERNEL_MASK[i *
                KERNEL_SIZE + j];
          }
```

```
30        }
31
32        // normalize the pixel value, then store it in the output image
33        pixelVal = min((float)CHANNELS_MAX_VALUE, max(pixelVal + pixelNormValue, 0.0f));
34        out[(row * width + col) * channels + c] = (unsigned char)pixelVal;
35      }
36    }
37  }
```

5 Risultati

Questa sezione illustra e commenta i risultati ottenuti, concentrandosi in particolare sulle prestazioni e sui potenziali miglioramenti ottenuti passando da un'esecuzione sequenziale a una parallela. Vengono anche mostrate e commentate alcune metriche di misurazione e confronto delle prestazioni tra un programma parallelo e uno sequenziale, tutte basate sul tempo di esecuzione del programma.

5.1 Soluzione

Questa breve sottosezione presenta con un esempio grafico i risultati del processo implementato. Per i test e le valutazioni riguardante le performance e la correttezza esecutiva del kernel image processing, sono state impiegate immagini di diverse dimensioni, partendo da immagini molto piccole ($100x100$ pixel) fino a immagini più grandi ($10000x6000$ pixel). La Figura 8[11] mostra un esempio di risultato del processo di convoluzione usando in particolare il filtro *sharpen*, che permette di simulare una maggiore nitidezza dell'immagine.

Figure 8: Filtro sharpen: immagine originale a sinistra e immagine processata a destra.

5.2 Performance

Questa sottosezione presenta e analizza i risultati in termini di performance della soluzione sviluppata. Vengono valutate le performance con diverse configurazioni del programma, quali ad esempio la dimensione dei blocchi CUDA. Tutti i test sono stati effettuati su una GPU NVIDIA MX250. Le Figure 9, 10 e 11 di seguito illustrano il tempo di esecuzione in relazione alla variazione della dimensione dell'immagine usata nella convoluzione, considerando le diverse modalità di esecuzione del programma. Nella Figura 9 si osserva che per immagini di dimensioni ridotte (fino a $512x512$ pixel), la quantità limitata di dati non permette all'esecuzione parallela di offrire un vantaggio significativo, e in questi casi l'esecuzione sequenziale tende a fornire prestazioni migliori. Tuttavia, per immagini di dimensioni maggiori, il tempo di esecuzione della versione sequenziale aumenta notevolmente, manifestando una crescita esponenziale.

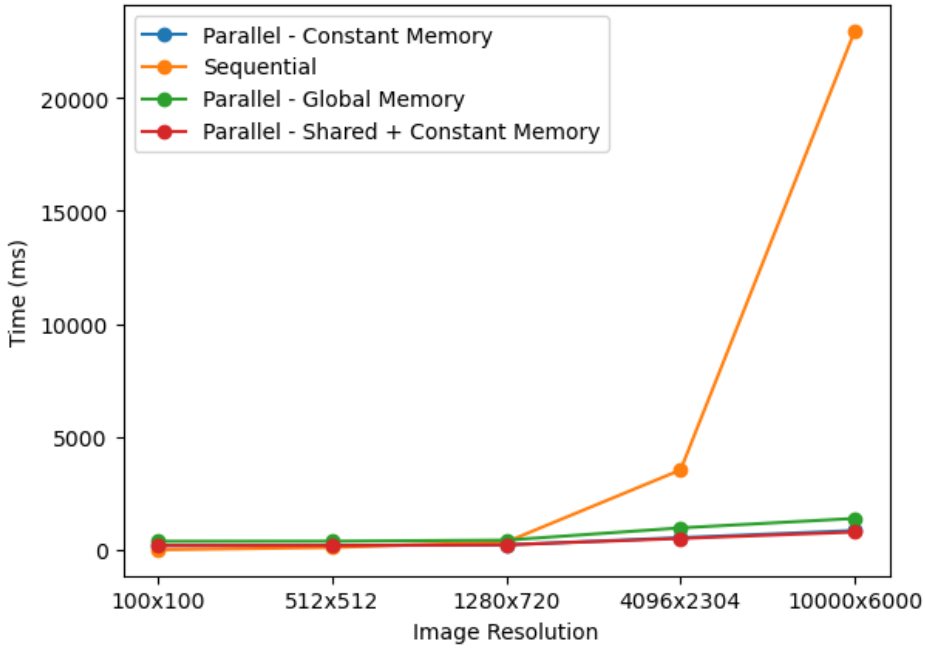

Figure 9: Tempo di esecuzione per dimensione immagine (distinzione per modalità di esecuzione).

