Bulletin of the *Transilvania* University of Braşov • Vol. 12 (61) No. 1 - 2019
Series I: Engineering Sciences
https://doi.org/10.31926/but.ens.2019.12.61.7

TIME PERFORMANCE COMPARISON BETWEEN GPU AND CPU COMPUTING BY IMPLEMENTING AN EDGE-DETECTION ALGORITHM USING SOBEL FILTER

Alen SMAILOVIĆ[1]

Abstract: *In the last few years, GPUs (Graphics Processing Units) have made rapid development. Their ever-increasing computing power and decreasing cost have attracted attention from both industry and academia. In addition to graphics applications, researchers are interested in using them for general purpose computing. NVIDIA released a new computing architecture, CUDA (compute united device architecture), for its GeForce 8 series, Quadro FX, and Tesla GPU products. This new architecture can change fundamentally the way in which GPUs are used. In this paper, we study the programmability of CUDA and its GeForce 8 GPU and compare its performance with general purpose processors, in order to investigate its suitability for general purpose computation.*

Key words: *gpu, cpu, computing, cuda, edge-detection.*

1. Introduction

Graphics Processing Units (GPUs) have in recent years become increasingly programmable, giving rise to an emerging field of General-Purpose computation on Graphics Processing Units (GPGPU) [2]. The theoretical processing power of GPU's have far surpassed that of CPU's due to the highly parallel structure of GPUs.

Due to physical limitations, the clock speed of CPUs has come to a maximum limit. The recent trend in the microprocessor industry is to put more cores (processors) into a single chip. A GPU is a good example of specialized massively parallel processors with over a hundred of cores in the latest products. Yet a GPU is known to be notoriously hard to program. In June 2007, NVIDIA released the CUDA programming guide, v1.0, for its G80 GPUs [3]. The CUDA programming model offers programmers a straightforward C-like interface instead of mapping computation using the traditional graphic API, which makes it significantly easier to utilize a GPU.

In this paper is described in detail a comparison between CPU computing and GPU computing by using an algorithm for edge detection. The images used are with different

[1] Dept. of Electronics and Computers, *Transilvania* University of Braşov, Romania.

resolution so it will affect proportionally the computing time for each unit. For the edge detection algorithm is used Sobel filter: the CPU implementation is in Python and the GPU implementation is into a kernel with a Python script as a wrapper.

2. Implementation Methodology

In order to show the differences between this two units, the final form of edge detection algorithm is build and running on platforms as:

- CPU: Intel(R) Core(TM) i7-5500 @ 2.40GHz [4],
- GPU: GeForce GTX 950M [5].

2.1. Edge detection algorithm

The Sobel operator performs a 2-D spatial gradient measurement on an image and so emphasizes regions of high spatial frequency that correspond to edges. Typically it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image. In theory, the operator consists of a pair of 3x3 convolution kernels as shown in Figure 1.

-1	0	+1
-2	0	+2
-1	0	+1

+1	+2	+1
0	0	0
-1	-2	-1

Fig. 1. *3x3 Sobel convolution kernels (Gx - left, Gy - right)*

These kernels are designed to respond maximally to edges running vertically and horizontally relative to the pixel grid, one kernel for each of the two perpendicular orientations. The kernels can be applied separately to the input image, to produce separate measurements of the gradient component in each orientation. These can then be combined together to find the absolute magnitude of the gradient at each point and the orientation of that gradient.

The gradient magnitude is given by:

$$|G| = \sqrt{Gx^2 + Gy^2}.$$

But typically an approximate magnitude is computed using:

$$|G| = |G_x| + |G_y|.$$

The angle of orientation of the edge (relative to the pixel grid) giving rise to the spatial gradient is given by:

$$\theta = \arctan(G_y / G_x).$$

A Gaussian convolution operator is used to “blur” images and remove detail and noise. It uses different kernel that represents the shape of a Gaussian hump.

2.2. CPU implementation

For the CPU, the edge detection algorithm is implemented using Python where are imported several packages in order to achieve the results. We imported numpy - a fundamental package for scientific computing, Image - a module which represents a PIL Image, time - a class to access the current tick and math - a module with mathematical functions.

The base function where all the logic is done will be called with an argument which is pointing to the path of image. The steps are straightforward, first of all we will open the image and save it in a local variable called „img”. After this, the image will be converted to RGB matrix and save it into a numpy array. The numpy array is passed to a function called „grayScale” and we are applying an algorithm for gray scaling the image. In this way it will be easier to apply the Sobel operator. Next we are detecting the edges in the current image and save it into a RGB image.

