Parallelization of a method for detecting non-stationary photometric perturbations in projection screens with CUDA

Antonio Díaz-Tula
Departamento de Ciencia de la Computación
Universidad de Oriente
Ave. Patricio Lumumba, 9500
Santiago de Cuba, Cuba
diaztula1@gmail.com

Miguel Castañeda-Garay
Departamento de Ciencia de la Computación
Universidad de Oriente
Ave. Patricio Lumumba, 9500
Santiago de Cuba, Cuba
mcgaray_cu@yahoo.es

Óscar Belmonte-Fernández
Departamento de Ingeniería y Cienca de los Computadores
Universitat Jaume I, Spain
Oscar.Belmonte@lsi.uji.es

ABSTRACT

The human-computer interaction using large projection screens is gaining more space nowadays. For these screens several computer vision techniques have been developed that allow the user to interact with the system through the projected images using laser pointers, special pens and the hands. On this work is presented the parallelization of a method for the real-time detection of non-stationary photometric perturbations in projection screens using the Computed Unified Device Architecture, in order to overcome the elevated running time of the serialized implementation on CPU. A comparison of the results is presented to establish the acceleration of the parallel algorithm against its original version on CPU.

Keywords

parallelization, photometric perturbation, projection screens, CUDA.

1. INTRODUCTION

High definition projection screens are gaining more space each day. Such screens are boosting the presence of multiple spectators, detailed model visualization, immersion sense and the creation of a natural environment of interactive collaboration between multiple users.

As a consequence, several computer vision techniques are being developed that allow users to interact with the system through projected images using laser pointers [Kirs98a], special pens [LaRo03a] and the hands [Koik01].

In [Mig09a] a local method for the real-time detection of non-stationary photometric perturbations in projected images was presented from modifications performed to the global method presented in [Jay04a] for the detection and removal of shadows in projected images.

The new method is based on computing the differences between the images of a projector frame buffer and the corresponding projected image captured by a camera. To carry out this comparison, a previous process of geometric and photometric calibration between the projector, the camera and the screen is needed.

To test this method a system prototype that uses a camera/projector pair was implemented, and it proved to be very robust when facing spatial variations of the projector's light intensity over the projection surface and the incidence on this surface of external locally-stationary factors.

But in the experimentation only about ten images per second were processed using a Core 2 Duo (2.66 GHz) processor, a NVIDIA GeForce 8400 GS GPU and a Logitech 9000 Pro Webcam, with a latency time of 95 milliseconds. This phenomenon cause visible differences of inaccuracy when detecting perturbations whose positions move across the screen [Mig09a].

Two main reasons were identified: the serial execution on CPU, and the latency time in the VRAM- to-RAM transfer of the projector frame buffer.

The method proposed in [Mig09a] is parallelizable as the information processing in several parts of the computation follows the Single Instruction Multiple Data (SIMD) model [Flynn72a].

The latency time is introduced during the copy of the projector frame buffer from VRAM to conventional RAM for the estimation process.

The Computed Unified Device Architecture (CUDA) gives the possibility to exploit the enormous parallel computing capabilities of the NVIDIA's Graphics Processing Unit (GPU), when applied to general purpose problems, as long as those problems are parallelizable by the SIMD model.

CUDA gives a subset of the C language to write kernels that run in parallel on GPU as a hierarchy of threads groups, with very low control and schedule overhead, fast barrier synchronization and shared memory usage [CUD09].

This makes CUDA a suitable technology to improve the running time of the method proposed in [Mig09a] ,because of two main reasons: it provides a powerful SIMD programming environment, and its global memory is located at VRAM as well as the projector frame buffer, that would decrease the latency time.

In this work is presented the **parallelization of the local method presented in** [Mig09a] **for the real-time detection of non-stationary photometric perturbations in projection screens using CUDA.**

The remainder of the content is structured as follows: section 2 gives a brief summary of the local method presented in [Mig09a] for the real-time detection of non-stationary photometric perturbations in projection screens; section 3 covers the parallelization of the previously mentioned method and finally, section 4 shows the results.

