Performance Analysis of Roberts Edge Detection Using CUDA and OpenGL

Marco Calì and Valeria Di Mauro
University of Catania, Catania, Italy

***Abstract*—The evolution of high-performance and programmable graphics processing units (GPUs) has generated considerable advancements in graphics and parallel computing. In this paper we present a Roberts filter based on edge detection algorithm using CUDA and OpenGL architectures. The basic idea is to use the Pixel Buffer Object (PBO) to create images with CUDA on a pixel-by-pixel basis and display them using OpenGL. The images can then be processed applying a Roberts filter for edge detection. Finally, it describes the results of an extensive measurement campaign as well as several comparisons among the code performance on CPUs and GPUs. The results are very promising since the GPU parallel version offers much higher performances than the CPU sequential version. The execution time of the GPU parallel version is much lower than the sequential equivalent execution time.**

***Index Terms*—CUDA, GPU, Image Processing, OpenGL, PBO, Roberts Edge Detection.**

I. Introduction

The technological development has increased computer performances and applications can perform more complex tasks [1], [2]. Recently the field of information retrieval has encountered a tremendous growth due to the newly available computational power and developed architectural supports. A large variety of scientific and industrial applications requires to analyze 2D images. In order to extract information from images it is often possible to apply specific filters to perform segmentation, edge identification, sharpening, etc. The most commonly used edge detection system is based on Roberts method.

In this paper we present a GPU parallel algorithm elaborating a great number of images in a short time. The solution can then be readily used for the realization of real-time devices and systems for industrial applications. A real-time execution requires to optimize the processing time for image elaboration and analysis. In this work we use the Common Unified Device Architecture (CUDA): an NVIDIA architecture which provides support to interact with the Graphic Processing Unit (GPU) for general-purpose computing. The solution presented is used to work with pgm grayscale images by OpenGL libraries implementing functions. An example of the features enabled by such functions are:

- loading an image in RAM;
- mapping an image to a Pixel Buffer Object (PBO);
- fast transferring the PBO data to GPU memory.

OpenGL libraries provide a direct control on image as well as making it possible to track the changes of the image without the developer. In this paper we have also measured the execution times and compared them to heterogeneous code execution on CPU, then we have observed the speedup advantages. The experiments were performed taking into account different image sizes and numbers.

In Section II the edge recognition operators basis will be introduced with particular attention to the Roberts operator and the respective Kernel. In Section III an introduction to GPU programming is presented as well as the multihreading logic used in CUDA, moreover the use of OpenGL libraries for processing image are explained relatively to the Pixel Buffer Object. Portions of the developed algorithm and the related comments are devised in Section IV. Finally, in Section V a performance analysis is presented as well as a comparison between the sequential and the parallel algorithm.

II. Image Processing

The digital filters used in image pre-processing can be represented as mathematical operators. Such digital filters may be used to reduce noise, improve contrast, separate objects from the background, etc. [3], [4], [5], [6]. It is possible to obtain substantial image improvement using parallel processing also for real time solutions [7]. Among the different types of image enhancement algorithms, spatial convolution kernel filtering produces the worst scenario.

A convolution kernel replaces every pixel with a new one so that its value is based on the relationship between the old pixel value and the values of pixels that surround it. In such a convolution procedure, two functions are overlaid: pixel values of the original image are stored in memory, and it is applied the mask of the convolution kernel. The kernels can vary in dimension, determining a different number of neighbouring pixels involved in the convolution. The kernel operates on the image by replacing the original pixels one by one, therefore the operation must be performed by applying the convolution operator to each pixel in the image. A typical convolution kernel mask operation is represented in Fig. 1.

The field of image processing and computer vision often make use of procedures like edge detection. Such a procedure is particularly useful for features extraction. The edge detection eliminates the informations in order to focus only on a specific variations set on the image based on the geometric and structural characteristics of the examined objects in order to recognize the borders. An example of such a variation is constituted by the region of pixels where the light intensity undergoes abrupt changes. Those rapidly variating regions can

