J Sign Process Syst (2010) 61:293–315
DOI 10.1007/s11265-010-0453-1

Improving the Performance of Hyperspectral Image and Signal Processing Algorithms Using Parallel, Distributed and Specialized Hardware-Based Systems

Antonio Plaza · Javier Plaza · Hugo Vegas

Received: 17 November 2009 / Revised: 5 January 2010 / Accepted: 5 January 2010 / Published online: 21 January 2010

Abstract Advances in sensor technology are revolutionizing the way remotely sensed data is collected, managed and analyzed. The incorporation of latest-generation sensors to airborne and satellite platforms is currently producing a nearly continual stream of high-dimensional data, and this explosion in the amount of collected information has rapidly created new processing challenges. For instance, hyperspectral signal processing is a new technique in remote sensing that generates hundreds of spectral bands at different wavelength channels for the same area on the surface of the Earth. Many current and future applications of remote sensing in Earth science, space science, and soon in exploration science will require (near) real-time processing capabilities. In recent years, several efforts have been directed towards the incorporation of high-performance computing (HPC) systems and architectures in remote sensing missions. With the aim of providing an overview of current and new trends in parallel and distributed systems for remote sensing applications, this paper explores three HPC-based paradigms for efficient implementation of the Pixel Purity Index (PPI) algorithm, available from the popular Kodak's Research Systems ENVI software package, as a representative case study for demonstration purposes. Several different parallel programming techniques are used to improve the performance of the PPI on a variety of parallel platforms, including a set of message passing interface (MPI)-based implementations on a massively parallel Beowulf cluster at NASA's Goddard Space Flight Center in Maryland and on a variety of heterogeneous networks of workstations at University of Maryland; a Handel-C implementation of the algorithm on a Virtex-II field programmable gate array (FPGA); and a compute unified device architecture (CUDA)-based implementation on graphical processing units (GPUs) of NVidia. Combined, these parts deliver an excellent snapshot of the state-of-the-art in those areas, and offer a thoughtful perspective on the potential and emerging challenges of adapting HPC systems to remote sensing problems.

Keywords Parallel systems · Hyperspectral imaging · Cluster computer systems · Heterogeneous parallel systems · FPGAs · GPUs

A. Plaza (✉) · J. Plaza
Department of Technology of Computers and Communications, University of Extremadura, Avda. de la Universidad s/n, 10071 Caceres, Spain
e-mail: aplaza@unex.es

J. Plaza
e-mail: jplaza@unex.es

H. Vegas
ArTeCS Group, Department of Computer Architecture, Complutense University, Ciudad Universitaria, 28040 Madrid, Spain
e-mail: hugovegas@fdi.ucm.es

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]. This new technique has gained tremendous popularity in recent years. Advances in sensor technology have led to the development of so-called hyperspectral instruments, which are capable of collecting hundreds

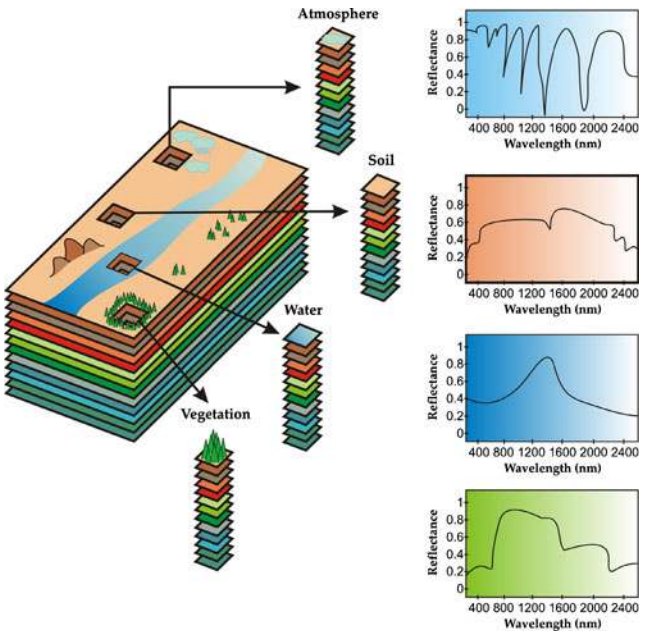


Figure 1 The concept of hyperspectral imaging.

of images corresponding to different wavelength channels for the same area on the surface of the Earth [2]. For instance, NASA is continuously gathering imagery data with hyperspectral Earth-observing sensors such as the Jet Propulsion Laboratory's Airborne Visible-Infrared Imaging Spectrometer (AVIRIS) [3], 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 to 12 km wide and several kilometers long using 224 spectral bands. The resulting hyperspectral data cube [4] is a stack of images (see Fig. 1) in which each pixel (vector) has an associated spectral signature or 'fingerprint' (signal) that uniquely characterizes the underlying objects, and the resulting data volume typically comprises several GBs per flight.

The extremely high computational requirements already introduced by hyperspectral imaging applications (and the fact that these systems will continue increasing their spatial and spectral resolutions in the near future) make them an excellent case study to illustrate the need for high performance computing (HPC) systems for image processing [5, 6], and remote sensing applications [7–9]. In particular, the development of computationally efficient techniques for transforming the massive amount of hyperspectral data collected on a daily basis into scientific understanding is critical for space-based Earth science and planetary exploration [10–12]. The wealth of spatial and spectral information provided by last-generation hyperspectral instruments has opened ground-breaking perspectives in many applications, including environmental modeling and assessment, target detection for military and defense/security purposes, urban planning and management studies, risk/hazard prevention and response including wild land fire tracking, biological threat detection, monitoring of oil spills and other types of chemical contamination [13]. Most of the above-cited applications require analysis algorithms able to provide a response in (near) real-time, which is a very ambitious goal since the price paid for the rich information available from hyperspectral sensors is the enormous amounts of data that they generate.

Specifically, the utilization of HPC systems in hyperspectral signal processing applications has become more and more widespread in recent years. The idea developed by the computer science community of using commercial off-the-shelf computer equipment, clustered together to work as a computational 'team,' is a very attractive solution in remote sensing applications [14]. This strategy is often referred to as Beowulf-class cluster computing [15], and has already offered access to greatly increased computational power at low cost (commensurate with falling commercial PC costs) in a number of remote sensing applications [16–19]. In theory, the combination of commercial forces driving down cost and positive hardware trends (e.g., CPU peak power doubling ever 18–24 months, storage capacity doubling every 12–18 months and networking bandwidth doubling every 9–12 months) offers supercomputing performance that can now be applied a much wider range of remote sensing problems.

Although most parallel techniques and systems for image information processing employed by NASA and other institutions during the last decade have chiefly been homogeneous in nature (i.e., they are made up of identical processing units, thus simplifying the design of parallel solutions adapted to those systems), a recent trend in the design of HPC systems for data-intensive problems is to utilize highly heterogeneous computing resources [20]. This heterogeneity is seldom planned, arising mainly as a result of technology evolution over time and computer market sales and trends. In this regard, networks of heterogeneous resources can realize a very high level of aggregate performance in remote sensing applications [21, 22], and the pervasive availability of these resources has resulted in the current notion of *Grid computing* [23], which endeavors to make such distributed computing platforms easy to utilize in different application domains, much like the World Wide Web has made it easy to distribute web content. It is expected that grid-based HPC systems will soon represent the tool of choice for the scientific community devoted to very high-dimensional data analysis in remote sensing.

Although remote sensing data processing algorithms map nicely to parallel systems made up of commodity CPUs, these systems are generally expensive and difficult to adapt to onboard remote sensing 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. In this regard, an exciting new development in the field of commodity computing is the emergence of field programmable gate arrays (FPGAs) [24–26] and graphic processing units (GPUs) [27], which can bridge the gap towards onboard and real-time analysis of remote sensing data. FPGAs are now fully reconfigurable, which allows one to adaptively select a data processing algorithm (out of a pool of available ones) to be applied onboard the sensor from a control station on Earth. On the other hand, the emergence of GPUs (driven by the ever-growing demands of the video-game industry) have allowed these systems to evolve from expensive application-specific units into highly parallel and programmable commodity components. The ever-growing computational demands of remote sensing applications can fully benefit from compact hardware components and take advantage of the small size and relatively low cost of these units as compared to clusters or networks of computers.