2. SUMMARY ON THE LOCAL METHOD FOR DETECTING PHOTOMETRIC PERTURBATIONS

The local method presented in [Mig09a] for the real-time detection of non-stationary photometric perturbations in projection screens is based on the comparison of two images: one of them represents the real image captured by the camera of the projected image (the camera image), and the other one is the image that "should" be captured by the camera (the estimated image).

Roughly speaking, if there were no lights or any other phenomena interfering with the projection process, the camera image and the estimated image should be very similar. Otherwise, these images must present differences in the regions where such perturbations are influenced.

To make a correspondence between the coordinates of the camera image, the projection surface image and projector frame buffer image, a geometric calibration process is needed, thus obtaining transference functions for the coordinate systems of such images. For the geometric calibration, the planar homography method described in [Suk01a] was used.

In the method presented in [Mig09a] the projector frame buffer resolution is higher than the camera resolution, which implies that a set of closely located pixels at the projector frame buffer are captured by the camera as a single pixel.

For this reason, the projector frame buffer is conceived as a matrix of rectangular regions, whose intersection is null and whose joint is equal to the buffer.. This partition is performed in order to obtain a matrix with a row and column number equal to the camera resolution, which in turn is the given resolution to the estimated image.

For each region of the projector frame buffer there is a single corresponding point in the estimated image. In turn, for each point of the estimated image there is a single corresponding point in the camera image, but for a given point in the camera image there may exist 0, 1 or more points in the estimated image (see Fig. 1).

To homogenize the color range of both the camera image and the estimated image a process of photometric calibration is needed.

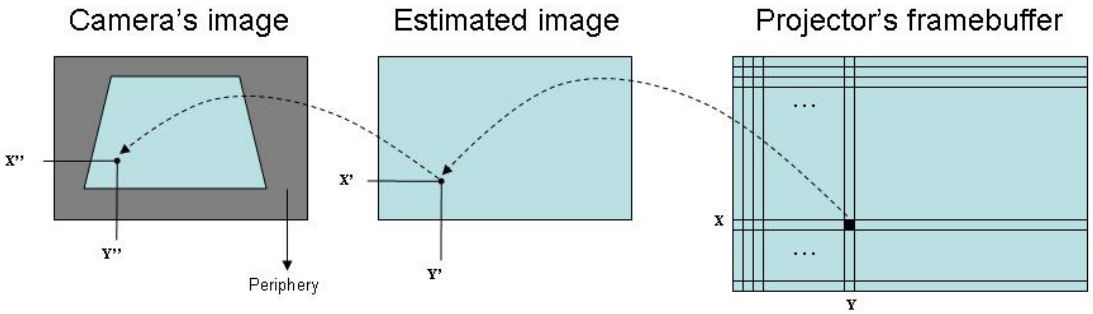

Figure 1 Correspondence between a region in the projector frame buffer, a point in the estimated image and a point in the camera image.

The photometric calibration process is established to make a correspondence between the color range of the camera image, the projection surface image and the projector frame buffer image.

This calibration is needed because the projector frame buffer image and the camera-captured image have different photometric ranges due to several internal and external factors such as: differences in the color spaces and brightness level between the projector and the camera; the location of the projector with respect to the camera, which cause brightness variations according to its position; the influence of the environmental light over the screen; the camera's internal features, as well as the adjustment of its intrinsic and extrinsic parameters; the existence of spots or irregularities on the projection surface, among others [Mig09a].

To perform this calibration, a model that produces transference functions between color spaces was used; those functions allow estimating the image that the camera should grab from the image of the frame buffer. For each region in the frame buffer such functions are obtained, one for each RBG color component. It differs from the method presented in [Jay04a] where the transference functions are obtained for the entire screen (global) and not for each region.

The color transference functions are previously evaluated for every possible value of each color component for each region of the frame buffer, and the results are stored in three-dimensional tables in order to avoid revaluating these functions each time during the estimation process.

