

UNIVERSITÀ DEGLI STUDI DI PERUGIA  
DIPARTIMENTO DI MATEMATICA E  
INFORMATICA

CALCOLO DISTRIBUITO E SISTEMI AD ALTE PRESTAZIONI

---

# GPU Programming

---

*Autore:*

Francesco GRADI

*Professore del corso:*

Oswaldo GERVASI

10 settembre 2018



# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
<b>2</b>	<b>Installazioni preliminari</b>	<b>3</b>
2.1	OpenCL . . . . .	3
2.1.1	I modelli . . . . .	4
<b>3</b>	<b>Configurazioni preliminari</b>	<b>5</b>
<b>4</b>	<b>I kernel</b>	<b>7</b>
4.1	Moltiplicazione di Matrici . . . . .	9
4.2	Ribaltamento di un'immagine . . . . .	11
4.3	Convoluzione di un'immagine . . . . .	13
<b>5</b>	<b>L'interfaccia grafica</b>	<b>16</b>
<b>6</b>	<b>Dati utili all'identificazione dell'autore</b>	<b>19</b>

# 1 Introduzione

L'obiettivo di questa esercitazione è stato impiegare le potenzialità della GPU per eseguire programmi specifici che altrimenti sfrutterebbero la CPU.

Per GPU (*Graphics Processing Unit*) si intende l'unità di elaborazione grafica solitamente utilizzata per il rendering di immagini, ma spesso anche utile per effettuare calcoli generici (*GPU computing*). Quelle più recenti sono composte da un elevato numero di core, il che le rende più funzionali di una CPU (*Central Processing Unit*) per l'esecuzione di codice parallelizzabile. Inoltre esse vengono anche spesso impiegate per computazioni estremamente esigenti per le quali le normali CPU non sarebbero sufficienti.

In questa esercitazione ho svolto del GPUPU (general-purpose computing on GPU) usufruendo dell'aiuto di *OpenCL*.

## 2 Installazioni preliminari

Il mio computer disponeva di un processore Intel e di una scheda grafica AMD. Per prima cosa è stato fondamentale aggiornare i driver Intel e AMD, ma soprattutto scaricare i driver specifici di AMD per *OpenCL*<sup>1</sup>. Dopodiché è stato finalmente possibile lavorare con *OpenCL*.

### 2.1 OpenCL

OpenCL è un framework Open-Source utilizzato per scrivere programmi che devono essere eseguiti su piattaforme eterogenee (CPU, GPU, DPS, ecc).

Esso utilizza un linguaggio simile a *C*, distinguendo però le funzioni normali dalle funzioni eseguite su un device OpenCL. Queste ultime vengono identificate con il nome di *kernel*.

Ogni device è costituito da unità di calcolo molteplici, ed ognuna di esse sfrutta elementi di elaborazione molteplici, detti *processing element* (PE). Di conseguenza una singola esecuzione del *kernel* viene eseguita su tutti i PE in parallelo.

Le specifiche di OpenCL sono quindi divise in 4 parti, dette modelli.

---

<sup>1</sup>Quest'ultima operazione è risultata più complicata del previsto poiché AMD sembra aver terminato il supporto driver *OpenCL* per la scheda grafica di cui disponevo, colpa probabile la sua obsolescenza. Tuttavia, grazie a dei siti esterni di appoggio, sono riuscito ad ottenere il file necessario, sebbene dedicando non poco tempo alla sua ricerca.

### 2.1.1 I modelli

- **Platform Model**

Questo modello stabilisce l'esistenza di un unico processore che coordina l'esecuzione (host), mentre possono esserci più processori dedicati all'esecuzione del *kernel* (devices). Definisce quindi un modello hardware astratto utilizzato spesso dai programmatori per scrivere le funzioni kernel.

- **Execution Model**

Questo modello stabilisce invece come è configurato l'ambiente di OpenCL nell'host e in che modo i kernel vengono eseguiti nei devices. Definisce inoltre un *Concurrency model* utilizzato per l'esecuzione del kernel nei devices.

- **Memory Model**

Questo modello definisce invece la gerarchia di memoria astratta usata dai kernel, indipendente dalla effettiva architettura di memoria sottostante.

- **Programming Model**

Quest'ultimo modello definisce infine come il *Concurrency model* viene mappato nell'hardware.

