Point to point processing of digital images using parallel computing

Eric Olmedo, Jorge de la Calleja, Antonio Benitez, and Ma. Auxilio Medina

Laboratorio de Percepción por Computadora
Universidad Politécnica de Puebla
Puebla, 72640, México

Abstract

This paper presents an approach the point to point processing of digital images using parallel computing, particularly for grayscale, brightening, darkening, thresholding and contrast change. The point to point technique applies a transformation to each pixel on image concurrently rather than sequentially. This approach used CUDA as parallel programming tool on a GPU in order to take advantage of all available cores. Preliminary results show that CUDA obtains better results in most of the used filters. Except in the negative filter with lower resolutions images OpenCV obtained better ones, but using images in high resolutions CUDA performance is better.

Keywords: *Digital image processing, parallel computing, CUDA.*

1. Introduction

Digital image processing has become an applied research area that goes from professional photography to several different fields such as astronomy, meteorology, computer vision, medical imaging, among others. The aim of digital image processing is to improve the pictorial information in order to perform subsequently other tasks such as image-based classification, feature extraction or pattern recognition. Image processing is usually an expensive and time-consuming task, for example in point to point processing, a grayscale image of 1024×1024 pixels, will require a CPU make to more than one million operations, and if it is a color image the number of operations must be multiplied by the number of channels.

In recent years, video cards or graphics processing units (GPU) have become tools for processing in parallel large amounts of information (in order of millions of data). NVIDIA developed the CUDA architecture that groups cores of GPU in a vector which can be programmed to reduce processing time over large amounts of data [14].

The use of a GPU to parallelize tasks started several years ago, for example, Fung and Mann [1] in 2004 proposed a new architecture using multiple GPUs for image processing and computer vision; they obtained significant speed up over a CPU implementation. In 2006 Farrugia et al [2] developed the GPUCV library to speed up the images time processing using GPUs; they observed that processing time with GPUCV varies from 18 to 1.2 times faster than native OpenCV function. With the emergence of CUDA, research was only focused on the design of algorithms and the task of how to operate a GPU was "forgotten" for the processing task. In 2007 Sham *et al* [16] presented an efficient method to compute mutual information (MI) between images. They improve the efficiency of MI calculations by a factor of 25 compared with a standard CPU-based implementation. Fung and Mann [3] in 2008 used CUDA to assist in "converting pictures into numbers" (i.e. computer vision). They obtained a speed up from 9.8 until 21 times than CPU implementation. In 2008 Zhiyi *et al* [4] improved execution times for some filters as histogram equalization, edge detection, DCT encode and decode algorithms; they obtained times from 8 to 200 faster than CPU. Tarabalka *et al* [17] in 2009 used GPUs for real-time processing of large data volumes recorded by a hyperspectral image; they reported that their GPU implementation run significantly faster, about 10 to 100 times.

This paper presents an approach, the point to point processing of digital images using parallel computing order to reduce execution times. This approach is tested over images of several resolutions, using grayscale, brightening, darkening, thresholding and contrast of images. The results show that the filters implemented in CUDA are faster than OpenCV functions and C implementations.

The remainder of the paper is organized as follows. In Sections 2, 3 and 4 we include a brief background on digital images processing, parallel computing, CUDA and OpenCV. Section 5 describes the proposed approach, while experimental results are showed in Section 6. Finally conclusions and future work are presented in Section 7.

2. Digital image processing

According to Jain [5], digital image processing consists of the application of functions that transform a two-dimensional image using a computer. Others authors as

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.

Crane [6] define this task as a science that manipulates digital images that covers an extend set of techniques to enhance or distort them.

A digital image is a set of bits that represents something; this set is obtained through a vision sensor, then this is transformed into digital format. In formal terms, a digital image is a two-dimensional function $f(x,y)$ where x and y are cartesian coordinates and f is the intensity of some point in the plane. The x, y and f values are finite and discrete amounts; any point in the image is often called a *picture element* or *pixel* [7].

A c*olor space* is a form to represent colors and relationships between them. Human vision is tri-chromatic, this means that the vision has three receptors that react to red (R), green (G) and blue (B) colors. The RGB space color works in the same way; it has three channels to represent each color [8]. In addition, a color digital image is a matrix of pixels where each element has three values (one for red, one for green and one for blue) in the range of [0, 255]. The combination of these three channels defines a color. It is possible to obtain until 16.8 million of colors (255 × 255 × 255).

There are several transformations or processing types that can be applied to digital images such as point to point, oriented to a region, geometrics and arithmetic, among others. In this paper the point to point operations are used because they are the simplest filter to implement in image processing.

2.1 Point to point processing

In the point to point processing type a transformation changes only one pixel in one channel. Let $f(x,y)$ be a pixel value in x and y coordinates of an image, $g(x,y)$ a transformation of $f(x,y)$ and T a function over $f(x,y)$. Then T maps $f(x,y)$ into $g(x,y)$ and the transformation only affects to the pixel in (x,y) coordinates as showed in the Eq.(1).

$$g(x, y) = T[f(x, y)] \quad (1)$$

Examples of this kind of image processing are negative, grayscale, brightening, darkening and thresholding filters.

2.2 Grayscale filter

Grayscale is a function that obtains as a result light intensity instead of colors. Sometimes this filter is called incorrectly black and white [9]. A way to obtain a grayscale image is averaging the three channels as showed in Eq.(2).

$$GrayI(x, y) = \frac{Red(x,y) + Green(x,y) + Blue(x,y)}{3} \quad (2)$$

where $GrayI$ is the image in grayscale.