La Figura 10 mostra più nel dettaglio i tempi delle versioni parallele del programma, con le diverse modalità di esecuzione.

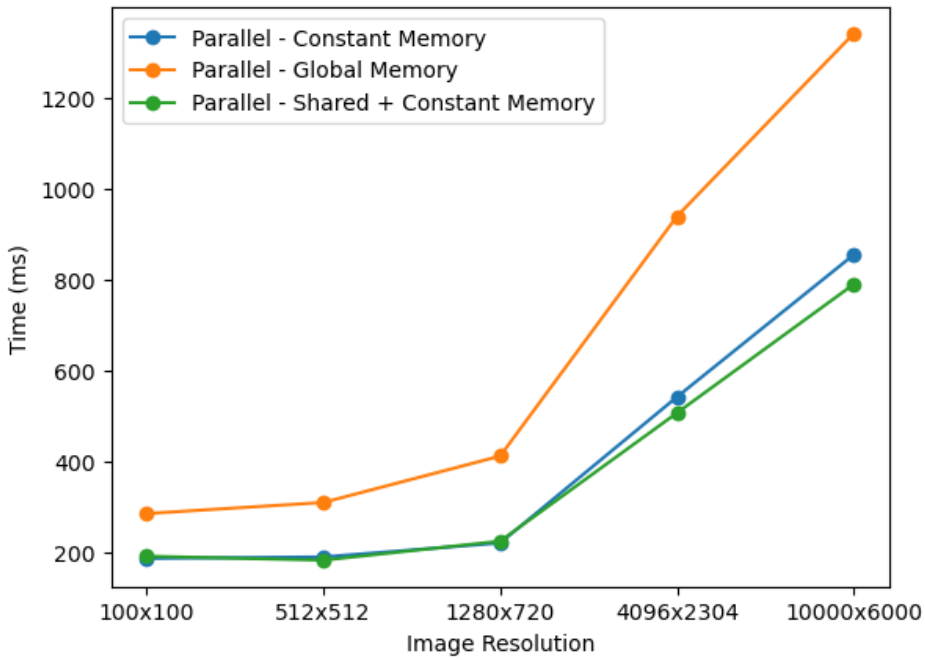

Figure 10: Tempo di esecuzione per modalità di esecuzione parallela, con allocamento e trasferimento di dati.

La Figura 11 rappresenta i tempi di esecuzione delle diverse versioni parallele del programma. A differenza della Figura 10 precedentemente discussa, questa figura considera esclusivamente i tempi della funzione kernel CUDA per la convoluzione, trascurando gli intervalli di tempo dedicati all'allocazione, copia e trasferimento dei dati tra l'host e il device. Questo consente di evidenziare come la versione *Parallel - Global Memory*, che memorizza la matrice del filtro e l'immagine stessa esclusivamente sulla

memoria globale (che ha una maggiore capacità ma è solitamente più lenta), fornisca prestazioni generalmente inferiori rispetto alle varianti che sfruttano la *constant* e *shared memory*. In particolare, la versione *Parallel - Shared + Constant Memory* raggiunge tempi di esecuzione quasi dimezzati.