### 3 Configurazioni preliminari

Una volta installati i driver, è stato necessario configurare opportunamente Visual Studio affinché rilevasse e utilizzasse i file necessari alla fruizione di OpenCL.

Il primo passo è stato aggiungere il percorso della cartella con i file necessari per OpenCL sotto la voce “Directory di inclusione aggiuntive” (Figura 1), nonché aggiungere sotto la voce “Directory librerie aggiuntive” il percorso della cartella con i file relativi all’SDK di AMD per OpenCL (Figura 2).

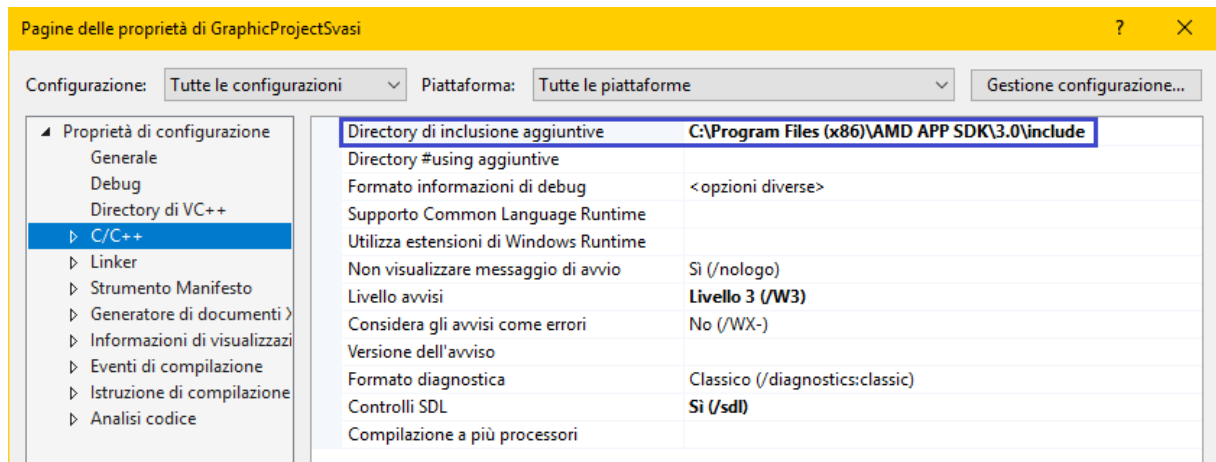


Figura 1: Inclusione della cartella con i file di OpenCL.

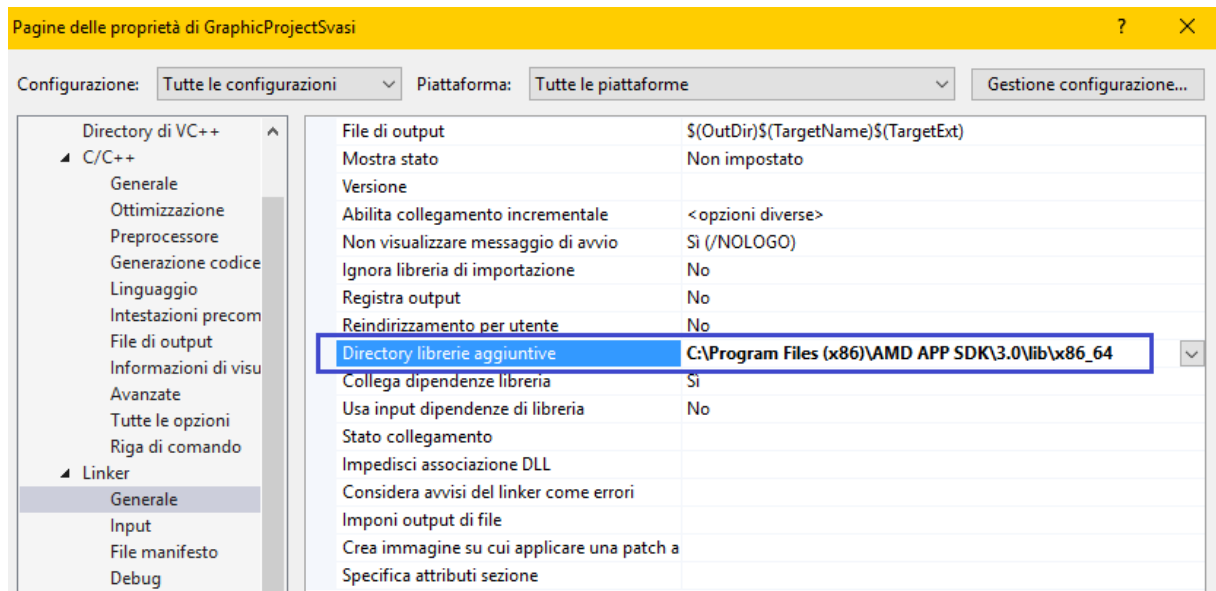


Figura 2: Inclusione della cartella con i file SDK di AMD.