The main purpose of this paper is to provide an experimental assessment of different HPC systems in the context of remote sensing applications. As a case study, this paper focuses on the pixel purity index (PPI) algorithm, one of the most widely used algorithms in the hyperspectral imaging community. The algorithm was originally developed by Boardman et al. [28], and was soon incorporated into Kodak's Research Systems ENVI [29] which is one of the most widely used commercial software packages by remote sensing scientists. Due to the algorithm's propriety and limited published results, its detailed implementation has never been made available in the public domain. Therefore, most of the scientists who use the PPI algorithm either appeal for ENVI software or implement their versions of the PPI based on whatever available in the literature. In this paper, we present our experience with the PPI algorithm and investigate several strategies for its efficient implementation on parallel, distributed and hardware-based systems, aimed at solving one the most significant drawbacks of the algorithm: its very high computational complexity. The description of several systems and strategies for implementation of the PPI algorithm provides an excellent snapshot of the state-of-the-art in the application of HPC models to remote sensing applications, and an in-depth study of a well-known commercial algorithm that will appeal to both practitioners and developers alike, thus providing a thoughtful perspective on the potential of applying HPC systems in current and planned remote sensing missions.

The remainder of the paper is organized as follows. Section 2 focuses on related work in the area of parallel hyperspectral imaging. Section 3 reviews the original PPI and presents a new optimized implementation of the algorithm. Section 4 develops several high-performance system architectures for efficient implementation of the PPI, including a commodity cluster-based parallel implementation, a distributed implementation for (fully or partially) heterogeneous networks of workstations, an FPGA-based implementation, and a GPU-based implementation. Section 5 provides an experimental comparison of the proposed implementations using several high-performance computing architectures. Specifically, we use a massively parallel Beowulf cluster at NASA's Goddard Space Flight Center, four distributed networks of workstations at University of Maryland, a Xilinx Virtex-II FPGA device, and an NVidia GeForce 8800 GTX GPU. The description of results is followed by a detailed discussion on the main observations and lessons learned after the detailed assessment of the different parallel implementations conducted in this work. Finally, Section 6 concludes the paper with some remarks and hints at plausible future research lines.

2 Related Work

In previous work, we have explored the implementation of different hyperspectral image and signal processing algorithms on a variety of parallel computing architectures [30]. Specifically, our efforts have been directed towards the parallelization of hyperspectral imaging algorithms that make use of spatial context, such as an algorithm called automatic morphological endmember extraction algorithm (AMEE) [31] which has been implemented in a variety of platforms such as commodity clusters [12], several heterogeneous networks of workstations [32, 33], and a GPU 7800 GTX from NVidia [27]. The results obtained in the parallelization of AMEE provided us with a good introspection and background on how to accomplish parallelization of other hyperspectral imaging algorithms, such as the PPI discussed in this paper. In previous related work we have also focused on the optimization of sequential versions of the PPI [34–36], thus allowing us to obtain a solid understanding of the main properties of the algorithm and the most relevant aspects for its parallelization, including the specific portions of the algorithm that are more interesting for optimization due to their computational complexity. In the past we have also explored the relevant topic of how to find the best possible initialization conditions for the PPI algorithm, which may be used in future work to improve the currently adopted random initialization module for the algorithm. In this work, we focus on optimizing the classic implementation of the PPI and provide a detailed inter-comparison of different parallel implementations of the algorithm in different parallel computing architectures, including commodity clusters [12], heterogeneous networks of workstations [32, 33], FPGAs [26] (although some of these approaches have been presented in previous work, these have never been inter-compared), as well a new GPU-based implementation of the PPI which represents a completely new contribution of this paper.

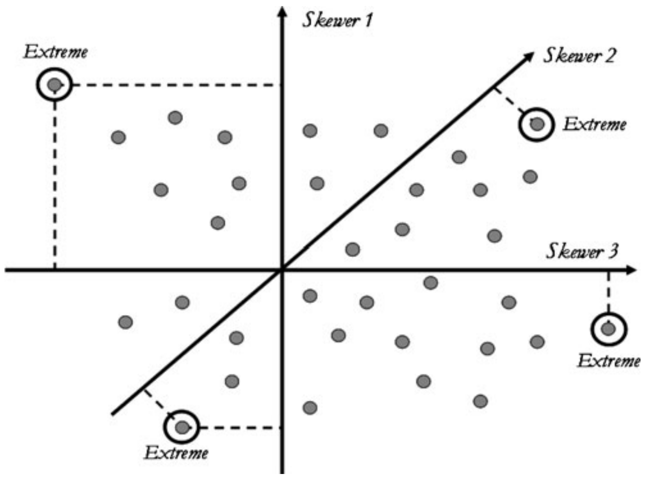


Figure 2 Toy example illustrating the performance of the PPI algorithm in a 2-dimensional space.

3 Pixel Purity Index (PPI) Algorithm

The PPI algorithm was originally developed by Boardman [28] and was soon incorporated into Kodak's Research Systems ENVI. The underlying assumption under the PPI algorithm is that the spectral signature associated to each pixel vector measures the response of multiple underlying materials at each site. For instance, it is very likely that the pixel vectors shown in Fig. 1 would actually contain a mixture of different substances (e.g., different types of vegetation, different types of soils, atmospheric interferers such as clouds, etc.). This situation, often referred to as the 'mixture problem' in hyperspectral analysis terminology [37], is one of the most crucial and distinguishing properties of spectroscopic analysis.

In hyperspectral images, mixed pixels exist for one of two reasons [34]. Firstly, if the spatial resolution of the sensor is not fine enough to separate different materials, these can jointly occupy a single pixel, and the resulting spectral measurement will be a composite of the individual spectra. Secondly, mixed pixels can also result when distinct materials are combined into a homogeneous mixture. This circumstance occurs independent of the spatial resolution of the sensor. A hyperspectral image is often a combination of the two situations, where a few sites in a scene are pure materials, but many other are mixtures of materials. To deal with the mixture problem in hyperspectral imaging, spectral unmixing techniques have been proposed as an inversion technique in which the measured spectrum of a mixed pixel is decomposed into a collection of spectrally pure constituent spectra, called *endmembers* in the literature, and a set of correspondent fractions, or *abundances*, that indicate the proportion of each endmember present in the mixed pixel [4, 37].

The PPI algorithm is a tool to automatically search for endmembers which are assumed to be the vertices of a convex hull [28]. The algorithm proceeds by generating a large number of random, N-dimensional unit vectors called 'skewers' through the dataset. Every data point is projected onto each skewer, and the data points that correspond to extrema in the direction of a skewer are identified and placed on a list (see Fig. 2). As more skewers are generated, the list grows, and the number of times a given pixel is placed on this list is also tallied. The pixels with the highest tallies are considered the final endmembers.

The inputs to the algorithm are a hyperspectral data cube $\mathbf{F}$ with N dimensions; a maximum number of endmembers to be extracted, E; the number of random skewers to be generated during the process, K; a cut-off threshold value, t_v, used to select as final endmembers only those pixels that have been selected as extreme pixels at least t_v times throughout the PPI process; and a threshold angle, t_a, used to discard redundant endmembers during the process. The output of the algorithm is a set of E final endmembers $\{\mathbf{e}_e\}_{e=1}^{E}$. The algorithm can be summarized by the following steps:

1. *Skewer generation*. Produce a set of K randomly generated unit vectors $\{\mathbf{skewer}_j\}_{j=1}^{K}$.
2. *Extreme projections*. For each $\mathbf{skewer}_j$, with $j = \{1, \cdots, K\}$, all sample pixel vectors $\mathbf{f}_i$ in the original data set $\mathbf{F}$, with $i = \{1, \cdots, T\}$, where T is the total number of pixels in the original data, are projected onto $\mathbf{skewer}_j$ via dot products given by the expression: $|\mathbf{f}_i \times \mathbf{skewer}_j| = \sum_{l=1}^{N} \mathbf{f}_i^{(k)} *$

$\mathbf{skewer}_j^{(k)}$, where $\mathbf{f}_i^{(k)}$ denotes the k-th band of pixel vector $\mathbf{f}_i$ and $\mathbf{skewer}_j^{(k)}$ denotes the k-th component of $\mathbf{skewer}_j$, to find sample vectors at its extreme (maximum and minimum) projections, thus forming an extrema set for $\mathbf{skewer}_j$ which is denoted by $S_{extrema}(\mathbf{skewer}_j)$. Despite the fact that a different $\mathbf{skewer}_j$ would probably generate a different extrema set $S_{extrema}(\mathbf{skewer}_j)$, it is very likely that some sample vectors may appear in more than one extrema set. In order to deal with this situation, we define an indicator function of a set $\mathbf{F}$, denoted by $I_S(\mathbf{x})$, to denote membership of an element $\mathbf{x}$ to that particular set as follows:

$$I_S(\mathbf{x}) = \begin{Bmatrix} 1 \text{ if } \mathbf{x} \in S \\ 0 \text{ if } \mathbf{x} \notin S \end{Bmatrix} \tag{1}$$