Pratt [10] suggests to use Eq. (3) due to eye sensibility, i.e., in the human vision there is an order in the degree of sensitivity to the colors: first the green, second the red and third the blue. Therefore, using Eq. (3), green color has more bright than the other ones; this is known as *luminescence* [10].

$$Gray(x, y)=0.3\times Red(x, y)+0.59\times Green(x, y)+0.11\times Blue(x, y) \quad (3)$$

2.3 Negative filter

The negative transformation inverts the intensity levels of an image. This is useful in grayscale images when is needed to find some significant characteristics and the predominant color is white (or black) like x-ray plate [9]. A color image has an intensity range of [0, 255], then Eq. (4) can be used to obtain a negative color.

$$Neg(x, y) = 255 - f(x, y) \quad (4)$$

2.4 Brightening filter

Brightening filter maps the pixel values to higher ones through a function applied to $f(x,y)$ or constants values [19] as showed in Eq. (5).

$$g(x, y) = af(x, y) + b \quad (5)$$

where $a>1$ and $b>0$ are constants.

One way to do this transformation is using the *sine function* in the range of $[0, \frac{\pi}{2}]$ as showed in Eq. (6).

$$Bright(x, y) = \mu \sin(k\, I(x, y)) \quad (6)$$

where k is the angular frequency, μ is the maximum amplitude [18] and $I(x,y)$ is the original image.

Eq. (6) has to be normalized due to the range of values of an image are between [0, 255], then k changes for $\frac{\pi}{2\Lambda}$ and μ for Λ. After of normalization process, the result is the Eq. (7).

$$Bright(x, y) = \Lambda \sin\frac{\pi\, I(x,y)}{2\Lambda} \quad (7)$$

where $\Lambda \in [0,255]$.

2.5 Darkening filter

There are cases where images appear very bright due to light exposure; thus they can be processed to be darker. The *cosine function* can be used for this purpose, due to this function works as the inverse of the *sine function* in the range of $[0, \frac{\pi}{2}]$, it decreases the intensities values, and as a result, the color image is darker than the original one [7]. Eq. (8) shows the general form to apply a darkening filter to an image.

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.

$$Dark(x,y) = \mu\,(1-\cos(k\,I(x,y))) \quad (8)$$

The normalization is the same that the used by the *sine function* so that the Eq. (9) is applied to a digital image.

$$Dark(x,y) = \Lambda\,(1-\cos\left(\frac{\pi\,I(x,y)}{2\Lambda}\right)) \quad (9)$$

where $\Lambda \in [0,255]$.

Figure 1 shows the behavior of the sine function (a) and the cosine function (b).

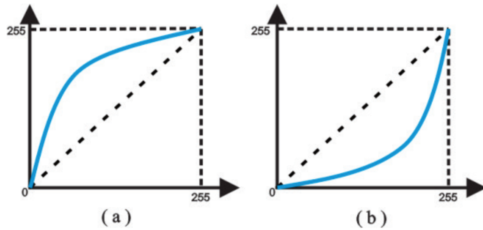

Fig. 1 Graphic for (a) bright image and (b) dark image.

2.6 Contrast filters

Sometimes the application of brightening or darkening does not highlight details of an image. Thus, we can handle two options:

1. To increase the intensity of the lighter tones and decrease the darker ones.
2. To increase the intensity of the darker and decrease the lighter tones.

This type of processing is called *contrast*. The contrast is achieved by means of a threshold from which the higher tones will increase its value and the lower will decrease and vice versa. There is a case called the *High Contrast* filter, where the values above a threshold will be the maximum value and those which are below will be the minimum, obtaining as result an image in two colors: white and black [7]. This type of contrast is also known as *image thresholding*. This transformation can be done applying Eq. (10). Figure 2 shows how the values change by the application of thresholding function [19].

$$Bin(x,y) = \begin{cases} 0 & I(x,y) \le threshold \\ 255 & I(x,y) > threshold \end{cases} \quad (10)$$

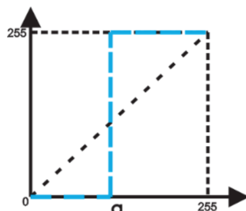

Fig. 2 High contrast function with threshold a.

There are several options in cases where the contrast will improve the tone without going as far as the thresholding [7]. In this work three techniques are used: 1) the *inverse sinusoidal function* (showed in Eq. 11), 2) the *hyperbolic tangent function* (showed in Eq. 12) for increasing the contrast and 3) *sine function* for decrease.

$$InvSin(x,y) = I(x,y) - \lambda \sin\left(\frac{2\pi\,I(x,y)}{\Lambda}\right) \quad (11)$$

$$HyTan(x,y) = \frac{\Lambda}{2}\left[1 - tanh\left[\alpha\left(I(x,y) - \frac{\alpha\Lambda}{2}\right)\right]\right] \quad (12)$$

where $\alpha > 0$.

The functions described above increase the contrast as shows in Figure 3.

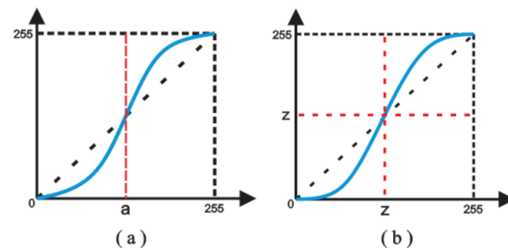

Fig. 3 (a) inverse sinusoidal and (b) hyperbolic tangent contrast.