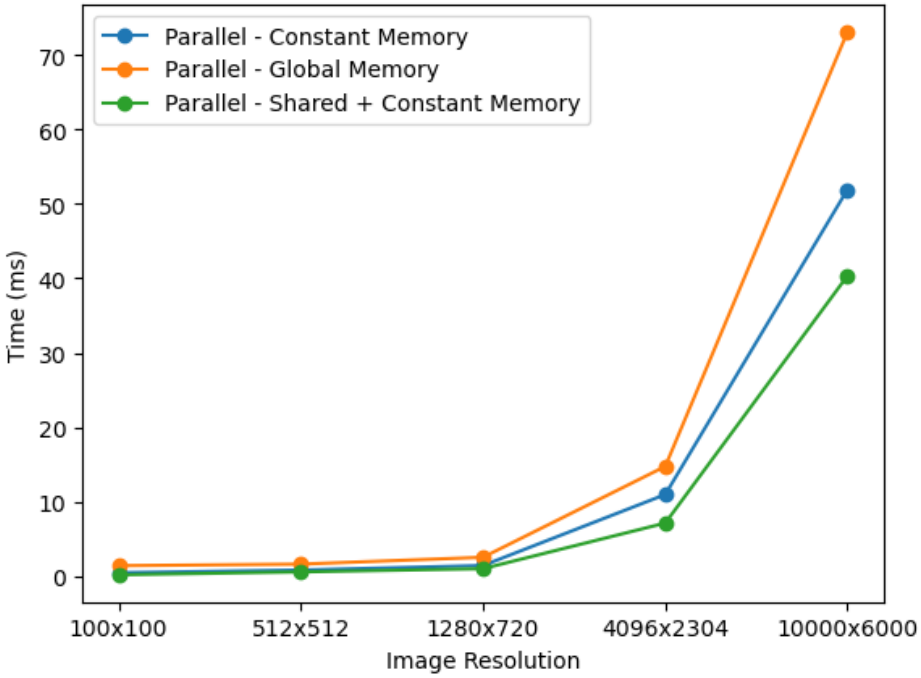

Figure 11: Tempo di esecuzione per la sola convoluzione, per modalità di esecuzione parallela.

La Figura 12 rappresenta il valore dello speedup medio per modalità di esecuzione e immagini impiegate. Lo speedup è un parametro che misura le prestazioni relative nell'elaborazione di un problema e, più tecnicamente, rappresenta il miglioramento della velocità di esecuzione di un compito su un sistema che utilizza una quantità variabile di risorse ad ogni esecuzione [12]. Emerge che la variante che sfrutta la *shared memory* fornisce generalmente le prestazioni migliori. La modalità che invece si basa esclusivamente sulla global memory risulta essere la meno efficiente.

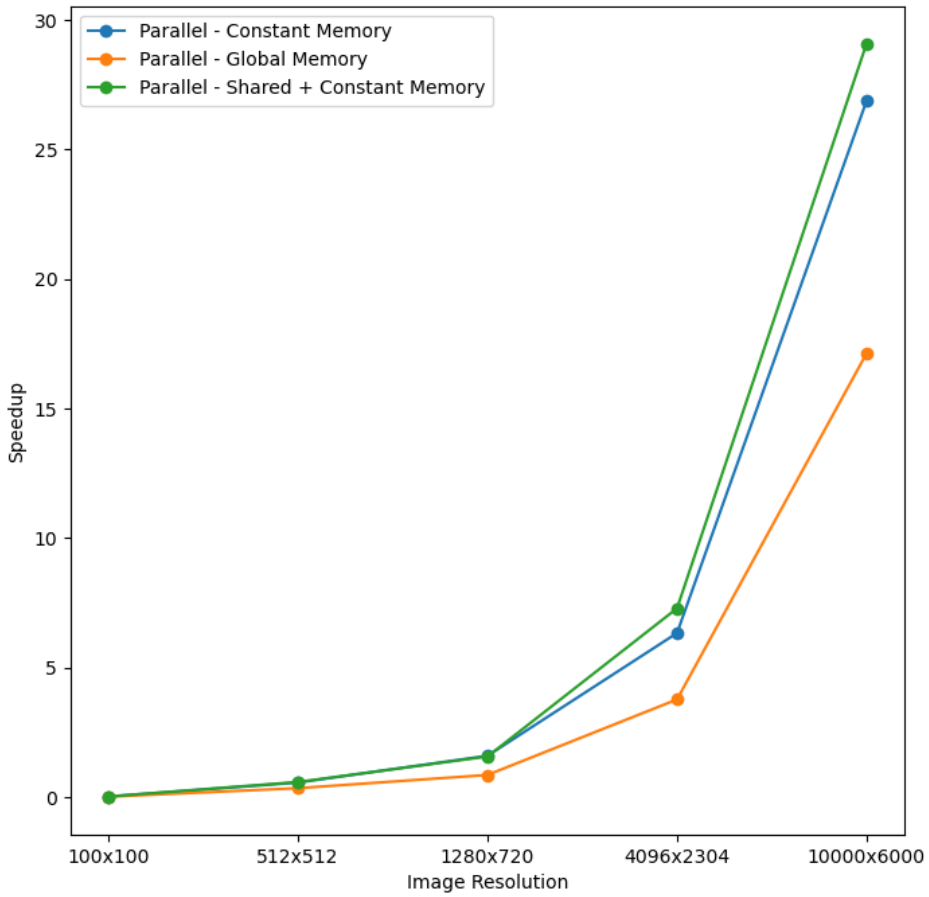

Figure 12: Speedup medio per dimensione immagine (distinzione per modalità di esecuzione).

La Tabella 1 mostra nel dettaglio i valori in millisecondi dei tempi di esecuzione per le diverse versioni del programma e diverse immagini di input. La Tabella 2 mostra invece i valori dettagliati dello speedup medio per diversa immagine e tipologia di esecuzione. Come precendentemente accennato, poiché per immagini di piccole dimensioni il tempo di esecuzione sequenziale è minore di quello parallelo, si osservano valori negativi di speedup per tutte le immagini con dimensioni inferiori a $1280x720$ pixel.