3. *Calculation of PPI scores*. Using the indicator function above, we calculate the PPI score associated to the sample pixel vector $\mathbf{f}_i$ (i.e., the number of times that given pixel has been selected as extreme in step 2) using the following equation:

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

4. *Endmember selection*. Find the pixel vectors with scores of $N_{PPI}(\mathbf{f}_i)$ which are above t_v, and form a unique set of endmembers $\{\mathbf{e}_e\}_{e=1}^{E}$ by calculating the spectral angle distance (SAD) for all possible vector pairs and discarding those pixels which result in an angle value below t_a. It should be noted that the SAD between a pixel vector $\mathbf{f}_i$ and a different pixel vector $\mathbf{f}_j$ is a standard similarity metric for remote sensing operations, mainly because it is invariant in the multiplication of the input vectors by constants and, consequently, is invariant to unknown multiplicative scalings that may arise due to differences in illumination and sensor observation angle.

$$\begin{aligned}\text{SAD}(\mathbf{f}_i, \mathbf{f}_j) &= \cos^{-1}(\mathbf{f}_i \times \mathbf{f}_j / \|\mathbf{f}_i\| \times \|\mathbf{f}_j\|) = \\ &= \cos^{-1}\left(\frac{\sum_{l=1}^{N} \mathbf{f}_i^{(k)} * \mathbf{f}_j^{(k)}}{\sqrt{\sum_{l=1}^{N} \mathbf{f}_i^{(k)^2}} * \sqrt{\sum_{l=1}^{N} \mathbf{f}_j^{(k)^2}}}\right).\end{aligned} \tag{3}$$

From the algorithm description above, it is clear that the PPI is not an iterative algorithm [35]. In order to set parameter values for the PPI, the authors recommend using as many random skewers as possible in order to obtain optimal results. As a result, the PPI can only guarantee to produce optimal results asymptotically and its computational complexity is very high. According to our experiments using standard AVIRIS hyperspectral data sets, the PPI generally requires a very high number of skewers (in the order of $K = 10^4$ or $K = 10^5$) to produce an accurate final set of endmembers [34], and results in processing times typically exceeding 1 h of computation in latest-generation desktop PCs. Such response time is unacceptable in many time-critical remote sensing applications. In the following section, we describe three different HPC system architectures specifically developed to speed up computational performance of the PPI algorithm.

4 Parallel Implementations

This section first develops a parallel implementation of the PPI algorithm which has been specifically designed to be run on massively parallel, homogeneous clusters of Beowulf type. Then, the parallel version is transformed into a heterogeneity-aware implementation by introducing an adaptive data partitioning algorithm specifically developed to capture the specificities of the underlying heterogeneous networks of distributed workstations. Finally, FPGA and GPU implementations aimed at onboard PPI-based processing at the same time as the data is collected on the sensor, are also provided.

4.1 Cluster-Based (Homogeneous) Parallel Implementation of the PPI

In this subsection, we describe a master-slave parallel version of the PPI algorithm. To reduce code redundancy and enhance reusability, our goal was to reuse much of the code for the sequential algorithm in the parallel implementation. For that purpose, we adopted a spatial-domain decomposition approach [38, 39] that subdivides the image cube into multiple blocks made up of entire pixel vectors, and assigns one or more blocks to each processing element (see Fig. 3).

It should be noted that the PPI algorithm is mainly based on projecting pixel vectors which are always

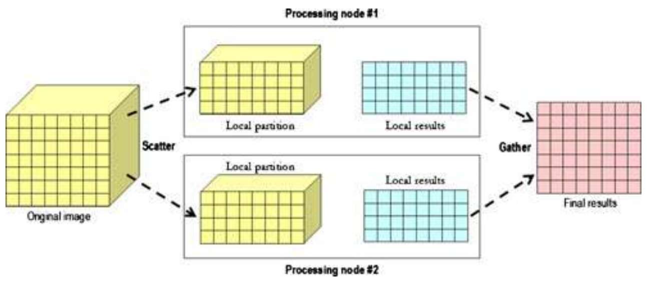


Figure 3 Spatial-domain decomposition for parallel implementation of the PPI.

treated as entire spectral signatures. This is a result of the convex geometry process implemented by the PPI, which relates to the spectral 'purity' or 'convexity' of the entire spectral signature associated to each pixel. Therefore, a spectral-domain partitioning scheme (which subdivides the whole multi-band data into blocks made up of contiguous spectral bands or sub-volumes, and assigns one or more sub-volumes to each processing element) is not appropriate in our application [12, 32]. This is because the latter approach breaks the spectral identity of the data because each pixel vector is split amongst several processing element. A further reason that justifies the above decision is that, in spectral-domain partitioning, the calculations made for each hyperspectral pixel need to originate from several processing elements, and thus require intensive inter-processor communication. Therefore, in our proposed implementation, a master-slave spatial domain-based decomposition paradigm is adopted (see Fig. 3, where the master processor sends partial data to the workers and coordinates their actions). Then, the master gathers the partial results provided by the workers and produces a final result.

As it was the case with the sequential version, the inputs to our cluster-based implementation of the PPI algorithm are a hyperspectral data cube $\mathbf{F}$ with N dimensions; a maximum number of endmembers to be extracted, E; the number of random skewers to be generated during the process, K; a cut-off threshold value, t_v; and a threshold angle, t_a. The output of the algorithm is a set of E endmembers $\{\mathbf{e}_e\}_{e=1}^{E}$. The parallel algorithm is given by the following steps:

1. *Data partitioning*. Produce a set of P equally-sized spatial-domain partitions of $\mathbf{F}$ and scatter all partitions by indicating all partial data structure elements which are to be accessed and sent to each of the workers.
2. *Skewer generation*. Generate k random unit vectors $\{\mathbf{skewer}_j\}_{j=1}^{K}$ in parallel, and broadcast the entire set of skewers to all the workers.
3. *Extreme projections*. For each $\mathbf{skewer}_j$, project all the sample pixel vectors at each local partition p onto $\mathbf{skewer}_j$ to find sample vectors at its extreme projections, and form an extrema set for $\mathbf{skewer}_j$ which is denoted by $S_{extrema}^{(p)}(\mathbf{skewer}_j)$. Now calculate the number of times each pixel vector $\mathbf{f}_i^{(p)}$ in the local partition is selected as extreme using the following expression:

$$N_{PPI}^{(p)}(\mathbf{f}_i^{(l)}) = \sum_{j=1}^{K} I_{S_{extrema}^{(p)}(\mathbf{skewer}_j)}(\mathbf{f}_i^{(l)}) \tag{4}$$

4. *Candidate selection*. Each worker now sends the number of times each pixel vector in the local partition has been selected as extreme to the master, which forms a final matrix of pixel purity indices N_{PPI} by combining all the individual matrices $N_{PPI}^{(p)}$ provided by the workers.
5. *Endmember selection*. The master selects those pixels with $N_{PPI}(\mathbf{f}_i) > t_v$ and forms a unique set $\{\mathbf{e}_e\}_{e=1}^{E}$ by calculating the SAD for all possible pixel vector pairs and discarding those pixels which result in angle values below t_a.

It should be noted that the proposed parallel algorithm has been implemented in the C++ programming language, using calls to message passing interface (MPI) [40]. We emphasize that, in order to implement step one of the parallel algorithm, we resorted to MPI *derived datatypes* to directly scatter hyperspectral data structures, which may be stored non-contiguously in memory, in a single communication step. As a result, we avoid creating all partial data structures on the root node (thus making a better use of memory resources and compute power).

4.2 Heterogeneous Parallel Implementation of the PPI

In this subsection, we adapt the cluster-based implementation of the PPI algorithm to a heterogeneous environment by reutilizing most of the code available for the cluster-based system [32, 33]. Before introducing our implementation of the PPI algorithm for heterogeneous systems, we must first formulate a general optimization problem in the context of fully heterogeneous systems (composed of different-speed processors that communicate through links at different capacities) [20]. Such a computing platform can be modeled as a complete graph where each node models a computing resource p_i weighted by its relative cycle-time w_i. Each edge in the graph models a communication link weighted by its relative capacity, where c_{ij} denotes the maximum capacity of the slowest link in the path of physical communication links from p_i to p_j (we assume that the system has symmetric costs, i.e., $c_{ij} = c_{ji}$). With the above assumptions in mind, processor p_i should accomplish a share of $\alpha_i \times W$ of the total workload, denoted by W, to be performed by a certain algorithm, with $\alpha_i \geq 0$ for $1 \leq i \leq P$ and $\sum_{i=1}^{P} \alpha_i = 1$. An abstract view of our problem can be simply stated in the form of a master-worker architecture, much like the commodity cluster-based homogeneous implementation described in the previous subsection. However, in order for such parallel algorithm to be also effective in fully heterogeneous systems, the master program must be modified to produce a set of P spatial-domain heterogeneous partitions of $\mathbf{F}$ in step one.