Figure 3 shows a softer change than thresholding, midrange tones increases higher than the last values, the same way is for the decremented tones [7]. There is a *sine* function for the inverse result almost identical to the *inverse sinusoidal*, an addition operation is used instead of subtraction, in Eq. (13) a plus operator is used and in Eq. (11) a minus operator.

$$Sin(x,y) = I(x,y) + \lambda \sin\left(\frac{2\pi\,I(x,y)}{\Lambda}\right) \quad (13)$$

where $\lambda \in [0,40)$.

Although the functions are similar, the result is not the same as showed in Figure 4. The curve is inverted and the darker values turn to brighter ones and vice versa.

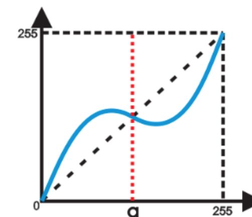

Fig. 4 *Sine function* to decrease contrast.

Figure 5 shows examples of applying the processing techniques described in this section. In (a) the original image, in (b) the grayscale image, in (c) the negative image, in (d) the thresholding, in (e) brightening of an image, in (f) darkening of an image, in (g) the contrast increase and in (h) the contrast decrease of an image.

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.

Fig. 5 Examples of point to point transformations.

3. Parallel computing

Parallel computing is an alternative to solve problems that require large times of processing or handling large amounts of information in "acceptable time" (according to each criterion). In parallel processing, a program is able to create multiple tasks that work together to solve a problem [11]. The main idea is to divide the problem into simple tasks and solve them concurrently, in such a way the total time can be divided between the total tasks (in the best case).

Parallel processing cannot be applied to all problems, in other words, not all the problems can be coded in a parallel form. A parallel program must have some features for a correct and efficient operation; otherwise, it is possible that runtime or operation does not have the expected performance. These features include the following [12]:

- G*ranularity*.- It is defined as the number of basic units and it is classified as:
  - *Coarse-grained*.- Few tasks of more intense computing.
  - *Fine grain*.- A large number of small parts and less intense computing.
- Type of parallel processing:
  - *Explicit*.- The algorithm includes instructions to specify which processes are built and executed in parallel way.
  - *Implicit*.- The compiler has the task of inserting the necessary instructions to run the program on a parallel computer.
- *Synchronization*.- This prevents the overlap of two or more processes.
- *Latency*.- This is the time transition of information from request to receipt.
- *Scalability*.- It is defined as the ability of an algorithm to maintain its efficiency by increasing the number of processors and the size of the problem in the same proportion [13].
- *Acceleration* and *efficiency* are metrics to assess the quality of a parallel implementation.

3.1 CUDA

Since a few years ago, two approaches was established about microprocessors design: the *multicores* processors addressed to keep the executions speed of sequential programs when there is movement between processor cores and *many-cores* processors focused to perform parallel applications [14].

In *many-cores* processors there are graphics cards or GPUs (Graphics Process Unit). From 2001 to 2005 the difference between CPU and GPU was small, since 2006 the performance of GPU increases significantly. In 2009 the peak floating-point calculation throughput was about 10 to 1, this means GPU reached 1 teraflop (1000 gigaflops[1]) and CPU only 100 gigaflops [14]. The difference in the performance is due to the philosophy of design from both processors approaches.

The idea of using CPU and GPUs for intense numeric computing motivates the design of CUDA (Compute Unified Device Architecture), a programming model for execution of an application in CPU and GPU [14].

Figure 6 shows the architecture of CUDA-capable GPU. It is organized in modules called Streaming Multi-processors (SMs), two SMs form a block each one with an independent parallel cache and a Global Memory. The access to the Global Memory space is sequential unlike random access of a CPU. The SMs are composed by Streaming Processors (SPs) that share a logic control and a space instruction cache, the total of SPs depend on GPUs model. The SPs can execute multiple threads per application (even thousands of threads). Meanwhile CPUs only support 2 or 4 threads per core according to a model [14]. For example, the G80 chip has 128 SPs organized in 16 SMs; each SM supports up to 768 threads, in total more than 12,000 threads for this chip.

Parallel programming in CUDA is explicit and fine grain, i.e., it is necessary to design how the task will be divided to be executed in parallel and how the communication between tasks can be done. These tasks should be as simple as possible, i.e. minimum operations that cannot be divided in simpler tasks. In a CUDA program, a kernel function specifies the code to be executed by all threads during a parallel phase. These functions are identified as *host*, *device* and *global* [14]. The first one refers to the section of a program that is

[1] One gigaflop means one thousand millions of floating-point operations

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.

executed only in the CPU, while the second one is executed only in the GPU. "Global" means that the CPU and the GPU can be able to communicate between them.

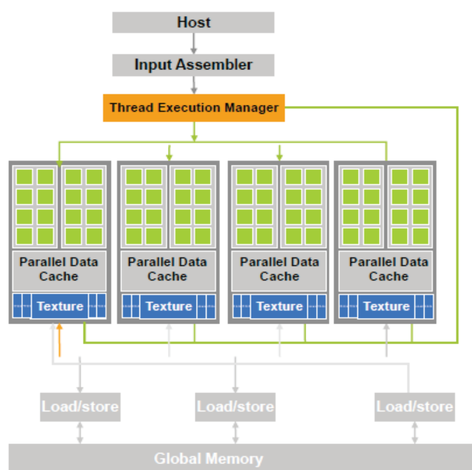

Fig. 6 CUDA diagram architecture.