Thus, the photometric calibration of each region in the frame buffer is given by three three-dimensional tables, one for each color component:

byte [camHeight][camWidth][64] redTable

byte [camHeight][camWidth][64] greenTable

byte [camHeight][camWidth][64] blueTable

To obtain the color components for a pixel in the estimated image, the average of the estimated color components for all the pixels in the frame buffer that belong to the corresponding region for the estimated pixel is computed.

Given a pixel in the frame buffer with coordinates *(x, y)* and color components *red*, *green* and *blue*, its corresponding coordinates in the estimated image are *(regX(x), regY(y))* and the estimated values are:

estRed = redTable [regX(x)] [regY(y)][red /4]

estGreen=greenTable[regX(x)][regY(y)][green/4]

estBlue=blueTable [regX(x)] [regY(y)][blue /4]

The function *regX(x)* gives the row of the estimated image that correspond to the pixel with row *x* in the frame buffer; the function *regY(y)* gives the column of the estimated image that correspond to the pixel with column *y* in the frame buffer.

The value of each color component is divided by 4 in order to reduce the amount of memory needed to store the tables of each region of the frame buffer.

The sequence of steps to detect the photometric perturbations is stated as follows:

1. Obtain the image of the projector frame buffer (in RGB format) in the variable *frameBuffer*.
2. For each pixel in this image with coordinates *(x, y)* obtain the coordinates *(x', y')* where *x'=regX(x)* y *y'=regY(y)*, then for the color components *red*, *green* and *blue* of *frameBuffer[x][y]* compute the estimated values by querying the entry *colorTable[x'][y'][color/4]*. Given that for several pixels in *frameBuffer* the same coordinates *(x', y')* will be obtained (for all that belong to the same region), the average must be computed for each color component.
3. Compare each pixel in the estimated image with its corresponding pixel in the camera image for each color component; if the difference between two values is greater than a given threshold, then a possible photometric perturbation may exist.

3. PARALLELIZATION OF THE LOCAL METHOD FOR DETECTING PHOTOMETRIC PERTURBATIONS

The first step is to copy the projector frame buffer to be processed with CUDA. One objective is to avoid the transference of this buffer to the RAM.

Reading the projector frame buffer for its processing with CUDA

It can be consulted in various CUDA SDK examples, that it is possible to write parallel algorithms to post-process the frame buffer of a window through the CUDA's interoperability with OpenGL. As described in [CUD09], OpenGL buffer objects can be mapped with CUDA to be accessed from the kernels.

In our problem we need to read the entire projector frame buffer, that is, the "Desktop". Whereas OpenGL does not provide any functions to create rendering contexts, this must be done using the underlying operating system's API (hence, loosing portability that way). The method was implemented for the Microsoft Windows XP operating system.

To carry out the reading of the projector frame buffer we follow these steps:

- Create a top-level, layered window that covers the entire Windows's desktop; this window will be invisible.
- Create a hardware-accelerated OpenGL rendering context associated with this window and make it current.
- Create a Pixel Buffer Object (PBO) with enough memory to store the entire projector frame buffer (this is related to the screen size and color depth). The PBO memory is usually allocated in VRAM and controlled by OpenGL.
- Use the OpenGL's function *glBindBuffer* to link our PBO to the reading operations over the frame buffer.
- Use the OpenGL's function *glReadPixels* to perform the copy of the frame buffer to the PBO.

Then we just need to use the CUDA's API function *cudaGLMapBufferObject* to map the PBO and its content is ready to be accesed from CUDA's threads.

Parallelization of the estimation and compare algorithms

3.2.1 Estimation algorithm in CPU.

The estimation algorithm takes as input the projector frame buffer, the color tables of each region in this buffer, the dimensions of the estimated image and the projector frame buffer, and returns the estimated image, that is, the image that the camera should capture at exactly that moment.

The estimation algorithm in CPU is as follows:

Remark: All the coordinates are row major order.

Input: Projector frame buffer, dimensions of the frame buffer and the estimated image, color tables of all the regions in the projector frame buffer.