In order to balance the load of the processors in the heterogeneous environment, each processor should execute an amount of work that is proportional to its speed. Therefore, two major goals of our partitioning algorithm are:

– to obtain an appropriate set of workload fractions $\{\alpha_i\}_{i=1}^{P}$ that best fit the heterogeneous environment, and
– to translate the chosen set of values into a suitable decomposition of the input data, taking into account the properties of the heterogeneous system.

In order to accomplish the above goals, we use a workload estimation algorithm (WEA) [32] that assumes that the workload of each processor p_i must be directly proportional to its local memory and inversely proportional to its cycle-time w_i. Below, we provide a description of WEA algorithm, which replaces the *data partitioning* step in the implementation of PPI provided in our previous section. Steps 2–5 of the parallel algorithm in the previous section would be executed immediately after the WEA algorithm below and remain exactly the same as those outlined in the algorithmic description provided in the previous section (thus enhancing code reutilization). The input to WEA is $\mathbf{F}$, an N-dimensional data cube, and the output is a set of P spatial-domain heterogeneous partitions of $\mathbf{F}$:

1. Obtain necessary information about the heterogeneous system, including the number of available processors P, each processor's identification number $\{p_i\}_{i=1}^{P}$, and processor cycle-times $\{w_i\}_{i=1}^{P}$.
2. Let V denote the total volume of data in the original hyperspectral image $\mathbf{F}$. Processor ps_i will be assigned a certain share $\alpha_i \times V$ of the input volume, where $\alpha_i \geq 0$ for $1 \leq i \leq P$ and $\sum_{i=1}^{P} \alpha_i = 1$. In order to obtain the value of α_i for processor p_i, calculate $\alpha_i = \frac{(1/w_i)}{\sum_{j=1}^{P}(1/w_j)}$.
3. Once the set $\{\alpha_i\}_{i=1}^{P}$ has been obtained, a further objective is to produce P spatial-domain partitions of the input hyperspectral data set. To do so, we proceed as follows:
   (a) Obtain a first partitioning of the hyperspectral data set so that the number of entire pixel vectors allocated to each processor p_i is proportional to its associated value of α_i.
   (b) If necessary, refine the initial partitioning taking into account the local memory associated to each processor.

The parallel algorithm described above has been implemented using two approaches. The first one is based on the C++4 programming language with calls to standard MPI functions. A second implementation was developed using HeteroMPI [41], a heterogeneous version of MPI which automatically optimizes the workload assigned to each heterogeneous processor (i.e., this implementation automatically determines the load distribution accomplished by our proposed WEA algorithm). Experimentally, we tested that both implementations resulted in very similar results and, hence, the experimental validation provided in the following section are based on the performance analysis achieved by the first implementation (i.e., implementing our proposed WEA algorithm using MPICH [42] to estimate the heterogeneous workloads).

4.3 FPGA-Based Parallel Implementation of the PPI

In this subsection, we describe a hardware-based parallel strategy for implementation of the hyperspectral data processing chain which is aimed at enhancing replicability and reusability of slices in FPGA devices through the utilization of systolic array design [43]. One of the main advantages of systolic array-based implementations is that they are able to provide a systematic procedure for system design that allows for the derivation of a well defined processing element-based structure and an interconnection pattern which can then be easily ported to real hardware configurations [44]. Using this procedure, we can also calculate the data dependencies prior to the design, and in a very straightforward manner. Before describing our implementation, we emphasize that our proposed design intends to maximize computational power of the hardware and minimize the cost of communications. These goals are particularly relevant in our specific application, where hundreds of data values will be handled for each intermediate result, a fact that may introduce problems related with limited resource availability and inefficiencies in hardware replication and reusability.

Our systolic array-based parallelization has been inspired by the work presented in [45], but presents several differences with regards to that work. First and foremost, our implementation is based on the use of a high-level language (called Handel-C [46]) for porting the PPI algorithm to hardware, while reference [45] presents a low-level implementation in Very High Speed Integrated Circuit Hardware Description Language (VHDL).[1] Second, the implementation in [45] is

[1] http://www.vhdl.org

targeted at a different FPGA platform. With the above issues in mind, our approach can be summarized as follows. It has been shown in previous sections that the PPI algorithm consists of computing a very large number of dot-products, and all these dot-products can be performed simultaneously. As a result, a possible way of parallelization is to have a hardware system able to compute K dot-products in the same time against the same pixel $\mathbf{f}_i$, where K is the number of skewers. Supposing such a system, the *extreme projections* step of the PPI algorithm can be simply written as described in Algorithm 1.

Algorithm 1 High-level implementation of the *extreme projections* step of the PPI algorithm

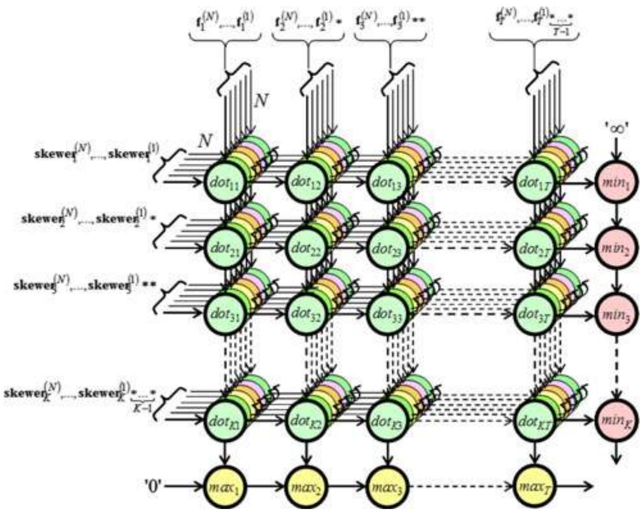
```
for ($n = 0$; $n < N$; $n$++) //$N$ denotes the number of bands
{
  par ($k = 0$; $k < K$; $k$++) //$K$ denotes the number of skewers
  {
    dp[$k$]=dot_product(pixels[$n$],skewers[$k$]);
    if (dp[$k$] < Min[$k$]) { Min[$k$]=dp[$k$]; Reg_Min[$k$]=$n$; }
    if (dp[$k$] > Max[$k$]) { Max[$k$]=dp[$k$]; Reg_Max[$k$]=$n$; }
  }
}
```

The **par** loop in Algorithm 1 expresses that K dot products are first performed in parallel, then K min and max operations are also computed in parallel. Now, if we suppose that we cannot simultaneously compute K dot products but only a fraction K/P, where P is the number of available processing units, then the *extreme projections* step of the PPI algorithm can be split into P passes, each performing $T \times K/P$ dot products, as indicated in Algorithm 2. From an architectural point of view, each processor receives successively the T pixels, computes T dot-products, and keeps in memory the two pixels having produced the *min* and the *max* dot products. In this scheme, each processor holds a different skewer which must be input before each new pass.

Algorithm 2 High-level implementation of the *extreme projections* step of the PPI algorithm, rewritten to be split into P algorithm passes

```
for ($p = 0$; $p < P$; $p$++) //$P$ is the number of algorithm passes
{
  $x = p \times (K/P)$; //$K$ denotes the number of skewers
  for ($n = 0$; $n < N$; $n$++) //$N$ denotes the number of bands
  {
    par($k = 0$; $k < K/P$; $k$++) //$K$ denotes the number of skewers
    {
      dp[$x + k$]=dot_product(pixels[$n$],skewers[$x + k$]);
      if (dp[$x + k$] < Min[$x + k$]) { Min[$x + k$]=dp[$x + k$]; Reg_Min[$x + k$]=$n$; }
      if (dp[$x + k$] > Max[$x + k$]) { Max[$x + k$]=dp[$x + k$]; Reg_Max[$x + k$]=$n$; }
    }
  }
}
```

With the above assumptions in mind, Fig. 4 describes the systolic array design adopted for the proposed



$\mathbf{f}_1^{(N)},\ldots,\mathbf{f}_1^{(1)}$
$\mathbf{f}_2^{(N)},\ldots,\mathbf{f}_2^{(1)}*$
$\mathbf{f}_3^{(N)},\ldots,\mathbf{f}_3^{(1)}**$
$\mathbf{f}_T^{(N)},\ldots,\mathbf{f}_T^{(1)}\underbrace{*\ldots*}_{T-1}$
N
N
'∞'
$\mathbf{skewer}_1^{(N)},\ldots,\mathbf{skewer}_1^{(1)}$
$\mathbf{skewer}_2^{(N)},\ldots,\mathbf{skewer}_2^{(1)}*$
$\mathbf{skewer}_3^{(N)},\ldots,\mathbf{skewer}_3^{(1)}**$
$\mathbf{skewer}_K^{(N)},\ldots,\mathbf{skewer}_K^{(1)}\underbrace{*\ldots*}_{K-1}$
dot_{11} dot_{12} dot_{13} dot_{1T} min_1
dot_{21} dot_{22} dot_{23} dot_{2T} min_2
dot_{31} dot_{32} dot_{33} dot_{3T} min_3
dot_{K1} dot_{K2} dot_{K3} dot_{KT} min_K
'0'
max_1 max_2 max_3 max_T

Figure 4 Systolic array design for the proposed FPGA implementation of the PPI algorithm.

