CUDA Gaussian Blur

LUT university

BM40A1400_07.01.2019 GPGPU Computing

Author: Gia Duy DUONG

Teacher: Aleksandr Bibov

Introduction

CUDA is a parallel computing platform and application programming interface model created by NVIDIA. It is recommended for most deep learning purposes. Training new models will be faster on a GPU computer than a normal computer without GPU, because GPU processes the tasks parallelly.

In this report, I will show the Gaussian Blur algorithm along with the implementation in PyCUDA. Gaussian Blur is widely used effect in many graphics software, including Photoshop, Google Photos.

Prerequisites

To complete this implementation, you will need a computer with a CUDA-capable GPU. You also need to install and configure some software:

- Python 3.7
- PyCUDA v2018.1.1
- NumPy 1.6.2
- CUDA Toolkit 10.1
- Visual Studio 2017 with C++ compiler

Please also make sure the Python, CUDA and Visual C++ compiler directories are in your system execution PATH, more info please see https://documen.tician.de/pycuda/ and https://developer.nvidia.com/cuda-zone

Implementation

The Gaussian Blur algorithm is easy to implement, it uses a convolution kernel. The algorithm can be slow as it's processing time is dependent on the size of the image and the size of the kernel.

Step 1 - Load the input image, extract all the color channels (red, green, blue) of the image:

```
img = Image.open(input_image)
input_array = np.array(img)
red_channel = input_array[:, :, 0].copy()
green_channel = input_array[:, :, 1].copy()
blue_channel = input_array[:, :, 2].copy()
```

Step 2 - Select the size of the kernel, then use the formula of a Gaussian function to generate the matrix kernel. In this sample source code, the size of the kernel is 5x5:

```
sigma = 2  # standard deviation of the distribution
kernel_width = int(3 * sigma)
if kernel_width % 2 == 0:
```

```
    kernel_width = kernel_width - 1  # make sure kernel width only sth 3,5,7 etc

# create empty matrix for the gaussian kernel #
kernel_matrix = np.empty((kernel_width, kernel_width), np.float32)
kernel_half_width = kernel_width // 2
for i in range(-kernel_half_width, kernel_half_width + 1):
    for j in range(-kernel_half_width, kernel_half_width + 1):
        kernel_matrix[i + kernel_half_width][j + kernel_half_width] = (
                np.exp(-(i ** 2 + j ** 2) / (2 * sigma ** 2))
                / (2 * np.pi * sigma ** 2)
        )
gaussian_kernel = kernel_matrix / kernel_matrix.sum()
```

Step 3 - Convolution of image with kernel. If the image has 3 color channels, we process all the individual color channel separately by multiple the pixel value of every pixel corresponding to its location in the convolution kernel. Since we want to use GPU, we write the corresponding CUDA C code which save in `gaussian_blur.cu` file, Then we transfer the data from the host to the device, after the process is done, fetch the data back from the GPU. Here is the CUDA code:

```
__global__ void applyFilter(const unsigned char *input, unsigned char *output, const unsigned int width, const unsig

    const unsigned int col = threadIdx.x + blockIdx.x * blockDim.x;
    const unsigned int row = threadIdx.y + blockIdx.y * blockDim.y;

    if(row < height && col < width) {
        const int half = kernelWidth / 2;
        float blur = 0.0;
        for(int i = -half; i <= half; i++) {
            for(int j = -half; j <= half; j++) {

                const unsigned int y = max(0, min(height - 1, row + i));
                const unsigned int x = max(0, min(width - 1, col + j));

                const float w = kernel[(j + half) + (i + half) * kernelWidth];
                blur += w * input[x + y * width];
            }
        }
        output[col + row * width] = static_cast<unsigned char>(blur);
    }
}
```

The CUDA function takes the individual color channel, width & height of the image, and the Gaussian Kernel as the input params, then produce result as the color channel which we will use for saving the result image in the next step.

Step 4 - Merge all the output arrays (red, green, blue) and save as an output result image which is already blurred:

```
output_array = np.empty_like(input_array)
output_array[:, :, 0] = red_channel
output_array[:, :, 1] = green_channel
output_array[:, :, 2] = blue_channel

# save result image
Image.fromarray(output_array).save(output_image)
```

Usage

In the Python commandline, run the follow command:

`main.py input_image output_img`

For example:

`python main.py test.tif result.tif`

will take test.tif as the input image, and then save the blurred image in result.tif

Testing

On my testing device with a GeForce GTX 970M Dedicated Graphics, it only take ~0.05s to blur an full HD image, and ~0.14s for a 4K image. That is an amazing result.

```
D:\Dropbox (Personal)\LUT\GPCPU\gaussian-blur>python main.py 4k.jpg 4k-result.jpg
Total processing time:  0.1413777999999999 s

D:\Dropbox (Personal)\LUT\GPCPU\gaussian-blur>python main.py fullhd.jpg fullhd-result.jpg
Total processing time:  0.05094489999999996 s
```

References

- https://www.youtube.com/watch?v=7LW_75E3A1Q
- https://www.youtube.com/watch?v=LZRiMS0hcX4
- https://www.youtube.com/watch?v=C_zFhWdM4ic
- https://en.wikipedia.org/wiki/Gaussian_blur