INTEGRATION, the VLSI journal 46 (2013) 89–103

Contents lists available at SciVerse ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

Use of FPGA or GPU-based architectures for remotely sensed hyperspectral image processing

Carlos González [a,*], Sergio Sánchez [b], Abel Paz [b], Javier Resano [c], Daniel Mozos [a], Antonio Plaza [b]

[a] Department of Computer Architecture and Automatics, Computer Science Faculty, Complutense University of Madrid, 28040 Madrid, Spain
[b] Hyperspectral Computing Laboratory, Department of Technology of Computers and Communications, University of Extremadura, 10003 Cáceres, Spain
[c] Department of Computer and Systems Engineering (DIIS), Engineering Research Institute of Aragon (I3A), University of Zaragoza, 50018 Zaragoza, Spain

ARTICLE INFO

Article history:
Received 22 January 2011
Received in revised form
5 April 2012
Accepted 5 April 2012
Available online 19 April 2012

Keywords:
Hyperspectral imaging
Hardware accelerators
FPGAs
GPUs
Application development experience

ABSTRACT

Hyperspectral imaging is a growing area in remote sensing in which an imaging spectrometer collects hundreds of images (at different wavelength channels) for the same area on the surface of the Earth. Hyperspectral images are extremely high-dimensional, and require advanced on-board processing algorithms able to satisfy near real-time constraints in applications such as wildland fire monitoring, mapping of oil spills and chemical contamination, etc. One of the most widely used techniques for analyzing hyperspectral images is spectral unmixing, which allows for sub-pixel data characterization. This is particularly important since the available spatial resolution in hyperspectral images is typically of several meters, and therefore it is reasonable to assume that several spectrally pure substances (called *endmembers* in hyperspectral imaging terminology) can be found within each imaged pixel. In this paper we explore the role of hardware accelerators in hyperspectral remote sensing missions and further inter-compare two types of solutions: field programmable gate arrays (FPGAs) and graphics processing units (GPUs). A full spectral unmixing chain is implemented and tested in this work, using both types of accelerators, in the context of a real hyperspectral mapping application using hyperspectral data collected by NASA's Airborne Visible Infra-Red Imaging Spectrometer (AVIRIS). The paper provides a thoughtful perspective on the potential and emerging challenges of applying these types of accelerators in hyperspectral remote sensing missions, indicating that the reconfigurability of FPGA systems (on the one hand) and the low cost of GPU systems (on the other) open many innovative perspectives toward fast on-board and on-the-ground processing of remotely sensed hyperspectral images.

1. Introduction

Hyperspectral imaging is concerned with the measurement, analysis, and interpretation of spectra acquired from a given scene (or specific object) at a short, medium or long distance by an airborne or satellite sensor [1]. The wealth of spectral information available from latest-generation hyperspectral imaging instruments, which have substantially increased their spatial, spectral and temporal resolutions, has quickly introduced new challenges in the analysis and interpretation of hyperspectral data sets. For instance, the NASA Jet Propulsion Laboratory's Airborne Visible Infra-Red Imaging Spectrometer (AVIRIS) [2] is now able to record the visible and near-infrared spectrum (wavelength region from 0.4 to 2.5 μm) of the reflected light of an area 2–12 km wide and several kilometers long using 224 spectral bands. The resulting data cube (see Fig. 1) is a stack of images in which each pixel (vector) has an associated *spectral signature* or 'fingerprint' that uniquely characterizes the underlying objects, and the resulting data volume typically comprises several Gigabytes per flight. This often leads to the requirement of hardware accelerators to speed-up computations, in particular, in analysis scenarios with real-time constraints in which on-board processing is generally required [3]. It is expected that, in future years, hyperspectral sensors will continue increasing their spatial, spectral and temporal resolutions (images with thousands of spectral bands are currently in operation or under development). Such wealth of information has opened groundbreaking perspectives in several applications [4] (many of which with real-time processing requirements) such as environmental modeling and assessment for Earth-based and atmospheric studies, risk/hazard prevention and response including wild land fire tracking, biological threat detection, monitoring of oil spills and other types

* Corresponding author.
E-mail addresses: carlosgonzalez@fdi.ucm.es (C. González), sersanmar@unex.es (S. Sánchez), apazgal@unex.es (A. Paz), jresano@unizar.es (J. Resano), mozos@fis.ucm.es (D. Mozos), aplaza@unex.es (A. Plaza).

0167-9260/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.vlsi.2012.04.002

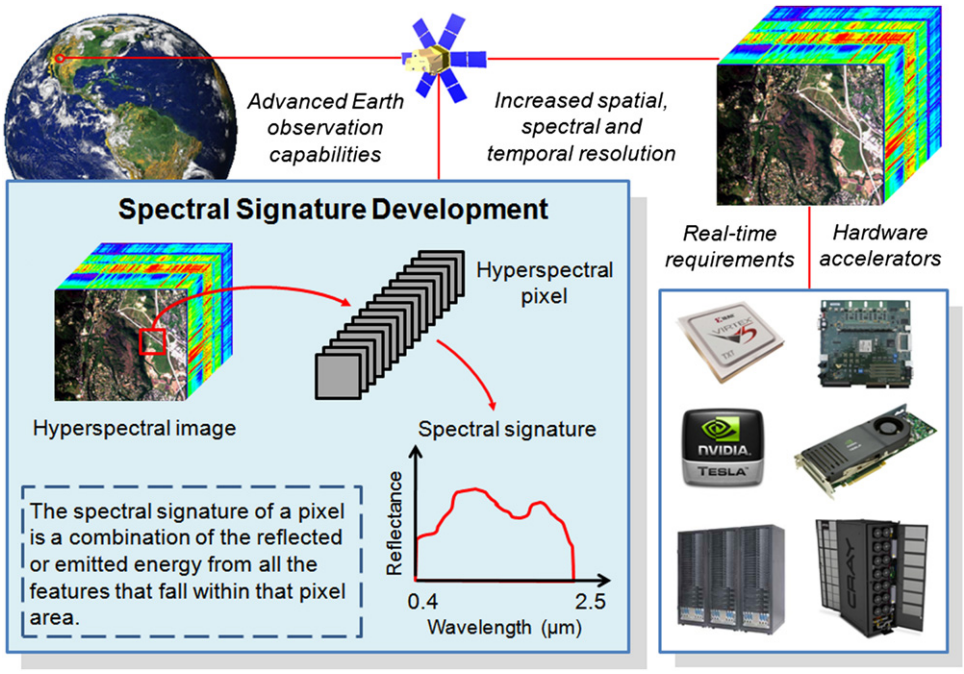


Fig. 1. An illustration of the processing demands introduced by the ever increasing dimensionality of remotely sensed hyperspectral imaging instruments.

of chemical contamination, target detection for military and defense/security purposes, urban planning and management studies, etc. [5].

Even though hyperspectral image processing algorithms generally map quite nicely to parallel systems such as clusters or networks of computers [6,7], these systems are generally expensive and difficult to adapt to on-board data processing scenarios, in which low-weight and low-power integrated components are essential to reduce mission payload and obtain analysis results in real-time, i.e. at the same time as the data is collected by the sensor [3]. Enabling on-board data processing introduces many advantages, such as the possibility to reduce the data down-link bandwidth requirements by both pre-processing data and selecting data to be transmitted based upon some predetermined content-based criteria [8]. In this regard, an exciting new development in the field of commodity computing is the emergence of programmable hardware accelerators such as field programmable gate arrays (FPGAs) [9] and graphic processing units (GPUs) [10], which can bridge the gap toward on-board and real-time analysis of hyperspectral data [8,11].

The appealing perspectives introduced by hardware accelerators such as FPGAs (on-the-fly reconfigurability [12] and software-hardware co-design [13]) and GPUs (very high performance at low cost [14]) also introduce significant advantages with regards to more traditional cluster-based systems. First and foremost, a cluster occupies much more space than an FPGA or a GPU. This aspect significantly limits the exploitation of cluster-based systems in on-board processing scenarios, in which the weight (and the power consumption) of processing hardware must be limited in order to satisfy mission payload requirements [3]. On the other hand, the maintenance of a large cluster represents a major investment in terms of time and finance. Although a cluster is a relatively inexpensive parallel architecture, the cost of maintaining a cluster can increase significantly with the number of nodes [6]. Quite opposite, FPGAs and GPUs are characterized by their low weight and size, and by their capacity to provide similar computing performance at lower costs in the context of hyperspectral imaging applications [11,12,14–18]. In addition, FPGAs offer the appealing possibility of adaptively selecting a hyperspectral processing algorithm to be applied (out of a pool of available algorithms) from a control station on Earth. This feature is possible thanks to the inherent re-configurability of FPGA devices [9], which are generally more expensive than GPU devices [14]. In this regard, the adaptivity of FPGA systems for on-board operation, as well as the low cost and portability of GPU systems, open innovative perspectives.

In this paper, we discuss the role of FPGAs and GPUs in the task of accelerating hyperspectral imaging computations. A full spectral unmixing chain is used as a case study throughout the paper and implemented using both types of accelerators in a real hyperspectral application (extraction of geological features at the Cuprite mining district in Nevada, USA) using hyperspectral data collected by AVIRIS. The remainder of the paper is organized as follows. Section 2 describes a hyperspectral processing chain based on spectral unmixing, a widely used technique to analyze hyperspectral data with sub-pixel precision. Section 3 describes an FPGA implementation of the considered chain. Section 4 describes a GPU implementation of the same chain. Section 5 provides an experimental comparison of the proposed parallel implementations using AVIRIS hyperspectral data. Finally, Section 6 concludes with some remarks and hints at plausible future research lines.