FPGA implementation. Here, local results remain static at each processing element, while pixel vectors are input to the systolic array from top to bottom and skewer vectors are fed to the systolic array from left to right. In Fig. 4, asterisks represent delays while $\mathbf{skewer}_j^{(n)}$ denotes the value of the n-th band of the j-th skewer, with $j \in \{1, \cdots, K\}$ and $n \in \{1, \cdots, N\}$, being N the number of bands of the input hyperspectral scene. Similarly, $\mathbf{f}_i^{(n)}$ denotes the reflectance value of the n-th band of the i-th pixel, with $i \in \{1, \cdots, T\}$, being T the total number of pixels in the input image. The processing nodes labeled as *dot* in Fig. 4 perform the individual products for the skewer projections. On the other hand, the nodes labeled as *max* and *min* respectively compute the maxima and minima projections after the dot product calculations have been completed. In fact, the *max* and *min* nodes can be respectively seen as part of a 1-D systolic array which avoids broadcasting the pixel while simplifying the collection of the results.

The operational functionality of each dot processing node in Fig. 4 is depicted in Fig. 5. Each processing node accumulates the positive or negative values of the pixel input according to the skewer input. For instance, if the i-th component of the skewer is 0, then the i-th component of the pixel vector is summed up to the accumulator (AC). If it equals 1, it is subtracted. This unit is composed of a single 16-bit Add/Sub module. This module computes the dotproduct by summing up positive or negative pixel values. The skewer is stored in a 1-bit, N-word memory, where N is the number of spectral channels. The initialization mechanism is not represented. It should be noted that Fig. 5 also

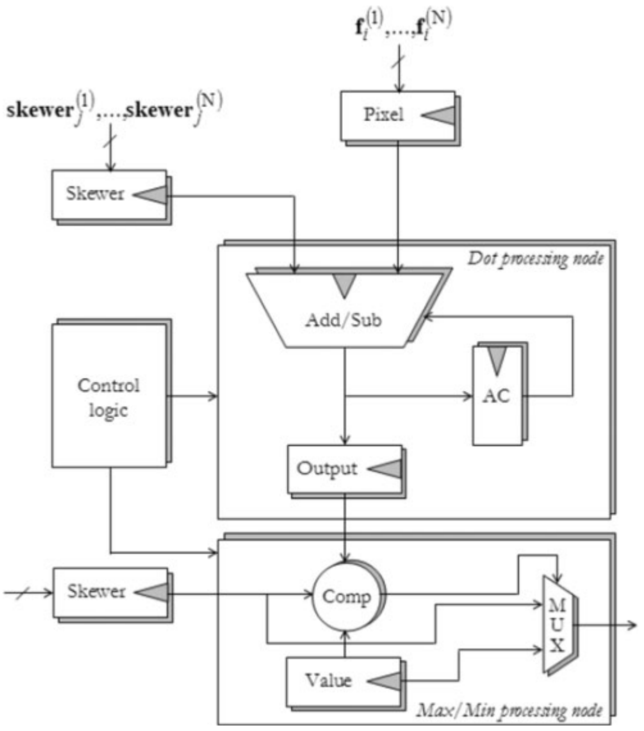

$\mathbf{f}_i^{(1)},\ldots,\mathbf{f}_i^{(N)}$
$\mathbf{skewer}_j^{(1)},\ldots,\mathbf{skewer}_j^{(N)}$
Pixel
Skewer
Dot processing node
Add/Sub
Control logic
AC
Output
Skewer
Comp
MUX
Value
Max/Min processing node

Figure 5 Architecture of each dot and max/min processing nodes in the proposed systolic array design.

illustrates the performance of the min and max processing nodes (their performance is similar and hence they are represented in the figure as a single unit called Min/Max). This unit performs bit-serially a comparison between a Min/Max value. If the value of the dot-product exceeds the corresponding Min/Max value, then the current dot-product value is substituted and the number of the pixel which has caused this change is memorized.

Basically, a systolic cycle in the architecture described in Figs. 4 and 5 consists in computing a single dot-product between a pixel and a skewer, and to memorize the index of the pixel if the dot-product is higher or smaller than a previously computed Min/Max value. Remember that a pixel is a vector of N spectral values, just like a skewer. A dot-product calculation (dp) between a pixel $\mathbf{f}_i$ and a $\mathbf{skewer}_j$ can be simply obtained using the expression $\sum_{k=1}^{N} \mathbf{f}_i^{(k)} * \mathbf{skewer}_j^{(k)}$. Therefore, a full dot product calculation requires N multiplications and $N-1$ additions, where N is the number of spectral bands. It has been shown in previous work that the skewer values can be limited to a very small set of integers when N is large, as in the case of hyper spectral images. A particular and interesting set is $\{1, -1\}$ since it avoids the multiplication [45]. The dot product is thus reduced to an accumulation of positive and negative values (the self-connections in the dot nodes of Fig. 4 represent the accumulation of intermediate results in those nodes). With the above assumptions in mind, the dot nodes only need 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 16-bit addition/subtraction operator. If we suppose that an addition or a subtraction is executed every clock cycle, then the calculation of a full dot-product requires N clock cycles. During the first systolic cycle, dot_{11} starts processing the first band of the first pixel vector, $\mathbf{f}_1$. During the second systolic cycle, the node dot_{12} starts processing the first band of pixel $\mathbf{f}_2$, while the node dot_{11} processes the second band of pixel $\mathbf{f}_1$, and so on.

The main advantage of the systolic array described in Figs. 4 and 5 is its scalability. Depending of the resources available on the reconfigurable board, the number of processors can be adjusted without modifying the control of the array. In order to reduce the number of passes, we may decide to allocate the maximum number of processors in the available FPGA components, but this option would limit the room in the FPGA for additional algorithms. In other words, although in Fig. 4 we represent an ideal systolic array in which T pixels can be processed, this is not the usual situation, and the number of pixels usually has to be divided by P, the number of available processors. In this scenario, after T/P systolic cycles, all the nodes are working. When all the pixels have been flushed through the systolic array, T/P additional systolic cycles will be required to collect the results for the considered set of P pixels, and a new set of P different pixels would be flushed until processing all T pixels in the original image. In summary, the principle of our parallelization framework is that K/P processors perform successively N dot products. The pixels are thus broadcast to all the processors and the computation is pipelined (systolized) to provide a highly scalable system.

Based on the system design described above, we have developed a high-level implementation of PPI using Handel-C [46], a design and prototyping language that allows using a pseudo-C programming style. The final decision on implementing our design using Handel-C instead of other well-known hardware description languages such as VHDL or Verilog was taken on the account that a high-level language may allow users to generate hardware versions of available hyperspectral analysis algorithms in relatively short time. For illustrative purposes, the source code in Handel-C corresponding to the *extreme projections* step of our FPGA implementation of the PPI algorithm is shown in Algorithm 3. The skewer initialization and endmember selection-related portions of the code are not shown for simplicity.

