ON THE OPTIMIZATION OF MEMORY ACCESS TO INCREASE THE PERFORMANCE OF SPATIAL PREPROCESSING TECHNIQUES ON GRAPHICS PROCESSING UNITS

J. Delgado[1], *G. Martín*[2], *J. Plaza*[1], *L. I. Jiménez*[1] *and A. Plaza* [1]

[1] Hyperspectral Computing Laboratory, University of Extremadura, Cáceres, Spain
[2]Instituto de Telecomunicações, Lisbon, Portugal

ABSTRACT

The use of spatial information prior to spectral unmixing of hyperspectral data is a very active research line in recent years. There are many approximations that consider spatial characteristics of the data in order to guide the endmember identification/extraction procedure. In particular, the spatial preprocessing (SPP) algorithm can be used prior to most existing spectral-based endmember identification techniques, thus promoting the selection of endmembers in spatially representative parts of the scene. The main concern regarding SPP and this kind of preprocessing techniques is that they are computational expensive, adding a significant burden to the spectral unmixing process which should be alleviated. In this paper we revisit and enhance a previously developed implementation of SPP for graphical processing units (GPUs) in order to increase its performance by exhaustively using the level one (L1)-cache level of the GPU. The performance of the proposed implementation is evaluated using an NVidiaTMGeForce GTX 580. Our experimental validation reveals that real-time processing performance can be obtained for real hyperspectral data sets collected by the Airborne Visible Infra-Red Imaging Spectrometer (AVIRIS).

Index Terms— Spectral unmixing, endmember identification, spatial preprocessing, graphics processing units (GPUs).

1. INTRODUCTION

Spectral mixture analysis aims at expressing the spectral signatures collected over a certain region as a linear/nonlinear mixture of the various materials found within the spatial extend of the ground instantaneous field of view of the imaging instrument [1, 2, 3]. Generally, this mixture occurs due to the insufficient spatial resolution of the acquisition instrument, and affects the correct localization of pure spectral signatures in spectrally heterogeneous areas of the analyzed scene. A well known possibility is, therefore, to guide the endmember identification process to spatially homogeneous areas, which are more likely to contain most of the purest signatures available in the scene [4, 5]. For this purpose, several spatial preprocessing (SPP) algorithms [6, 7, 8] have been used in combination with traditional endmember identification techniques [9], thus allowing that the pure signatures be obtained using spatial and spectral features. As mentioned before, the main problem of these techniques is that they add extra computational costs to the unmixing processing chain. Despite the availability of several techniques to accelerate the performance of spectral unmixing algorithms on GPUs [10, 11], very few efforts have been devoted to the generation of efficient implementations of SPP techniques to be used prior to spectral unmixing [12]. In this paper we revisit the GPU-based parallel implementation of the SPP algorithm in [12], implementing a different version that incorporates a more efficient memory management strategy. The presented implementation has been tested on two different NVidia GPU architectures: GeForce GTX 580 and GeForce GTX870M, using hyperspectral data collected by NASA's AVIRIS over the Cuprite mining district in Nevada. Our experimental validation shows that a significant reduction in the execution time can be achieved by taking advantage of the L1-cache level of the GPU.

2. SPATIAL PREPROCESSING ALGORITHM

The goal of the original SPP algorithm is to spatially weight the spectral information of each pixel in the scene [6]. Fig. 1 illustrates a toy example based on two bands of a hyperspectral dataset. The idea behind SPP is to center each spectral feature in the data cloud around its mean value, and then shift each feature toward the centroid of the data cloud. Each spectral feature is shifted proportionally to a similarity measure calculated using the spectral information of the pixel under consideration and a spatial neighborhood around it [6, 12]. In the end, pixels located in spatially homogeneous areas (such as pixel 1 in Fig. 1) are expected to have a smaller displacement regarding their original location than pure anomalous pixels (pure pixels surrounded by spectrally different materials, such as pixels 2 and 3 in Fig. 1). As a result, a new simplex (in blue in Fig. 1) is generated with regards to the original one (in red in Fig. 1), taking into account not only spectral but also spatial information.