2. Hyperspectral unmixing chain

In this section, we present spectral unmixing [19] as a hyperspectral image processing case study. No matter the spatial resolution, the spectral signatures collected in natural environments are invariably a mixture of the signatures of the various materials found within the spatial extent of the ground instantaneous field view of the imaging instrument [20]. The availability of hyperspectral instruments with a number of spectral bands that exceeds the number of spectral mixture components allow us to approach this problem as follows. Given a set of spectral vectors acquired from a given area, spectral unmixing aims at inferring the pure spectral signatures, called *endmembers* [21,22], and the material fractions, called *fractional abundances* [23], at each pixel. Let us assume that a

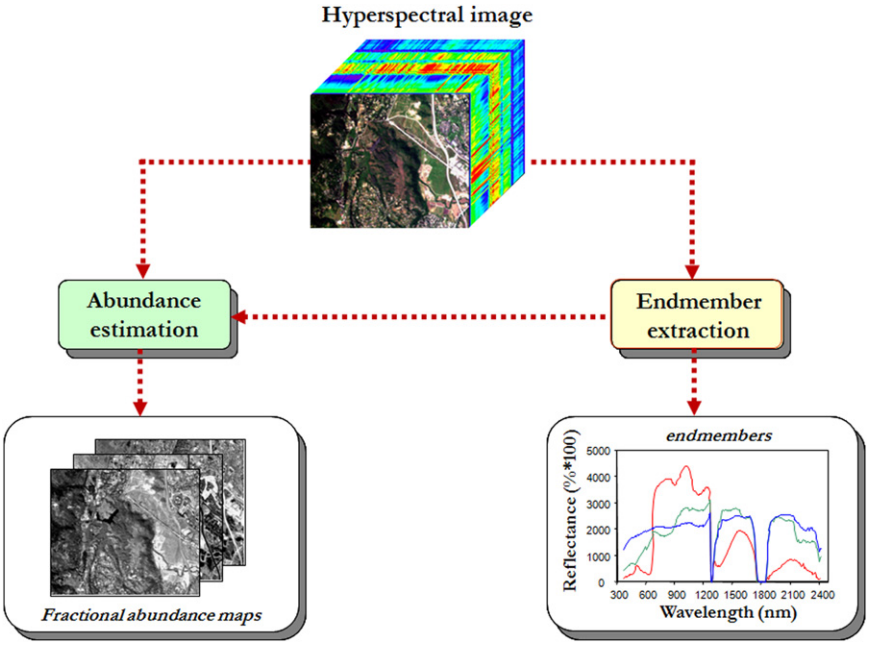


Fig. 2. Hyperspectral unmixing chain.

hyperspectral scene with N bands is denoted by $\mathbf{F}$, in which a pixel of the scene is represented by a vector $\mathbf{f}_i = [f_{i1}, f_{i2}, \ldots, f_{in}] \in \Re^N$, where $\Re$ denotes the set of real numbers in which the pixel's spectral response f_{ik} at sensor wavelengths $k = 1,\ldots,N$ is included. Under the linear mixture model assumption [19], each pixel vector can be modeled using the following expression [24]:

$$\mathbf{f}_i = \sum_{j=1}^{p} \mathbf{e}_j \cdot \Phi_j + \mathbf{n}, \tag{1}$$

where $\mathbf{e}_j = [e_{j1}, e_{j2}, \ldots, e_{jn}]$ denotes the spectral response of an endmember, Φ_j is a scalar value designating the fractional abundance of the endmember $\mathbf{e}_j$, p is the total number of endmembers, and $\mathbf{n}$ is a noise vector. The solution of the linear spectral mixture problem described in (1) relies on the correct determination of a set $\{\mathbf{e}_j\}_{j=1}^{p}$ of endmembers and their abundance fractions $\{\Phi_j\}_{j=1}^{p}$ at each pixel $\mathbf{f}_i$.

In this work we have considered a standard hyperspectral unmixing chain which is available in commercial software packages such as ITTVis Environment for Visualizing Images (ENVI).[1] The unmixing chain is graphically illustrated by a flowchart in Fig. 2 and consists of two main steps: (1) endmember extraction, implemented in this work using the pixel purity index (PPI) algorithm [25], and (2) non-negative abundance estimation, implemented in this work using the image space reconstruction algorithm (ISRA), a technique for solving linear inverse problems with positive constraints [26]. It should be noted that an alternative approach to the hyperspectral unmixing chain described in Fig. 2 is based on including a dimensionality reduction step prior to the analysis. However, this step is mainly intended to reduce processing time but often discards relevant information in the spectral domain. As a result, in our implementation we do not include the dimensionality reduction step in order to work with the full spectral information available in the hyperspectral data cube. With the aforementioned observations in mind, we describe below the two steps of the considered hyperspectral unmixing chain.

[1] http://www.ittvis.com.

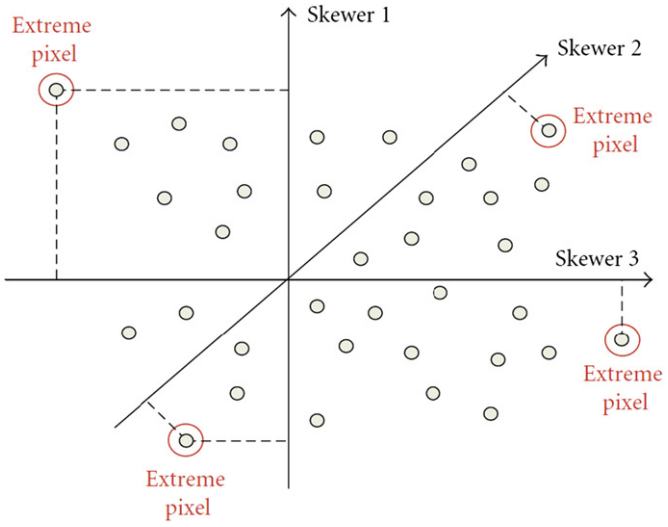


Fig. 3. Toy example illustrating the PPI endmember extraction algorithm in a two-dimensional space.

2.1. Endmember extraction

In this stage we use the PPI algorithm, a popular approach which calculates a spectral purity score for each n-dimensional pixel in the original data by generating random unit vectors (called *skewers*), so that all pixel vectors are projected onto the skewers and the ones falling at the extremes of each skewer are counted. After many repeated projections to different skewers, those pixels that count above a certain cut-off threshold are declared "pure" (see Fig. 3).

The inputs to the PPI algorithm in Fig. 3 are a hyperspectral image cube $\mathbf{F}$ with N spectral bands; a maximum number of projections, K; a cut-off threshold value, v_c, used to select as final endmembers only those pixels that have been selected as extreme pixels at least v_c times throughout the process; and a threshold angle, v_a, used to discard redundant endmembers during the

process. The output is a set of p endmembers $\{\mathbf{e}_j\}_{j=1}^{p}$. The algorithm can be summarized by the following steps:

1. *Skewer generation*. Produce a set of K randomly generated unit vectors, denoted by $\{\mathbf{skewer}_j\}_{j=1}^{K}$.
2. *Extreme projections*. For each $\mathbf{skewer}_j$, all sample pixel vectors $\mathbf{f}_i$ in the original data set $\mathbf{F}$ are projected onto $\mathbf{skewer}_j$ via dot products of $\mathbf{f}_i \cdot \mathbf{skewer}_j$ to find sample vectors at its extreme (maximum and minimum) projections, forming an extrema set for $\mathbf{skewer}_j$ which is denoted by $S_{extrema}(\mathbf{skewer}_j)$.
3. *Calculation of pixel purity scores*. Define an indicator function of a set S, denoted by $I_S(\mathbf{f}_i)$, to denote membership of an element $\mathbf{f}_i$ to that particular set as $I_S(\mathbf{f}_i)=1$ if $(\mathbf{f}_i \in S)$ else 0. Using the function above, calculate the number of times that a given pixel has been selected as extreme using the following equation

$$N_{PPI}(\mathbf{f}_i) = \sum_{j=1}^{K} I_{S_{extrema}(\mathbf{skewer}_j)}(\mathbf{f}_i). \quad (2)$$

4. *Endmember selection*. Find the pixels with value of $N_{PPI}(\mathbf{f}_i)$ above v_c and form a unique set of p endmembers $\{\mathbf{e}_j\}_{j=1}^{p}$ by calculating the spectral angle (SA) [20,27] for all possible endmember pairs and discarding those which result in an angle value below v_a. The SA is invariant to multiplicative scalings that may arise due to differences in illumination and sensor observation angle [24].

The most time consuming stage of the PPI algorithm is given by step 2 (extreme projections). For example, running step 2 on a hyperspectral image with 614×512 pixels (the standard number of pixels produced by NASA's AVIRIS instrument in a single frame, each with $N=224$ spectral bands) using $K=400$ skewers (a configuration which empirically provides good results based on extensive experimentation reported in previous work [21]) requires the calculation of more than 2×10^{11} multiplication/accumulation (MAC) operations, i.e. a few hours of non-stop computation on a 500 MHz microprocessor with 256 MB SDRAM [6]. In [12], another example is reported in step 2 of the PPI algorithm on ENVI 4.5 version took more than 50 min of computation to project every data sample vector of a hyperspectral image with the same size reported above onto $K=10^4$ skewers in a PC with AMD Athlon 2.6 GHz processor and 512 MB of RAM. Fortunately, the PPI algorithm is well suited for parallel implementation. In [28,29], two parallel architectures for implementation of the PPI are proposed. Both are based on a 2-dimensional processor array tightly connected to a few memory banks. A speedup of 80 is obtained through an FPGA implementation on the Wildforce board (4 Xilinx XC4036EX plus 4 memory banks of 512 KB) [30]. More recent work presented in [31,13] uses the concept of blocks of skewers (BOSs) to generate the skewers more efficiently. Although these works have demonstrated the efficiency of a hardware implementation on reconfigurable FPGA boards, they rely on a different strategy for generating the skewers and, hence, we do not to use them for comparisons at this point. To the best of our knowledge, no GPU implementations of the PPI algorithm are reported in the literature.

2.2. Abundance estimation

Once a set of p endmembers $\mathbf{E}=\{\mathbf{e}_j\}_{j=1}^{p}$ have been identified (note that $\mathbf{E}$ can also be seen as an $n \times p$ matrix where n is the number of spectral bands and p is the number of endmembers), a positively constrained abundance estimation, i.e. $\Phi_j \geq 0$ for $1 \leq j \leq p$, can be obtained using ISRA [26,32], a multiplicative algorithm based on the following iterative expression:

$$\hat{\Phi}_j^{k+1} = \hat{\Phi}_j^{k}\left(\frac{\sum_{l=1}^{n}(e_{jl} \cdot f_{il})}{\sum_{l=1}^{n}(e_{jl} \cdot e_{jl}) \cdot \hat{\Phi}_j^{k}}\right), \quad (3)$$

where the endmember abundances at pixel $\mathbf{f}_i=[f_{i1}, f_{i2}, \ldots, f_{in}]$ are iteratively estimated, so that the abundances at the $k+1$-th iteration for a given endmember $\mathbf{e}_j$, $\hat{\Phi}_j^{k+1}$, depend on the abundances estimated for the same endmember at the k-th iteration, $\hat{\Phi}_j^{k}$. The procedure starts with an initial estimation (e.g. equal abundance for all endmembers in the pixel) which is progressively refined in a given number of iterations. It is important to emphasize that the calculations of the fractional abundances for each pixel are independent, so they can be calculated simultaneously without data dependencies, thus increasing the possibility of parallelization.

3. FPGA implementation

In this section we present an FPGA implementation for the endmember extraction and the abundance estimation parts of the spectral unmixing chain described in Section 2. The proposed architecture specification can be easily adapted to different platforms. Also, our architecture is scalable depending on the amount of available resources.