Successivamente è bastato specificare quali dipendenze aggiuntive di OpenCL sarei andato ad utilizzare (Figura 3) per terminare la configurazione iniziale complessiva.

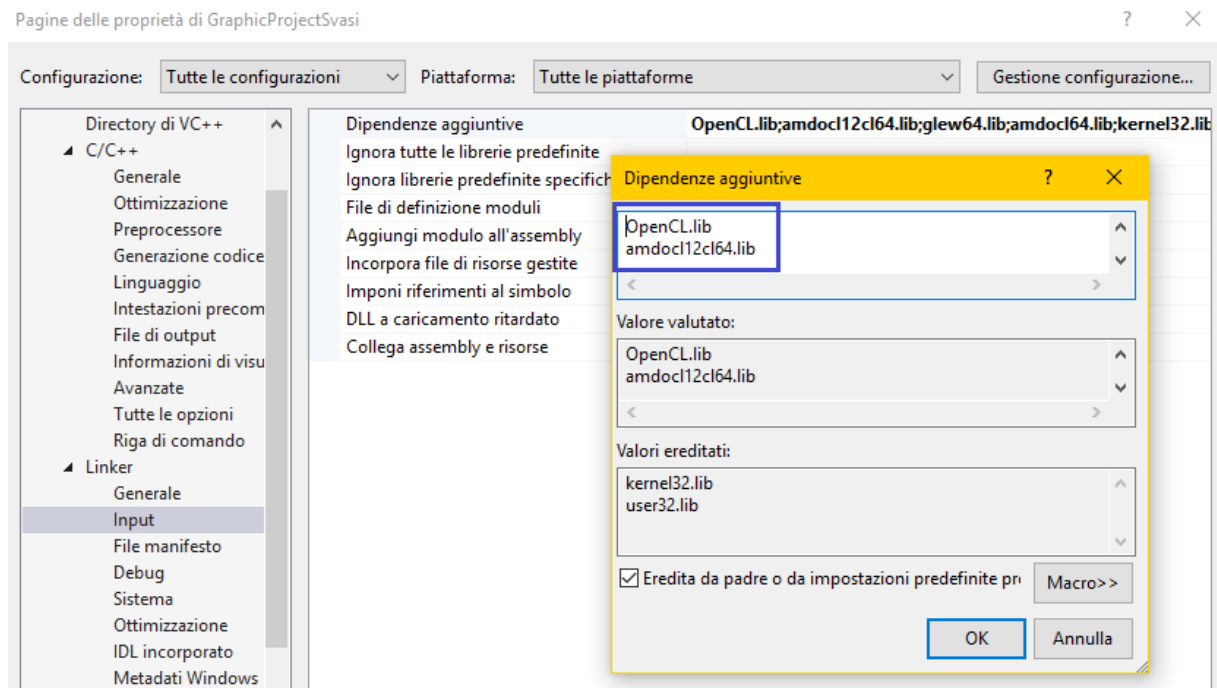


Figura 3: Aggiunta delle dipendenze necessarie al progetto.

## 4 I kernel

Come suggerito prima, i *kernel* rappresentano i file contenenti il codice utile allo svolgimento delle operazioni che questa esercitazione richiede. Di norma vengono chiamati dal *main* e fanno uso di funzioni specifiche:

- **clGetPlatformIDs()**: questa funzione serve per identificare le piattaforme disponibili del sistema su cui si esegue il kernel. Nel



mio caso è stato ovviamente necessario far uso della GPU.