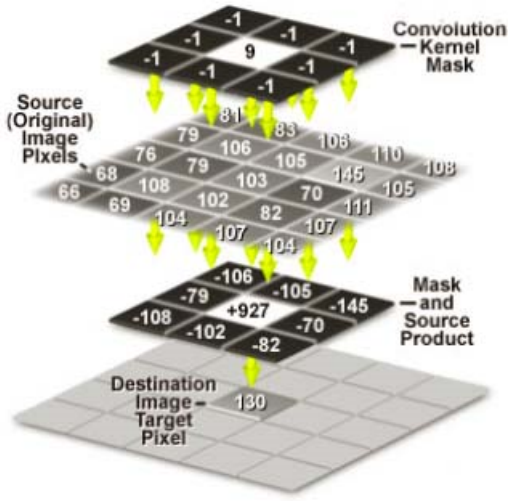

Fig. 1. Convolution kernel mask operation

be identified as the edges of a certain object on the image. The edge detection method is based on digital Roberts filter. The Roberts filter operator approximates the intensity gradient of the brightness using two different kernels:

$$G_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \qquad G_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \tag{1}$$

G_x, also called horizontal kernel, is able to enhance the horizontal component of the intensity gradient, the G_y, also called vertical kernel, at each point, enhances the vertical component. When combined G_x and G_y give the Roberts kernel

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

When G is applied to an image, a threshold is used to determine which values indicate a boundary and which do not. By increasing the pixels gradient value the lines tend to white.

III. GPU COMPUTING

The GPU architecture consists of a scalable number of streaming multiprocessors (SMs), each containing eight streaming processor (SP) cores, two special function units (SFUs), a multithreaded instruction fetch and issue unit, a read-only constant cache, and a 16KB read/write shared memory [8]. The SM executes a batch of 32 threads together called a warp [9], unlike SIMD instructions the concept of warp is not exposed to programmers, rather programmers write a program for one thread and then specify the number of parallel threads in a block, and the number of blocks in a kernel grid.

In an SM all threads block should be executed all together. On the other hand, a SM can manage multiple concurrently running blocks. The number of blocks running on a SM is determined by the resource requirements of each block as well as the registers and shared memory usage. We will use the adjective *active* for the executed blocks on one SM at a certain moment. One block typically is executed on several warps and the number of warps consists of the total number of GPU cores divided by the number of cores contained in one warp. The latter on Nvidia GPU cards has generally been 32, but could change in the future for new card models.

CUDA architecture makes it possible to have direct access to the GPU instruction set enabling us to use such GPU card for general purpose parallel computing. The management and programming is supported by the CUDA APIs [10]. Each CUDA thread is mapped to a GPU core. The GPU card can execute one or more kernel grids, as well as the streaming multiprocessors (SM) which execute one or more blocks. CUDA architecture provides APIs and directives in order to be compatible with different standard programming languages such as C, C++, Fortran, etc. The main advantages is then the easy implementation possibilities offered to the developers which have experience in the said standard programming languages, and the possibility to achieve highly modular software systems [11], [12], [13]. Moreover, CUDA architecture has been proven a more efficient method to port developed software systems with respect to other GPU oriented technologies such as Direct3D and OpenGL.

Nevertheless, the OpenGL (Open Graphics Library) [14], [15], [16] offers a great number of software tools for the development of GPU oriented code. OpenGL offers cross-language and multi-platform application programming interfaces (API) generally used in rendering applications. These APIs are used to interact with a GPU to achieve hardware-accelerated rendering, therefore OpenGL allows applications to use advanced graphics on relatively small systems. It is possible to mix OpenGL and CUDA technology in order to enhance the performances of an application. An example is given by the use of PBO (Pixel Buffer Object) which makes it possible to import multidimensional structures on a pixel-by-pixel basis and display them using the related OpenGL APIs. The advantage in such an approach is to directly obtain an efficient mapping of pixels directly into threads [17]. In facts, PBO is a specific portion of the video memory in which it is possible to render images that can be transformed into textures. Another important advantage of the approach is the speed of pixel data transfer through DMA (Direct Memory Access) channel [18]. The OpenGL PBO mechanism also permits asynchronous data transfers between the host and the device [19]. It is therefore important to correctly schedule the workload between different memory transfers in order to maximise the performance obtainable with the asynchronous approach.

The texture data are loaded from an image source (image file or video stream) that can be directly loaded into a PBO, which is controlled by OpenGL. Fig. 2 gives a simplified schema of the texture transfer using the Pixel Buffer Object. Of course, while the transfer is asynchronous, it is anyway needed a certain amount of CPU workload in order to transfer data from the host memory to the PBO. Then, after such transfers,

SIMD and MIMD are types of parallel architectures identified in Flynn's taxonomy, which basically says that computers have Single or Multiple streams of Instructions processing single or Multiple Data

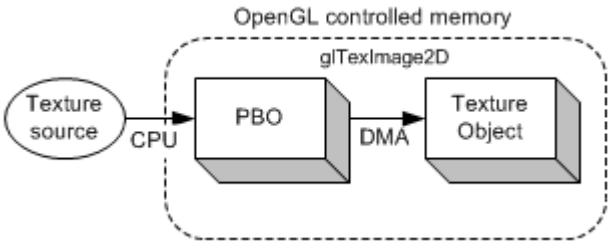

Fig. 2. Texture loading with PBO.

the GPU controllers (driven by the OpenGL drivers) manage to copy data from a PBO to a texture object. This means that OpenGL performs a DMA transfer operation without wasting CPU cycles, so the CPU benefits from a lower workload and can perform other operations without waiting such a transfer to be completed.