3.1. FPGA implementation of endmember extraction

The most time consuming stage of the PPI algorithm (extreme projections) computes a very large number of dot products, all of which can be performed simultaneously. At this point, it is important to recall that the dot product unit in our FPGA implementation is intended to compute the dot product between a pixel vector $\mathbf{f}_i$ and a $\mathbf{skewer}_j$. If we consider a simple basic unit such as the one displayed in Fig. 4(a) as the baseline for parallel computations, then we can perform the parallel computations by pixels [see Fig. 4(b)], by skewers [see Fig. 4(c)], or by pixels and skewers [see Fig. 4(d)]. The max/min units are intended to calculate the maximum on the pixel vector $\mathbf{f}_i$ for a given $\mathbf{skewer}_j$.

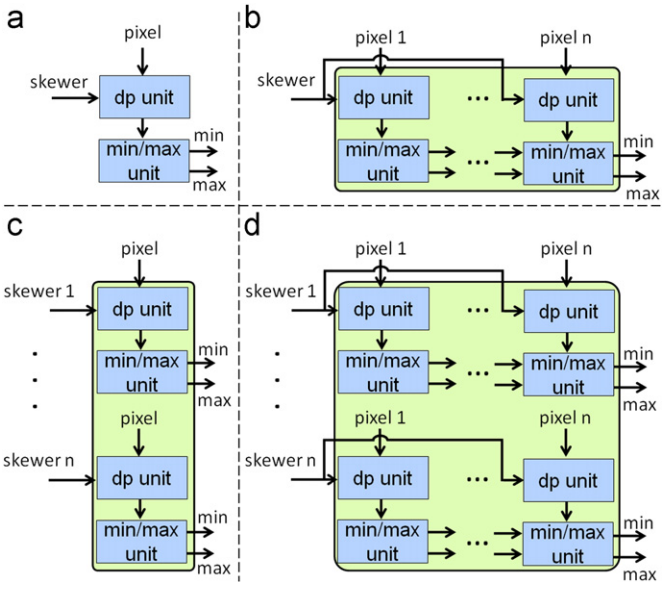


Fig. 4. (a) Basic unit. (b) Parallelization strategy by pixels. (c) Parallelization strategy by skewers. (d) Parallelization strategy by skewers and pixels.

The sequential design of the basic units ensures that the increase in the number of parallel computations (in any considered parallel scenario) does not increase the critical path, therefore the clock cycle remains constant. Furthermore, if we increased the number of parallel computations, the required area would grow proportionally with the number of basic units (again, in any considered parallel scenario). In parallelization by skewers [see Fig. 4(c)] once we have calculated the projections of all hyperspectral pixels with a skewer, we know the maximum and the minimum for this skewer. However, in the parallelization by pixels [see Fig. 4(b)] or by skewers and pixels [see Fig. 4(d)], a few extra clock cycles are required to evaluate the maximum and minimum of a skewer using a cascade computation.

Taking in mind the above rationale, in this work we have selected the parallelization strategy based on skewers. Apart from the aforementioned advantages with regards to other possible strategies, the main reason for our selection is that the parallelization strategy based on skewers fits very well the procedure for data collection (in a pixel-by-pixel fashion) at the imaging instrument. Therefore, parallelization by skewers is the one that best fits the data entry mechanism since each pixel can be processed immediately as collected. Specifically, our hardware system should be able to compute K dot products against the same pixel $\mathbf{f}_i$ at the same time, K being the number of skewers.

Fig. 5 shows the architecture of the hardware used to implement the PPI algorithm, along with the I/O communications. For data input, we use a DDR2 SDRAM and a DMA (controlled by a PowerPC) with a FIFO to store pixel data. For data output, we use a PowerPC to write the position of the endmembers in the DDR2 SDRAM. Finally, a systolic array, a random generation module and an occurrences memory are also used. For illustrative purposes, Fig. 6 describes the architecture of the dot product processors

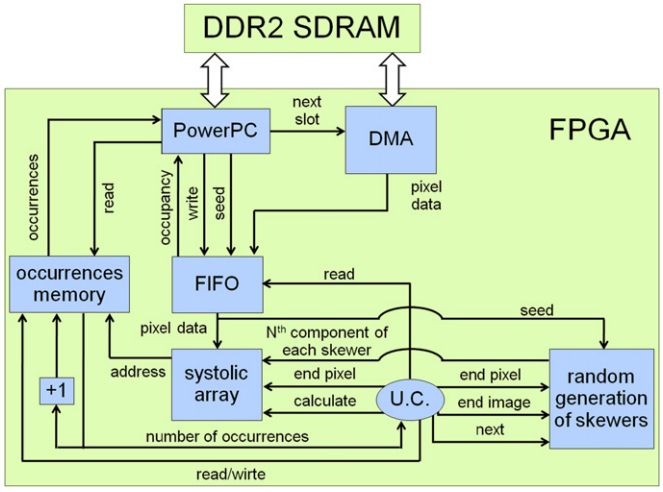


Fig. 5. Hardware architecture to implement the endmember extraction step.

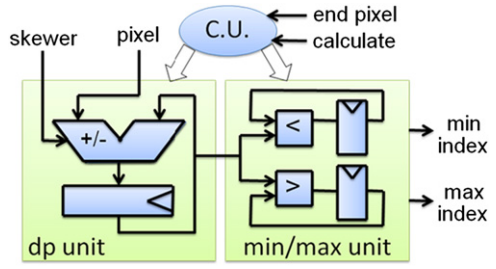


Fig. 6. Hardware architecture of a dot product processor.

used in our systolic array. Basically, a systolic cycle consists of computation of a single dot product between a pixel and a skewer, and memorization of the index of the pixel if the dot product is higher or smaller than a previously computed max/min value. It has been shown the in previous work [28,29] that the skewer values can be limited to a very small set of integers when their dimensionality is large, as in the case of hyperspectral images. A particular and interesting set is $\{1, -1\}$ since it avoids the multiplication. The dot product is thus reduced to an accumulation of positive and negative values. As a result, each dot product processor only needs to accumulate the positive or negative values of the pixel input according to the skewer input. These units are thus only composed of a single addition/subtraction operator and a register. The min/max unit receives the result of the dot product and compares it with the previous minimum and maximum values. If the result is a new minimum or maximum, it will be stored for future comparisons.

On the other hand, the incorporation of a hardware-based random generation module is one of the main features of our system. This module significantly reduces the I/O communications that were the main bottleneck of the system in the previous implementations [15,28,29]. It should be noted that, in a digital system, it is not possible to generate 100% random numbers. In our design, we have implemented a random generator module which provides pseudo-random and uniformly distributed sequences using registers and XOR gates. It requires an affordable amount of space (178 slices for 100 skewers) and it is able to generate the next component of every skewer in only one clock cycle and operates at a high clock frequency (328 MHz).

To calculate the number of times each pixel has been selected as extreme (step 3 of the algorithm) we use the occurrences memory, which is initialized to zero. Once we have calculated the minimum and maximum value for each of the skewers, we update the number of occurrences by reading the previous values stored for the extremes in the occurrences memory and then, writing these values increased by one. When this step is completed, the PowerPC reads the total number of occurrences for each pixel. If this number exceeds the threshold value v_c, it is selected as an endmember. After that, the PowerPC calculates the SA for all possible endmember pairs and discards those which result in an angle value below v_a (step 4 of the algorithm). Finally, the PowerPC writes the non-redundant endmember positions in the DDR2 SDRAM.

In order to make a flexible description of the design, most parts of it are done using generic parameters that allow us to instantiate the design following the characteristic of the FPGA without the need to make any change in the code. We only have to set the appropriate value to each design parameter (e.g. the number of parallel units). This reconfigurability property is one of the most important advantages of FPGAs (dynamic and flexible design) over application-specific integrated circuits (ASICs), characterized by their static design. Moreover, in our FPGA-based architecture we first read from the DDR2 SDRAM all configuration parameters. With this strategy we provide our design with the flexibility needed to adapt (in runtime) to different hyperspectral scenes, different acquisition conditions, even from different hyperspectral sensors.

To conclude this section, we provide a step-by-step description of how the proposed architecture performs the extraction of a set of endmembers from a hyperspectral image

- Firstly, we read from the DDR2 SDRAM all configuration parameters, i.e. the number of skewers K, the number of pixels and bands N of the hyperspectral scene, the cut-off threshold value v_c to decide which pixels are to be selected as endmembers, and the threshold angle v_a to discard redundant endmembers. These values are stored in registers.

- To initialize the random generation module, the PowerPC generates two seeds of K bits (where K is the number of skewers) and writes them to the FIFO.
- Afterwards, the control unit reads the seeds and sends them to a random generation module which stores them. Hence, the random generation module can provide the systolic array with 1 bit for each skewer on each clock cycle.
- After the PowerPC has written the two seeds, it places an order to the DMA so that it starts copying a piece of the image from the DDR2 SDRAM to the FIFO. As mentioned before, the main bottleneck in this kind of system is data input which is addressed in our implementation by the incorporation of a DMA that eliminates most I/O overheads. Moreover, the PowerPC monitors the input FIFO and sends a new order to the DMA every time that it detects that the FIFO is half-empty. This time, the DMA will bring a piece of the image that occupies half of the FIFO.
- When the data of the first pixel has been written in the FIFO, the systolic array and the random generation module start working. Every clock cycle a new pixel is read by the control unit and sent to the systolic array. In parallel, the k-th component of each skewer is also sent to the systolic array by the random generation module.
- During N clock cycles data of a pixel are accumulated positively or negatively depending of the skewer component. In the next clock cycle, the min/max unit updates the pixel and the random generation module restores the original two seeds, concluding the systolic cycle. In order to process the hyperspectral image, we need as many systolic cycles as pixels in the image.
- When the entire image is processed, we update the number of occurrences as extreme for all minima and maxima through reads and writes to the occurrences memory.
- The aforementioned steps are repeated many times according to the number of skewers that can be parallelized and the total number of skewers we want to evaluate.
- Now the PowerPC reads the pixel purity score associated to each pixel. If it exceeds the preset threshold value, v_c, this pixel is selected as an endmember. After this, the PowerPC discards redundant endmembers and writes the non-redundant endmember positions in the DDR2 SDRAM.

3.2. FPGA implementation of abundance estimation

Fig. 7 shows the architecture of the hardware used to implement the ISRA for abundance estimation, along with the I/O communications, following a scheme similar to the previous subsection. The general structure can be seen as a three-stage pipelined architecture: the first stage provides the necessary data for the system (endmembers and image data), the second stage carries out the abundance estimation process for each pixel, and finally the third stage sends such fractional abundances via an RS232 port. Furthermore, in our implementation these three steps work in parallel. Additional details about the hardware implementation in Fig. 7 can be found in [32].

Since the fractional abundances can be calculated simultaneously for different pixels, the calculation of the ISRA algorithm can be parallelized by replicating each ISRA basic unit u times and by adding two referees, one to read from the write FIFO and to send data to the corresponding basic unit or units, and the other to write the abundance fractions to the read FIFO. The number of times that we can replicate the ISRA basic unit is determined by the amount of available hardware resources and fixes the speedup of the parallel implementation. Additional details about the behavior of the referees, along with a step-by-step description of how the proposed architecture performs the abundance estimation can be found in [32].