Algorithm 3 Source code of the Handel-C (high level) FPGA implementation of the PPI algorithm

```
void main(void) {
  unsigned int 16 max[E]; //E is the number of endmembers
  unsigned int 16 end[E];
  unsigned int 16 i;
  unsigned int 10000 k; //k denotes the number of skewers
  unsigned int 224 N; //N denotes the number of bands
  par (i = 0; i < E; i++) max[i] = 0;
  par (i = 0; i < T; i++) { //T denotes the number of pixels
    par (k = 0; k < E; k++) {
      par (j = 0; j < N; j++) {
        Proc_Element[i][k](pixels[i][j],skewers[k][j],0@i,0@k);
      }
    }
  }
  for (i = 0; i < E; i++) {
    max[i]=Proc_Element[i][k](0@max[i], 0, 0@i, 0@k);
  }
  phase_1_finished=1
  while (!phase_2) { //Waiting to enter phase 2 }
  for (i = 0; i < E; i++) end[i]=0;
  for (i = 0; i < T; i++) { //T denotes the number of pixels
    par (k = 0; k < E; k++) {
      par (j = 0; j < N; j++) {
      end[i]=end[i]&&Proc_Element[i][k](pixels[i][j],skewers[k][j]);
      }
    }
  }
  phase_2_finished=1
  global_finished=0
  for (i = 0; i < E; i++)   global_finished=global_finished&&end[i];
```

For a detailed understanding of the piece of code shown in Algorithm 3, we point to reference material [46]. The implementation was compiled and transformed to an EDIF specification automatically by using the DK3.1 software package [47]. With this specification, and using other tools such as Xilinx ISE [48] to simplify the final steps of the hardware implementation, we also incorporated hardware-specific limitations and constraints to the mapping process into a Virtex-II FPGA. These tools allowed us to evaluate the total amount of resources needed by the whole implementation, along with sub-totals related to different functional units available in the FPGA that will be discussed in the following section, along with our results in terms of algorithm performance and execution time for the three parallel systems developed in this section.

4.4 GPU-Based Parallel Implementation of the PPI

GPUs can be abstracted in terms of a *stream* model, under which all data sets are represented as streams (i.e., ordered data sets) [49]. Algorithms are constructed by chaining so-called *kernels*, which operate on entire streams, taking one or more streams as inputs and producing one or more streams as outputs. Thereby, data-level parallelism is exposed to hardware, and kernels can be concurrently applied without any sort of synchronization. Modern GPU architectures such as NVidia GeForce cards adopt this model and implement a generalization of the traditional rendering pipeline, which consists of two main stages:

1. *Vertex processing*. The input to this stage is a stream of vertices from a 3-D polygonal mesh. Vertex processors transform the 3-D coordinates of each vertex of the mesh into a 2-D screen position, and apply lighting to determine their colors (this stage is now fully programmable).
2. *Fragment processing*. In this stage, the transformed vertices are first grouped into rendering primitives, such as triangles, and scan-converted into a stream of pixel fragments. These fragments are discrete portions of the triangle surface that corresponds to the pixels of the rendered image. Apart from identifying constituent fragments, this stage also interpolates attributes stored at the vertices, such as texture coordinates, and stores the interpolated values at each fragment. Arithmetical operations and texture lookups are then performed by fragment processors to determine the ultimate color for the fragment. For this purpose, texture memories can be indexed with different texture coordinates, and texture values can be retrieved from multiple textures.

It should be noted that fragment processors currently support instructions that operate on vectors of four RGBA components (Red/Green/Blue/Alpha channels) and include dedicated texture units that operate with a deeply pipelined texture cache [50]. As a result, an essential requirement for mapping non-graphics algorithms onto GPUs is that the data structure can be arranged according to a stream-flow model, in which kernels are expressed as *fragment* programs and data streams are expressed as *textures* [51]. Using C-like, high-level languages such as the compute unified device architecture (CUDA), programmers can write fragment programs to implement general-purpose operations.

In this section, we develop a CUDA-based technique for mapping two of the steps of the PPI algorithm (i.e. extreme projections, and calculation of PPI scores) onto a GPU using a stream-based processing approach that makes use of kernels. These steps account for most of the execution time involved in the PPI algorithm and exhibit enough data parallelism for a GPU implementation.

The first issue that needs to be addressed is how to map a hyperspectral image onto the memory of the GPU. Since the size of hyperspectral images usually exceeds the capacity of such memory, we split them into multiple spatial-domain partitions made up of entire pixel vectors (see Fig. 3), i.e., each spatial-domain parti-

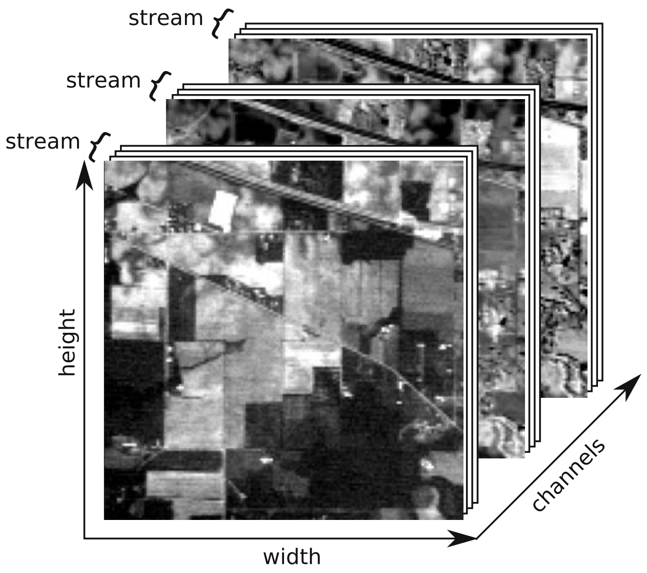


Figure 6 Mapping of a spatial-domain hyperspectral data partition onto the GPU memory.

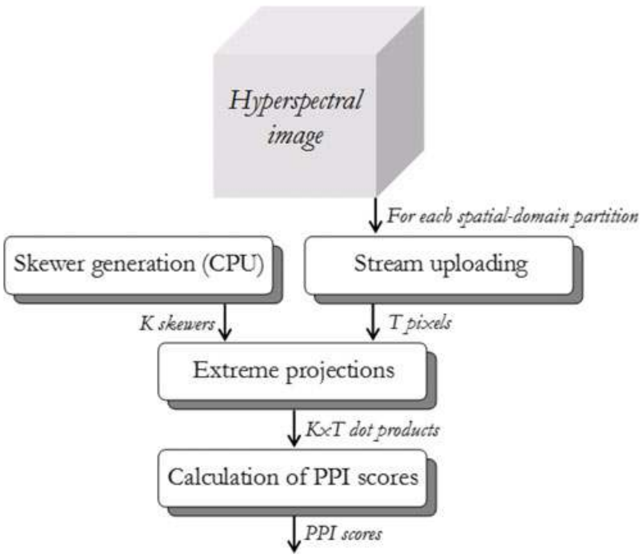


Figure 7 Flowchart of the proposed stream-based GPU implementation.

tion incorporates all the spectral information on a localized spatial region and is composed of spatially adjacent pixel vectors. As shown by Fig. 6, each spatial-domain partition is further divided into 4-band tiles (called spatial-domain tiles), which are arranged in different areas of a 2-D texture. Such partitioning allows us to map four consecutive spectral bands onto the RGBA color channels of a texture (memory) element. Apart from the tiles, we also allocate additional memory to hold other information, such as the skewers which are generated at the host (CPU) and then transmitted to the GPU, and also intermediate results such as dot products, norms, and pointwise distances.

Figure 7 shows a flowchart describing our GPU-based implementation. The *stream uploading* stage performs the data partitioning and mapping operations described above, i.e., dividing the image into spatial-domain partitions and uploading them as a set of tiles onto the GPU memory. The *skewer generation* step performs the generation of skewers in the host (CPU) and transmission of these skewers to the GPU. The remaining stages perform the actual PPI computation and comprise the following kernels:

1. *Extreme projections*. The tiles are input streams to this stage, which obtains all the inner products and norms necessary to compute the required projections. Since streams are actually tiles, the implementation of this stage is based on a *multiply and add* (MAD) operation, a multi-pass kernel that implements an element-wise multiply and add operation, thus producing four partial inner products stored in the RGBA channels of a texture element.
2. *Calculation of PPI scores*. Finally this kernel uses as inputs the projection values generated in the previous stage, and produces a stream containing (for each pixel vector) the relative coordinates of the pixels with maximum and minimum distance after the projection onto each skewer, thus completing the first three steps of the PPI algorithm.

5 Experimental Results

This section provides an assessment of the effectiveness of the parallel versions of PPI described in the previous section. Before describing our study on performance analysis, we first describe the HPC system architectures used in this work for evaluation purposes. These include Thunderhead, a massively parallel Beowulf cluster made up of homogeneous commodity components and available at NASA's GSFC; four different networks of heterogeneous workstations distributed among different locations at University of Maryland; a Xilinx Virtex-II XC2V6000-6 FPGA; and an NVidia 8800 GTX GPU. Next, we describe the hyperspectral data sets used for evaluation purposes. A detailed survey on algorithm performance in a real mineral mapping application, supported by hyperspectral data collected by NASA's AVIRIS sensor, is then provided. The section concludes with a detailed discussion on the main observations and lessons learned from the application of each particular system to the considered remote sensing problem.

