Real Time Video Filtering with CUDA

Luis Alfredo Benjume Tovar A01066516

Abstract – In the following document explains the implementation of real time vide filtering using CUDA and OpenCV using the visual Studio Environment.

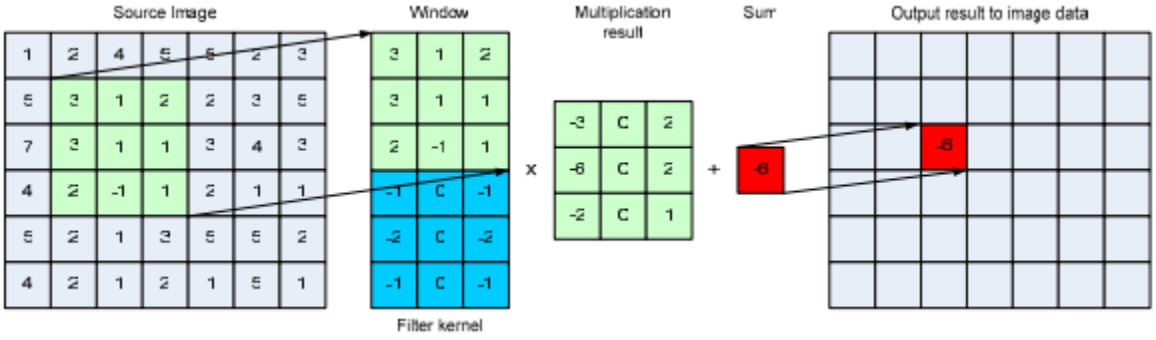
1. Introduction

Graphics cards are widely used to accelerate gaming and 3D graphics applications. The GPU (Graphics Processing Unit) of a graphics card is built for compute-intensive and highly parallel computations. Image processing is a natural fit when it comes to parallel data processing. With CUDA the power of the GPU is being leveraged to accelerate more general purpose and high-performance applications. Image processing is a well-known and established research field. It is a form of signals processing in which the input is an image, and the output can be an image or anything else that undergoes some meaningful processing. Commonly processing happens on the entire image, and the same steps are applied to every pixel of the image. This means a lot of repetition of the same work. Thus, we can say that image processing is a perfect fit for parallel computing.

In this case we will be using CUDA in conjunction with the OpenCV library to perform real time convolutions and other operations to a live video feed with source on the user's camera.

2. Solution

To solve this problem, we should first start by understanding what a convolution is. Convolution filtering is a technique that can be used for a wide array of image processing tasks, some of which may include smoothing and edge detection. Mathematically, a convolution measures the amount of overlap between two functions. In the context of image processing a convolution filter is just the scalar product of the filter weights with the input pixels within a window surrounding each of the output pixels. In the next image we will se an example of a basic convolution method:



Convolutional kernel array is typically much smaller than the input array and iterates through the input array and at each iteration it computes a weighted sum of the current input element as well as its neighboring input elements and the result is placed in the output array. A two-dimensional convolution filter requires n*m multiplications for each output pixel, where n and m are the width and height of the filter kernel. In this case we will be working with a two-dimension convolution.

Starting off with the data previously provided we can start implementing a CUDA implementation of a convolution. We can start off with a simple implementation:

1. Read the input elements in its neighbourhood.
2. Read the convolutional kernel.
3. Perform the convolution computation.
4. Write the result of the computation to its index in the output array.

```
__global__ void convolution_global_memory(float *N, float *M, float *P, int Width){

        int i = blockIdx.x*blockDim.x+threadIdx.x;

        float Pvalue = 0;

        int n_start_point = i-(MASK_WIDTH/2);

        for(int j =0; j<MASK_WIDTH;j++){
                if(n_start_point+j >=0 && n_start_point+j < Width){
                        Pvalue+= N[n_start_point+j]*M[j];
                }
        }

        P[i]=Pvalue;
}
```