4. OpenCV

OpenCV (Open Source Computer Vision) is an open source library originally developed by Intel, which provides functions for creating real time applications of computer vision and machine learning. Figure 7 shows the three modules of OpenCV, one for image processing and computer vision, another one for automatic learning and the last one that provides functions for handling image and video and graphic user interface for presentation. This library is written in C and C++ and can be run in environments such as Linux, Windows and Mac OS X. It is possible to obtain optimized codes using the "Integrated Performance Primitives" (IPP) library that has low-level optimized routines used in several algorithms [15].

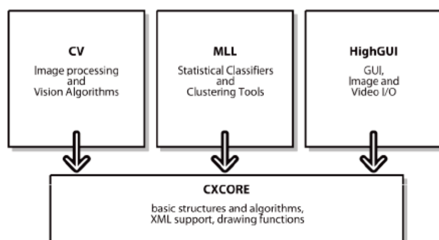

Fig. 7 Components of the OpenCV library [15].

OpenCV consists of over 500 functions of vision. There are functions for digital images processing with filters and masks that can be applied to an image to improve the quality or to find information that is not easily available at a glance. These functions are used in several areas such as industry inspection, medical imaging, security, user interfaces, camera calibration, stereo vision and robotics.

5. Digital image processing using parallel computing

Our approach is divided into two sections: the first one obtains images in grayscale and binary; and the second one transforms the images using the brightening, the darkening and contrast filters.

Algorithm 1 shows the set of steps used for image transformation performed in a GPU. The step 4 can be changed for every proposed filter.

The CPU will create 32×32 blocks, a total of 1024 blocks. Each block executes a number of threads defined as follows: *width*×*height* of the image divided by 1024. Thus, for example, for images of 1024×1024 pixels, each block will have 1024 threads. This is done because the card model does not support more than 1024 threads per block.

Algorithm 1. Image transformation in CUDA.
Input: *I* original image
Output: *I'''* resulting image of the transformation

1. Load an image *I*
2. This image *I* is transformed to a vector *I'* of integer values with the three consecutive channels in each pixel, it means that each pixel in the vector is ordered as follows:
   - $r_1, g_1, b_1, r_2, g_2, b_2, \ldots, r_n, g_n, b_n$

   where *r* is the red one, *g* is the green one and *b* is the blue one and the subindexes are used to indicate the number of pixel.
3. Then, *I'* is stored in the GPU memory
4. After that, the new transformation is applied and the result is stored in a new vector *I''*. One or three channels can be used.
5. Then, the vector *I''* is transferred to RAM memory
6. Finally, the vector *I''* is transformed to an image *I'''* to verify the correct performance of the filters

5.1 Grayscale and thresholding

Eq. (2) is used for grayscale transformation, the average of three channels become in an image with just one channel. Figure 8 shows the general steps for a gray scale transformation for a color image. This is decomposed in three channels (red, green and blue), each GPU core processes a pixel. The three colors are added and divided by 3, and then they are stored in a new array in the same position using only one channel.

Algorithm 2 shows the algorithm to obtain a grayscale image in a CPU. In line 1, the algorithm is performed through the entire image. Eq. (2) is applied to each pixel and the result is stored in a vector called *GI* (Grayscale Image) in the same pixel position. In all algorithms the

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.

index i is one pixel in position i and the indexes $i\times3$, $i\times3+1$ and $i\times3+2$ refer to the red, green and blue channels respectively.

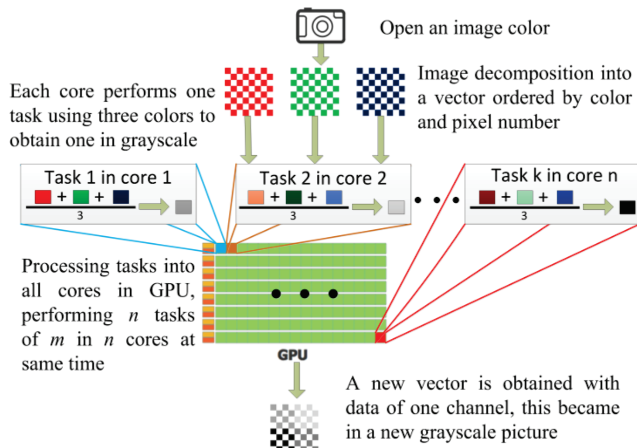

Fig. 8. General diagram for the transformation in grayscale. Each core in GPU processes the average of the three channels of the pixels and stored it in a new image.

Algorithm 2. Grayscale transformation in a CPU.
Input: *I* image vector **Output**: *GI* grayscale image 1. for i = 0 to (*width(I)* × *height(I)*) do 2. *GI[i]* = (*I[i×3]* + *I[i×3+1]* + *I[i×3+2]*) /3 3. endfor

Algorithm 3 shows the algorithm for a GPU using CUDA. In this Algorithm can be noticed the difference between the CPU and GPU version. First, there is not a loop strictly, in this case an identifier is assigned to a variable *i;* this is the combination of thread number in matrix threads and the block number as showed in Figure 9.