Output: Estimated image.

Step 1. Initialize the estimated image with 0.

Step 2. For each pixel in *frameBuffer* with coordinates *(i, j)* do steps 3 to 7:

Step 3. Compute the region to which belongs the pixel *(i, j)* in the estimated image:

*i'=i*cammeraHeight /frameBufferHeight*

*j'=j*cammeraWidth /frameBufferWidth*

Step 4. Obtain the pixel's color components:

red = frameBuffer[i][j] & 0x000000FF

green=(frameBuffer[i][j] & 0x0000FF00)>>8

blue =(frameBuffer[i][j] & 0x00FF0000)>>16

Step 5. Compute the estimated color components:

estRed = redTable [i'][j'][red /4]

estGreen = greenTable[i'][j'][green /4]

estBlue = blueTable [i'][j'][blue /4]

Step 6. Add the estimated values to the corresponding pixel in the estimated image:

redEstimImg [i'][j'] += red

greenEstimImg [i'][j'] += green

blueEstimImg [i'][j'] += blue

Step 7. Increase the count of pixels from *frameBuffer* that belong to the computed region:

regionCount[i'][j']++

Step 8. For each pixel in the estimated image with coordinates *(i',j')* divide by the pixel count for each color component:

redEstim [i'][j'] /= regionCount[i'][j']

greenEstim[i'][j'] /= regionCount[i'][j']

blueEstim [i'][j'] /= regionCount[i'][j']

Algorithm 1. Estimation algorithm in CPU.

3.2.2 Decomposition of the algorithm.

As can be easily seen, the same steps repeat over different data, this allows a data parallelism over a shared address space [Gra03a].

We used the output data decomposition technique, where each output element can be independently computed as a function of the input. The value of each pixel in the estimated image (output) depends only on the corresponding frame buffer's region and its color tables (input).

This partition leads to the definition of a task as computing a pixel in the estimated image. The number of tasks is equal to the product of the estimated image's width and height. For example, 76800 tasks are obtained from a resolution of 320x240. This decomposition can be classified as fine texture according to its granularity. Figure 2 gives us a graphical scheme of the parallel algorithm.

Still an issue must be analyzed: all the regions of the frame buffer do not have the same size, i.e., the rows and columns number may be different for two or more regions. This may influence in the performance of the algorithm when the threads of the same warp diverge in their execution paths.

But there are several reasons in favor of this approach:

(i) There is uniformity in the sense that each thread computes exactly a pixel of the estimated image, thus being unnecessary any communication and synchronization mechanisms between threads.

(ii) Each thread will write on a single pixel in the estimated image, so there is no need to use atomic instructions (available only for computing capabilities 1.1 or above).