Each thread computes it's P[i] value by iterating over its neighbourhood of input values starting at i-(MASK_WIDTH/2) and ending at i+(MASK_WIDTH/2). Both N and M are in global memory so at each iteration the operation N[n_start_point+j]*M[j] involves two calls to global memory. This implementation is okay, but we can increase efficiency by putting the convolutional kernel in constant memory. The CUDA runtime will initially read the convolutional kernel from global memory as before however now it will cache it since it knows it will never be modified.

```
#define MASK_WIDTH 5
__constant__ float M[MASK_WIDTH];
```

Applying this to our code we came out with:

```
__constant__ float KernelStore[256];

__global__ void convolve(unsigned char* source, int width, int height, int paddingX, int paddingY, size_t kOffset, int kWidth, int kHeight, unsigned char* destination)
{
    int x = (blockIdx.x * blockDim.x) + threadIdx.x;
    int y = (blockIdx.y * blockDim.y) + threadIdx.y;

    float sum = 0.0;
    int   pWidth = kWidth / 2;
    int   pHeight = kHeight / 2;

    if (x >= pWidth + paddingX &&
        y >= pHeight + paddingY &&
        x < (blockDim.x * gridDim.x) - pWidth - paddingX &&
        y < (blockDim.y * gridDim.y) - pHeight - paddingY)
    {
        for (int j = -pHeight; j <= pHeight; j++)
        {
            for (int i = -pWidth; i <= pWidth; i++)
            {
                int ki = (i + pWidth);
                int kj = (j + pHeight);
                float w = KernelStore[(kj * kWidth) + ki + kOffset];


                sum += w * float(source[((y + j) * width) + (x + i)]);
            }
        }
    }

    destination[(y * width) + x] = (unsigned char)sum;
}
```

As we can se the idea is the same, but we have more parameters in the CUDA kernel, since we define all the image filter kernels in the same piece of constant memory and we need other data in the function to know which convolutional kernel to use in the function.

For the project we also made a Sobel Filter, is an algorithm used in computer vision and image processing for identifying regions characterized by sharp changes in intensity (edges). It computes an approximation of the gradient of the image intensity function by convolving the input image with a 3x3 kernel in the horizontal and vertical directions. The description of this algorithm is as follows:

1. Convert input image to a gray-scale image.
2. Compute Gx (gradient x), an approximation of the horizontal gradient of the image, by convolving the image with a 3x3 kernel:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * B$$

3. Compute Gy (gradient y), an approximation of the vertical gradient of the image, by convolving the image with a 3x3 kernel:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * B$$

4. Compute G, the final gradient magnitude, by combining the horizontal gradient Gx and the vertical gradient Gy with next formula:

$$G = \sqrt{G_x^2 + G_y^2}$$

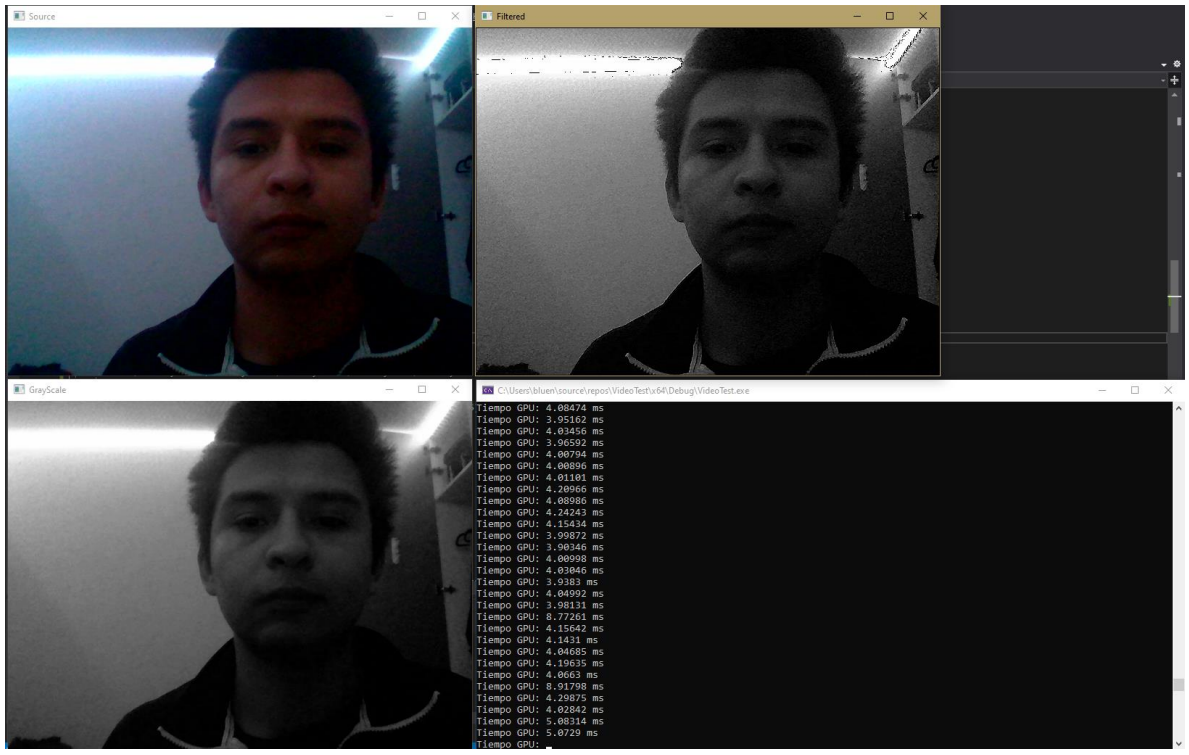
For the implementation of the Sobel filter, I reused the convolution already made to calculate gradients x and y and the implement another kernel to calculate the magnitude of both gradients and obtain the gradient magnitude:

```
44  __global__ void pythagoras(unsigned char* a, unsigned char* b, unsigned char* c)
45  {
46      int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
47
48      float af = float(a[idx]);
49      float bf = float(b[idx]);
50
51      c[idx] = (unsigned char)sqrtf(af * af + bf * bf);
52  }
```