Algorithm 3. Grayscale transformation using CUDA in a GPU.
Input: *I* image vector **Output**: *GSC* grayscale image 1. For each GPU task *i* = *blockIdx.x* × *(blockDim.x* × *blockDim.y)* + *blockDim.x* × *threadIdx.y* + *threadIdx.x*; 2. *GSC[i]* = (*I[i×3]* + *I[i×3+1]* + *I [i×3+2]*) /3 3. endfor

Algorithm 4 shows the algorithm to obtain a thresholding image using CUDA. First, given an input threshold value, which will be the reference for applying the Eq. (10), every pixel value above this threshold will be set up to 255 and lower values will be set up to 0. Line 2 can be replaced by the comparison of the expression (*I[i×3]* + *I[i×3+1]* + *I[i×3+2]*) /3 to avoid the compute of the gray scale before the thresholding.

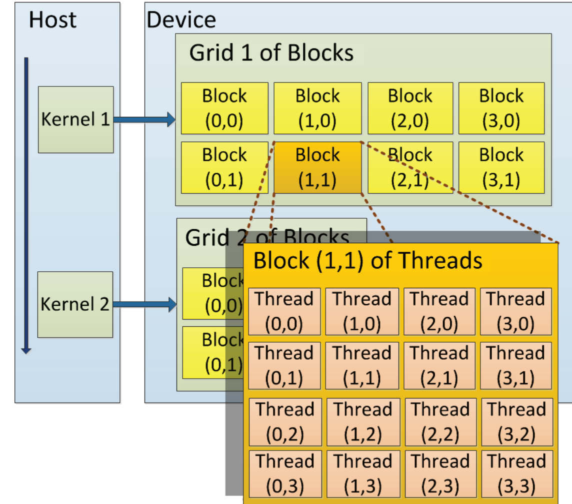

Fig. 9. Blocks and threads in CUDA. It can be seen as a matrix formed of a small matrix, then the location of a single thread in the complete matrix is a combination of a block number and a thread number in the block.

Algorithm 4. Thresholding image transformation using CUDA in a GPU.
Input: *GSC* the grayscale image and a *threshold* **Output**: *IT* the binary image 1. For each GPU task *i* = *blockIdx.x* × *(blockDim.x* × *blockDim.y)*+ *blockDim.x* × *threadIdx.y* + *threadIdx.x*; 2. if *GSC[i]* < *threshold* 3. *IT [i] = 0;* 4. else 5. *IT[i] = 255;* 6. endfor

5.2 Color image processing

Two cases are handled for color image processing: filters using OpenCV functions and equivalent functions developed in C. In these cases the transformation function is applied to the three channels and is stored in the final vector.

First for the brightening filter, using Eq. (7), the lambda value is established as input parameter. Algorithm 5 shows a section of the algorithm section where the function is applied, the k value is defined as $\pi/(2\times lamda)$ due to normalization of the sine function. Lines 3, 4 and 5 of Algorithm 5 shows that the Eq. (7) is applied to each channel of the image and all channels are stored in a new image *BI*.

A similar process is carried on darkening filter, where the *cosine function* corresponding to Eq. (8). Also the value of *lambda* is established as parameter and k value is defined as a part of the normalization function.

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.

Algorithm 5. Brightening image transformation using CUDA in a GPU.

Input: *I* original image vector and *lamda*
Output: *BI* the brightening image

1. k = π/(2×*lamda*)
2. For each GPU task *i = blockIdx.x × (blockDim.x × blockDim.y)+ blockDim.x × threadIdx.y + threadIdx.x*
3. *BI[i×3] = lambda (sin(k × I[i×3]))*
4. *BI[i×3+1] = lamda(sin(k × I[i×3+1]))*
5. *BI[i×3+2] = lamda(sin(k × I[i×3+2]))*
6. endfor

Algorithm 6 shows the algorithm using the Eq. (8) in parallel, as the case of the *sine function*, all channels are used for this filter and these are stored in the new vector *DI*.

Algorithm 6. Darkening image transformation using CUDA in a GPU.

Input: *I* original image vector and *lamda*
Output: *DI* the darkening image

1. *k* = π/(2×*lamda*)
2. For each GPU task *i = blockIdx.x × (blockDim.x × blockDim.y) + blockDim.x × threadIdx.y + threadIdx.x*
3. *DI[i×3] = lambda (1-cos(k × I[i×3*]))
4. *DI[i×3+1] = lamda*(*1-cos* (*k × I[i×3+1*]))
5. *DI[i×3+2] = lamda*(*1-cos* (*k × I[i×3+2*]))
6. endfor

Algorithm 7 shows that the function filter were used on maximum range of pixel values, so that Λ from Eq.(10) was set up to 255, then *k* was equal to $\frac{2\pi}{255}$. Another contrast function is the *hyperbolic tangent* corresponding to Eq. (12). This function and *inverse sinusoidal function* increases the image contrast. The algorithm for *hyperbolic tangent* is showed in Algorithm 8 also set up Λ to 255 and *k* to $\frac{2}{255}$.

Algorithm 7. Inverse sinusoidal contrast image transformation using CUDA in a GPU.

Input: *I* original image vector and *lamda*
Output: *ISCI* the inverse sinusoidal contrast image

1. k = 2π/255
2. For each GPU task *i = blockIdx.x × (blockDim.x × blockDim.y) + blockDim.x × threadIdx.y + threadIdx.x*
3. *ISCI [i×3] = I[i×3] – lambda(sin(k× I[i×3]))*
4. *ISCI[i×3+1]= I[i×3+1]–lambda(sin(k× I[i×3+1]))*
5. *ISCI [i×3+2]= I[i×3+2]–lambda(sin(k× I[i×3+2]))*
6. endfor

Algorithm 8. Hyperbolic tangent contrast image transformation using CUDA in a GPU.