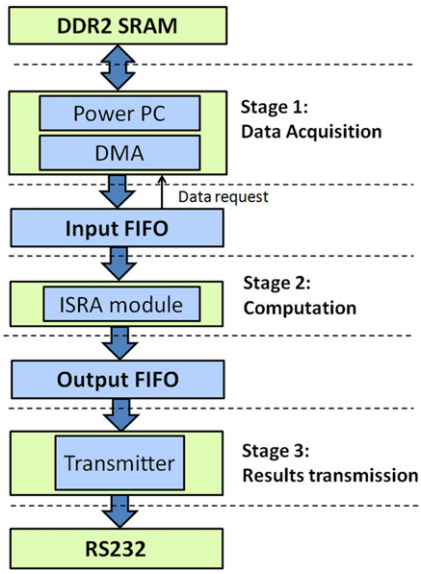


Fig. 7. Hardware architecture to implement the abundance estimation step.

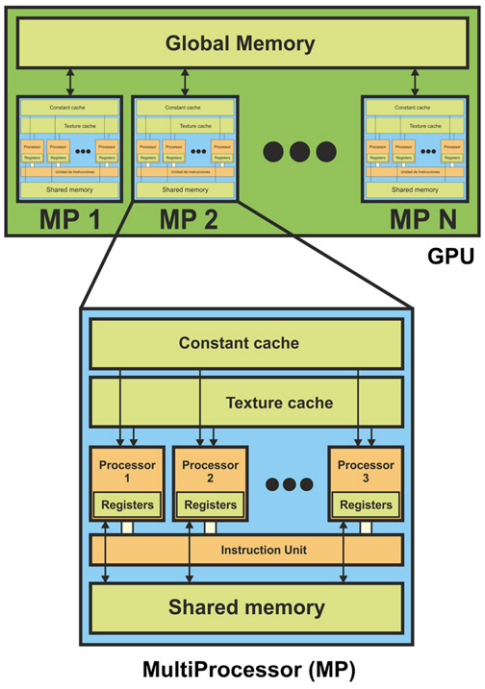


Fig. 8. Schematic overview of a GPU architecture.

4. GPU implementation

GPUs can be abstracted in terms of a stream model, under which all data sets are represented as streams (i.e. ordered data sets) [33]. Fig. 8 shows the architecture of a GPU, which can be seen as a set of multiprocessors (MPs). Each multiprocessor is characterized by a single instruction multiple data (SIMD) architecture, i.e. in each clock cycle each processor executes the same instruction but operating on multiple data streams. Each processor has access to a local shared memory and also to local cache memories in the multiprocessor, while the multiprocessors have access to the global GPU (device) memory. Algorithms are constructed by chaining so-called *kernels* which operate on entire streams and which are executed by a multiprocessor, taking one

or more streams as inputs and producing one or more streams as outputs. The kernels can perform a kind of batch processing arranged in the form of a grid of blocks (see Fig. 9), where each block is composed by a group of threads which share data efficiently through the shared local memory and synchronize their execution for coordinating accesses to memory. As a result, there are different levels of memory in the GPU for the thread, block and grid concepts (see Fig. 10). There is also a maximum number of threads that a block can contain but the number of threads that can be concurrently executed is much larger (several blocks executed by the same kernel can be managed concurrently, at the expense of reducing the cooperation between threads since the threads in different blocks of the same grid cannot synchronize with the other threads, as explored in the context of hyperspectral imaging implementations in [34]). Our implementation of the

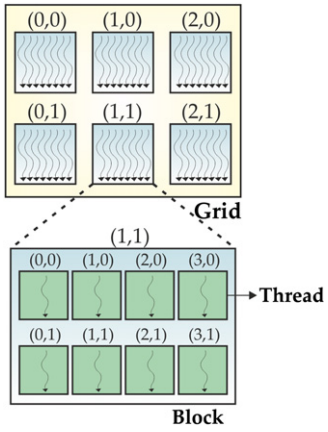


Fig. 9. Batch processing in the GPU: grids of blocks of threads.

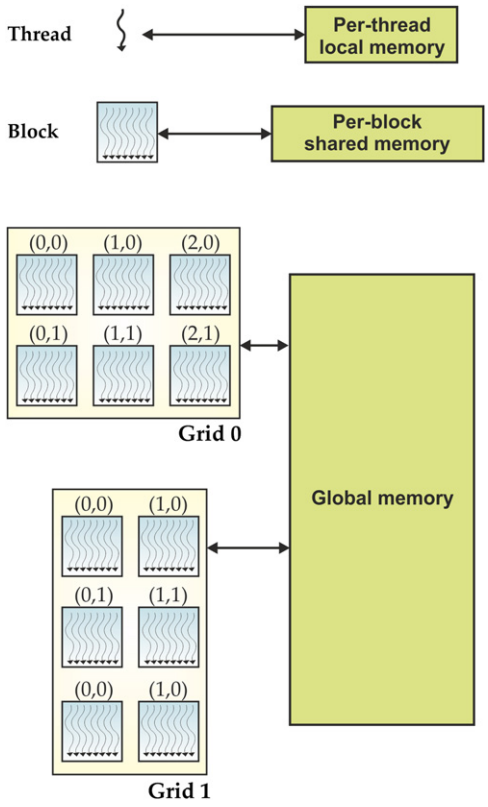


Fig. 10. Different levels of memory in the GPU for the thread, block and grid concepts.

endmember extraction and the abundance estimation parts of the considered hyperspectral unmixing chain has been carried out using the compute unified device architecture (CUDA).[2]

4.1. GPU implementation of endmember extraction

The first issue that needs to be addressed is how to map a hyperspectral image onto the memory of the GPU. If the size of hyperspectral image exceeds the capacity of the GPU memory, we split the image into multiple spatial-domain partitions [6] made up of entire pixel vectors. In order to perform random generation of skewers in the GPU, we have adopted one of the most widely used methods for random number generation in software, the Mersenne twister, which has extremely good statistical quality [35]. This method is implemented in the `RandomGPU` kernel available in `CUDA`.

Once the skewer generation process is completed, the most time-consuming kernel in our GPU implementation is the one taking care of the *extreme projections* step of the endmember extraction step. This kernel is denoted by `PPI` and displayed in Fig. 11 for illustrative purposes. The first parameter of the `PPI` kernel, `d_image`, is the original hyperspectral image. The second parameter is a structure that contains the randomly generated skewers. The third parameter, `d_res_partial`, is the structure in which the output of the kernel will be stored. The kernel also receives as input parameters the dimensions of the hyperspectral image, i.e. `num_lines`, `num_samples` and `num_bands`. The structure `l_rand` (local to each thread) is used to store a skewer through a processing cycle. It should be noted that each thread works with a different skewer and the values of the skewer are continuously used by the thread in each iteration, hence it is reasonable to store skewers in the registers associated to each thread since these memories are hundreds of times faster than the global GPU memory. The second structure used by the kernel is `s_pixels`, which is shared by all threads. Since each thread needs to perform the dot product using the same image pixels, it is reasonable to make use of a shared structure which can be accessed by all threads, thus avoiding that such threads access the global GPU memory separately. Since the `s_pixels` structure is also stored in shared memory, the accesses are also much faster. It should be noted that the local memories in a GPU are usually quite small in order to guarantee very fast accesses, therefore the `s_pixels` structure can only accommodate a block of ν pixels.

Once a skewer and a group of ν pixels are stored in the local memory associated to a thread, the next step is to perform the dot product between each pixel vector and the skewer. For this purpose, each thread uses two variables `pemin` and `pemax` which store the minima and maxima projection values, respectively, and other two variables `imin` and `imax` which store the relative index of the pixels resulting in the maxima and minima projection values. Once the projection process is finalized for a group of ν pixels, another group is loaded. The process finalizes when all hyperspectral image pixels have been projected onto the skewer. Finally, the `d_res_partial` structure is updated with the minima and maxima projection values. This structure is reshaped into a matrix of pixel purity indices by simply counting the number of times that each pixel was selected as extreme during the process and updating its associated score with such number, thus producing a final structure `d_res_total` that stores the final values of $N_{PPI}(\mathbf{f}_i)$ for a given pixel $\mathbf{f}_i$. Additional `CUDA` kernels (not described here for space considerations) have been developed to perform the other steps involved in the PPI algorithm, thus leading to the selection of a final set of p endmembers $\{\mathbf{e}_j\}_{j=1}^{p}$.

[2] http://www.nvidia.com/object/cuda_home_new.html.

```
__global__ void PPI (float *d_image, float *d_random, int *d_res_partial,
int num_lines, int num_samples, int num_bands)
{

int idx = blockDim.x * blockIdx.x+threadIdx.x;
float pemax; // Maximum value of dot product
float pemin; // Minimum value of dot product
float pe;    // Scalar product

int v,d; int imax = 0; int imin = 0; pemax = MIN_INT; pemin = MAX_INT;

__shared__ float s_pixels[Tam_Vector]; float l_rand[224];

//Copy a skewer from GPU global memory to GPU registers
for (int k=0; k < num_bands; k++){
      l_rand[k] = d_random[idx*num_bands+k];
}

for (int it = 0; it < num_lines*num_samples/N_Pixels; it++){

      //Copy N_Pixels pixels to shared memory
      if (threadIdx.x < N_Pixels){
      for (int j=0; j<num_bands; j++){
            s_pixels[threadIdx.x+N_Pixels*j] =
            d_imagen[(it*N_Pixels+threadIdx.x)+(num_lines*num_samples*j)];
      }
}

__syncthreads();

//For each pixel
for (v=0; v < N_Pixels; v++){

      //Calculate dot product
      pe = 0;
      for (d = 0; d < num_bands; d++){
            pe = pe + l_rand[d]*s_pixels[v+N_Pixels*d];
      }

      //Calculate extreme values
      if (pe > pemax){
            imax = it*N_Pixels+v; pemax = pe;
      }

      if (pe < pemin){
            imin = it*N_Pixels+v; pemin = pe;
      }
}

//Update d_res_partial structure
d_res_partial[idx*2] = imax;
d_res_partial[idx*2+1] = imin;
}
```

Fig. 11. CUDA kernel PPI developed to implement the extreme projections step of the PPI endmember extraction algorithm on the GPU.

4.2. GPU implementation of abundance estimation

Our GPU version of the abundance estimation step is based on a CUDA kernel called ISRA, which implements the ANC-constrained abundance estimation procedure described in Section 2.2. The ISRA kernel is shown in Fig. 12. Here, d_image_vector is the structure that stores the hyperspectral image in the device (GPU) and d_image_unmixed is the final outcome of the process, i.e. the abundances associated to the p endmembers $\{\mathbf{e}_j\}_{j=1}^{p}$ derived by the PPI algorithm. These endmembers are stored in a structure called s_end. The kernel performs (in parallel) the calculations associated to Eq. (3) in iterative fashion for each pixel in the hyperspectral image, providing a set of positive abundances.