Let $\mathbf{y}_{i,j}$ represent the pixel in spatial coordinates i, j. With this notation in mind, the scalar factor is calculated as

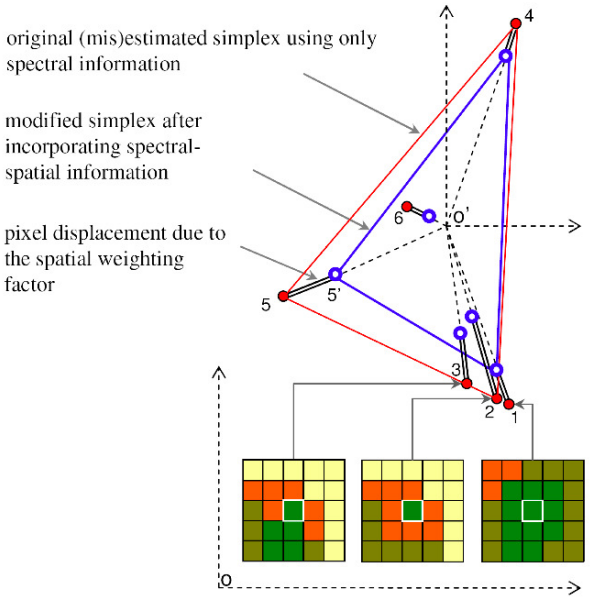

Fig. 1. Graphical illustration of the spatial preprocessing (SPP) technique.

follows:

$$\alpha(i,j) = \sum_{r=i-d}^{i+d} \sum_{s=j-d}^{j+d} \beta[r-i, s-j] \cdot \gamma[\mathbf{y}_{i,j}, \mathbf{y}_{r,s}], \quad (1)$$

where $\mathbf{y}_{i,j}$ is the pixel for which we are calculating the scalar factor, and $\mathbf{y}_{r,s}$ are the spatial neighbours. Here d represents half of the window size, so that the full window size is $ws = 2 \cdot d + 1$. The γ function computes the spectral angle between the pixel and its neighbors, and the β function computes a weight factor based on the distance between the pixel and the neighbors. The spectral angle is computed as follows:

$$\gamma[\mathbf{y}_{i,j}, \mathbf{y}_{r,s}] = \arccos \frac{< \mathbf{y}_{i,j}, \mathbf{y}_{r,s} >}{||\mathbf{y}_{i,j}|| \cdot ||\mathbf{y}_{r,s}||}, \quad (2)$$

where $< \cdot, \cdot >$ denotes the dot product between two vectors and $|| \cdot ||$ denotes the euclidean norm of a vector. As we can see in (3) the closest neighbours are given more relevance. Also the β function is normalized to sum to one as follows:

$$\beta(a,b) \propto \frac{1}{a^2 + b^2}. \quad (3)$$

Once the scalar factor has been computed, every pixel is displaced to the simplex centroid depending on the scalar factor. Expressions (4) and (5) show how to displace the image pixels depending on the scalar factor, as detailed in [13].

$$\rho(i,j) = (1 + \sqrt[2]{\alpha(i,j)})^2 \quad (4)$$

$$\mathbf{y}_{i,j}{}' = \frac{1}{\rho(i,j)}(\mathbf{y}_{i,j} - \overline{\mathbf{c}}) + \overline{\mathbf{c}} \quad (5)$$

Here, $\overline{\mathbf{c}}$ is the simplex centroid, computed as the average of all the image pixels; $\mathbf{y}_{i,j}{}'$ is the new displaced pixel and $\mathbf{y}_{i,j}$ is the original pixel at the spatial coordinates i, j. Finally, n_l and n_c are the number of lines and columns of the hyperspectral image, respectively.