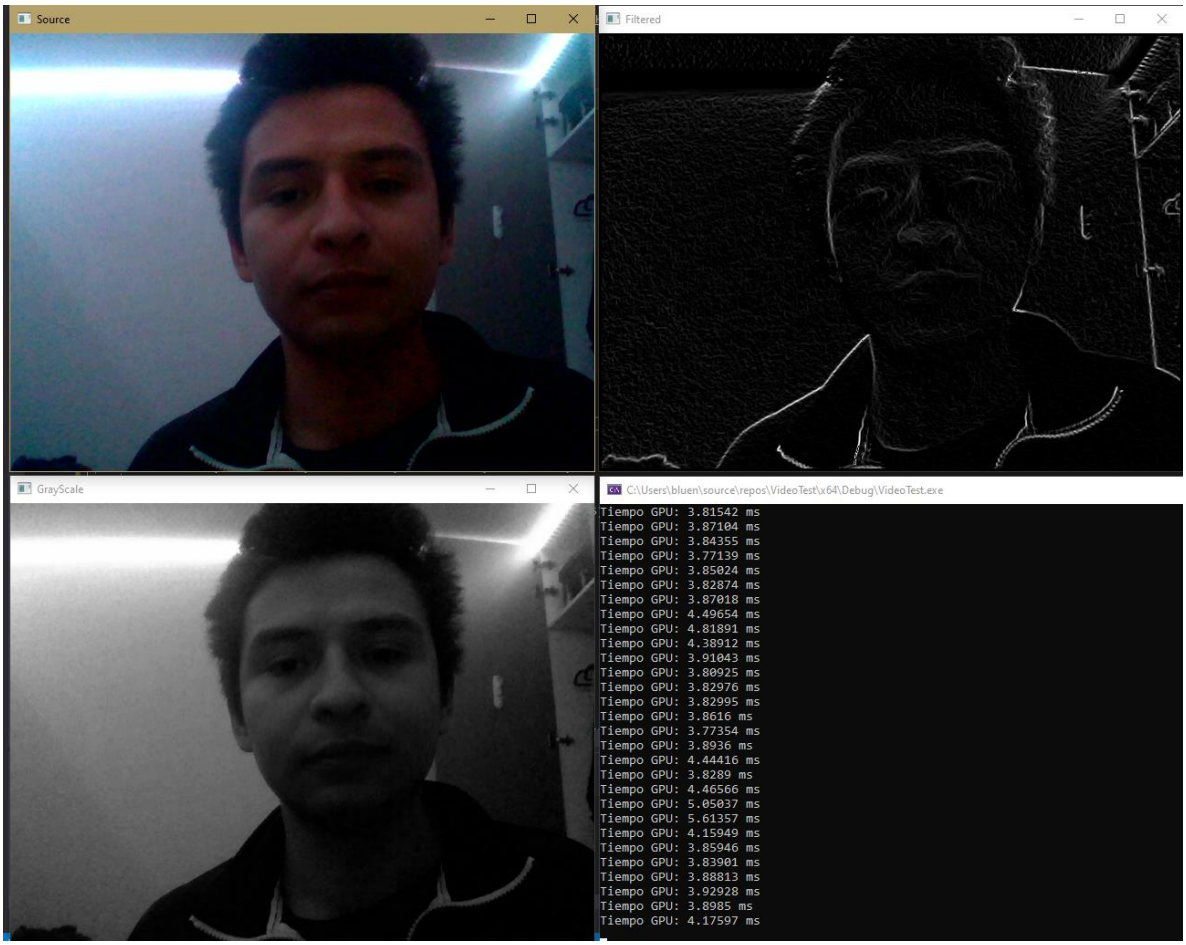
3. Results

As result of the previous implementations of the filters we obtain the next images an measurements:

In the image above we can see a gaussian blur filter being applied to the image in the window with title “filtered” we can se the result of the convolution, in which we should be able to distinguish a blurred image as the result. For this filter the kernel results to have a dimension of 5*5 and each element is a floating point number, which makes the calculation harder to make, as we can see reflected on the time on GPU as being the highest of all three filters.



For the image above the filtered image is the result of applying a sharpen kernel which as the name tell us “sharpens” the details or edges of the image to give a feel of higher definition or better quality taking to the GPU less than half the time of the blurring filter time since it is only a convolutional kernel of 3*3.



For the last case we will analyze the Sobel Filter, thus we could think it is the “heaviest” operation because of it having to depend on two other convolutional kernels calculations we can appreciate in the time measurements that in fact results to be one of the fastest. This is a great example of how parallelization can help to make some processes more efficient.

4. Conclusions

The most challenging part of the project was the implementation of the 2-dimensional convolution since the approach of it was to be able to receive different kernels depending on the one selected by the user. At first the idea was to pass the kernel as a parameter to the function and apply it to the source image, but for some reason that I couldn’t figure out the output always just showed black, and so I decided to try by defining different kernels on the __global__ function and switch between them and, that’s when I started getting some results. As I continued to investigate, I realized that the best way to manage a matrix that is not meant to change and only read is to use the constant memory of the device and that this would also make the process more efficient. After starting to implement this solution the problem now was how to implement all convolutional kernels in 1 simple variable and how to decide which part to use depending on the selection of the user, but after applying the same logic that we saw in class for handling indexes linearization it was simple. As a conclusion using CUDA library for real time image processing was harder than expected,

but the result were very smooth and they were up to the expectations with no lag between windows and almost immediate reaction to filter change.