To conclude this section, we emphasize that our GPU implementation of the full unmixing chain (endmember extraction plus abundance estimation) has been carefully optimized taking into account the considered architecture (summarized in Fig. 8), including the global memory available, the local shared memory in each multiprocessor, and also the local cache memories. Whenever possible, we have accommodated blocks of pixels in small local memories in the GPU in order to guarantee very fast accesses, thus performing block-by-block processing to speed up the computations as much as possible. In the following, we analyze the performance of our parallel implementations in different architectures.

5. Experimental results

This section is organized as follows. In Section 5.1 we describe the hardware used in our experiments. Section 5.2 describes the hyperspectral data set that will be used for demonstration purposes. Section 5.3 evaluates the unmixing accuracy of the considered implementations. Section 5.4 inter-compares the parallel

```
__global__ void ISRA(float *d_image_vector, float *d_image_unmixed,
float *d_endmembers, float *d_endmembersT, int num_lines, int num_samples,
int num_bands, int N_END, int MAX_ITER)
{
// d_image_vector is the structure that stores the hyperspectral image in the device
// d_image_unmixed is the structure that stores the abundance estimation maps
// idx is the identification number of the thread

      int idx = blockDim.x * blockIdx.x + threadIdx.x;
      float s_end[NUM_BANDS]; float s_endt[NUM_END];
      float l_pixel[NUM_BANDS]; float l_abu[NUM_END];
      float numerator; float denominator; float dot = 0.0;

      //For all pixels
      for (int t=0; t<num_bands; t++){
            l_pixel[t]=d_image_vector[idx+(num_lines*num_samples*t)];
      }

      //Calculate abundances using ISRA for one pixel
      for (int it=0; it<=MAX_ITER; it++){
            numerator=0; denominator=0;

            //For all endmembers
            for (int e=0; e<N_END; e++){

                  //Read an endmember and store it in shared memory
                  if (threadIdx.x==0){
                        for (int i=0; i<num_bands; i++){
                        s_end[i]=d_endmembers[e*num_bands+i];}
                  }
                  syncthreads();

                  //For all bands
                  for (int k=0; k<num_bands; k++){
                        numerator=numerator+s_end[k]*l_pixel[k];

                        if (threadIdx.x==0){
                              for (int i=0; i<num_bands; i++){
                                    s_endt[i]=d_endmembersT[k*N_END+i];
                              }
                        }
                        syncthreads();

                        //Calculate dot product
                        for (int s=0; s<N_END; s++){
                              dot+=s_endt[s]*l_abu[s];
                        }
                        denominator+=dot*s_end[k];
                        dot=0;}

                  //Calculate a new abundance
                  l_abu[e]*=(numerator/denominator);
                  numerator=0; denominator=0;
            }
            //Store abundance in global memory
            for (int k=0; k<N_END; k++){
            d_image_unmixed[pixel+(num_lines*num_samples*k)]=l_abu[k];
      }
}
```

Fig. 12. CUDA kernel ISRA that computes endmember abundances in each pixel of the hyperspectral image.

performance of the FPGA and GPU implementations respectively described in Sections 3 and 4. Finally, Section 5.5 discusses the obtained results.

5.1. Hardware accelerators

The hardware architecture described in Section 3 has been implemented using VHDL language for the specification of the systolic array. Further, we have used the Xilinx ISE environment and the Embedded Development Kit (EDK) environment[3] to specify the complete system. The full system has been implemented on a low-cost reconfigurable board (XUPV2P type) with a single Virtex-II PRO XC2VP30 FPGA component, a DDR SDRAM DIMM memory slot with 2 GB of main memory, a RS232 port, and some additional components not used by our implementation (see Fig. 13). This FPGA has a total of 13,696 slices, 27,392 slice flip flops, and 27,392 four input LUTs available. In addition the FPGA includes some heterogeneous resources, such as two PowerPCs and distributed block RAMs. This is a rather old low-cost board but we have selected it for this study because it is similar to other FPGAs that have been certified by several international agencies for remote sensing applications.[4]

On the other hand, the implementation described in Section 4 has been tested on the NVidia Tesla C1060 GPU (see Fig. 14), which features 240 processor cores operating at 1.296 GHz, with single precision floating point performance of 933 Gflops, double precision floating point performance of 78 Gflops, total dedicated memory of 4 GB, 800 MHz memory (with 512-bit GDDR3 interface) and memory bandwidth of 102 GB/s. The GPU is connected

[3] http://www.xilinx.com/ise/embedded/edk_pstudio.htm.

[4] http://www.xilinx.com/publications/prod_mktg/virtex5qv-product-table.pdf.

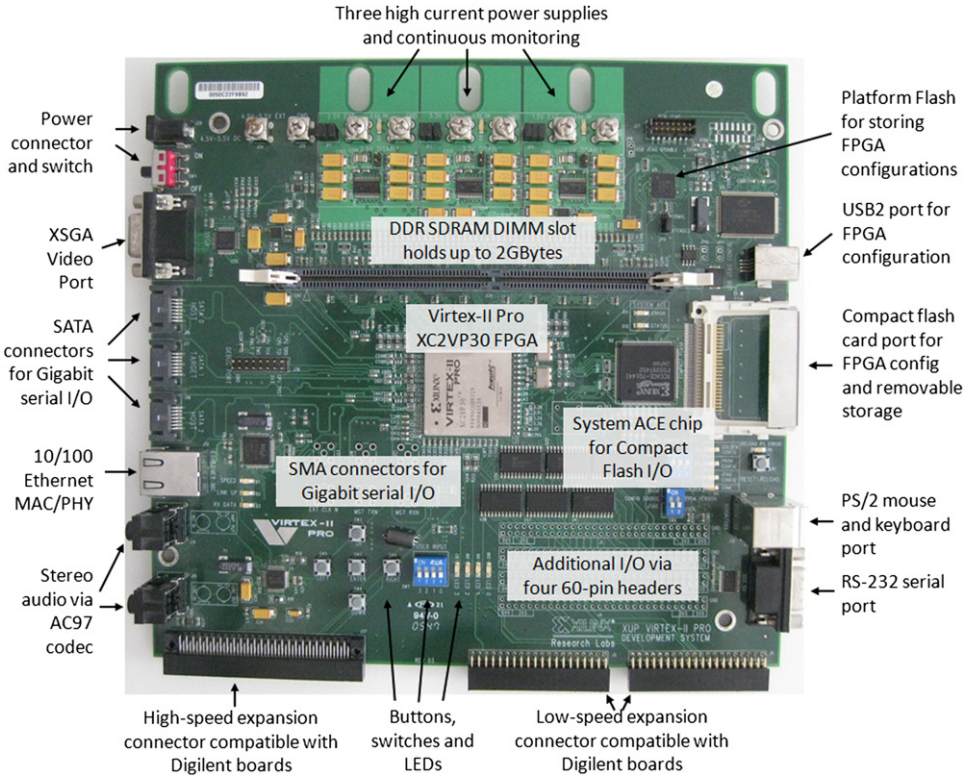


Fig. 13. Xilinx reconfigurable board XUPV2P (http://www.xilinx.com/univ/XUPV2P/Documentation/ug069.pdf).

Fig. 14. NVidia Tesla C1060 GPU (http://www.nvidia.com/object/product_tesla_c1060_us.html).

to an Intel core i7 920 CPU at 2.67 GHz with eight cores, which uses a motherboard Asus P6T7 WS SuperComputer.

5.2. Hyperspectral data

The hyperspectral data set used in experiments is the well-known AVIRIS Cuprite scene [see Fig. 15(a)], available online in reflectance units.[5] This scene has been widely used to validate the performance of endmember extraction and unmixing algorithms. The scene comprises a relatively large area (350 lines by 350 samples and 20-m pixels) and 224 spectral bands between 0.4 and 2.5 μm, with nominal spectral resolution of 10 nm. Bands 1–3, 105–115 and 150–170 were removed prior to the analysis due to water absorption and low SNR in those bands. The site is well understood mineralogically, and has several exposed minerals of interest including alunite, buddingtonite, calcite, kaolinite and muscovite. Reference ground signatures of the above minerals [see Fig. 15(b)], available in the form of a U.S. Geological Survey (USGS) library,[6] will be used to assess endmember signature purity in this work.

5.3. Unmixing accuracy

Before empirically investigating the parallel performance of the proposed implementations, we first evaluate their unmixing accuracy in the context of the considered application. Prior to a full examination and discussion of results, it is important to outline the parameter values used for the considered unmixing chain. In all the considered implementations, the number of endmembers to be extracted was set to $p=22$ after estimating the dimensionality of the data using the *virtual dimensionality* concept [36]. In addition, the number of skewers was set to $K=10^4$ (although values of $K=10^3$ and $K=10^5$ were also tested, we experimentally observed that the use of $K=10^3$ resulted in the loss of important endmembers, while the endmembers obtained using $K=10^5$ were essentially the same as those found using $K=10^4$). The threshold angle parameter was set to $\nu_a = 10^\circ$, which is a reasonable limit of tolerance for this metric, while the cut-off threshold value parameter ν_c was set to the mean of N_{PPI} scores obtained after $K=10^4$ iterations. These parameter values are in agreement with those used before in the literature [21]. The number of iterations for the ISRA algorithm for abundance estimation was set to 50 after empirically observing that the root mean square error (RMSE) between the original and the

[5] http://aviris.jpl.nasa.gov/html/aviris.freedata.html.

[6] http://speclab.cr.usgs.gov/spectral-lib.html.

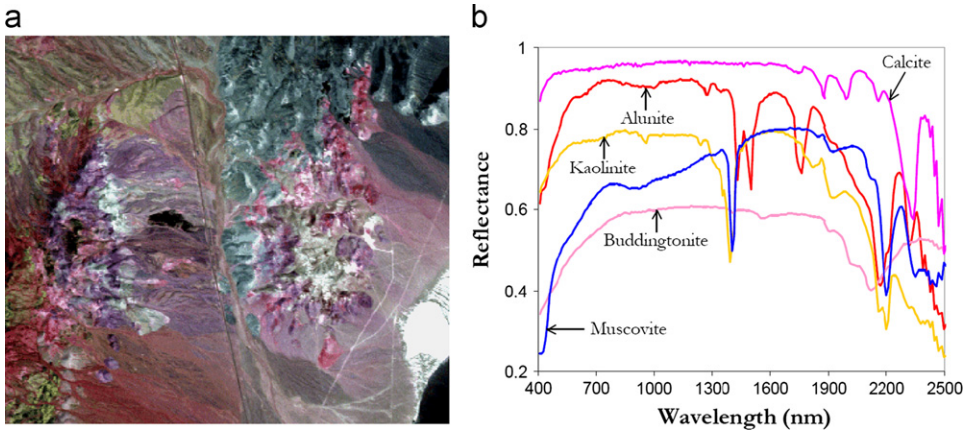


Fig. 15. (a) False color composition of the AVIRIS hyperspectral image collected over the Cuprite mining district in Nevada. (b) U.S. Geological Survey mineral spectral signatures used for validation purposes. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this article.)