3. GPU IMPLEMENTATION

The parallel implementation is based on four main kernels:

1. The first kernel computes the centroid of the simplex $\overline{\mathbf{c}}$.
2. The second kernel computes the euclidean norms of each image pixel (Φ) that will be used to compute the γ function in (2).
3. The third kernel, which is the most time consuming one, computes the similarity factor $\alpha(i,j)$ for each pixel as given by the expression (1). In this kernel there are as many blocks as pixels: $B = n_l \cdot n_c$, and there are as many threads as the window size: $T = ws^2 = (2d+1)^2$. Each of the threads of the kernel compute the dot product between the central and the corresponding neighbor pixel in the window given by $< \mathbf{y}_{i,j}, \mathbf{y}_{r,s} >$, then computes $\gamma[\mathbf{y}_{i,j}, \mathbf{y}_{r,s}]$ as in (2). After that, the kernel weights this value using the β function (precomputed using the CPU). Finally the kernel performs a reduction to sum all the values inside the window, as a result the kernel obtains $\alpha(i,j)$.
4. The fourth kernel computes the displacement to the centroid for each pixel as in (5).

In this work, we focus on improving the performance of the third kernel (as it is the most time-consuming one). Specifically, we reworked an straightforward parallel implementation of the SPP [12] in order to compute the similarity factor $\alpha(i,j)$ for each pixel taking advantage of a more efficient memory management during this process. To achieve this, we considered two implementation strategies.

3.1. One pixel per block

The most straightforward parallel implementation is the one that processes one pixel per each block using the main memory to store the required data. This kernel uses the shared memory and the GPU registers to store temporal data, but $\mathbf{Y}$, β and Φ matrices are stored and read from the global (video) memory of the GPU. In this case, the grid configuration is a mesh of $n_l \times n_c$ blocks, each one containing $ws \times ws$ threads. As we can see in Fig. 2, different blocks will access the neighboring pixels in order to process the corresponding pixels (i.e. P1 and P2). As shown in Fig. 2, both of them need to process their overlapping neighbors (represented in yellow color), thus the memory access to those neighbors are duplicated. If we extrapolate this issue to the case of several

blocks, it results in a significant decrease of parallel performance.

3.2. Several pixels per block

Trying to avoid the aforementioned issue, we propose a second implementation which processes several pixels per each block. The main goal of this implementation is to use the first level of cache memory in the GPU to cache the memory accesses to $\mathbf{Y}$. It should be noted that both L1-cache and shared memory are much faster than the local and global memories. In the previous version each block is in charge of processing one pixel, thus, they cannot use the shared memory or the L1-cache to avoid several repeated memory accesses, as illustrated in Fig. 2.

The latest NVidia architectures (Kepler and Fermi) include cache systems and, therefore, if we process several pixels in the same block (see red region in Fig. 2), the access to the neighbors of the pixels in the block (see orange region in Fig. 2), which correspond to overlapping neighbors, will be effectively cached, thus improving the performance. In Fig. 2, when the pixel P1 is processed, the memory in blue will be accessed. Thus it is likely that, when the pixel P2 in green is processed, the yellow memory positions are already in the L1-cache (see Fig. 2). Furthermore, the access to the β matrix can be also cached for the pixels processed inside the same block. The number of pixels processed by each block will be $P = \lfloor T_{max}/ws^2 \rfloor$, where T_{max} is the maximum number of threads supported by the architecture. Therefore, the number of blocks in the optimized version will be $B = (n_l \cdot n_c)/P$ and each block will contain $T = ws^2 \cdot P$ threads.

4. EXPERIMENTAL RESULTS

The dataset used in our experiments was collected by the AVIRIS instrument, operated by the NASA's Jet Propulsion Laboratory, over the Cuprite mining district in Nevada (available online [1]). The portion used in experiments corresponds to a 350×350-pixel subset, which comprises 188 spectral bands in the range from 400 to 2500 nm and a total size of around 50 MB.