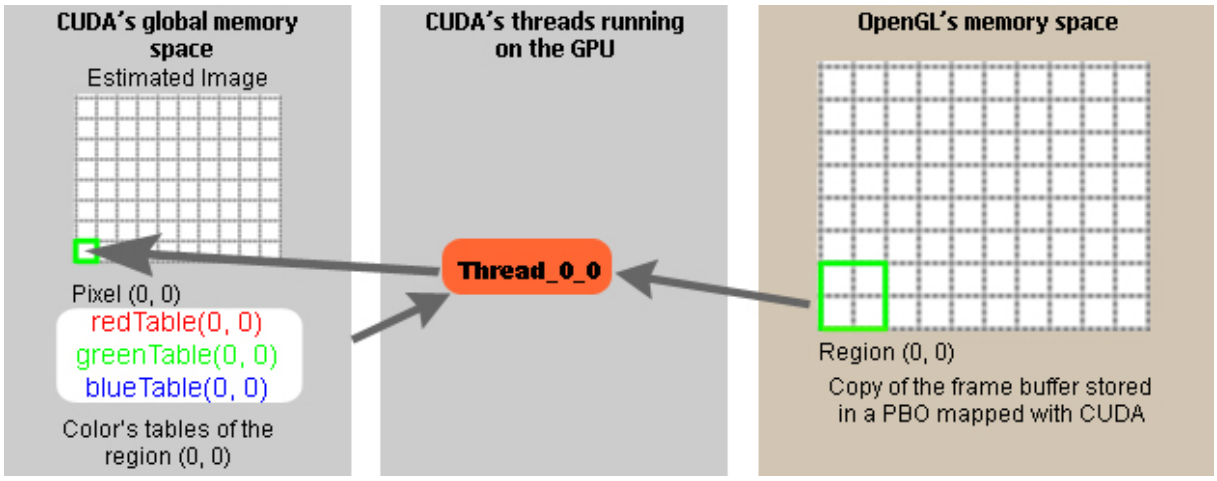

Figure 2. Scheme of the estimation algorithm if GPU.

3.2.3 Determining the execution configuration.

When data decomposition is applied to a problem, generally the task mapping is static [Gra03a], where the tasks are distributed among the available processors before the algorithm execution.

Our problem adjusts to a Blocks Distribution Scheme [Gra03a], where the resulting matrix (the estimated image) is divided in areas of k_1 columns and k_2 rows, so that each thread must compute all the estimated pixels of a given area.

CUDA's threads are organized in a hierarchy of one-dimensional, bi-dimensional or three-dimensional thread blocks; those in turn are organized in a one-dimensional or bi-dimensional grid of blocks.

The GPU have a number of multiprocessors, and each multiprocessor have eight streaming processors. When the CPU launches the execution of a grid, its blocks are enumerated and distributed to the available multiprocessors. When a multiprocessor finishes the execution of a block, it gets assigned another non-executed block. This execution model is scalable, so we can define any number of threads without worrying about the number of multiprocessors.

When the number of thread blocks increases to a large amount, GPUs with a few multiprocessors will not be favored, as plenty of time will be used in distributing the non-executed blocks to the multiprocessors as they become available, and this time may be significant against the running time of each block.

For that reason the value of k_1 and k_2 must be obtained so each thread computes an area of k_1 x k_2 pixels of the estimated image. These values can be adjusted depending on the number of available multiprocessors, thus giving more scalability to the implementation.

The thread blocks are set to be bi-dimensional and have 16x16 = 256 threads, a value that is recommended in [CUD09] to obtain a good performance. The blocks grid is also bi-dimensional and it size will be:

$grid.x = ceil(cameraWidth / 16 * k_1)$

$grid.y = ceil(cameraHeight / 16 * k_2)$

For instance, if we want a maximum number of 512 blocks per multiprocessor we proceed as follows:

$$ceil(\frac{cameraWidth}{16 * k_1}) \geq \frac{cameraWidth}{16 * k_1}$$

$$ceil(\frac{cameraHeight}{16 * k_2}) \geq \frac{cameraHeight}{16 * k_2}$$

$$\frac{cameraWidth * cameraHeight}{256 * k_1 * k_2 * mpCount} \leq 512$$

$$k_2 \geq \frac{cameraWidth * cameraHeight}{131072 * k_1 * mpCount}$$

It is desirable that k_1 be a divisor of *cameraWidth* and k_2 be a divisor of *cameraHeight*, because, then the number of pixels to estimate is uniformly distributed among all threads.

Table 1 shows some possible execution configurations.

Camera resolution	Multiprocessors (MPs) count	k_1	k_2	Blocks per MPs
320x240	2	1	1	150
800x600	2	1	2	469
1024x768	2	1	3	512

Table 1. Execution configurations for different camera resolution and multiprocessors count.

3.2.4 Parallel estimation and compare algorithms.

We expose next the parallel estimation algorithm:

Remark: All the coordinates are row major order.

Input: Initial coordinates of the estimated image area that the thread will compute, values of k_1 and k_2, color tables and projector frame buffer regions of the estimation area. Size of the projector frame buffer and the estimated image.

Output: Corresponding area of the estimated image.

Step 1. For each pixel in the estimated image area with coordinates *(i', j')* do steps 2 to 8:

Step 2. Initialize *redSum*, *greenSum* and *blueSum* with zero.

