Migrating multiespectral image processing to the GPU

Juan D'Amato[(1)(3)], Aldo Rubiales[(1)(3)], Fernando Mayorano[(1)(3)], Paula Tristan[(2)(3)] y Jose Massa[(2)]

(1) PLADEMA, Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires . **(2)** INTIA, Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires, Paraje Arroyo Seco, Campus Universitario (7000), Tandil, Argentina Tel.(02293) 439682 Int. 49. **(3)** CONICET, Rivadavia 1917, Ciudad Autónoma de Buenos Aires, Argentina

Abstract Multispectral satellite images contain large volumes of data distributed in bands. Therefore, the manipulation of the information contained in each image is a task that demands a lot of CPU resources. Traditionally, multispectral image manipulation has been done using exclusively time and CPU resources; but in recent years, the improvement of the graphics card's technology has provided additional processing units, besides CPUs. Firstly these units were designed to perform tasks associated with display issues or specific processes for which they were developed. This paper proposes a strategy based on the use of graphics processing units (GPUs) in order to perform a set of tasks on multispectral satellite images. In this work several algorithms have been implemented, for example: atmospheric correction and subsequent false true color display. These tasks usually require a substantial computational effort, even more when an interactive environment is required for the user. Independent pixel level processing was performed, ensuring an efficient implementation over graphic cards. This implementation improves the time required for the processing over traditional CPUs.

1 Introduction

The development of remote sensing has grown dramatically in recent years and faster growth is still expected in the future. There are numerous applications based on analysis of satellite images covering different areas of science as cartography, agriculture, forestry and military logistics, among others.

A satellite image contains a large amount of information that cannot be recognized immediately, but it may be emphasized by using adequate image processing techniques.

The digital image processing generally refers to an image processed by a computer. A more precise definition might be "put a numerical representation of an object to a series of operations to obtain a desired result". [1]

Traditionally, image processing operations rely on computer CPU, while other tasks are being executed, which may result in a decrease of performance.

An alternative to improve the performance of these tasks is to have multiple processors and a control unit that allows the parallel execution. However, it is not always feasible because of its economic cost.

In recent times, the growing necessity for display solutions has encouraged the development of programmable graphics processing units. Even though these units were designed to provide efficient visualization, they also can be used for other purposes.

The evolution of these process units (known as programmable GPUs) is a new exciting development. From the viewpoint of a general purpose programmer, these platforms can be abstracted using a stream processing model in which all data blocks are considered as ordered sets of data [5], and the applications are constructed by chaining multiple cores. At present, the latest GPUs on the market are capable of providing peaks of performance over 300 Gflops.

Image analysis algorithms can be benefited from the performance of GPU's based hardware, as remarked in [7] and [8], and programming models.

In this paper, an implementation of processing satellite multispectral image using programmable GPUs is presented. The following section presents the most relevant concepts involving GPU's, such as the rendering pipeline and the programming language. Later, we present implementation issues and finally a comparison of execution times between GPU and CPU implementations is performed.

2 GPU Concepts

The model of parallel computation SPMD (Single Process Multiple Data) proposed by [3] and now implemented, with slightly differences, in the graphic platforms (GPUs), is very convenient for real time graphics rendering, since many of the data is almost identical, in counterpart to the general use proposed by the CPU.

For the particular case of images, the adoption of this model is almost natural, already evaluated in [2], the same instructions are executed repeatedly on the same data structure, allowing the programm to divide the process in many threads, accesing to common memory spaces.

The use of the GPU as a calculation unit requires that programmers change the data structures according to GPU's flow model, and understand the data pipeline in order to optimize its performance.

2.1 Rendering Pipeline

The GPU manipulates the data through polygons, generally triangles, and textures. Each triangle is associated with a material or textured image, by bidimensional mapping which indicates the position of the pixel regarding the initial vertex.