Since in this work we make use of both OpenGL and CUDA directives, in order to correctly map the pixels using the PBO, it is firstly needed to create a GL context (which is OS specific) and a CUDA buffer registration. After that it is necessary to set up the GL view port and coordinate system [20], generate one or more GL buffers to be shared with the CUDA application and, subsequently, register these buffers within the application itself (see Fig. 3).

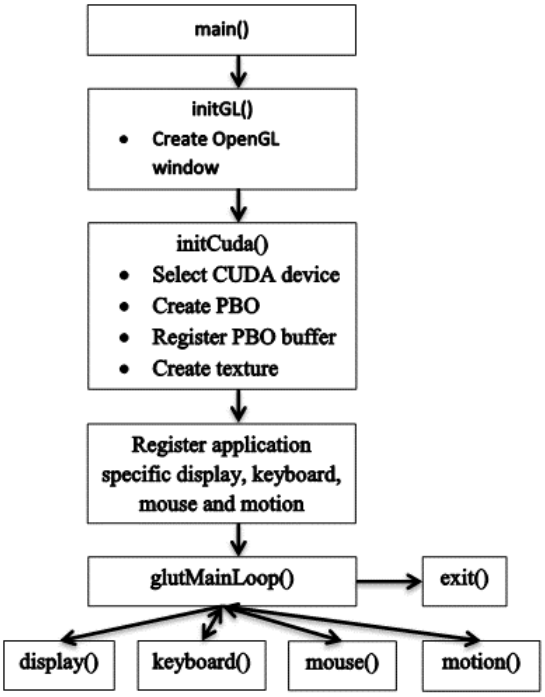

Fig. 3. CUDA Buffer Registration.

A. Upload

The algorithm used to upload data using PBOs [21] is as follows.

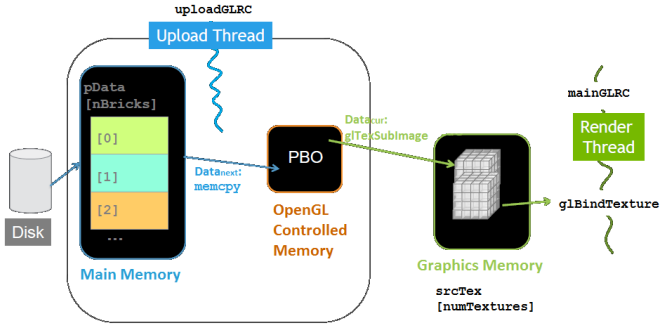

Fig. 4. Upload Application Layout.

1) Creating a PBO on the GPU using `glGenBuffers`
2) Binding PBO to unpack the buffer target
3) Allocating buffer space on GPU with `glBufferData`
4) Mapping PBO to CPU memory denying GPU access for now, `glMapBuffer` returns a pointer to a place in GPU memory where the PBO resides
5) Copying data from CPU to GPU using pointer from `glMapBuffer`
6) Unmapping PBO (`glUnmapBuffer`) to allow GPU full access of the PBO again
7) Transfering data from buffer to a texture target
8) Unbinding the PBO to allow for normal operation again

Steps one to three are only necessary during initialization, while steps four to eight have to be performed every time the texture needs to be updated.

Fig. 4 shows a generic upload application layout.

B. Download

PBOs can also be used to download data back to the CPU. There is one problem which must be addressed though; as the download is an asynchronous operation (using DMA) one must make sure that the GPU does not clear the render buffer before the transfer is complete.

The following is a description of how to download data using PBOs.