5.1 Parallel Computing Systems

5.1.1 Beowulf Cluster

The first parallel system used for experimental validation in this work is the Thunderhead system at NASA's Goddard Space Flight Center in Maryland. This Beowulf cluster can be seen as an evolution of the HIVE (Highly Parallel Virtual Environment) project, started in spring of 1997 to build a commodity cluster intended to be exploited by different users in a wide range of scientific applications. The idea was to have workstations distributed among many offices and a large number of compute nodes (the compute core) concentrated in one area. The workstations would share the compute core as though it was apart of each. At the time of the experiments, Thunderhead was composed of 284 dual 2.4 Ghz Intel 4 Xeon nodes, each with 1 GB of memory and 80 GB of hard disk (see http://thunderhead.gsfc.nasa.gov for additional details). The total disk space available in the system was 21.44 Tbyte, and the theoretical peak performance of the system was 2.5728 Tflops (1.2 Tflops on the Linpack benchmark). The estimated cost of the Thunderhead system is 1.25M U.S. dollars. Along with the 568-processor computer core (out of which 256 were used for experiments), Thunderhead has several nodes attached to the core with Myrinet 2000 connectivity. Our parallel algorithms were run from one of such nodes, called thunder1. The operating system is Linux Fedora Core, and MPICH was the message-passing library used.

5.1.2 Heterogeneous Networks Of Computers

To explore the performance of the heterogeneity-aware implementation of PPI developed in this chapter, we have considered four different heterogeneous networks. All of them were custom-designed in order to approximate a recently proposed framework for evaluation of heterogeneous parallel algorithms [52], which relies on the assumption that a heterogeneous algorithm cannot be executed on a heterogeneous network faster than its homogeneous version on the equivalent homogeneous network. Let us assume that a heterogeneous network consists of $\{p_i\}_i^P$ heterogeneous workstations with different cycle-times w_i, which span m communication segments $\{s_j\}_{j=1}^m$, where $c^{(j)}$ denotes the communication speed of segment s_j. Similarly, let $p^{(j)}$ be the number of processors that belong to s_j, and let $w_t^{(j)}$ be the speed of the t-th processor connected to s_j, where $t = 1, ..., p^{(j)}$. Finally, let $c^{(j,k)}$ be the speed of the communication link between segments s_j and s_k, with $j, k = 1, ..., m$. According to [52], the above network can be considered equivalent to a homogeneous one made up of $\{q_i\}_{i=1}^P$processors with constant cycle-time and interconnected through a homogeneous communication network with speed c if the following expressions are satisfied:

$$c = \frac{\sum_{j=1}^m c^{(j)} \cdot [\frac{p^{(j)}(p^{(j)}-1)}{2}]}{\frac{P(P-1)}{2}} + \\ + \frac{\sum_{j=1}^m \sum_{k=j+1}^m p^{(j)} \cdot p^{(k)} \cdot c^{(j,k)}}{\frac{P(P-1)}{2}}, \quad (5)$$

$$w = \frac{\sum_{j=1}^m \sum_{t=1}^{p^{(j)}} w_t^{(j)}}{P}, \quad (6)$$

where Eq. (5) states that the average speed of point-to-point communications between the processors $\{p_i\}_{i=1}^P$ in the heterogeneous network should be equal to the speed of point-to-point communications between processors $\{q_i\}_{i=1}^P$ in the homogeneous network, with both networks having the same number of processors. On the other hand, Eq. (6) states that the aggregate performance of processors $\{p_i\}_{i=1}^P$ should be equal to the aggregate performance of processors $\{q_i\}_{i=1}^P$.

With the above principles in mind, a heterogeneous algorithm may be considered optimal if its efficiency on a heterogeneous network is the same as that evidenced by its homogeneous version on the equivalent homogeneous network. This allows using the parallel performance achieved by the homogeneous version as a benchmark for assessing the parallel efficiency of the heterogeneous algorithm. The four considered networks are considered approximately equivalent under the above framework. Their detailed description follows:

– **Fully heterogeneous network**. Consists of 16 different workstations, and four communication segments. Table 1 shows the properties of the 16 heterogeneous workstations, where processors $\{p_i\}_{i=1}^4$ are attached to communication segment s_1, processors $\{p_i\}_{i=5}^8$ communicate through s_2, processors $\{p_i\}_{i=9}^{10}$ are interconnected via s_3, and processors $\{p_i\}_{i=11}^{16}$ share the communication segment s_4. The communication links between the different segments $\{s_j\}_{j=1}^4$ only support serial communication. For illustrative purposes, Table 2 also shows the capacity of all point-to-point communications in the heterogeneous network, expressed as the time in milliseconds to transfer a one-megabit message between each processor pair (p_i, p_j) in the hetero-

Table 1 Specifications of heterogeneous computing nodes in a fully heterogeneous network of distributed workstations.

Processor number	Architecture overview	Cycle-time (seconds/Mflop)	Memory (MB)	Cache (KB)
p_1	Intel Pentium 4	0.0058	2,048	1024
p_2, p_5, p_8	Intel Xeon	0.0102	1,024	512
p_3	AMD Athlon	0.0026	7,748	512
p_4, p_6, p_7, p_9	Intel Xeon	0.0072	1,024	1,024
p_{10}	UltraSparc-5	0.0451	512	2,048
$p_{11} - p_{16}$	AMD Athlon	0.0131	2,048	1,024

geneous system. As noted, the communication network of the fully heterogeneous network consists of four relatively fast homogeneous communication segments, interconnected by three slower communication links with capacities $c^{(1,2)} = 29.05$, $c^{(2,3)} = 48.31$, $c^{(3,4)} = 58.14$ in milliseconds, respectively. The first group of processors ($p_1 - p_4$) is connected to the second group ($p_5 - p_8$) via $c^{(1,2)}$; the second group of processors ($p_5 - p_8$) is connected to the third group ($p_9 - p_{10}$) via $c^{(2,3)}$, and finally the third group of processors ($p_9 - p_{10}$) is connected to the fourth group ($p_{11} - p_{16}$) via $c^{(3,4)}$. Although this is a simple architecture, it is also a quite typical and realistic one as well.

– **Fully homogeneous network**. Consists of 16 identical Linux workstations with processor cycle-time of $w = 0.0131$ s per megaflop, interconnected via a homogeneous communication network where the capacity of links is $c = 26.64$ ms.
– **Partially heterogeneous network**. Formed by the set of 16 heterogeneous workstations in Table 1 but interconnected using the same homogeneous communication network with capacity $c = 26.64$ ms.
– **Partially homogeneous network**. Formed by 16 identical Linux workstations with cycle-time of $w = 0.0131$ s per megaflop, interconnected using the communication network in Table 2.

5.1.3 Field Programmable Gate Array

In order to test the proposed systolic array design in a hardware-based computing architecture, our parallel design was implemented on a Virtex-II XC2V6000-6 FPGA of the Celoxica's ADMXRC2 board. It contains 33,792 slices, 144 Select RAM Blocks and 144 multipliers (of 18 × 18-bit). Concerning the timing performances, we decided to pack the input/output registers of our implementation into the input/output blocks in order to try and reach the maximum achievable performance.

Table 2 Capacity of communication links (time in milliseconds to transfer a one-megabit message) in a fully heterogeneous network of distributed workstations.

Processor	$p_1 - p_4$	$p_5 - p_8$	$p_9 - p_{10}$	$p_{11} - p_{16}$
$p_1 - p_4$	19.26	48.31	96.62	154.76
$p_5 - p_8$	48.31	17.65	48.31	106.45
$p_9 - p_{10}$	96.62	48.31	16.38	58.14
$p_{11} - p_{16}$	154.76	106.45	58.14	14.05

5.1.4 Graphics Processing Unit

Our GPU-based experiments were performed on a 2006-model HP xw8400 workstation based on dual Quad-Core Intel Xeon processor E5345 running at 2.33 GHz with 1.333 MHz bus speed and 3 GB RAM. The computer was equipped with an NVidia GeForce 8800 GTX with 16 multiprocessors, each composed of 8 SIMD processors operating at 1,350 Mhz. Each multiprocessor has 8,192 registers, a 16 KB parallel data cache of fast shared memory, and access to 768 MB of global memory. This card is used most efficiently in a data-parallel fashion, when the ratio of computations to memory access is high and when many computations are performed concurrently, which ideally suits the implementation of the PPI algorithm. The algorithm was implemented using NVidia's CUDA, which is a collection of C extensions and a runtime library. CUDAs functionality primarily allows a developer to write C functions to be executed on the GPU. CUDA also includes memory management and execution configuration so that, with CUDA, a developer can control the number of GPU processors and threads that are to be invoked during a functions execution.