Resolution	Sequential	Global	Constant	Shared + Const
100x100	3.859	285.458	186.760	191.501
512x512	106.328	309.910	189.981	182.898
1280x720	351.694	411.974	220.706	224.676
4096x2304	3538.819	939.415	541.970	506.833
10000x6000	22962.614	1341.567	854.934	789.941

Table 1: Tempo di esecuzione medio per modalità di esecuzione e dimensione immagine.

Resolution	Global	Constant	Shared + Const
100x100	0.014	0.021	0.020
512x512	0.343	0.560	0.581
1280x720	0.854	1.593	1.565
4096x2304	3.767	6.530	6.982
10000x6000	17.116	26.859	29.069

Table 2: Speedup medio per modalità di esecuzione e dimensione immagine.

Le altre tabelle che seguono, mostrano i tempi e speedup relativi a esecuzioni con diverse dimensioni di blocchi CUDA. Le Tabelle 3 e 4 riguardano blocchi di dimensione $32x32$, le Tabelle 5 e 6 si riferiscono a blocchi di dimensione $16x16$, mentre le Tabelle 7 e 8 riguardano blocchi di dimensione $8x8$.

Resolution	Global	Constant	Shared + Const
100x100	280.173	151.902	166.611
512x512	314.764	153.312	144.080
1280x720	356.346	162.285	187.109
4096x2304	897.122	537.617	463.731
10000x6000	1294.665	818.207	734.694

Table 3: Tempo di esecuzione medio con blocco $32x32$.

Resolution	Global	Constant	Shared + Const
100x100	0.014	0.025	0.023
512x512	0.338	0.694	0.738
1280x720	0.987	2.167	1.880
4096x2304	3.945	6.582	7.631
10000x6000	17.736	28.065	31.255

Table 4: Speedup medio con blocco $32x32$.