1) Generating a PBO on the GPU using glGenBuffers
2) Binding PBO to unpack buffer target
3) Allocating buffer space on GPU according to data size using glBufferData
4) Deciding what framebuffer to read using glReadBuffer. One can also read directly from a texture using glGetTexImage, then skipping the next step
5) Using glReadPixels to read pixel data from the targeted framebuffer to the bound PBO. This call does not block as it is asynchronous when reading to a PBO as opposed to CPU controlled memory. Map PBO to CPU memory denying GPU access for now. glMapBuffer returns a pointer to a place in GPU memory where the PBO resides
6) Copying data from GPU to CPU using pointer from glMapBuffer

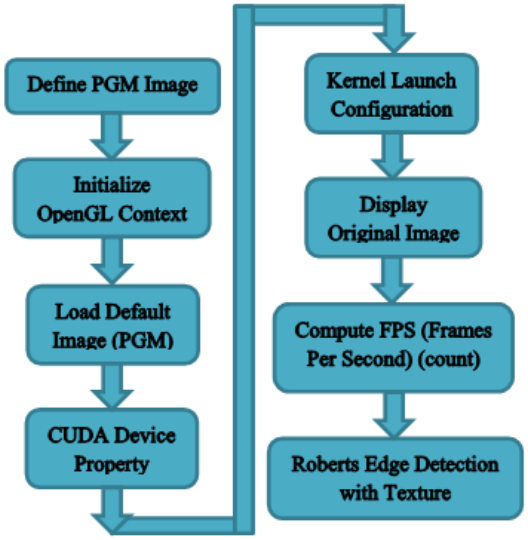

Fig. 5. Roberts Edge Detection procedure.

7) Unmaping PBO (glUnmapBuffer) to allow GPU full access of the PBO again
8) Unbinding the PBO to allow for normal operation again

Steps one to three are only necessary during initialization, while steps four to nine need to be performed every time new data have to be downloaded. However, downloads and uploads still involve GPU context switch and cannot be done in parallel with the GPU processing or drawing. Multiple PBOs can potentially speed up the transfers.

IV. ROBERTS ALGORITHM

Roberts edge detection filter can be broken down in different steps. First we need to set up the image data objects which will store the intermediate and final results, obtaining the size of the input image with the OpenGL libraries. Next, we have to do the computation using Roberts operators and the sum of the Roberts operators to create and edge image on the CUDA. Roberts Edge Detection procedure is displayed in Fig. 5.

The algorithm using the features provided by OpenGL and CUDA will be described below [22].

We now describe how to create a convolution kernel of Roberts and apply it for an image processing in CUDA. The following code is for Roberts Edge Detection.

```
// Roberts Edge Detection
Matrix2x2_original_image(upper_left, upper_right,
                         lower_left, lower_right)

// Horizontal kernel
principal_diagonal = lower_right - upper_left;

// Vertical kernel
secondary_diagonal = lower_left - upper_right;

// Module of gradient
Median = square(abs(principal_diagonal)+abs(secondary_diagonal))
```

The following code is for copying the original image. The main operations are: copy of image, insert original data.

```
// Boundary control Median_range_control
if Median lower inf then
    return inf
else
    if Median higher sup then
        return sup
    end
end
return (unsigned char) Median
// because the PBO works using data char
```

```
// Definition space memory to be used in GPU

*pointer new array = pointer original array
              + blockIdx.x*Dimension z

cycle (start to threadIdx.x; stop to widht;
               increase of blockDim.x)
```

Then, perform a texture lookup in a given 2D sampler with the function `tex2D` and associate it to pointer of new array in the position (step cycle).

The elaboration of image is performed by means of the following steps: initializing an array in CUDA for the elaborate image, inserting Original data.

```
// Definition space memory to be used in GPU

*pointer new array = pointer original array
              + blockIdx.x*Dimension z

cycle (start to threadIdx.x; stop to Dimension z;
               increase of blockDim.x)
```

Then, performing a texture lookup in a given 2D sampler with the function `tex2D` and insert it in the matrix of Roberts.

```
upper_left  = tex2D(lookup, coordinate_to_perform_lookup -1,
                            dimension_x-1)

upper_right = tex2D(lookup, coordinate_to_perform_lookup,
                            dimension_x-1)

lower_left  = tex2D(lookup, coordinate_to_perform_lookup -1,
                              dimension_x)

lower_right = tex2D(lookup, coordinate_to_perform_lookup,
                              dimension_x)
```

Finally, the matrix of Roberts is associated to the pointer of a new array.

Then, it is necessary to define the setup of the texture in an external C function, according to the following steps.

- Describe the format of the value that is returned when fetching the texture through the `"cudaChannelFormatDesc"` function

```
if the image is "pgm" then
    assigned format unsigned char
else
    assigned format uchar4
end
```

- Allocate a CUDA array using `cudaMallocArray()` function that is included in `checkCudaErrors`, used to correct the string in case of errors.