This is the input data needed to define in the application, and which feeds the rendering process. The vertex and fragment processors are the programmable elements of the pipeline, and the programs that they execute call vertex and fragment shaders, respectively.

The vertexes processing stage carries out operations on vertexes sent to the GPU. The vertex processor transforms each of these into a vertex in the projection space. Once transformed, the vertexes are reassembled forming triangles and they are passed through the fragment flow. These fragments are the discrete portions of the surface of the triangle corresponding to the pixels of the represented image. Aside from identifying the fragments that constitute the triangle, the rastering stage interpolates attributes stored in the vertexes, such as the texture coordinates, and store these values as attributes of each fragment. The fragment processors calculate the resulting color using arithmetical operations and simultaneous access to multiple textures. In order to increase the computer efficiency, these processors support short vectorial instructions that operate on 4 components vectors (channels Red/Green/Blue/Alpha), and include units of access to textures.

The latency of the accesses to data is hidden using pre-search and efficient texture cache. Finally, the results obtained by the fragment processor are combined with the existing information stored in the position 2D associated in frame buffer, to produce the final color.

2.2 CPUs – GPUs Interaction

Developing applications that take advantages of the GPUs is still not a standarized process. Operative Systems are not prepared for the extended architecture that graphic cards represent, so it's necessary to program the interoperation and communication CPU-GPU summing up the architectural differences named before.

This communication requires some steps, shown in Figure 1, and listed below:

- Load data into the GPU memmory
- Define and pass parameters to the active programms
- Execute programms

For these requierements, some proporsals arose from the main graphics SDK experiences. The creators of OpenGL and DirectX defined each a high level language, similar to C++, alowing programmers to modify the vertex and fragment steps. The high level standard language promoted by the OpenGL consortium is GLSL [3], implemented in principle by all the manufacturers. The greater facility that has brought this scheme is the capability of processing each pixel, applying an arbitrary conversion.

According to the GPU version, the render pipeline varies certain capacities like the amount of calculation cores, the amount of simultaneous accessible textures and others hardware related improvements.

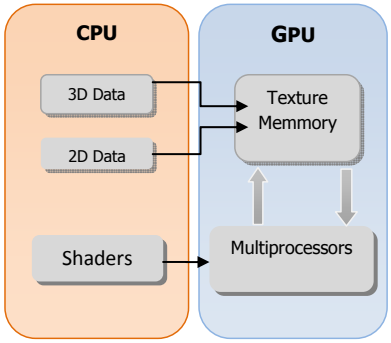

Figure 1: CPU – GPU data integration model

Recently, some new CPU-GPU integration models are emerging, like Nvidia CUDA [6] or ATI Stream Computing, but they are still quite new technologies only supported by last graphic cards and suffering of constant modifications. In this work, we promote the programming technique based on Shaders that maintains compatibility with a greater variety of graphic cards. It's necessary comment that this application is intended to run on rather old notebooks. The different processes required (Vegetation Index, False Color, Rayleigh correction) have been implemented with GLSL, generating a fragment shader for each.

3 Processing on the GPU

The first step in the process, is the preparation of the data, this task is accomplished on the CPU side and immediately passed to the rendering Pipeline. The processing of Radar images, such as SACC or SAT5, requires combining multi band information (commonly RGB or infrared data).

For the Rayleigh correction, applied on SACC's radar images, each pixel is corrected using the expression as shown in Eq. 1.

$$I = I_0 |\sigma(\theta, \phi)|^2 \frac{(2\pi)^2}{(\lambda R)^2} \quad \text{Eq. 1}$$

$$\sigma(\theta, \phi) = A(\theta) sen(\phi) \hat{e}_\phi + B(\theta) cos(\phi) \hat{e}_\theta$$

Where σ is the dispersión coefficient, θ and Φ the Azimuthal and Zenithal angles, R is the distance to the particles . I_0 is the *Solar irradiance* spectrum at top of atmosphere and λ is a constant.These aditional data will be coded in extra textures.