Table 1
Spectral angle scores (in degrees) between the endmembers extracted from the AVIRIS Cuprite hyperspectral image by different implementations and some selected USGS reference signatures.

USGS mineral	ENVI software (deg.)	C++ implementation (deg.)	FPGA implementation (deg.)	GPU implementation (deg.)
Alunite	4.81	4.81	4.81	4.81
Buddingtonite	4.07	4.06	4.06	4.06
Calcite	5.09	5.09	5.09	5.09
Kaolinite	7.72	7.67	7.67	7.67
Muscovite	5.27	6.47	6.47	6.47

reconstructed hyperspectral image (using the $p=22$ endmembers derived by PPI and their per-pixel abundances estimated by ISRA) was very low, and further decreased very slowly for a higher number of iterations.

Table 1 shows the SA between the most similar endmembers detected by the original implementation of the chain available in ENVI 4.5, an optimized implementation of the same chain described in Section 2 (coded using the C++ programming language), and the proposed FPGA and GPU-based implementations. The SA between an endmember $\mathbf{e}_j$ selected by the PPI and a reference spectral signature $\mathbf{s}_i$ is given by

$$\mathrm{SA}(\mathbf{e}_j,\mathbf{s}_i)=\cos^{-1}\frac{\mathbf{e}_j\cdot\mathbf{s}_i}{\|\mathbf{e}_j\|\cdot\|\mathbf{s}_i\|}. \quad (4)$$

It should be noted that Table 1 only reports the SA scores associated to the most similar spectral endmember with regards to its corresponding USGS signature. Smaller SA values indicate higher spectral similarity. As shown in Table 1, the endmembers found by our parallel implementations were exactly the same as the ones found by the serial implementation of the processing chain implemented in C++. However, our parallel implementations produced slightly different endmembers than those found by ENVI's implementation. In any event, we experimentally tested that the SA scores between the endmembers that were different between the ENVI and the parallel algorithms were always very low (below 0.85°), a fact that reveals that the final endmember sets were almost identical in the spectral sense. Finally, it is worth noting that an evaluation of abundance estimation in real analysis scenarios is very difficult due to the lack of ground-truth to quantitatively substantiate the obtained results. However, we qualitatively observed that the abundance maps derived by ISRA for the $p=22$ endmembers derived by PPI were in good agreement with those derived by other methods in previous work (see for instance [24,37,38]). These maps are not displayed here for space considerations.

5.4. Parallel performance

5.4.1. Endmember extraction

Table 2 shows the resources used for our FPGA implementation of the PPI-based endmember extraction process for different numbers of skewers (ranging from $K=10$ to $K=100$), tested on the Virtex-II PRO xc2vp30 FPGA of the XUPV2P board. As shown in Table 2, we can scale our design up to 100 skewers (therefore, $P=100$ algorithm passes are needed in order to process $K=10^4$ skewers). In our design the clock cycle remains constant at 187 MHz. It should be noted that, in the current implementation, the complete AVIRIS hyperspectral image is stored in an external DDR2 SDRAM. However, with an appropriate controller, other options could be supported, such as using flash memory to store the hyperspectral data.

In previous FPGA designs of the PPI algorithm [15,31], the module for random generation of skewers was situated in an external processor. Hence, frequent communications between the host (CPU) and the device (FPGA) were needed. However, in our implementation the hardware random generation module is implemented internally in the FPGA board. This approach significantly reduced the communications, leading to increased parallel performance. To further reduce the communication overheads we have included a DMA and applied a prefetching approach in order to hide the communication latency. Basically, while the systolic array is processing a set of data, the DMA is fetching the following set, and storing it in the FIFO. Having in mind the proposed optimization concerning the use of available resources, it is important to find a balance between the number of DMA operations and the capacity of the destination FIFO. In other words, we need to fit enough information in the FIFO so that the systolic array never needs to stop. In addition, the greater the FIFO capacity, the fewer DMA operations will be required. We have evaluated several FIFO sizes and identified that, for 1024 positions or more, there are no penalties due to reading of the input data. With the above considerations in mind, we emphasize that our

Table 2
Summary of resource utilization for the FPGA-based implementation of endmember extraction from the AVIRIS Cuprite hyperspectral image.

Number of skewers	Number of slice flip flops	Number of four input LUTs	Number of slices	Percentage of total	Maximum operation frequency (MHz)
10	1120	1933	1043	7.61	187
20	2240	3865	2085	15.22	187
30	3360	5796	3127	22.83	187
40	4480	7728	4170	30.44	187
50	5600	9659	5212	38.05	187
60	6720	11,591	6254	45.66	187
70	7840	13,522	7296	53.27	187
80	8960	15,454	8339	60.88	187
90	10,080	17,385	9381	68.49	187
100	11,200	19,317	10,423	76.1	187

implementation in the considered FPGA board was able to achieve a total processing time for the considered AVIRIS scene of 31.23 s. Compared with a recent FPGA-based implementation using the same hyperspectral scene and presented in [39], our FPGA implementation of the PPI shows a significant increase in performance, with a speedup of 2 with regard to that implementation. We must consider that the FPGA used in [39] has 2.5 times more slices than the one used in our implementation of the PPI algorithm. This result is far from real-time endmember extraction, since the cross-track line scan time in AVIRIS, a push-broom instrument, is quite fast (8.3 ms to collect 512 full pixel vectors). This introduces the need to process the considered scene (350×350 pixels) in approximately 2 s in order to fully achieve real-time performance. In this regard, it is worth noting that we used the internal clock of 100 MHz (the maximum frequency for the XUPV2P board) for the execution instead of any external clock. Therefore, if faster FPGAs are certified for aerospace use, it would be straightforward to achieve a 1.9 speedup with regards to our current implementation. Moreover, as FPGA technology has greatly improved in recent years, soon larger FPGAs will be ready for aerospace applications, and since our design is fully scalable we could also achieve important speedups. Theoretically, if we scale our design to an FPGA with 10 times more area using a 187 MHz clock it should be possible to fully achieve real-time endmember extraction performance. These FPGAs are already available in the market although they have not been yet been certified for aerospace applications.

On the other hand, the execution time for the endmember extraction stage implemented on the NVidia C1060 Tesla GPU was 17.59 s (closer to real-time performance than the FPGA implementation but still far). In our experiments on the Intel Core i7 920 CPU, the C++ implementation took 3454.53 s to process the considered AVIRIS scene (using just one of the available cores). This means that the speedup achieved by our GPU implementation with regards to the serial implementation (using just one of the available cores) was approximately 196.39.

For illustrative purposes, Fig. 16 shows the percentage of the total execution time consumed by the `PPI` kernel described in Fig. 11, which implements the *extreme projections* step of the unmixing chain, and by the `RandomGPU` kernel, which implements the *skewer generation* step in the chain. Fig. 16 also displays the number of times that each kernel was invoked (in the parentheses). These values were obtained after profiling the implementation using the `CUDA` Visual Profiler tool.[7] In the figure, the percentage of time for data movements from host (CPU) to device (GPU) and from device to host are also displayed. It should be noted that two data movements from host to device are needed to transfer the original hyperspectral image and the skewers to the GPU, while only one movement from device to host (final result) is needed. As shown by Fig. 16, the `PPI` kernel consumes about 99% of the total GPU time devoted to endmember extraction from the AVIRIS Cuprite image, while the `RandomGPU` kernel comparatively occupies a much smaller fraction. Finally, the data movement operations are not significant, which indicates that most of the GPU processing time is invested in the most time-consuming operation, i.e. the calculation of skewer projections and identification of maxima and minima projection values leading to endmember identification.

5.4.2. Abundance estimation

Table 3 shows the resources used for our hardware implementation of the proposed ISRA design for different values of the ISRA basic units that can operate in parallel, u. As shown in Table 3, our design can be scaled up to $u=10$ ISRA basic units in parallel. An interesting behavior of the proposed parallel ISRA module is that it can be scaled without significant increase in the critical path delay (the clock frequency remains almost constant). For the maximum number of basic units used in our experiments ($u=10$), the percentage of total hardware utilization is 90.07% thus reaching almost full hardware occupancy. However, other values of u tested in our experiments, e.g. $u=7$, still leave room in the FPGA for additional algorithms. In any case, this value could be significantly increased in more recent FPGA boards. For example, in a Virtex-4 FPGA XQR4VLX200 (89,088 slices) certified for space, we could have 6.5 times more ISRA units in parallel simply by synthesizing the second stage of the hardware architecture for the new number of units in parallel. In this way, we could achieve a speedup of 6.5 without modifications in our proposed design. In the case of an airborne platform without the need for space-certified hardware, we could use a Virtex-6 XQ6VLX550T (550,000 logics cells) which has nearly 40 times more logic cells than the FPGA used in our experiments.

In our current implementation, the processing time of ISRA for 50 iterations was 1303.1 s in the FPGA version (which is far away from real-time abundance estimation performance) and 47.84 min in the CPU version, which has been extensively fine-tuned using autovectorization capabilities and optimization flags. This resulted in a speedup of 2.2. We emphasize that other FPGA implementations of ISRA are available in the literature [40,41]. However, the authors indicate that their full FPGA design and implementation (tested on a Virtex II Pro FPGA) required more execution time than the serial implementation presented in [42]. This was due to data transmission and I/O bottlenecks. Since our approach optimizes I/O significantly, a comparison with regards to the approach in [40,41] was not deemed suitable.

Regarding the execution time of the abundance estimation step in the GPU, we measured a processing time of 24.37 s in the NVidia C1060 Tesla GPU. This means that the speedup achieved by the GPU implementation with regards to the Intel Core i7 920 CPU implementation (using just one of the available cores) was approximately 60.81. In another study focused on GPU implementation of ISRA [43], the execution time obtained after running the algorithm with 50 iterations for a smaller hyperspectral image collected also by AVIRIS (132 samples by 167 lines and 224 spectral bands, for a total size of 9.73 MB) was much higher (205 s and about 2.67 speedup with regards to the serial version). Although our results are more encouraging in terms of speedup, it should be noted that the times measured in [43] were reported for an older NVidia GPU, the 8800 GTX with 128 cores operating at 1.33 GHz. Note also that the difference in the speedup achieved in the NVidia C1060 Tesla GPU with regards to the speedup

[7] http://developer.nvidia.com/object/cuda_3_1_downloads.html.