Step 3. Compute in *pixelCount* the number of pixels in the corresponding projector frame buffer region.

Step 4. For each pixel in the corresponding *frameBuffer* region with coordinates *(i, j)* do steps 5 to 7:

Step 5. Obtain the color components of the pixel from *frameBuffer*:

```
red   = frameBuffer[i][j]& 0x000000FF
green =(frameBuffer[i][j]& 0x0000FF00)>>8
blue  =(frameBuffer[i][j] & 0x00FF0000)>>16
```

Step 6. Compute the estimated values for each color component:

```
estRed   = redTable [i'][j'][  red /4]
estGreen = greenTable[i'][j'][green /4]
estBlue  = blueTable [i'][j'][ blue /4]
```

Step 7. Add the estimated values:

```
redSum   [i'][j'] += estRed
greenSum[i'][j'] += estGreen
blueSum  [i'][j'] += estBlue
```

Step 8. Write the averaged results in the estimated image:

```
redEstimImg   [i'][j']= redSum/pixelCount
greenEstimImg[i'][j']=greenSum/pixelCount
blueEstimImg  [i'][j']= blueSum/pixelCount
```

Algorithm 2. Parallel estimation algorithm in GPU.

The algorithm for comparing the images was also parallelized. The same execution configuration that was previously exposed is use for this algorithm.

The parallel version of the compare algorithm introduces some execution overhead, since the camera image must be copied to the CUDA's global memory to perform the comparison with the estimated image (that already is at global memory).

Next we expose the parallel compare algorithm:

Remark: All the coordinates are row major order.

Input: Initial coordinates of the estimated image area and the camera image area that the thread will compare, values of k_1 and k_2, size of the images.

Output: Binary matrix area with 1 in the pixels where a photometric perturbation may exists.

Step 1. For each pixel in the estimated image area and the camera area with coordinates *(i, j)* do steps 2 to 3:

Step 2. Compute the differences between color components:

```
red= abs(redEstimImg  [i][j] – redCamera  [i][j])
green = abs(greenEstimImg[i][j] – greenCamera[i][j])
blue=abs(blueEstimImg[i][j] – blueCamera [i][j])
```

Step 3. Compare and write the results in the output matrix:

```
outImg[i][j]=(red>redThreshold||
green>greenThreshold || blue>blueThreshold)?1:0
```

Algorithm 3. Parallel compare algorithm in GPU.

Some implementation details.

3.3.1 Execution phases.

The process of working with the GPU was divided in three main phases:

- Initialization phase: the necessary memory is allocated in CUDA and in the host, the color tables for each region are copied to CUDA's global memory, several constants are initialized and the execution configuration for the algorithms is determined.

- Execution phase: the parallel algorithms are invoked. The normal order should be:

  Estimation algorithm, requires no transference between the host and the device.

  Compare algorithm, requires two transferences between host and device: receives the camera image, and return the resulting binary matrix.

  There is some relaxation in the sense that both the estimation algorithm and the compare algorithm can be called more than once repeatedly, although if the compare algorithm is called before any call to the estimation algorithm then the results are inconsistent.

- Termination phase: all resources are released from the GPU and CPU, and the working session with CUDA is closed.

3.3.2 *Use of shared memory.*

Global memory accesses are less time-expensive while less memory transactions are require. If all threads in a half-warp (0 to 15 or 16 to 31) follow a memory access pattern (which differs according to the computing capability) then the memory accesses can be **coalesced** and only a few (one or two) memory transactions are required, thus improving performance.

Due to the nature of our problem, when the threads of a half-warp accesses its color tables to estimate a color, say the red, each can have access to any of 64 bytes, so the total area they can address is 64*16=1024 bytes, and one of the requirements for coalescence is that all the threads in the half-warp accesses an aligned memory segment having at most 128 bytes [CUD09] for computing capabilities 1.2 or above. For computing capabilities 1.0 and 1.1 the coalescence requirement are stricter, so no coalescence will be met for any computing capabilities.