Modern graphic cards are capable of 3D textures managing, which is the more natural and simple representation. However, GPUs work better with bidimensional textures. Because of this, the multi-spectral image was separated into a set of 2D

textures, assigning each band or phantom to a color channel in the textures, as it is shown in Figure 2.

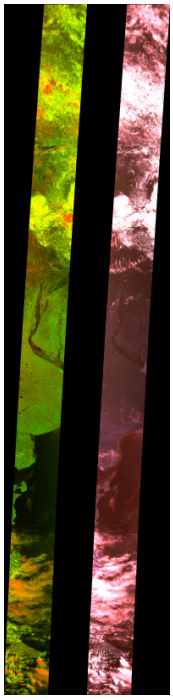

FIGURE 2: 7 bands distributed in 2 RGBA textures

As it was said, some additional data is required that have the same spatial distribution as the visible spectrum intensities, such as the Zenithal and Azimuthal angles of Satellite SACC. These data must be also coded as a Texture, commonly limited to 1 byte. This causes precision limitations, when originally data had floating precision. For this case, a 2 bytes data decomposition is proposed, one byte for the sign, and other for the absolute value. This calculation scheme, even quite inexact, is simple and efficient to recover the original value. Other implementation schemes are feasible, as long as they're not already supported by the platform and considering how they can affect the GPU performance.

In Figure 3 the obtained angle images mapped to a color space are showened. The dark red values represent the range 0° to 90°, the light red represent the 90° to 180° range and the blue ones are the 180° to 360°.

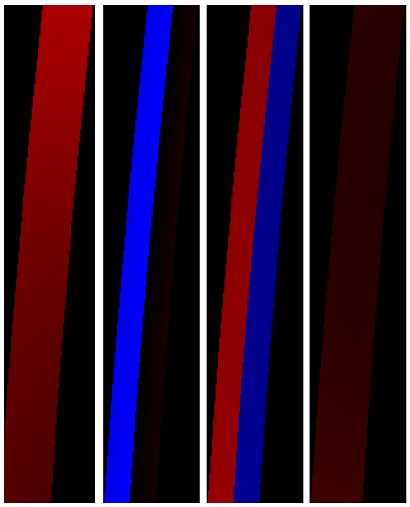

Figure 3a)solar cenithal angle texture, b) solar azimuthal angle texture, c) satellital cenithal angle texture, d) satellital azimuthal angle texture

The angle resolution is much lower than the one of the color bands, requiring to, at fragment step in the pipeline, estimate the angle value for the evaluated pixel. This interpolation is made by the GPU automatically through texture mapping, in a more efficient way than CPU traditional way.

3.2 Large Images Treatment

An issue that demands additional effort is the treatment of large images. The tipical size of radar images surpasses the capacity of the graphic card, for example, SACC images are represented with 32000 pixels by 3500 pixels resolution, while the maximum resolution allowed by the cards is 2048x2048 pixels. In order to solve this, a scheme of double zoom is used, obtaining precise visualizations of the zone, without losing the efficiency of the algorithm. This implies the definition of two levels of detail:

- A macro level to visualize all the image, with lower resolution
- A detailed level with great resolution in a defined region.

The implemented scheme of detail levels, among other possible alternatives considered is the real time generation of the textures that represent the inspected region. The user interacts with the macro level and when the load of the detailed is finished, the textures indexes are updated for the rendering. This solution has the benefit to use less amount of memory, although is a bit slower on CPUs with one core. In the Figure 4 both zoom levels are showned:

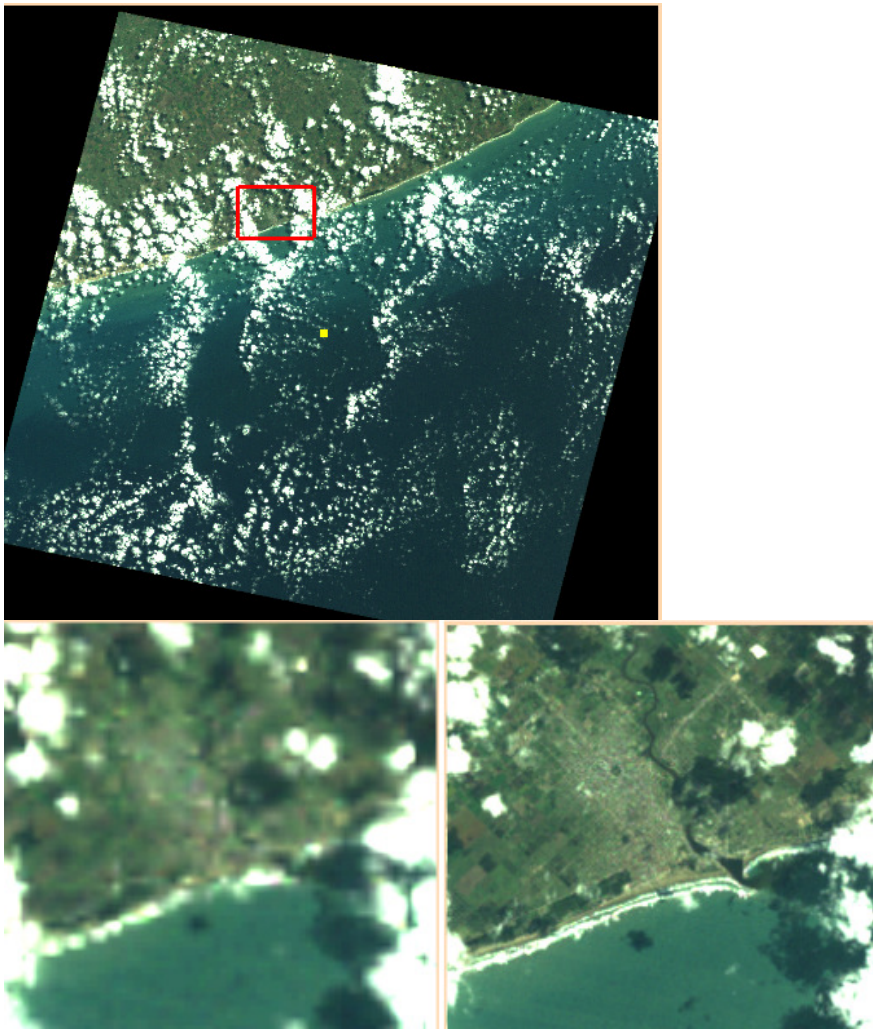

Figure 4 : Total Image with selected region, [Above], low-detail image (left), high-detail image (right)

3.3 Image Processing and Visualization

As was named previously, the Per Pixel Processing is programmed with GLSL. Each one of the required operations is implemented like different shaders, with their set of parameters.

To complete this process, the parameters of the actual view and the remote sensor are required. Such parameters are the constants of Solar Irradiance for the given sensor: ($E_{0,}\tau_r$, τ_g), active bands and the associated textures. These values are sent to the GPU, before rendering. For example, to apply False Color (a simply way to identify which band has more energy), the user chooses the active bands and each band is mapped to a RGBA channel. When the Normalized Difference Vegetation Index (NDVI) is required, the shader is concerned to mapp a value to a color.

Other typical process that visualice the temperature band is presented below. A convenient way to do this is to use the color space Hue/Saturation and Lightness (HSL). Each temperature value is associated with a chromaticity, as shown in Figure 5.