- Copy from the memory area pointed to by src to the CUDA array dst using `cudaMemcpyToArray()` function, and specifying the direction of the copy with `cudaMemcpyHostToDevice`.

For deleting the texture with an external C function, the following steps are needed.

- Release the CUDA array using `cudaFreeArray()` function, which must have been returned by a previous call.
- Wrap for the __global__ call that sets up the texture and threads in an external void C function.
- Bind the CUDA array to the texture reference tex through `cudaBindTextureToArray()` function.

switch *for different cases* **do**
  case *original image*
    sets up the texture and threads break
  case *elaborate image with ROBERTS*
    sets up the texture and threads break
  end
endsw

- Unbind the texture bound to tex.

For elaborating an image with OpenGL the code below is used. Such a graphic library is designed to aid in rendering computer graphics. This typically involves providing optimized versions of functions that handle common rendering tasks. It can be realized purely by code running on the CPU, common in embedded systems, or code running on hardware accelerated by a GPU, more common in PCs. By employing these functions, a program can prepare an image to be output to a monitor. These libraries load the image data into memory, map between screen and world-coordinates, generate of texture mipmaps and also ensure interoperability with other third party libraries and SDK.

More specifically, for function `display()` the following steps have to be performed.

- Map the graphics resources (PBO) in resources for access by CUDA using `cudaGraphicsMapResources()` function.
- Return a pointer through which the mapped graphics resource may be accessed using `cudaGraphicsResourceGetMappedPointer()` function.
- Unmap the graphics resources in PBO (and once unmapped, the resources can not be accessed by CUDA until they are mapped again) using `cudaGraphicsUnmapResources()`.
- Use function `glClear()` for the bitwise of masks that indicate the buffers to be cleared; using the mask `GL_COLOR_BUFFER_BIT`, indicating the buffers currently are enabled for color writing.
- Bind a named texture to a texturing target using `glBindTexture()` function; then bind a named buffer object using `glBindBuffer()` function, and taking it from PBO buffer.
- Specify a two-dimensional texture subimage, with target, texture, level, xoffset, yoffset, width, height, format, type, pixels, using `glTexSubImage2D()` function.
- Disable server-side `GL_DEPTH_TEST` by `glDisable()`, or enable server-side `GL_TEXTURE_2D` by `glEnable()`.
- Set texture parameters by means of `glTexParameterf()`, with target `GL_TEXTURE_2D`, which specifies the target texture of the active texture unit, and pname `GL_TEXTURE_MIN_FILTER`, `GL_TEXTURE_MAG_FILTER`, `GL_TEXTURE_WRAP_S`, `GL_TEXTURE_WRAP_T`, which specifies the symbolic name of a single-valued texture parameter, and finally param `GL_LINEAR`, `GL_REPEAT`, which specifies the value of pname.
- Specify the primitive or primitives that will be created from vertices presented between `glBegin` and the subsequent `glEnd` (the primitives are specified by `glVertex2f` and `glTexCoord2f`)
- Perform a buffer swap on the layer in use for the current window through `glutSwapBuffers`

Function `reshape()` consists of the following main steps.

- Specify the affine transformation of x and y from normalized device coordinates to window coordinates using `glViewport`.
- Specify `GL_PROJECTION` is the target for subsequent matrix operations by means of `glMatrixMode`.
- Replace the current matrix with the identity matrix through `glLoadIdentity`, and describe a transformation that produces a parallel projection using `glOrtho`.
- Specify `GL_MODELVIEW` is the target for subsequent matrix operations by means of `glMatrixMode`.
- Replace the current matrix with the identity matrix using `glLoadIdentity`.

For function `cleanup()` the following steps are performed.

- Unregister a graphics resource for access by CUDA using `cudaGraphicsUnregisterResource`, with resource `cuda pbo`.
- Bind a named buffer object using `glBindBuffer` taking it from `GL_PIXEL_UNPACK_BUFFER`.
- Delete a buffer object named by the elements of pbo buffer by means of `glDeleteBuffers`, and delete a texture named by the elements of texid using `glDeleteTextures`.

In addition to the above functions, a `main()` function has been implemented in order to initialize data, load a default image, run the processing functions, and save results.