The GPU implementations of SPP have been tested on a desktop computer with a GPU NVidia GTX 580, which features 512 processor cores operating at 1.54 GHz. The GPU is connected to an Intel core i7 920 CPU at 2.67 GHz with eight cores.

It is important to emphasize that according to our experiments, we can assume that the difference between serial and GPU implementations are negligible. Hence, the only relevant difference between the serial and parallel algorithms is the time they need to complete their calculations. The serial algorithm was executed in one of the available cores of the desktop computer, and the parallel times were measured in the considered GPU platform. For each experiment, 10 runs were performed and the mean values are reported (these times were always very similar, with differences on the order of a few milliseconds only).

[1] http://aviris.jpl.nasa.gov

Table 1. Mean execution times for the parallel and serial implementations of the SPP algorithm after 10 Monte-Carlo runs.

window size	3	5	7	9	11
SPP	7.86	18.63	38.87	77.72	141.80
1-pixel/block	0.69	0.71	0.78	**0.87**	**1.06**
Speedup	11.39	26.24	49.83	**89.33**	**133.77**
N-pixels/block	**0.35**	**0.42**	**0.53**	0.89	1.31
Speedup	**22.45**	**44.35**	**73.33**	87.32	108.24

Table 1 summarizes the obtained results by the C implementation and by the two GPU implementations. An optimization has been considered for the CPU implementation, namely the inclusion of the –O3 optimization flag in the compiler. Best execution times and speedups for each GPU and window size are highlighted in bold typeface.

As revealed by Table 1, the N-pixels/block version performs substantially better than the 1-pixel/block for reasonable window sizes. At this point, it is worth noting that selection of very large window sizes make no sense, due to the fact that the pixels located far away are still considered neighbors, but very low weights are assigned to the pixels near the border of the window as defined in Eq. (3). These results clearly indicate the importance of efficiently using the cache memory in order to achieve the best possible performance of GPU implementations.

5. CONCLUSIONS AND FUTURE LINES

In this paper, we have presented a new GPU implementation of a spatial preprocessing algorithm based on efficient memory management policy. The experimental results indicate that it is possible to increase the parallel performance of the algorithm by making an adequate use of L1-cache memory. This is an important contribution, as the new generations of GPUs are all including cache memories with different levels and it is of particular importance to adequately exploit these cache memories in order to optimize the implementations. Future work will be focused on the improvement of this implementation studying different L1-cache configuration parameters, and also on the development of other implementations of the full spectral unmixing chain (including spatial preprocessing) on alternative hardware devices such as field programmable gate arrays (FPGAs).

6. REFERENCES

[1] N. Keshava and J. F. Mustard, "Spectral unmixing," *IEEE Signal Process. Mag.*, vol. 19, no. 1, pp. 44–57,