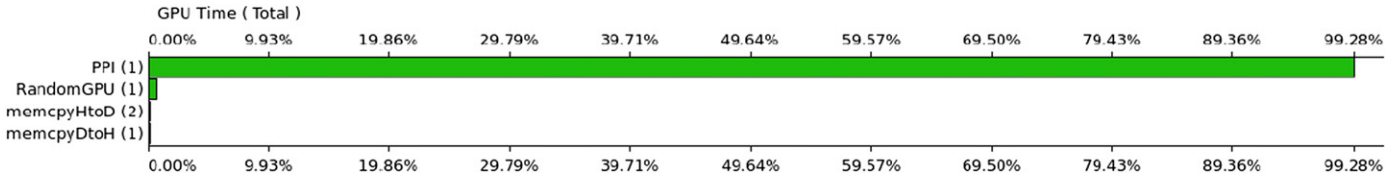


Fig. 16. Summary plot describing the percentage of the total GPU time consumed by the endmember extraction step of the unmixing chain on the NVidia Tesla C1060 GPU.

Table 3
Summary of resource utilization for the FPGA-based implementation of ISRA for different numbers of modules in parallel.

Component	Number of modules (u)	Number of MULT18X18s	Number of slice flip flops	Number of four input LUTs	Number of slices	Percentage of total	Maximum operation frequency (MHz)
Parallel ISRA module	4	32	1334	5410	3084	22.51	44.7
	5	40	2001	8115	4626	33.77	44.5
	6	48	2668	10,820	6168	45.03	44.4
	7	56	3335	13,525	7710	56.29	44.3
	8	64	4002	16,230	9252	67.55	44.1
	9	72	4669	18,935	10,794	78.81	43.9
	10	80	5336	21,640	12,336	90.07	43.8
RS232 transmitter	–	0	69	128	71	0.28	208
DMA controller	–	0	170	531	367	1.45	102

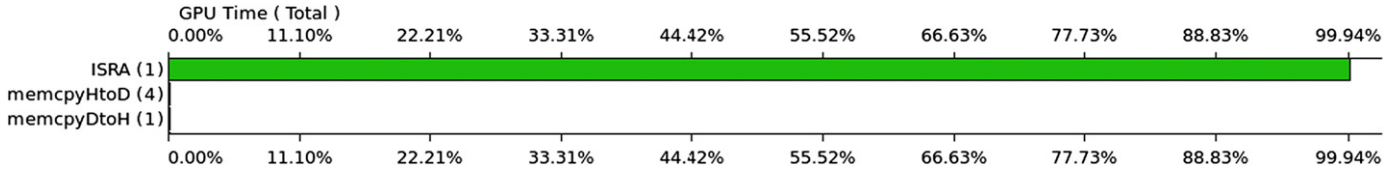


Fig. 17. Summary plot describing the percentage of the total GPU time consumed by the abundance estimation step of the unmixing chain on the NVidia Tesla C1060 GPU.

reported for the Virtex-II PRO XC2VP30 FPGA. This is mainly because the ISRA algorithm relies on iterative multiplication operations which cannot be implemented in this small FPGA as efficiently as in the GPU. For illustrative purposes, Fig. 17 shows the percentage of the total execution time employed by each of the `CUDA` kernels obtained after profiling the abundance estimation step for the AVIRIS Cuprite scene in the NVidia Tesla C1060 architecture. The first bar represents the percentage of time employed by the `ISRA` kernel, while the second and third bars respectively denote the percentage of time for data movements from host (CPU) to device (GPU) and from device to host. The reason why there are four data movements from host to device is because not only the hyperspectral image, but also the endmember signatures, the number of endmembers, and the structure where the abundances are stored need to be forwarded to the GPU, which returns the structure with the estimated abundances after the calculation is completed. As shown in Fig. 17, the `ISRA` kernel dominates the computation and consumes more than 99% of the GPU time.

5.5. Discussion and result comparison

The performance results obtained by our two proposed implementations of the full spectral unmixing chain are quite satisfactory, achieving important speedups when compared to the results obtained by a high performance CPU. This demonstrates that both options represent promising technologies for aerospace applications, although the number of GPU platforms which have been certified for space operation is still very limited as compared to FPGAs. Although in our experiments the GPU implementation clearly achieved the best performance results (17.59+24.37 s for unmixing (endmember extraction + abundance estimation) the considered hyperspectral data set), it should be noted that the FPGA implementation is using an older technology. As a result, the performance results reported for the FPGA (31.23+1303.1 s for unmixing the same scene) are also reasonable.

Regarding the platform cost, the cost of the FPGA used is just 300$ (although an equivalent board certified for aerospace applications may be significantly more expensive), whereas the cost of the GPU is around 1000$. Hence, both architectures are very affordable solutions for aerospace applications. Another important feature to be discussed is the scalability of the provided solutions. In this case, the FPGA implementation has been carried out in order to be fully scalable as it can work in parallel with any number of skewers as long as enough hardware area is available. In the case of the GPU implementation, the developed code can also be made scalable but it is certainly more dependent on the considered GPU platform, hence its portability to other GPU platforms is more challenging than in the case of the FPGA implementation, which could be easily adapted to other boards.

Flexibility is also an important parameter to be discussed in aerospace applications, since the available space is very restricted. In this regard, both platforms offer a low-weight solution compliant with mission payload requirements, although power consumption is higher in the GPU solution than in the FPGA solution. In both cases the solutions can be easily reused to deal with other different processing problems. In the GPU case, this is as simple as executing a different code, whereas in the FPGA case this can be achieved by reconfiguring the platform.

Finally it is also important to discuss the design effort needed in both cases. Compared to a C++ code developed for conventional CPUs, both solutions require additional design efforts specially since designers must learn different design paradigm and development

environments, and also take into account several implementation low-level details. After comparing the design effort complexity of the two options that we have implemented, we believe that probably FPGA design is a little bit more complex than GPU design due to two main reasons. On the one hand, in the FPGA implementation the platform needs to be designed whereas in the GPU implementation the platform just needs to be used. On the other hand, for large FPGA designs hardware debugging is a complex issue. However, we believe that, in both cases, the performance achievements are more significant than the increase in the design complexity.

6. Conclusions and future research lines

Through the detailed analysis of a representative spectral unmixing chain, we have illustrated the increase in computational performance that can be achieved by incorporating hardware accelerators to hyperspectral image processing problems. A major advantage of the incorporation of such hardware accelerators aboard airborne and satellite platforms for Earth observation is to overcome an existing limitation in many remote sensing and observatory systems: the bottleneck introduced by the bandwidth of the down-link connection from the observatory platform. Experimental results demonstrate that our hardware implementations make appropriate use of computing resources in the considered FPGA and GPU architectures, and further provide significant speedups using only one hardware device as opposed to previous efforts using clusters, and with fewer on-board restrictions in terms of cost, power consumption and size, which are important when defining mission payload in remote sensing missions (defined as the maximum load allowed in the airborne or satellite platform that carries the imaging instrument). Although the response times measured are not strictly in real-time, they are still believed to be acceptable in most remote sensing applications. In addition, the reconfigurability of FPGA systems (on the one hand) and the low cost of GPU systems (on the other) open many innovative perspectives from an application point of view, ranging from the appealing possibility of being able to adaptively select one out of a pool of available data processing algorithms (which could be applied on the fly aboard the airborne/satellite platform, or even from a control station on Earth), to the possibility of providing a response in applications with real-time constraints. It should be noted that on-board processing is not an absolute requirement for our proposed developments, which could also be used in on-ground processing systems.

Although the experimental results presented in this work are encouraging, further work is still needed to optimize the proposed implementation and fully achieve real-time performance. Radiation-tolerance and power consumption issues for these hardware accelerators should be explored in more detail in future developments. Optimal parallel designs and implementations are still needed for other more sophisticated processing algorithms that have been used extensively in the hyperspectral remote sensing community, such as advanced supervised classifiers (e.g. support vector machines) which do not exhibit regular patterns of computation and communication [4]. Future work will also be focused on optimizing the ISRA abundance estimation algorithm discussed in this work on FPGA platforms and we will also study other abundance estimation algorithms, such as the FCLSU, to check if they are more suitable for FPGA implementation.

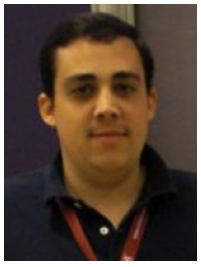
Acknowledgments

This work has been supported by the European Community's Marie Curie Research Training Networks Programme under reference MRTN-CT-2006-035927, Hyperspectral Imaging Network (HYPER-I-NET). This work has also been supported by the Spanish Ministry of Science and Innovation (HYPERCOMP/EODIX project, reference AYA2008-05965-C04-02), AYA2009-13300-C03-02 y TIN2009-09806.