- **clGetDeviceIDs()**: utile a identificare l'insieme dei dispositivi disponibili di una data piattaforma.
- **clCreateContext()**: crea un nuovo contesto e lo associa al dispositivo opportuno. Un contesto consente di fornire comandi e dati al suddetto dispositivo, fungendo inoltre da contenitore astratto nell'host con lo scopo di gestire gli oggetti di memoria disponibili.
- **clCreateCommandQueue()**: definisce una coda di comandi da associare al dispositivo.
- **clCreateBuffer()**: crea i buffer necessari per contenere gli oggetti di memoria.
- **clEnqueueWriteBuffer()**: scrive i dati all'interno dei buffer, associando il comando alla coda.
- **clCreateProgramWithSource()**: crea il programma a partire dal file kernel.
- **clBuildProgram()**: compila il programma.
- **clCreateKernel()**: crea il kernel a partire dal programma compilato, passandogli la funzione da eseguire (che si trova all'interno del file kernel).
- **clSetKernelArg()**: passa alla funzione sopra citata i buffer necessari.

- **clEnqueueNDRangeKernel()**: esegue la funzione del kernel, prendendo come parametri la coda dei comandi e il numero di *work-items* per ogni dimensione. Ogni *work-item* esegue la funzione del kernel una sola volta, ma poiché ognuno di essi esegue in parallelo, il tempo computazionale è drasticamente ridotto e l'insieme dei loro output costituisce quindi il risultato.
- **clEnqueueReadBuffer()**: legge il buffer di output.
- **clCreateImage2D()**: definisce lo spazio necessario per l'allocazione di un'immagine.
- **clCreateSampler()**: crea un *sampler*, ovvero un oggetto che descrive le modalità di accesso ad un'immagine da parte di un dispositivo.

## 4.1 Moltiplicazione di Matrici

Questo kernel si è quindi occupato di prendere in input la dimensione di due matrici e, previo popolamento randomizzato, di effettuarne la moltiplicazione.

Come ogni kernel ha sfruttato le funzioni sopra elencate, allocando inoltre lo spazio necessario in memoria per contenere le matrici, inclusa quella risultante. Tramite due cicli ho fatto sì che il programma scorresse le due matrici e le scrivesse in tre file di appoggio, per potervi accedere successivamente.

La Figura 4 mostra il codice che ho scritto del kernel in questione.

```

.c 7  moltiplicazioneMatrici.cl  X  vector_add_kernel.cl
__kernel void moltiplicazioneMatrici(__global int *A, __global int *B, int righeA, int colonneB, __global int *C) {

    // Get the index of the current element to be processed
    int i = get_global_id(0);
    int j = get_global_id(1);
    int val = 0;

    for (int k = 0; k < righeA; k++){
        val = val + A[k + j * righeA] * B[k * colonneB + i];
    }

    C[i + righeA * j] = val;
}

```

Figura 4: Il kernel per la moltiplicazione matriciale.

Ho eseguito un test con due matrici rispettivamente 2x3 e 3x4 (Figura 5). Il kernel ha quindi calcolato correttamente la matrice risultante (Figura 6).

matAdata.txt			matBdata.txt - Blocco note				
File	Modifica	F	File	Modifica	Formato	Visualizza	?
3	2	10	6	7	1	7	
6	10	6	10	6	2	1	
			3	2	9	9	

Figura 5: Le matrici A[2x3] e B[3x4].

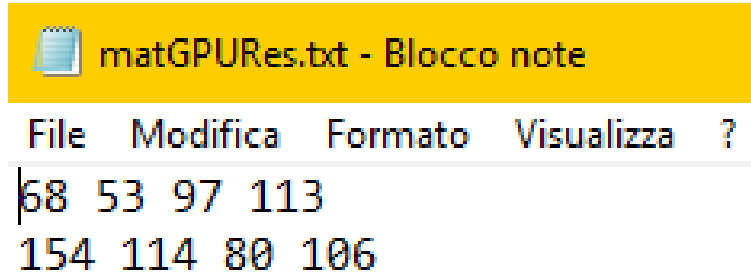


Figura 6: La matrice  $C[2 \times 4]$  risultante.

## 4.2 Ribaltamento di un'immagine

Il funzionamento di questo kernel è simile a quello precedente, ma stavolta invece di allocare lo spazio necessario per tre matrici, ho dovuto far sì che il programma allocasse lo spazio necessario per un'immagine, passandone al buffer altezza e larghezza.

A fini precauzionali, ho deciso di lavorare con immagini in estensione *.bmp*, poiché OpenCL sembra supportarli meglio.