The Sobel operator function is shown in Figure 2.

```
46  def sobelOperator( pxImg, width, height ):
47      # Sobel matrix for edge detection X & Y
48      sobelX = np.array([[-1, 0, 1],[-2, 0, 2],[-1, 0, 1]])
49      sobelY = np.array([[-1, -2, -1],[0, 0, 0],[1, 2, 1]])
50
51      # Apply sobel convolution for edge detecting
52      pxImg = conv_norm(sobelX, sobelY, pxImg, width, height)
```

Fig. 2. *Implementation of Sobel operator*

The Sobel convolution algorithm will compute the magnitude for X (pixel_x) and then for Y (pixel_y), afterwards to compute the gradient magnitude for each pixel of the image (as is shown in Figure 3).

```
28          # Compute the gradient magnitude
29          lenght[index] = (( pixel_x * pixel_x + pixel_y * pixel_y ) ** 0.5)
```

Fig. 3. *Compute the magnitude gradient for each pixel*

2.3. GPU implementation

The CUDA programming model is a heterogeneous model in which both the CPU and GPU are used. In CUDA, the „host” refers to the CPU and its memory, while the device” refers to the GPU and its memory. Code run on the host can manage memory on both the host and device, and also launches kernels which are functions executed on the device. These kernels are executed by many GPU threads in parallel [1].

The kernels are written in specific file with a .cu extension. The code will be compiled with CUDA C Compiler, nvcc which is part of NVidia CUDA Toolkit. The command in

order to compile the code is:

nvcc -o out kernel.cu.

The kernel used for writing this paper implements two functions: one for Sobel filter and one for gray scaling. Both are using threads which are running in parallel on GPU. The source code for gray scale function is presented in Figure 4.

```
109  extern "C" __global__ void gray_scale(const float * pixin, float * pixout, const int width, const int height)
110  {
111      int idx = (threadIdx.x) + blockDim.x * blockIdx.x;
112
113      if(idx < width * height) {
114
115          // Compute pixels to obtain a grayscale image
116          float px = 0.2126 * pixin[idx * 3 + 0] +
117                     0.7152 * pixin[idx * 3 + 1] +
118                     0.0722 * pixin[idx * 3 + 2];
119
120          // Save pixel into the output image (RGB matrix)
121          pixout[idx * 3 + 0] = px;
122          pixout[idx * 3 + 1] = px;
123          pixout[idx * 3 + 2] = px;
124      }
125  }
```

Fig. 4. *Source code for gray scaling the image in CUDA*

The input variable „pixin” represents the image send it from the Python wrapper. The „idx” represents the index on horizontal of each pixel from the image which is computed in parallel according with the size of block. In our case is 1024. The same strategy is used when the Sobel filter is applied, but there is one more index for vertical pixels.

Each time when we are using an algorithm implemented in CUDA (described above) and is running on GPU we must allocating memory inside the GPU for the image. In order to store data on the GPU that can be communicated back to the host, we need to have allocated memory that lives until it is freed, see global memory as the heap space with life until the application closes or is freed, it is visible to any thread and block that have a pointer to that memory region. Global memory resides on the device. However, there are ways to use the host memory as "global" memory using mapped memory. After allocating the memory, we will need to copy the image from host (CPU) to the device (GPU) and allocating the necessary memory inside the GPU for output image. When all of this are done we can run the kernel safe.

3. Results and Discussion

After running both applications, an example of the result image is presented in Figure 5. For testing the code, we used 5 types of images with different resolutions: VGA (640x480), HD (1280x720), FHD (1920x1080), QHD (3840x2160) and 4K (5120x3200).

As you can see in the result image, the details of edge detection are strong and you can easier apply other algorithms in order to detect different kind of objects.

Fig. 5. *Edge detection applied on image*

Talking about the time of running the application, on CPU is growing from less than 100 seconds to more than 500 seconds (almost 10 minutes) for the 4K type image. As you can see in Figure 6, the orange line represents the CPU computing-time.