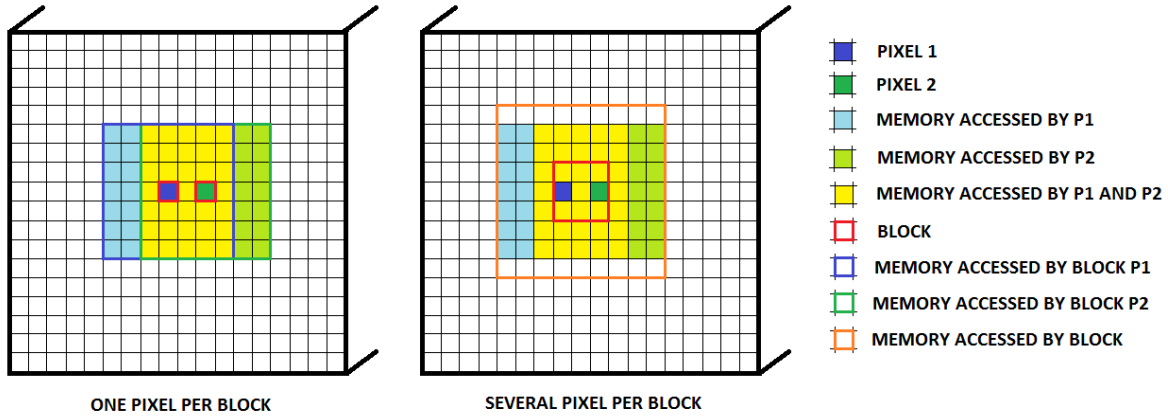

Fig. 2. Illustration of the memory accessed when there is one pixel per block and when there are several pixels per block.

2002.

[2] A. Marinoni; J. Plaza; A. Plaza; P. Gamba, "Nonlinear hyperspectral unmixing using nonlinearity order estimation and polytope decomposition," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 8, no. 6, pp. 2644–2654, 2015.

[3] J. Plaza; R. Pérez; A. Plaza; P. Martínez; D. Valencia, "Mapping oil spills on sea water using spectral mixture analysis of hyperspectral image data," in *Proceedings of SPIE Optics East 2005, III Conference on Chemical and Biological Standoff Detection*, J. O. Jensen and J. Thriault, Eds., 2005, vol. 5995, pp. 79–86.

[4] A. Plaza, P. Martinez, R. Perez, and J. Plaza, "Spatial/spectral endmember extraction by multidimensional morphological operations," *IEEE Trans. Geosci. Remote Sens.*, vol. 40, pp. 2025–2041, 2002.

[5] D. M. Rogge, B. Rivard, J. Zhang, A. Sanchez, J. Harris, and J. Feng, "Integration of spatial–spectral information for the improved extraction of endmembers," *Remote Sens. Environ.*, vol. 110, no. 3, pp. 287–303, 2007.

[6] M. Zortea and A. Plaza, "Spatial preprocessing for endmember extraction," *IEEE Trans. Geosci. Remote Sens.*, vol. 47, no. 8, pp. 2679–2693, 2009.

[7] G. Martin and A. Plaza, "Region-based spatial preprocessing for endmember extraction and spectral unmixing," *IEEE Geosci. Remote Sens. Lett.*, vol. 8, no. 4, pp. 745–749, 2011.

[8] G. Martin and A. Plaza., "Spatial-spectral preprocessing prior to endmember identification and unmixing of remotely sensed hyperspectral data," *IEEE J. Sel. Topics Appl. Earth Observations Remote Sens.*, vol. 5, no. 2, pp. 380–395, 2012.

[9] J. Plaza; E. M. T. Hendrix; I. García; G. Martín; A. Plaza, "On endmember identification in hyperspectral images without pure pixels: A comparison of algorithms," *J. Math. Imag. Vis*, vol. 42, no. 2, pp. 163–175, 2012.

[10] G. M. Gallicó; S. Lopez; B. Aguillar; J. F. López; R. Sarmiento, "Parallel implementation of the modified vertex component analysis algorithm for hyperspectral parallel implementation of the modified vertex component analysis algorithm for hyperspectral unmixing using opencl," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 7, no. 8, pp. 3650–3659, 2014.

[11] A. Plaza; J. Plaza; A. Paz; S. Sánchez, "Parallel hyperspectral image and signal processing," *IEEE Signal Processing Magazine*, vol. 28, no. 3, pp. 119–126, 2011.

[12] J. Delgado; G. Martin; J. Plaza; L. I. Jimenez; A. Plaza, "Gpu implementation of spatial preprocessing for specgtral unmixing of hyperspectral data," in *Proceedings of the 2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, IEEE, Ed., 2015, pp. 5043–5046.

[13] M. Zortea and A. Plaza, "Spatial preprocessing for endmember extraction," *IEEE Trans. Geosci. Remote Sens.*, vol. 47, pp. 2679–2693, 2009.