La Figura 7 mostra il codice che ho scritto del kernel in questione.

```

specchiaImmagine.cl ➤ ✕ input.bmp
__kernel
void specchia_immagine(__global float* dest_image, __global float* src_image, int W, int H)
{
    const int ix = get_global_id(0);
    const int iy = get_global_id(1);

    int specchiaX = (int)(W - ix);

    if ((specchiaX >= 0) && (specchiaX < W) && (iy >= 0) && (iy < H))
    {
        dest_image[iy*W + ix] = src_image[iy*W + specchiaX];
    }
}

```

Figura 7: Il kernel per il ribaltamento di un'immagine.

Testando quindi il programma, ho potuto constatare come dall'immagine di partenza (Figura 8) il kernel ha creato in output l'immagine specchiata (Figura 9).



Figura 8: L'immagine di partenza.



Figura 9: L'immagine specchiata.

### 4.3 Convoluzione di un'immagine

La convoluzione è una tecnica che modifica il valore di ogni pixel di un'immagine sfruttando le informazioni che ottiene dai pixel adiacenti. Un kernel di convoluzione (detto anche filtro) stabilisce come ogni pixel verrà influenzato dai suoi vicini.

Sfruttando sempre le funzioni dei kernel sopra descritte, in quest'ultimo programma ho utilizzato la stessa immagine di partenza (resa in scala di grigi per una migliore esaltazione del risultato, Figura 10) e una matrice di convoluzione data per ottenere il risultato in Figura 11.