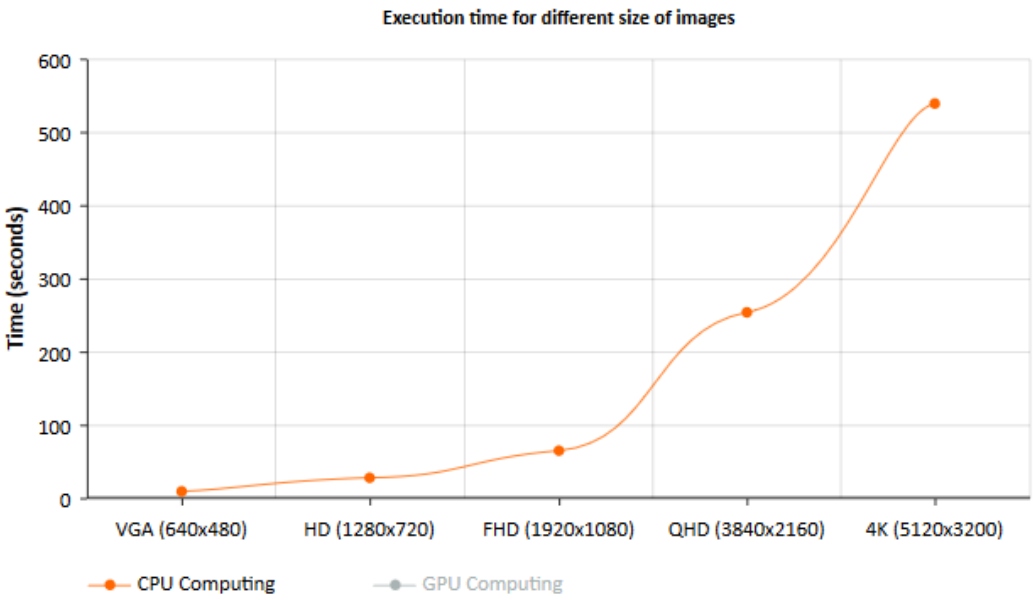

Fig. 6. *Execution time of Sobel Edge Detection on CPU*

The computing time for GPU is presented in Figure 7. You can clearly see that for an image with 4K resolution, the processing time is less than 5 seconds.

The operations involve the summation of an image buffer. The methods which are required the summation are moment calculations and solution to linear system of equations.

Experimental results show that GPU implementation achieve a better speedup over the other CPU implementations. This speedup value increases as increase the size of the image. As example computing time for VGA and HD resolution are almost the same.

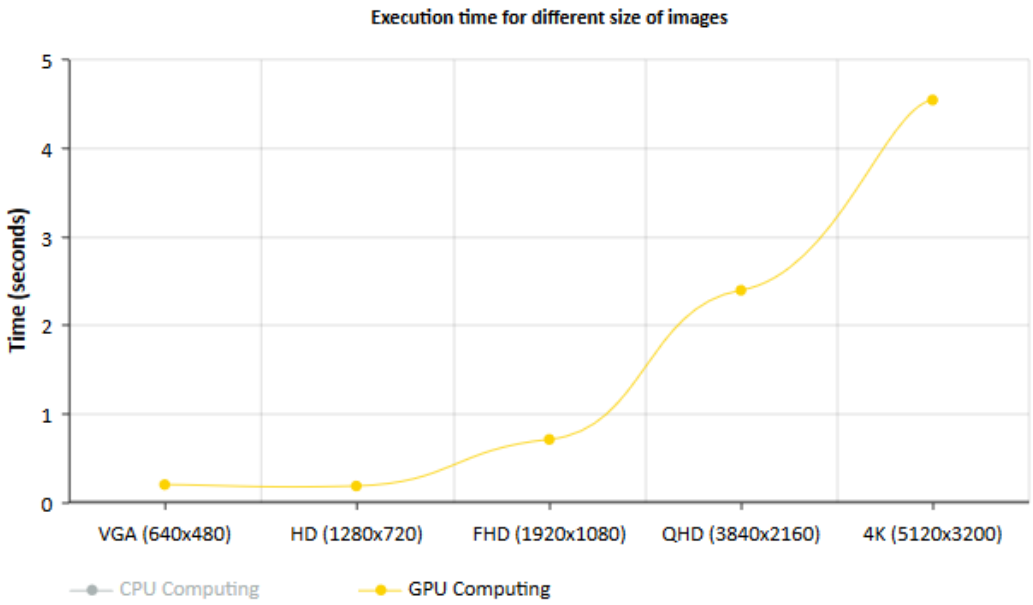

Fig. 7. *Execution time of Sobel Edge Detection on GPU*

4. Conclusion

The purpose of this project and course was to implement our first kernel and to compare performance between CPU and GPU. After we implemented our Sobel edge detector, we clearly saw the differences between the CPU and GPU (as it can be seen from our tables above). It is observed that NVIDIA graphics processing is able to solve many complex computational problems in fractions of the time required on a CPU. The capability to achieve faster speed depends upon parallelism in the program, but also to be more easily and with less effort developing that programs it will be required a friendly interface.

Acknowledgements

This work was developed in the framework of Lappeenranta University of Technology.

References

1. Storti, D., Yurtoglu, M.: *CUDA for Engineers: An Introduction to High-Performance Parallel Computing*. USA. Addison-Wesley Professional, 2015.
2. https://developer.nvidia.com/cuda-zone. Accessed: 10.01.2019.
3. https://videocardz.net/gpu/nvidia-g80/. Accessed: 12.01.2019.
4. https://ark.intel.com/products/85214/Intel-Core-i7-5500U-Processor-4M-Cache-up-to-3-00-GHz-. Accessed: 13.01.2019.
5. https://www.geforce.com/hardware/notebook-gpus/geforce-gtx-950m/specifications. Accessed: 13.01.2019.