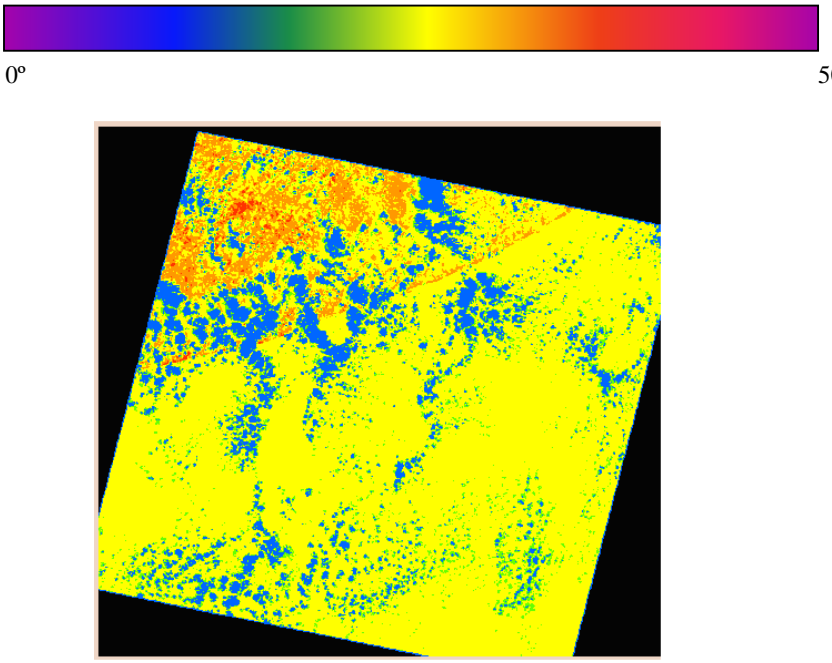

Figure 5:[Up] Temperature – color association [down] Temperature distribution

3.4 View Update

At certain elapsed time, or when something changed, the view is updated. After this event, the generated data, the application parameters and the programmed shaders recreate the visualization.

The core of the visualization is distributed in a previous part of data loading in GPU memory and an iterative part of updating the rendering, selecting shaders according to the wished process and assigning the corresponding parameters.

4 CPU – GPU Performance Comparison

Processing CPU differs from GPU essentially in the way the problem must be understood, because it requires to determine the type architecture on which the system is executed and also to know the the problem domain.

Radar	Nº Textures	Operation	Res. Texture	CPU	GPU GForce 7600
SACC	1	False_Color	100px	0,9	1,2
SACC	1	False_Color	200px	2,62	1,9
SACC	1	False_Color	400px	8,9	2,0
SACC	1	False_Color	800px	33,3	2,1
SACC	1	GAIN	100px	3	2,0
SACC	1	GAIN	200px	13	2,0
SACC	1	GAIN	400px	43,5	2,0
SACC	1	GAIN	800px	167,3	2,1
SACC	5	RAYL	100px	3	1.7
SACC	5	RAYL	200px	17	2
SACC	5	RAYL	400px	48,6	2,3
SACC	5	RAYL	800px	177	2,3
LNST5	5	RAYL	100px	0	1,8
LNST5	5	RAYL	200px	16	2,0

Table 1. Implementation comparison times in milliseconds between a CPU and a standar GPU

The CPU is still today a sequential architecture, and lies in the shadow of graphic cards with its ability to process simultaniously multiple pixels. Migrating to a parallel technology implies to be able to apply easily a transformation on data with low cohesion. This is possible when images are treated.

After that, we will evaluate the processing times of some study cases. The results were obtained on the basis of selecting squared image regions, and the times were compared to realize the different types of processing, especially on the SACC radar images, that are the most complex ones. Both implementations, CPU as GPU one, were programmed in a similar and efficient way. The execution in GPU adds a calling time that affects the overall performance. In [Table I] the run times of the proposed calculations for both implementations are shown. For the tests, it was used a Gforce 7600 graphic card, that is at the moment quite standard; on a CPU Athlon of 2.0 GHZ. In the obtained results, it is observed immediately that the CPU responds efficiently when the operations are simple and the images small and always GPU adds the latency time because of the data transfer capacity.

Even when many data sources are accessed, for instance the case of Rayleigh correction (indicated as Rayl in the picture), the GPU responds satisfactorily. It's observed that for NVIDIA 7600 graphic card, the times are very short.