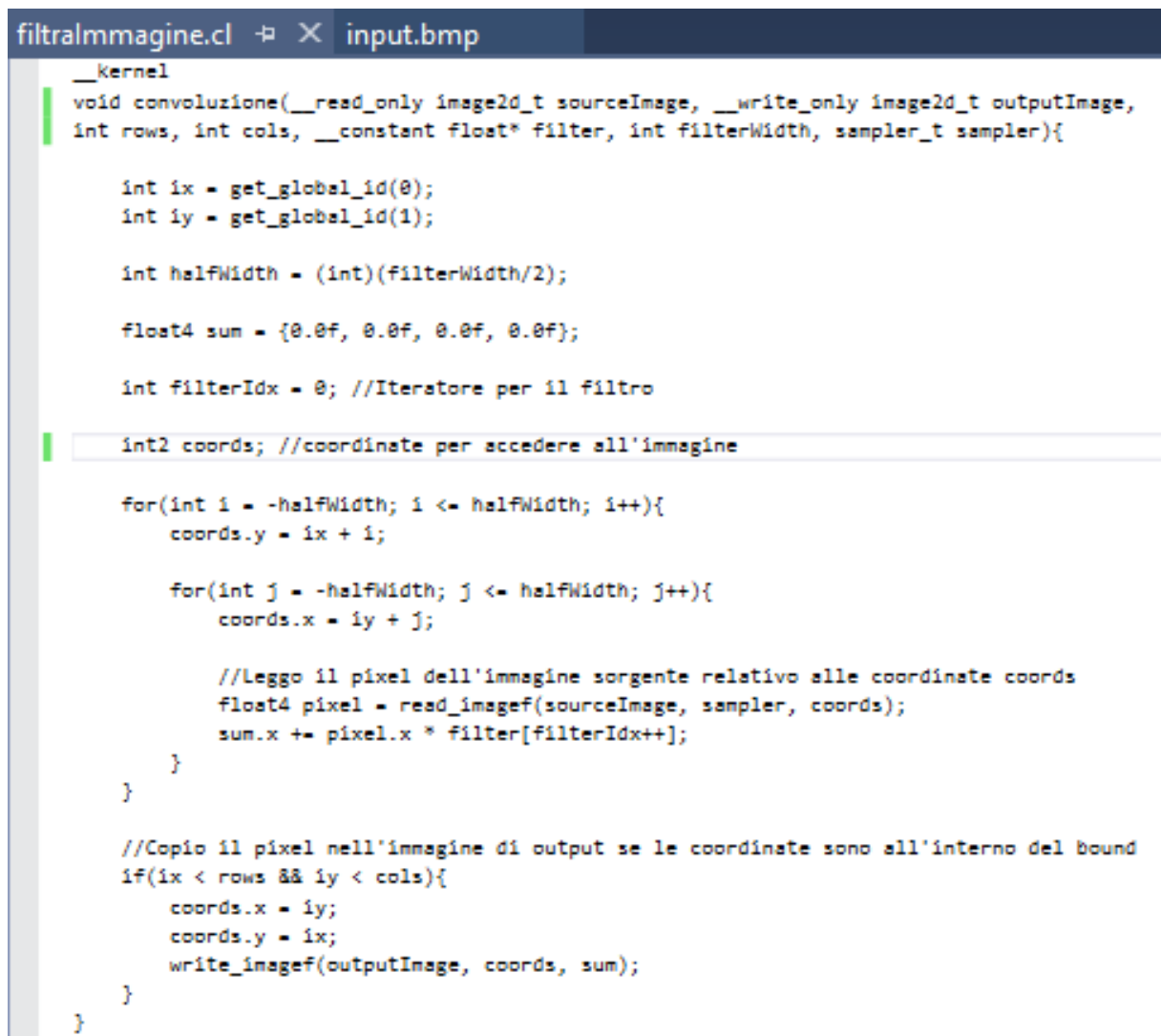


Figura 10: L'immagine di partenza in scala di grigi.



Figura 11: L'immagine con il filtro applicato.

La Figura 12 mostra il codice kernel che mi ha permesso l'attuazione della suddetta modifica all'immagine.

The image is a screenshot of a code editor window. The title bar at the top shows two tabs: 'filtralimmagine.cl' and 'input.bmp'. The code is written in C++ and is a CUDA kernel. It starts with a function signature 'void convoluzione' with parameters for source and output images, dimensions, a filter, filter width, and a sampler. Inside the function, it gets global IDs for x and y coordinates, calculates the half-width of the filter, initializes a sum vector, and sets a filter index. A nested loop iterates over the filter area, reading pixels from the source image and multiplying them by the filter values. Finally, it checks if the coordinates are within the output image bounds and writes the result. The code is formatted with indentation and comments in Italian.

```
__kernel
void convoluzione(__read_only image2d_t sourceImage, __write_only image2d_t outputImage,
int rows, int cols, __constant float* filter, int filterWidth, sampler_t sampler){

    int ix = get_global_id(0);
    int iy = get_global_id(1);

    int halfWidth = (int)(filterWidth/2);

    float4 sum = {0.0f, 0.0f, 0.0f, 0.0f};

    int filterIdx = 0; //Iteratore per il filtro

    int2 coords; //coordinate per accedere all'immagine

    for(int i = -halfWidth; i <= halfWidth; i++){
        coords.y = iy + i;

        for(int j = -halfWidth; j <= halfWidth; j++){
            coords.x = ix + j;

            //Leggo il pixel dell'immagine sorgente relativo alle coordinate coords
            float4 pixel = read_imagef(sourceImage, sampler, coords);
            sum.x += pixel.x * filter[filterIdx++];
        }
    }

    //Copio il pixel nell'immagine di output se le coordinate sono all'interno del bound
    if(ix < rows && iy < cols){
        coords.x = iy;
        coords.y = ix;
        write_imagef(outputImage, coords, sum);
    }
}
```

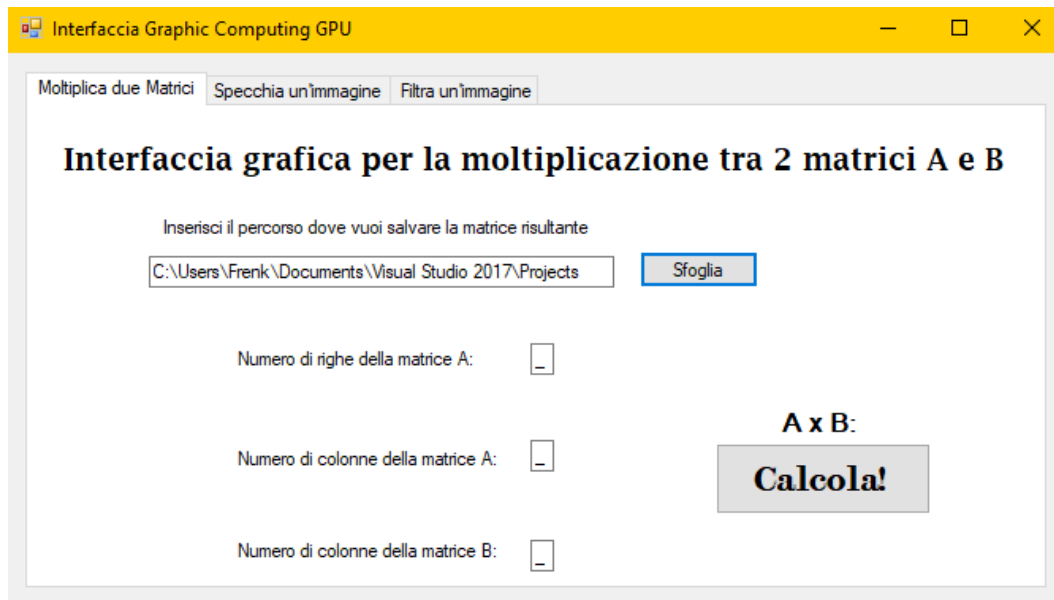
Figura 12: Il kernel per la convoluzione di un'immagine.