V. Performance Analysis

The performance analysis has been done by comparing execution time of parallel computing with its sequential counterpart. Intel i7 processor and NVIDIA GEFORCE 845m devices have been used. Detailed technical specifications are

TABLE I
GPU TECHNICAL SPECIFICATION

Memory Bandwidth	16 GB/s
Pixel Rate	13.8 GPixel/s
Texture Rate	27.6 GTexel/s
Floating Point Performance	883.7 GFLOPS
Shading Units	512
Texture Mapping Units	32

TABLE II
CPU TECHNICAL SPECIFICATION

Processor Number	i7-4710HQ
Intel®Smart Cache	6 MB
DMI2	5 GT/s
# of Cores	4
# of Threads	8
Processor Base Frequency	2.5 GHz
Max Turbo Frequency	3.5 GHz
Processor Graphics	Intel®HD Graphics 4600
Graphics Base Frequency	400 MHz
Graphics Max Dynamic Frequency	1.2 GHz

given in TABLE I and TABLE II. The code has been compiled using CUDA runtime v.7 and CUDA compute capability 5.

The tests were carried out using the same grayscale image in four different sizes, in order to evaluate temporal variations for different sizes.

The image size has been varied from 128x128 pixels to 1024x1024 pixels. For each of them the algorithm works so that, after loading the image, it is possible to select the original image or the one processed (through the use of Roberts edge detection) by using a selection menu, as shown in Fig. 6

Roberts operator provides the edge detection of the image at small scales. If an object with a jagged boundary is present, as it is shown in Fig. 6 (left), Roberts operator will find the edges at each spike and twist of the perimeter as in Fig. 6 (right). The operator is sensitive to high frequency noise in the image and will generate only local edge data instead of recovering the global structure of a boundary.

For each image, multiple runs were carried out and the greatest amounts of processing time were recorded, by using `CudaEventElapsedTime()` function for the GPU version and `SDK Timer` function for the CPU version. Then it was calculated the average for different image sizes, as shown in TABLE III.

The execution times for the sequential and parallel versions on CPU and GPU are plotted in Fig. 7. Looking at the data in Fig. 7, we can state that the average processing CPU time is greater than the average processing GPU time, particularly when the image size increses then the sequential version of the program has a considerable loger execution time. For each image size, it has been computed the average processing time ratio between CPU and GPU, which is in the range between 4.1 and 4.4, and it is shown in Fig. 8.

Fig. 9 shows the comparison between the average processing time on the GPU and CPU for a number of images equal to 50. It illustrates that the time of the CPU is higher than the

TABLE III
AVERAGE EXECUTION TIME

Image Size (Pixel)	**CPU Average Processing Time**	**GPU Average Processing Time**	**Average Processing Time Ratio**
128 x 128	0,000488625	0,000114875	4,2535365
256 x 256	0,001517857	0,000360125	4,2148064
512 x 512	0,005681625	0,001378625	4,1212259
1024 x 1024	0,0205595	0,00464925	4,4221111

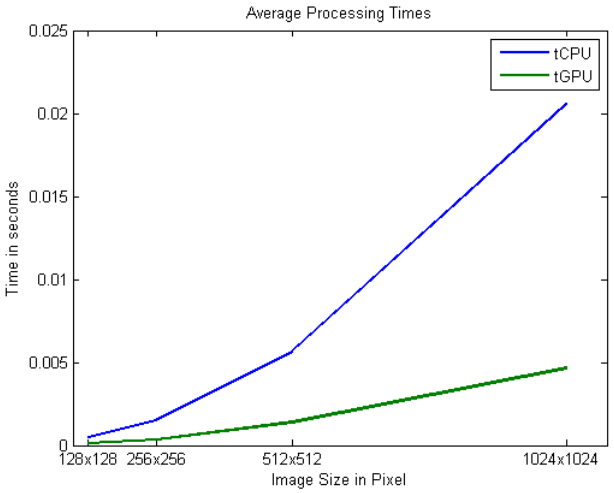

Fig. 7. Average Processing Times.

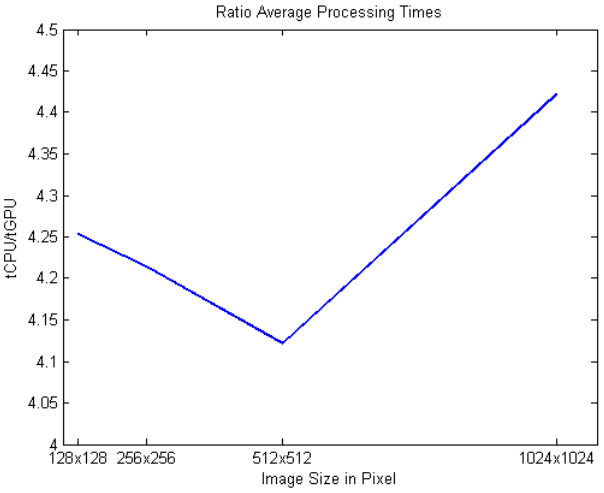

Fig. 8. Average Processing Times Ratio.