In Figure 6, the table results are visualized for the different analyzed configurations involving SAC-C Rayleigh correction and False color visualization.

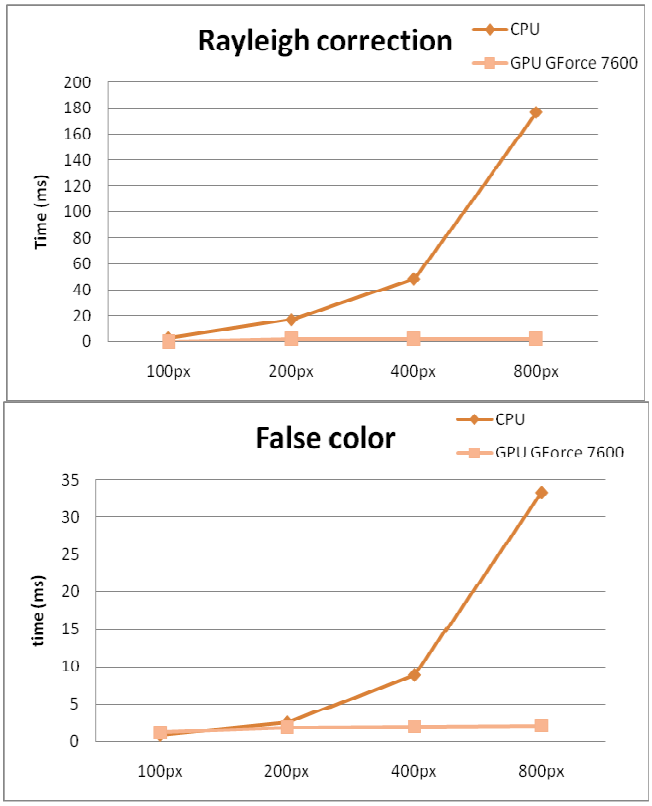

Figure 6: Time responses on GPUs and CPU for both processings of SACC Radar Images, varing the size of the inspected region

5 Conclusions

In this work we have explored the viability to use graphic hardware to implement processing algorithms on multi spectral images with great computer requirements. The propose approach represents an efficient alternative compared to standard implementations in CPU and the results show important accelerations in the visualization. In spite of these advantages, the use of the GPU also has certain limitations, generally referred to the inconvenience of using a processor for completely different aims from which were thought.

A common problem is the complexity to adapt systems to this architecture, problem that is being solved with the development of new technologies, such as CUDA. Another critic is the precision limitations of the GPU registers. Generally, they use 2 or 4 bytes to represent a float number in a GPU, which in comparison with the 4, 8 or more bytes used in the CPU, are not enough for many scientific applications. In newer versions of the pipeline these problems are being solved.

6 References

[1] K. R. Castleman Castelman. *"Digital Image Processing"*. Ed. Prentice Hall, New Jersey, (1996).

[2] Clark, A., Martinez, K. and Welsh, B. *"Parallel architectures for image processing. In: Image Processing"*, pp. 169-189, McGraw-Hill. (1991).

[3] Randi J.Rost. *The OpenGL Shading Language*. Ed. Addison-Wesley, 2nd Edition. (2006)

[4] Frederica Darema. *Architecture for Parallel Processing*. PHD Thesis (1984)

[5] IkkJin Ahn. *Image Processing on the GPU*, University of Pennsylvania. (2005).

[6] Compute Unified Device Architecture - CUDA – http://www.nvidia.es/CUDA

[7] Jean-Philippe Farrugia, Patrick Horain, Erwan Guehenneux, Yannick Alusse, *GPUCV: A framework for image processing accelaration with graphic processors*, 2006

[8] James Fung, Steve Mann, and Chris Aimone, *Openvidia: Parallel gpu computer vision*, in Proceedings of the ACM Multimedia 2005, November 2005, pp. 849–852.

.