Two choices are available: the use texture fetches or shared memory; we chose the second one since it is possible to obtain a bank-conflicts free distribution with a probability of 1/16 that the desired value exists in shared memory for each thread and each color component.

As the blocks have 16x16 threads, we define a matrix of shared memory with the same size for each color component, which totalize (256 threads)*(4 bytes)*(3 colors) = 3 Kb of shared memory per block. This low value allows for several active blocks per multiprocessor, improving the overall performance.

Figure 3 gives an idea of the use of shared memory in the estimation algorithm.

This distribution is bank-conflicts free. Table 2 shows the distribution of each word of shared memory to the banks of shared memory.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
…															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Table 2. Distribution of each word of shared memory in a 16x16 word matrix to the banks.

The number inside each cell represents the bank of shared memory on which the 32-bit word of shared memory is located. As can be seen, all threads of each half-warp own a word of shared memory that lies in a distinct bank, thus avoiding bank conflicts.

This shared memory size per thread gives the possibility to estimate up to 16 different values per color component without accessing the global memory, because 256/4 = 64 entries in the color table, divided by 4 bytes on each shared memory word = 16 values.

It should also be considered that it's not likely that there exist many sudden variations of color in an image, because generally the changes of colors are softened and progressive.

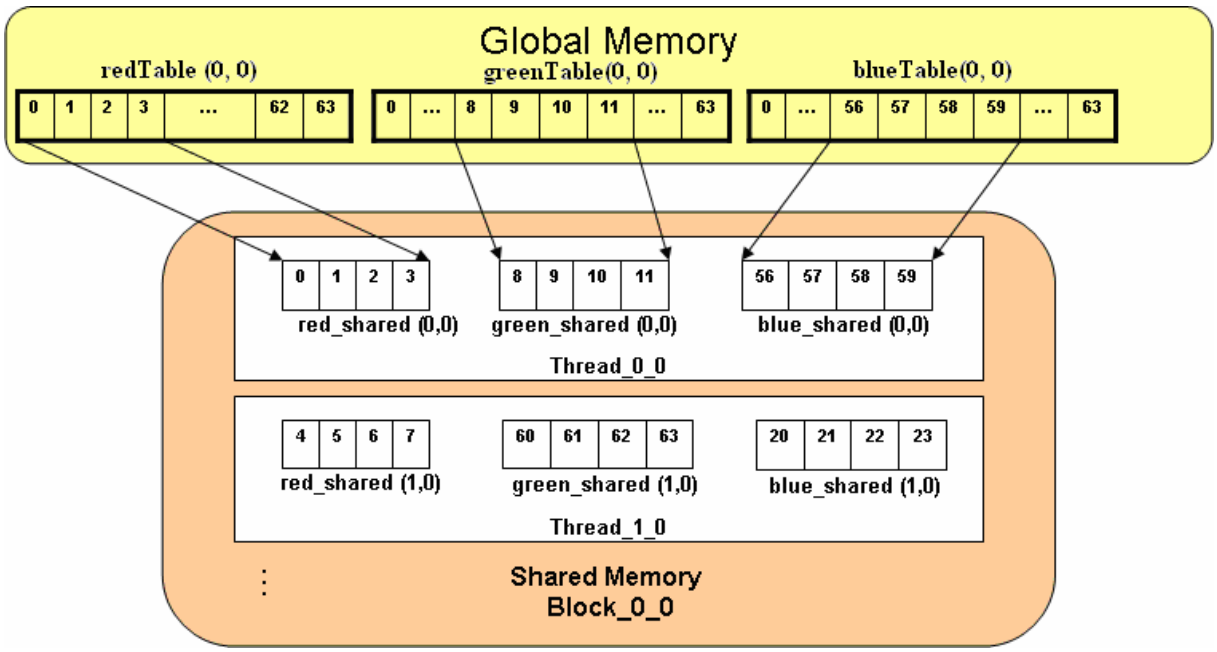

Figure 3. Use of shared memory.