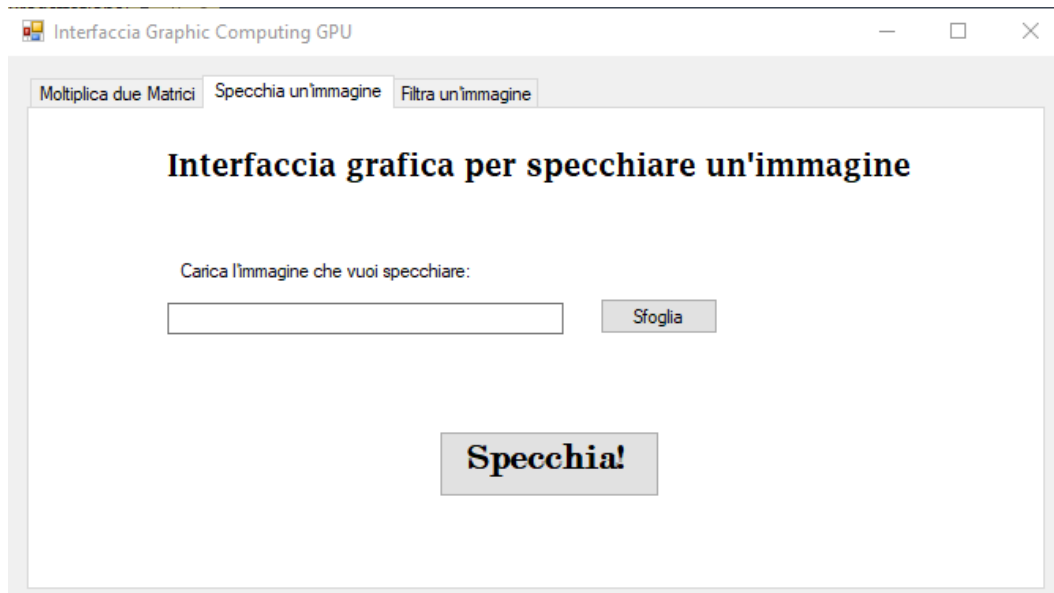


## 5 L'interfaccia grafica

Per la realizzazione del *punto bonus* dell'intero progetto, ho quindi costruito - tramite Windows Form - un'interfaccia grafica che permettesse l'avvio immediato dei tre kernel.

In essa è possibile scegliere il percorso dove salvare le matrici o le immagini e, nel caso della moltiplicazione matriciale, dare direttamente in input righe e colonne delle due matrici. Successivamente l'interfaccia si occuperà di avviare il programma corrispondente e di creare i rispettivi file di output.





Per poter quindi effettuare le azioni sopra descritte, ho inserito il codice in Figura 13 all'interno della funzione di click di ogni bottone (opportunamente modificato per ogni kernel), affinché il programma salvasse il percorso stabilito dall'utente e le matrici e/o le immagini in file specifici, per il corretto funzionamento dello stesso.

```
private void button2_Click(object sender, EventArgs e)
{
    System.IO.File.WriteAllText("../.../Files/path.txt", textBox1.Text);
    System.IO.File.WriteAllText("../.../Files/righeA.txt", maskedTextBox1.Text);
    System.IO.File.WriteAllText("../.../Files/colonneA.txt", maskedTextBox2.Text);
    System.IO.File.WriteAllText("../.../Files/colonneB.txt", maskedTextBox3.Text);
    //altre textbox
    Process.Start(@"C:/Users/Frenk/Documents/Visual Studio 2017/Projects/GraphicProjectSv
```

Figura 13: Il codice relativo al bottone per la moltiplicazione di matrici.

## 6 Dati utili all'identificazione dell'autore



**Francesco Gradi**

*matricola:* 3 0 9 6 4 5