Resolution	Global	Constant	Shared + Const
100x100	290.259	178.545	194.807
512x512	315.438	200.058	189.788
1280x720	403.931	235.996	253.695
4096x2304	912.676	538.398	518.728
10000x6000	1342.289	831.517	801.058

Table 5: Tempo di esecuzione medio con blocco $16x16$.

Resolution	Global	Constant	Shared + Const
100x100	0.013	0.022	0.020
512x512	0.337	0.531	0.560
1280x720	0.871	1.490	1.386
4096x2304	3.877	6.573	6.822
10000x6000	17.107	27.615	28.665

Table 6: Speedup medio con blocco $16x16$.

Resolution	Global	Constant	Shared + Const
100x100	283.543	229.833	213.086
512x512	296.200	216.574	216.507
1280x720	477.656	263.836	233.225
4096x2304	1015.134	556.146	521.980
10000x6000	1399.473	907.235	834.070

Table 7: Tempo di esecuzione medio con blocco $8x8$.

Resolution	Global	Constant	Shared + Const
100x100	0.014	0.017	0.018
512x512	0.359	0.491	0.491
1280x720	0.736	1.333	1.508
4096x2304	3.486	6.363	6.780
10000x6000	16.408	25.311	27.531

Table 8: Speedup medio con blocco $8x8$.

6 Conclusioni

La soluzione implementata consente l'applicazione di tecniche di elaborazione delle immagini mediante l'utilizzo di filtri vari. Il progetto è stato sviluppato con CUDA, sfruttando la potente capacità computazionale delle GPU. Le valutazioni sulle performance evidenziano che le diverse versioni parallele offrono notevoli vantaggi rispetto a quella sequenziale per immagini di dimensioni significative, raggiungendo per le immagini più grandi ($10000x6000$ pixel) uno speedup medio di circa 25. Questi risultati costituiscono un punto di partenza solido per potenziali sviluppi futuri e ottimizzazioni. Ad esempio, si potrebbe esplorare la possibilità di ottimizzare ulteriormente l'applicativo, dando anche la possibilità di applicare un maggior numero di filtri o filtri di dimensioni differenti.

References

[1] *Kernel (image processing) - Wikipedia*. URL: https://en.wikipedia.org/wiki/Kernel_(image_processing). (accessed: 12.02.2024).

[2] *CUDA Toolkit*. URL: https://developer.nvidia.com/cuda-toolkit. (accessed: 11.02.2024).

[3] *Digital image processing - Wikipedia*. URL: https://en.wikipedia.org/wiki/Digital_image_processing. (accessed: 17.02.2024).

[4] *Convolution - Wikipedia*. URL: https://en.wikipedia.org/wiki/Convolution. (accessed: 17.02.2024).

[5] *Image Convolution From Scratch*. URL: https://medium.com/analytics-vidhya/image-convolution-from-scratch-d99bf639c32a. (accessed: 18.02.2024).

[6] M. Bertini. *Introduction to Data parallelism: GPU computing with CUDA*. Dipartimento di Ingegneria dell'Informazione (DINFO). University of Florence, IT.

[7] *Graphics Processing Units (GPUs)*. URL: https://gistbok.ucgis.org/bok-topics/graphics-processing-units-gpus. (accessed: 17.02.2024).

[8] G. Piqué. *Pikerozzo / cuda-kernel-img-proc*. URL: https://github.com/Pikerozzo/cuda-kernel-img-proc. (accessed: 13.02.2024).

[9] S. Barrett. *nothings / stb*. URL: https://github.com/nothings/stb. (accessed: 13.02.2024).

[10] *Image embossing - Wikipedia*. URL: https://en.wikipedia.org/wiki/Image_embossing#Example. (accessed: 15.02.2024).

[11] *Lenna - Wikipedia*. URL: https://en.wikipedia.org/wiki/Lenna. (accessed: 13.02.2024).

[12] *Speedup - Wikipedia*. URL: https://en.wikipedia.org/wiki/Speedup. (accessed: 15.02.2024).