Input: *I* original image vector
Output: *THCI* hyperbolic tangent contrast

1. *k = 255/2*
2. For each GPU task *i = blockIdx.x × (blockDim.x × blockDim.y) + blockDim.x × threadIdx.y + threadIdx.x*;
3. *THCI[i×3] = k(1+tanh(I[i×3] - k*))
4. *THCI[i×3+1] = k(1+tanh(I[i×3+1] - k*))
5. *THCI[i×3+2] = k(1+tanh(I[i×3+2] - k*))
6. endfor

The last filter in color is the *sine* contrast function. This function decreases the contrast in an image. Algorithm 9 shows the algorithm for the *sine function* to adjust the contrast. In this filter an addition operation was performed and the other operation is a subtraction. Again a *lambda* value is designed as parameter and Λ was set up to 255.

Algorithm 9. *Sine* contrast image transformation using CUDA in a GPU.

Input: *I* original image vector and *lamda*
Output: *SCI* the sine contrast image

1. *k = 2π/255*
2. For each GPU task *i = blockIdx.x × (blockDim.x × blockDim.y) + blockDim.x × threadIdx.y + threadIdx.x*;
3. *SCI [i×3] = I[i×3i] + lambda(sin(k× I[i×3]))*
4. *SCI[i×3+1] = I[i×3+1] + lambda (sin (k× I[i×3+1]))*
5. *SCI[i×3+2] = I[i×3+2] + lambda (sin (k× I[i×3+2]))*
6. endfor

6. Experimental results

Eight modules were implemented in CUDA which apply the transformation with the restriction that the image to be processed must be stored in GPU memory previously to process it. It is important to notice that we are only considering the execution time, not the time used to transfer data between RAM memory and the GPU memory. Our approach is tested using images of different sizes, this in order to verify if the execution times are kept in proportion to apply the transformation to an image of larger sizes. The dimensions of the images used for this experimentation are of 256×256, 512×512, 1024×1024, 1800×1400 and 4000×3000 pixels.

The computer used for experiments was a desktop PC with AMD Phenom II Quad-core to 3.2 GHz, 12 GB of RAM, operating system 64-bit Linux Fedora 14 and OpenCV version 2.3. For CUDA processing a GeForce 430 GT video card with 96 cores and 1 GB of RAM DDR3 is used.

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.

The transformation is applied on 10 different images for each resolution; first with an OpenCV version and later using the CUDA kernels. There are 5 modules that are not developed with OpenCV. The reason is that darkening, brightening and contrast filters have not equivalent in this library, and therefore they are developed in C language.

The results are presented in Tables 1-8 for the grayscale, brightening, darkening, image thresholding and contrast with the *inverse sinusoidal*, *hyperbolic tangent* and *sine* functions, respectively. Tables 1, 2 and 5 show execution times for CUDA modules and OpenCV functions. Tables 3, 4, 6, 7 and 8 show execution times for CUDA modules and C language module for CPU.

We can notice in Tables 1 and 5 with the grayscale transformation and image threshold that CUDA obtains better results than OpenCV for all cases. However, in Table 2 the negative image for small resolutions (256×256 and 512×512) CUDA module is slower than OpenCV function and when the resolution is increased CUDA became better than OpenCV.

Using a module programed in C language for Tables 3, 4, 6, 7 and 8 in all cases CUDA modules obtain better times in comparison with the version programed in C language. In these cases the difference in times is larger than OpenCV, for example 2040ms of CUDA versus 34ms of C language module for *inverse sinusoidal* contrast transformation in the higher resolution.

Table 1. Grayscale transformation execution times.

Resolution	OpenCV	CUDA
256 × 256	0.178176	0.1022784
512 × 512	0.6816352	0.3869152
1024 × 1024	2.8476448	1.5157472
1800 × 1400	6.7490382	3.64163556
4000 × 3000	31.591331	17.2692191

Table 2. Negative transformation execution times.

Resolution	OpenCV	CUDA
256 × 256	0.1439168	0.1556832
512 × 512	0.5186912	0.5487712
1024 × 1024	2.3863488	2.10624
1800 × 1400	5.5122464	5.0195488
4000 × 3000	25.7547744	23.7847103

Table 3. Brightening image transformation execution times.

Resolution	CPU in C	CUDA
256 × 256	8.847117	0.19903
512 × 512	36.12278	0.761405
1024 × 1024	142.6773	2.995606
1800 × 1400	342.4271	7.20223
4000 × 3000	1610.854	33.97197

Table 4. Darkening image transformation execution times.

Resolution	CPU in C	CUDA
256 × 256	9.5512192	0.2016992
512 × 512	38.731661	0.7718848
1024 × 1024	151.256079	3.0336608
1800 × 1400	336.17552	7.2744512
4000 × 3000	1719.52881	34.4515743

Table 5. Threshold image transformation execution times.

Resolution	OpenCV	CUDA
256 × 256	0.2222272	0.146016
512 × 512	0.868816	0.555968
1024 × 1024	3.478688	2.1979296
1800 × 1400	8.2935616	5.2435488
4000 × 3000	42.4278015	24.6556834

Table 6. *Inverse sinusoidal* contrast transformation execution times.