Fig. 6. Original image (left) and Processed image (right).

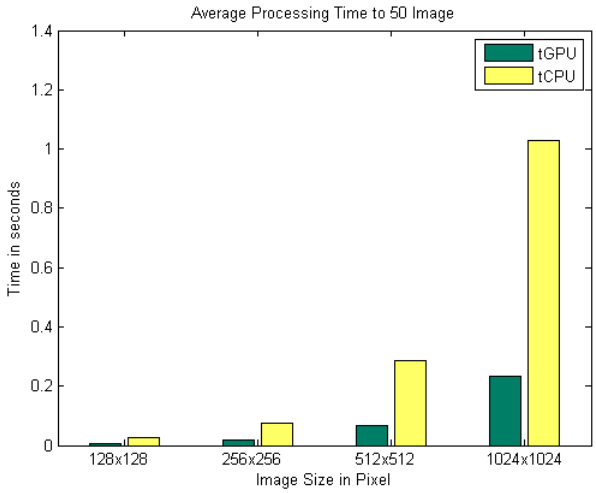

Fig. 9. Average Processing Times to 50 Image.

time of the GPU. Moreover, if we process a greater number of images, the time difference is considerable.

VI. Related Works

The edge detection can be carried out by different types of operators such as Roberts, Canny, Deriche, Differential, Prewitt, Sobel. An example regarding how to use Sobel Edge Detection can be seen in the following articles.

In [23], authors have used the Sobel kernel for edge detection. The processing times were compared using code developed for CPU and GPU, in order to observe the improvements reported by the GPU computing. Furthermore, a study has been performed about the timing given by the use of field-programmable gate array (FPGA), noting that the running times are comparable to GPU. The authors have included in the code the steps for loading the file, storing it in memory, the data transport on CPU to GPU and back, and the visualization on a window screen, without direct control of images by OpenGL libraries.

Shah [20] has proposed the implementation of the Sobel filter on a GeForce GT 130. Image processing is performed in CUDA environment with the GPU computing; OpenGL libraries provide a control of frame and the insertion into the memory. In the end, the execution times were compared with each other taking into account a heterogeneous code on CPU and observing the improvements of speedup on GPU.

VII. Conclusions

As shown by the performance between execution time of parallel computing with its sequential counterpart, the performance GPU results indicate that significant speedup can be achieved. In fact, Roberts edge detection can get a substantial speedup, compared to the CPU based implementations, obtaining an average processing time CPU over GPU ratio in the range 4.1 to 4.4.

The time reduction is significant in a great number of processing tasks, allowing the detection of images in high quality and in real time. This is due to GPUs, which provide a novel and efficient acceleration technique for image processing, cheaper than hardware implementation.

In the future work, we plan to implement a code which allows independent and parallel execution cores, so as to further reduce the execution times. The unstructured programming logic provides freedom of architecture and an implementation of algorithms able to execute more articulated functions. The capacity to achieve high speeds depends on the core number and the program architecture using the parallelism in the best way, to reduce the execution times.

REFERENCES

[1] C. Napoli, G. Pappalardo, and E. Tramontana, "An agent-driven semantical identifier using radial basis neural networks and reinforcement learning," in *XV Workshop "From Objects to Agents" (WOA)*, vol. 1260. Catania, Italy: CEUR-WS, September 2014.

[2] C. Napoli, G. Pappalardo, E. Tramontana, R. Nowicki, J. Starczewski, and M. Woźniak, "Toward work groups classification based on probabilistic neural network approach," in *Proceedings of International Conference on Artificial Intelligence and Soft Computing (ICAISC)*, ser. Springer LNCS, Zakopane, Poland, June 2015, vol. 9119, pp. 79–89.

[3] C. Napoli, G. Pappalardo, E. Tramontana, Z. Marszałek, D. Połap, and M. Woźniak, "Simplified firefly algorithm for 2d image key-points search," in *Symposium on Computational Intelligence for Human-like Intelligence (CHILI)*, ser. Symposium Series on Computational Intelligence (SSCI). IEEE, 2014, pp. 118–125. [Online]. Available: http://dx.doi.org/10.1109/CIHLI.2014.7013395

[4] M. Woźniak, C. Napoli, E. Tramontana, G. Capizzi, G. Lo Sciuto, R. Nowicki, and J. Starczewski, "A multiscale image compressor with rbfnn and discrete wavelet decomposition," in *Proceedings of IEEE International Joint Conference on Neural Networks (IJCNN)*, Killarney, Ireland, July 2015, pp. 1–7, DOI: 10.1109/IJCNN.2015.7280461.