EXPERIMENTATION AND RESULTS

To evaluate the parallel implementation of the local method presented in [Mig09a] for the real-time detection of non-stationary photometric perturbations in projection screens some experimentation was made.

The parallel algorithms were integrated to an existing system prototype implemented in Java through the Java Native Interface (JNI). A little effort was needed to make the system call either, the existing serial implementation on CPU or the new parallel implementation with CUDA.

The experimentation was carried out in a Core 2 Duo processor at 2.66 MHz and a GeForce 8400 GS GPU (having two multiprocessors). The screen resolution was 1024x768 and the camera resolution was 320x240, both with 32 bit color in RGB format. The development environment for the experiment was NetBeans IDE 6.5 over Microsoft Windows XP SP2, and the version of CUDA 2.2.

Table 3 shows the results of running both the serial and parallel algorithms in the system prototype. For the parallel algorithms, experimentation was made both using shared memory and not using shared memory.

As can be seen, without using shared memory a speedup of 1.7x was achieved, in contrast with the higher **2.8x** speedup obtained when using shared memory.

	Serial implementation	Parallel implementation	Parallel implementation (without using shared memory)
Latency time	95 ms	35 ms	57 ms
Average FPS	10	28	17

Table 3. Experimentation results.

The number of concurrent threads per iteration in the parallel implementation is equal to 320x240 = 76800 for the estimation algorithm, plus 76800 for the compare algorithm, making a total of 153600 threads.

CONCLUSIONS

In this work, it was presented the parallelization of the local method presented in [Mig09a] for the real-time detection of non-stationary photometric perturbations in projection screens using the Computed Unified Device Architecture.

The implementation requires neither communication nor synchronization between threads. It is also designed to be scalable and compatible with computing capabilities 1.0 or above, and a bank conflicts free access to shared memory is used in order to improve performance, obtaining a speedup of 2.8x in the experimentation.

Still some other optimizations may be introduced to the implementation in the future for trying to achieve better results.

REFERENCES

[CUD09] NVIDIA CUDA™ Programming Guide 2.2, 2009 NVIDIA Corporation.

[Flynn72a] Flynn, M. J.: Some Computer Organizations and Their Effectiveness", IEEE Transactions on Computers, vol. C-21, Sept. 1972.

[Gra03a] Grama, A., Karypis G. et al., Introduction to parallel computing, Addison-Wesley, 2003.

[Jay04a] JAYNES C., WEBB S., STEELE M.: Camera-based detection and removal of shadows from interactive multiprojector displays. IEEE Transactions on Visualization and Computer Graphics 10, 3 (2004).

[Kirs98a] Kirstein, C., Muller, H.: Interaction with a projection screen using a camera-tracked laser pointer. In: Proceedings of the International Conference on Multimedia Modeling. IEEE Computer Society Press (1998).

[Koik01] Koike, H., Sato, Y., Kobayashi, Y.: Integrating paper and digital information on EnhancedDesk: a method for real-time finger tracking on augmented desk system. In: ACM Trans. On CHI, 8 (4), pp. 307--322 (2001).

[LaRo03a] La Rosa, F., Costanzo, C, Lannizzotto, G.: VisualPen: A Physical Interface for natural human-computer interaction. In: Physical Interaction (PI03) – Workshop on Real World User Interfaces. 2003.

[Mig09a] Castañeda-Garay, M., Belmonte-Fernández, O., Gil-Altaba, J., Pérez-Rosés, H., Un Método para la Detección en Tiempo Real de Perturbaciones Fotométricas en Imágenes Proyectadas, Congreso Español de Informática Gráfica CEIG'09, San Sebastian, Sept. 9-11 (2009), Páginas 239-242. The Eurographics Digital Library, http://diglib.eg.org.

[Suk01a] Sukthankar, R., Stockton, R.G., Mullin, M.D.: Smarter presentation: Exploiting homography in camera-projector systems. In: Proceedings of International Conference on Computer Vision, pp 247--253. Vancouver, Canada, July 9-12 (2001)