Resolution	CPU in C	CUDA
256 × 256	11.26008	0.203088
512 × 512	46.3111264	0.776816
1024 × 1024	184.731943	3.057152
1800 × 1400	448.714719	7.3071904
4000 × 3000	2040.16223	34.6397216

Table 7. *Hyperbolic tangent* contrast transformation execution times.

Resolution	CPU in C	CUDA
256 × 256	5.681072	0.1872896
512 × 512	22.7657601	0.7213312
1024 × 1024	91.6525903	2.823536
1800 × 1400	219.788937	6.8108832
4000 × 3000	1054.94413	32.2680701

Table 8. *Sine* contrast transformation execution times.

Resolution	CPU in C	CUDA
256 × 256	12.5076096	0.2077856
512 × 512	51.3341637	0.7901344
1024 × 1024	205.06483	3.1164224
1800 × 1400	500.958685	7.4421792
4000 × 3000	2273.02695	35.2699835

In Figures 10, 11, and 14 we present a graphic comparison for the CUDA and OpenCV version. For module in C the graphics are showed in Figure 12, 13, 15, 16 and 17.

In Figure 11 both curves are very close together and do not separate in the most points, in resolution 256×256 and 512×512 the curve of OpenCV is under of CUDA for remaining resolutions CUDA curve is slightly above.

In Figures 10 and 14 the behavior is similar in both curves and the difference between CUDA and OpenCV is not large in comparison with the Figures 12, 13, 15, 16 and 17 where the gain of CUDA is larger.

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.

Fig. 10. Comparative graphic of CUDA vs. OpenCV for grayscale transformation.

Fig. 11. Comparative graphic of CUDA vs. OpenCV for negative image.

Fig. 12. Comparative graphic of CUDA vs. module in C for brightening image.

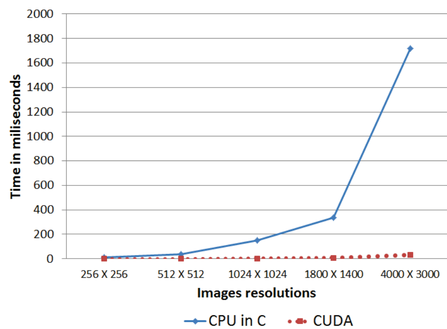

Fig. 13. Comparative graphic of CUDA vs. module in C for darkening image transformation.

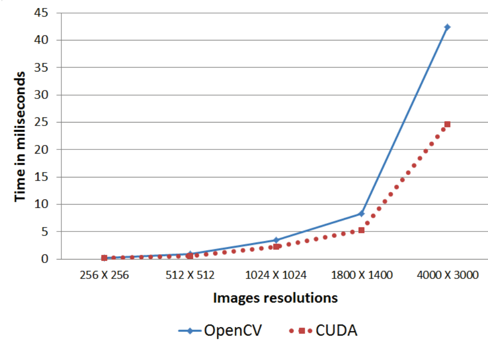

Fig. 14. Comparative graphic of CUDA vs. OpenCV for threshold image.

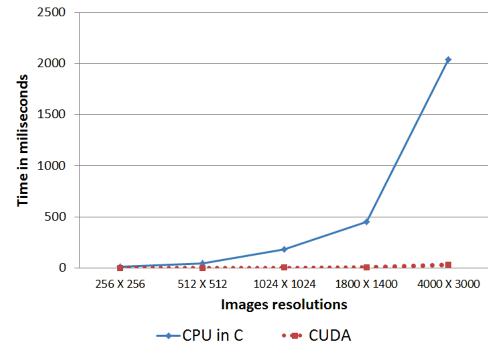

Fig. 15. Comparative graphic of CUDA vs. module in C for *inverse sinusoidal* contrast function.

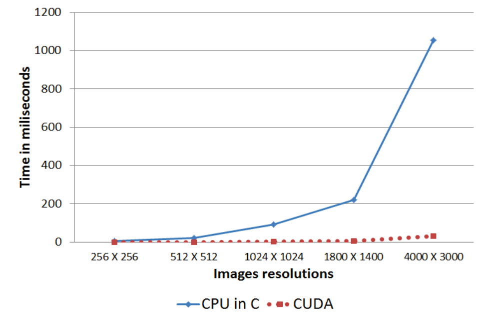

Fig. 16. Comparative graphic of CUDA vs. module in C for *hyperbolic tangent* contrast function.

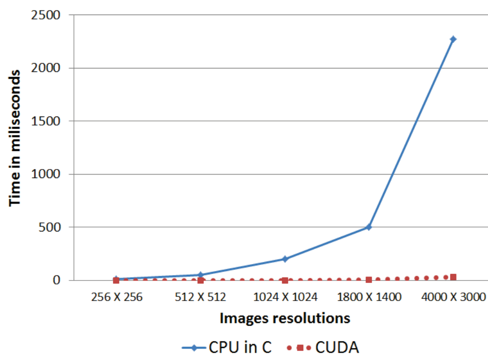

Fig. 17. Comparative graphic of CUDA vs. module in C for *sine* contrast function

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.

7. Conclusions

We have presented an approach for digital image processing using parallel computing. For this purpose, eight modules were implemented using CUDA for running in a GPU. Due to some filters are not implemented in OpenCV, two filters were developed using C without optimization; and this was reflected in the large difference in execution times. When the comparison was done with OpenCV functions, the gain proportion was less than the C module. The fact of this large difference allows us to conclude two things: 1) CUDA increases the performance execution of some tasks that need millions of operations and 2) it is possible obtain better times if the code is optimized, for this purpose, the gain in parallel may be not very significant. However, from the results we can conclude that CUDA obtained better results in most cases than OpenCV. Future works include testing our approach to compute Haar-like features for object classification.