[5] M. Woźniak, D. Połap, M. Gabryel, R. Nowicki, C. Napoli, and E. Tramontana, "Can we process 2d images using artificial bee colony?" in *Proceedings of International Conference on Artificial Intelligence and Soft Computing (ICAISC)*, ser. Springer LNCS, Zakopane, Poland, June 2015, vol. 9119, pp. 660–671.

[6] G. Capizzi, G. Lo Sciuto, C. Napoli, E. Tramontana, and M. Woźniak, "Automatic classification of fruit defects based on co-occurrence matrix and neural networks," in *Proceedings of IEEE Federated Conference on Computer Science and Information Systems (FedCSIS)*, Lodz, Poland, September 2015, pp. 873–879.

[7] J. Wade and R. Millar, "Fpga-powered display controllers enhance isr video in real time," Z Microsystems, May 2012. [Online]. Available: http://mil-embedded.com/articles/fpga-powered-isr-video-real-time/

[8] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39 – 55, March 2008.

[9] (2009) Nvidias next generation cuda™ compute architecture: Fermi™. NVIDIA Corporation. [Online]. Available: http://www.nvidia.it/page/home.html

[10] A. Lal, Shimpi, Derek, and Wilson, "Nvidia's geforce 8800 (g80): Gpus re-architected for directx 10," *AnandTech*, November 2006. [Online]. Available: http://www.anandtech.com/show/2116

[11] R. Giunta, G. Pappalardo, and E. Tramontana, "AODP: refactoring code to provide advanced aspect-oriented modularization of design patterns," in *Proceedings of ACM Symposium on Applied Computing (SAC)*, Riva del Garda, Italy, March 2012, pp. 1243–1250.

[12] G. Pappalardo and E. Tramontana, "Suggesting extract class refactoring opportunities by measuring strength of method interactions," in *Proceedings of Asia Pacific Software Engineering Conference (APSEC)*. Bangkok, Thailand: IEEE, December 2013, pp. 105–110.

[13] E. Tramontana, "Automatically characterising components with concerns and reducing tangling," in *Proceedings of IEEE Computer Software and Applications Conference (COMPSAC) Workshop QUORS*, Kyoto, Japan, July 2013, pp. 499–504.

[14] "Opengl wiki," OpenGL.org. [Online]. Available: https://www.opengl.org/wiki/Main_Page

[15] M. Segal and K. Akeley, *The OpenGL Graphics System: A Specification (Version 4.0)*. OpenGL.org, March 2010. [Online]. Available: https://www.opengl.org/registry/doc/glspec40.core.20100311.pdf

[16] D. Shreiner, G. Sellers, J. Kessenich, and B. Licea-Kane, *OpenGL Programming Guide Eighth Edition*. Addison-Wesley, March 2013.

[17] S. A. MahMoudi and P. Manneback, "Parallel image processing on gpu with cuda and opengl."

[18] S. H. Ahn. Opengl pixel buffer object (pbo). [Online]. Available: http://www.songho.ca/opengl/gl_pbo.html

[19] K. Group. Arb_pixel_buffer_object, architecture review board (arb). [Online]. Available: http://www.opengl.org/registry/specs/ARB/pixel_buffer_object.txt

[20] K. Shah, "Performance analysis of sobel edge detection filter on gpu using cuda & opengl," *International Journal for Research in Applied Science and Engineering Technology (IJRASET)*, vol. 1 Issue III ISSN: 2321-9653, October 2013. [Online]. Available: http://www.ijraset.com/

[21] J. Sandgren, "Transfer time reduction of data transfers between cpu and gpu," *Teknisk-naturvetenskaplig fakultet UTH-enheten*, July 2013. [Online]. Available: http://www.teknat.uu.se/student

[22] Nvidia cuda library documentation 4.1. NVIDIA Corporation. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/modules.html

[23] M. Chouchene, F. E. Sayadi, Y. Said, M. Atri, and R. Tourki, "Efficient implementation of sobel edge detection algorithm on cpu, gpu and fpga," *International Journal of Advanced Media and Communication*, vol. 5, pp. 105 – 117, April 2014.