References

[1] A.F.H. Goetz, G. Vane, J.E. Solomon, B.N. Rock, Imaging spectrometry for Earth remote sensing, Science 228 (1985) 1147–1153.
[2] R.O. Green, et al., Imaging spectroscopy and the airborne visible/infrared imaging spectrometer (AVIRIS), Remote Sensing of Environment 65 (1998) 227–248.
[3] A. Plaza, Special issue on architectures and techniques for real-time processing of remotely sensed images, Journal of Real-Time Image Processing 4 (2009) 191–193.
[4] A. Plaza, J.A. Benediktsson, J. Boardman, J. Brazile, L. Bruzzone, G. Camps-Valls, J. Chanussot, M. Fauvel, P. Gamba, J. Gualtieri, M. Marconcini, J.C. Tilton, G. Trianni, Recent advances in techniques for hyperspectral image processing, Remote Sensing of Environment 113 (2009) 110–122.
[5] A. Plaza, C.-I. Chang, High Performance Computing in Remote Sensing, Taylor & Francis, Boca Raton, FL, 2007.
[6] A. Plaza, D. Valencia, J. Plaza, P. Martinez, Commodity cluster-based parallel processing of hyperspectral Imagery, Journal of Parallel and Distributed Computing 66 (2006) 345–358.
[7] A. Plaza, D. Valencia, J. Plaza, An experimental comparison of parallel algorithms for hyperspectral analysis using homogeneous and heterogeneous networks of workstations, Parallel Computing 34 (2008) 92–114.
[8] Q. Du, R. Nekovei, Fast real-time onboard processing of hyperspectral imagery for detection and classification, Journal of Real-Time Image Processing 22 (2009) 438–448.
[9] S. Hauck, The roles of FPGAs in reprogrammable systems, Proceedings of the IEEE 86 (1998) 615–638.
[10] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA Tesla: a unified graphics and computing architecture, IEEE Micro 28 (2008) 39–55.
[11] U. Thomas, D. Rosenbaum, F. Kurz, S. Suri, P. Reinartz, A new software/hardware architecture for real time image processing of wide area airborne camera images, Journal of Real-Time Image Processing 5 (2009) 229–244.
[12] C. Gonzalez, J. Resano, D. Mozos, A. Plaza, D. Valencia, FPGA implementation of the pixel purity index algorithm for remotely sensed hyperspectral image analysis, EURASIP Journal on Advances in Signal Processing 969806 (2010) 1–13.
[13] J. Theiler, J. Frigo, M. Gokhale, J.J. Szymanski, Co-design of software and hardware to implement remote sensing algorithms, Proceedings of SPIE 4480 (2001) 200–210.
[14] A. Paz, A. Plaza, Clusters versus GPUs for parallel automatic target detection in remotely sensed hyperspectral images, EURASIP Journal on Advances in Signal Processing 915639 (2010) 1–18.
[15] A. Plaza, C.-I. Chang, Clusters versus FPGA for parallel processing of hyperspectral imagery, International Journal of High Performance Computing Applications 22 (2008) 366–385.
[16] E. El-Araby, T. El-Ghazawi, J.L. Moigne, R. Irish, Reconfigurable processing for satellite on-board automatic cloud cover assessment, Journal of Real-Time Image Processing 5 (2009) 245–259.
[17] J. Setoain, M. Prieto, C. Tenllado, F. Tirado, GPU for parallel on-board hyperspectral image processing, International Journal of High Performance Computing Applications 22 (2008) 424–437.
[18] Y. Tarabalka, T.V. Haavardsholm, I. Kasen, T. Skauli, Real-time anomaly detection in hyperspectral images using multivariate normal mixture models and GPU processing, Journal of Real-Time Image Processing 4 (2009) 1–14.
[19] J.B. Adams, M.O. Smith, P.E. Johnson, Spectral mixture modeling: a new analysis of rock and soil types at the Viking Lander 1 site, Journal of Geophysical Research 91 (1986) 8098–8112.
[20] N. Keshava, J.F. Mustard, Spectral unmixing, IEEE Signal Processing Magazine 19 (2002) 44–57.
[21] A. Plaza, P. Martinez, R. Perez, J. Plaza, A quantitative and comparative analysis of endmember extraction algorithms from hyperspectral data, IEEE Transactions on Geoscience and Remote Sensing 42 (2004) 650–663.
[22] Q. Du, N. Raksuntorn, N.H. Younan, R.L. King, End-member extraction for hyperspectral image analysis, Applied Optics 47 (2008) 77–84.
[23] D. Heinz, C.-I. Chang, Fully constrained least squares linear mixture analysis for material quantification in hyperspectral imagery, IEEE Transactions on Geoscience and Remote Sensing 39 (2000) 529–545.
[24] C.-I. Chang, Hyperspectral Data Exploitation: Theory and Applications, John Wiley & Sons, New York, 2007.
[25] J.W. Boardman, F.A. Kruse, R.O. Green, Mapping target signatures via partial unmixing of aviris data, in: Proceedings of the JPL Airborne Earth Science Workshop, 1995, pp. 23–26.
[26] M.E. Daube-Witherspoon, G. Muehllehner, An iterative image space reconstruction algorithm suitable for volume ECT, IEEE Transactions on Medical Imaging 5 (1985) 61–66.
[27] C.-I. Chang, Hyperspectral Imaging: Techniques for Spectral Detection and Classification, Kluwer Academic/Plenum Publishers, New York, 2003.

[28] D. Lavernier, E. Fabiani, S. Derrien, C. Wagner, Systolic array for computing the pixel purity index algorithm on hyperspectral images, Proceedings of SPIE 4480 (1999) 130–138.
[29] D. Lavernier, J. Theiler, J. Szymanski, M. Gokhale, J. Frigo, FPGA implementation of the pixel purity index algorithm, Proceedings of SPIE 4693 (2002) 30–41.
[30] Wildfore Reference Manual, Technical Report, Revision 3.4, Annapolis Micro System Inc., 1999.
[31] M. Hsueh, C.-I. Chang, Field programmable gate arrays (FPGA) for pixel purity index using blocks of skewers for endmember extraction in hyperspectral imagery, International Journal of High Performance Computing Applications 22 (2008) 408–423.
[32] C. Gonzalez, J. Resano, A. Plaza, D. Mozos, FPGA implementation of abundance estimation for spectral unmixing of hyperspectral data using the image space reconstruction algorithm, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 5 (2012) 248–261.
[33] J. Setoain, M. Prieto, C. Tenllado, A. Plaza, F. Tirado, Parallel morphological endmember extraction using commodity graphics hardware, IEEE Geoscience and Remote Sensing Letters 43 (2007) 441–445.
[34] S. Sanchez, A. Paz, G. Martin, A. Plaza, Parallel unmixing of remotely sensed hyperspectral images on commodity graphics processing units, Concurrency and Computation: Practice and Experience 23 (2011) 1538–1557.
[35] M. Matsumoto, T. Nishimura, Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator, ACM Transactions on Modeling and Computer Simulation 8 (1998) 3–30.
[36] Q. Du, C.-I. Chang, Estimation of number of spectrally distinct signal sources in hyperspectral imagery, IEEE Transactions on Geoscience and Remote Sensing 42 (2004) 608–619.
[37] M.E. Winter, N-FINDR: an algorithm for fast autonomous spectral endmember determination in hyperspectral data, Proceedings of SPIE Image Spectrometry V 3753 (2003) 266–277.
[38] J.M.P. Nascimento, J.M. Bioucas-Dias, Vertex component analysis: a fast algorithm to unmix hyperspectral data, IEEE Transactions on Geoscience and Remote Sensing 43 (2005) 898–910.
[39] D. Valencia, A. Plaza, M.A. Vega-Rodriguez, R.M. Perez, FPGA design and implementation of a fast pixel purity index algorithm for endmember extraction in hyperspectral imagery, Chemical and Biological Standoff Detection III, Proceedings of SPIE 5995 (2006) 69–78.
[40] J. Morales, N. Medero, N.G. Santiago, J. Sosa, Hardware implementation of image space reconstruction algorithm using FPGAs, in: Proceedings of the IEEE International Midwest Symposium on Circuits and Systems, vol. 1, 2006, pp. 433–436.
[41] J. Morales, N.G. Santiago, A. Morales, An FPGA implementation of image space reconstruction algorithm for hyperspectral imaging analysis, Proceedings of SPIE 6565 (2007) 1–18.
[42] S. Rosario, Iterative Algorithms for Abundance Estimation on Unmixing of Hyperspectral Imagery, Master Thesis, University of Puerto Rico, Mayaguez, 2004.
[43] D. Gonzalez, C. Sanchez, R. Veguilla, N.G. Santiago, S. Rosario-Torres, M. Velez-Reyes, Abundance estimation algorithms using NVIDIA CUDA technology, Proceedings of SPIE 6966 (2008) 1–9.



Carlos González received the M.Sc. degree in 2008 and is currently a Teaching Assistant in the Department of Computer Architecture and Automatics, Complutense University of Madrid, Spain. In his work he mainly focuses on Applying run-time reconfiguration in aerospace applications. He has recently started with this topic, working with algorithms that deal with hyperspectral images. He is also interested in the acceleration of artificial intelligence algorithms applied to games. He won the Design Competition of the IEEE International Conference on Field Programmable Technology in 2009 (FPT'09) and in 2010 (FPT'10).

Sergio Sánchez received the M.Sc. degree in 2010 and is currently a Research Associate with the Hyperspectral Computing Laboratory, Department of Technology of Computers and Communications, University of Extremadura, Spain. His main research interests comprise hyperspectral image analysis and efficient implementations of large-scale scientific problems on commodity graphical processing units (GPUs).

Abel Paz received the M.Sc. degree in 2007 and is currently a staff member of Bull Spain working in the Center for Advanced Technologies of Extremadura (CETA), and also a Research Associate with the Hyperspectral Computing Laboratory, Department of Technology of Computers and Communications, University of Extremadura, Spain. His main research interests comprise hyperspectral image analysis and efficient implementations of large-scale scientific problems on commodity Beowulf clusters, heterogeneous networks of computers and grids, and specialized computer architectures such as clusters of graphical processing units (GPUs).

Javier Resano received the Bachelor Degree in Physics in 1997, a Master Degree in Computer Science in 1999, and the Ph.D. degree in 2005 at the Universidad Complutense of Madrid, Spain. Currently he is Associate Professor at the Computer Eng. Department of the Universidad of Zaragoza, and he is a member of the GHADIR research group, from Universidad Complutense, and the GAZ research group, from Universidad de Zaragoza. He is also member of the Engineering Research Institute of Aragon (I3A). His research has been focused in hardware/software co-design, task scheduling techniques, Dynamically Reconfigurable Hardware and FPGA design. His FPGA designs have received several international awards including the first prize in the Design Competition of the IEEE International Conference on Field Programmable Technology in 2009 (FPT'09) and in 2010 (FPT'10).

Daniel Mozos is a permanent professor in the Department of Computer Architecture and Automatics of the Complutense University of Madrid, where he leads the GHADIR research group on dynamically reconfigurable architectures. His research interests include design automation, computer architecture, and reconfigurable computing. Mozos has a B.S. in physics and a Ph.D. in computer science from the Complutense University of Madrid.

Antonio Plaza received the M.Sc. degree in 1999, and the Ph.D. degree in 2002, all in Computer Engineering. Dr. Plaza is the Head of the Hyperspectral Computing Laboratory and an Associate Professor with the Department of Technology of Computers and Communications, University of Extremadura, Spain. His main research interests comprise hyperspectral image analysis, signal processing, and efficient implementations of large-scale scientific problems on high performance computing architectures, including commodity Beowulf clusters, heterogeneous networks of computers and grids, and specialized computer architectures such as field programmable gate arrays (FPGAs) or graphical processing units (GPUs). He is an Associate Editor for the IEEE Transactions on Geoscience and Remote Sensing journal in the areas of Hyperspectral Image Analysis and Signal Processing. He is also an Associate Editor for the Journal of Real-Time Image Processing. He is the project coordinator of the Hyperspectral Imaging Network, a four-year Marie Curie Research Training Network designed to build an interdisciplinary European research community focused on hyperspectral imaging activities. He is also a member of the Management Committee and coordinator of a working group in the Open European Network for High Performance Computing on Complex Environments, funded by the European Cooperation in Science and Technology programme. Dr. Plaza is a Senior Member of IEEE, and received the recognition of Best Reviewers of the IEEE Geoscience and Remote Sensing Letters journal in 2009, a Top Cited Article award of Elsevier's Journal of Parallel and Distributed Computing in 2005–2010, and the 2008 Best Paper award at the IEEE Symposium on Signal Processing and Information Technology. Additional information about the research activities of Dr. Plaza is available at http://www.umbc.edu/rssipl/people/aplaza.