References

[1] J. Fung and S.Mann, "Using multiple graphics cards as a general purpose parallel computer : Applications to computer vision," in Proceedings of the 17th International Conference on Pattern Recognition (ICPR2004) 2004, vol. 1, pp. 805-808.
[2] E. G. Farrugia J.-P., Horain P. and Y. Alusse, "Gpucv: A framework for image processing acceleration with graphics processors," In 2006 IEEE International Conference on Multimedia and Expo, 2006, pp. 585–588.
[3] J. Fung and S. Mann, "Using graphics devices in reverse: Gpu-based image processing and computer vision," in 2008 IEEE International Conference on Multimedia and Expo. IEEE, June 2008, pp. 9-12.
[4] Z. Yang, Y. Zhu, and Y. Pu, "Parallel image processing based on CUDA," In Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 03 (CSSE '08) , 2008, Vol. 3, pp. 198-201.
[5] A. K. Jain, Digital Image Processing. Prentice Hall, 1989.
[6] R. Crane, A simplified approach to image processing: classical and modern techniques. Prentice Hall, 1997.
[7] R. C. Gonzalez and R. E. Woods, Digital Image Processing, 2nd edition. Prentice-Hall, Inc, 2002.
[8] S. Montabone, Beginning Digital Image Processing: Using Free Tools for Photographers. CRC Press, 2010.
[9] A. Bovik, The essential guide to image processing. San Diego, California: Academic Press, 2009.
[10] W. K. Pratt, Fundamentals of Digital Image Processing. New York: John Wiley & Sons, 1991.
[11] P. Pacheco, An Introduction to Parallel Programming. Morgan Kaufmann, 2011.
[12] A. G. López, J. Delgado, and S. Castañeda, "Metodologías de paralelización en la supercomputadora cicese2000," Departamento de Cómputo Dirección de Telemática Centro de Investigación Científica y de Educación Superior de Ensenada, Tech. Rep., February 2000.
[13] R. T. Rasúa, "Algoritmos paralelos para la solución de problemas de optimización discretos aplicados a la decodificación de señales," Ph.D. dissertation, Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia, Valencia, España, 2009.
[14] D. Kirk and W. Hwu, Programming Massively Parallel Processors. A Hands-on Approach. Morgan Kaufmann Publishers, January 2010.
[15] G. Bradski and A. Kaehler, Learning OpenCV, ser. Nutshell Handbook. USA: O´ Reilly Media, Inc., 2008.
[16] Shams, Ramtin and Barnes, Nick, "Speeding up Mutual Information Computation Using NVIDIA CUDA Hardware," In Proceedings of the 9th Biennial Conference of the Australian Pattern Recognition Society on Digital Image Computing Techniques and Applications (DICTA '07), 2007, pp. 555-560.
[17] Yuliya Tarabalka and Trym Vegard Haavardsholm, Ingebjørg Kåsen and Torbjørn Skauli, "Real-time anomaly detection in hyperspectral images using multivariate normal mixture models and GPU processing", Journal of Real-Time Image Processing, 2009, Vol. 4, No.3, pp. 287-300.
[18] W. Burger and M. J. Burge, Principles of digital image processing: core algorithms. Springer, 2009.
[19] R. Szeliski, Computer Vision: Algorithms and Applications. Springer, 2011.

Eric Olmedo is currently a student of the Master of Systems Engineering and Intelligent Computing at the Universidad Politéctica de Puebla (UPP). He holds a BEng (2009) degree in Computer Systems from Benemérita Universidad Autonoma de Puebla (BUAP). His current interest areas include machine learning and computer vision.

Jorge de la Calleja is an associate professor of computer science at the Universidad Politéctica de Puebla (UPP). He holds MSc (2003) and PhD (2008) degrees in Computer Science from the National Institute of Astrophysics, Optics and Electronics (INAOE) and a BEng (2001) degree in Computer Systems from Benemérita Universidad Autonoma de Puebla (BUAP). His current research areas include machine learning and computer vision. Since January 2012, Dr. De la Calleja has been awared as national researcher from the National Council for Science and Technology (CONACyT).

Antonio Benitez Ruiz is an associate professor of computer science at the Universidad Politéctica de Puebla (UPP). He holds MSc (1997) and PhD (2005) degrees in Computer Science from Universidad de las Américas Puebla (UDLAP) and a BEng (1990) degree in Computer Systems from Benemérita Universidad Autonoma de Puebla (BUAP). He has participated in projects related to the development of robotics and description of virtual environments. His current research areas include computer perception and virtual reality. Dr. Benitez coordinates the graduate education department at the UPPuebla.

María Auxilio Medina Nieto is an associate professor of computer science at Universidad Politécnica de Puebla (UPP). She holds MSc (2001) and PhD (2008) degrees in Computer Science from Universidad de las Américas Puebla (UDLAP) and a BEng (1999) degree in Computer Systems from Benemérita Universidad Autónoma de Puebla (BUAP). She has participated in projects related to the development of software agents and their applicationsto digital libraries. Currently, her research topics are information retrieval, knowledge representation based on ontologies, and information and communications technologies. Dr. Medina coordinates ICT's academic community at the UPPuebla.

Copyright (c) 2012 International Journal of Computer Science Issues. All Rights Reserved.