5.2 Hyperspectral Data

A well-known hyperspectral data set collected over the Cuprite mining district in Nevada was used in experiments to evaluate the algorithms in the context of a real mineral mapping application. The data set[2] consists of 1,939 × 677 pixels and 224 bands in the wavelength range 0.4–2.5 μm (574 MB in size). The Cuprite data set is atmospherically corrected and available in reflectance units (it has been atmospherically and geometrically corrected by JPL [3]), thus allowing direct comparison of pixel vectors to ground spectral signatures. The Cuprite site has been extensively mapped by the U.S. Geological Survey (USGS) in the last 20 years, and there is extensive ground-truth information available, including a library of mineral signatures collected

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

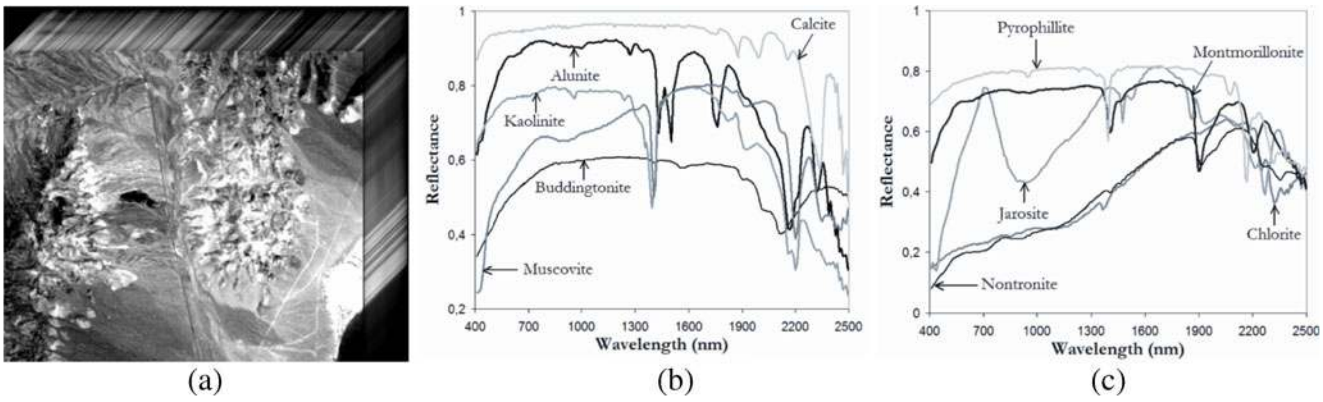


Figure 8 **a** AVIRIS scene over Cuprite mining district. **b**, **c** Ground-truth mineral spectra provided by USGS.

on the field.[3] Figure 8a shows the spectral band at 587 nm wavelength of the AVIRIS scene. The spectra of USGS ground minerals: alunite, buddingtonite, calcite, kaolinite, muscovite [Fig. 8b], chlorite, jarosite, montmorillonite, nontronite, pyrophyllite [Fig. 8c] are also displayed. These selected spectral signatures will be used in this work to evaluate endmember extraction accuracy of the proposed implementations of PPI algorithm.

5.3 Performance Evaluation

Before analyzing the parallel properties of the proposed implementations, we first conducted an experiment-based cross-examination of endmember extraction accuracy to assess the SAD-based spectral similarity scores obtained after comparing the ten USGS library spectra with the corresponding endmembers extracted by the three parallel implementations of the PPI algorithm. Table 3 shows the SAD between the most similar target pixels detected by the original ENVI implementation and our three proposed parallel implementations with regards to the USGS signatures. In all cases, the total number of endmembers to be extracted was set to $E = 16$ for all versions after estimating the virtual dimensionality (VD) of the data [4], although only ten endmembers were available for quantitative assessment due to the limited number of ground-truth signatures in our USGS library. In order to display the results in a more effective manner, we only report the SAD score associated to the most similar spectral endmember (out of 16 endmembers obtained for each implementation of the PPI) with regards to its corresponding USGS signature. It is important to emphasize that smaller SAD values indicate higher spectral similarity [36]. Table 3 revealed that the three considered parallel implementations did not produce exactly the same results as those obtained by the original PPI algorithm implemented in Kodak's Research Systems ENVI 4.0 [29], although the spectral similarity scores with regards to the reference USGS signatures were very satisfactory in all cases. Prior to a full examination and discussion of results, it is also important to outline parameter values used for the PPI. It is worth noting that, in experiments with the Cuprite AVIRIS scene, we observed that the PPI produced the same final set of experiments when the number of randomly generated skewers was set to $K = 10^4$ and above (values of $K = 10^3$, 10^5 and 10^6 were also tested). Based on the above simple experiments, we empirically set parameter t_v (threshold value) to the mean of N_{PPI} scores obtained after $K = 10^4$ iterations. In addition, we set the threshold angle value used to discard redundant endmembers during the process to $t_a = 0.01$. These parameter values are in agreement with those used before in the literature [34].

5.3.1 Parallel Performance on a Beowulf Cluster System

To empirically investigate the parallel properties of our multiprocessor PPI implementation, we tested its performance on NASAs Thunderhead Beowulf cluster. For that purpose, we measured the speedups achieved by the multiprocessor runs over a single-processor run of our sequential C++ implementation of the PPI algorithm using only one Thunderhead processor. It should be noted that the speedup factors were calculated as follows: the real time required to complete a task on P processors, T_P, was approximated by $T_P = A_P + \frac{B_P}{P} + C_P$, where A_P is the sequential (non-parallelizable) portion of the computation, B_P is the parallel portion, and C_P is the communication time. In our parallel codes, A_P corresponds to the *data partitioning* and

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

Table 3 SAD-based spectral similarity scores between the endmembers extracted by different system implementations of PPI and ten USGS reference signatures.

USGS Mineral	ENVI software	Homogeneous PPI	Heterogeneous PPI	FPGA-based PPI	GPU-based PPI
Alunite	0.084	0.084	0.084	0.084	0.084
Buddingtonite	0.071	0.071	0.071	0.073	0.071
Calcite	0.099	0.099	0.099	0.099	0.099
Chlorite	0.065	0.065	0.065	0.065	0.065
Jarosite	0.091	0.091	0.091	0.102	0.102
Kaolinite	0.136	0.136	0.136	0.136	0.136
Montmorillonite	0.106	0.106	0.106	0.112	0.106
Muscovite	0.092	0.092	0.092	0.092	0.092
Nontronite	0.099	0.102	0.102	0.099	0.096
Pyrophyllite	0.094	0.094	0.094	0.097	0.097

endmember selection steps (performed by the master), while B_P corresponds to the *skewer generation*, *extreme projections* and *candidate selection* steps, which are performed (in 'embarrassingly parallel' fashion) at the different workers. Since communications only take place at the beginning (initial data scatter in the *data partitioning* step) and at the end (final data gather in the *endmember selection* step) of the parallel algorithm, these are not overlapped with the computations in our parallel implementation. With the above assumptions in mind, we can define the speedup for P processors, S_P, as follows:

$$S_P = \frac{T_1}{T_P} \approx \frac{A_P + B_P}{A_P + \frac{B_P}{P} + C_P}, \tag{7}$$

where T_1 denotes the time measured for the sequential version of the algorithm in a single processor. The relationship above is known as Amdahl's Law [53]. It is obvious from this expression that the speedup of a parallel algorithm does not continue to increase with increasing the number of processors. The reason is that the sequential portion A_P is proportionally more important as the number of processors increase and, thus, the performance of the parallelization is generally degraded for a large number of processors. With the above definitions in mind, Table 4 shows the total time spent by the parallel implementation in communications and computations in the Thunderhead Beowulf cluster, where two types of computation times were analyzed, namely, sequential (those performed by the root node with no other parallel tasks active in the system, labeled as A_P in the table) and parallel (the rest of computations, i.e., those performed by the root node and/or the workers in parallel, labeled as B_P/P in the table). The latter includes the times in which the workers remain idle. In addition, Table 4 also displays the communication times C_P, the total execution times T_P, and the speedups S_P.

It can be seen from Table 4 that, although A_P scores were non-zero mainly due to the *endmember selection* step of the parallel algorithm, which is performed at the master node once the workers have finalized their parallel computations, these scores were always very low and irrelevant when compared to the B_P/P scores, which anticipates high parallel efficiency of the multiprocessor algorithm, even for a very high number of processors. On the other hand, it can also be seen from Table 4 that the impact of communications is not particularly significant since C_P scores are always very similar, regardless of the number of processors used, while B_P/P scores are generally higher than C_P scores except in those cases in which the number of processors is 196 and above, in which the ratio of parallel computations to communications is reduced resulting from the fact

Table 4 Sequential computation (A_P), parallel computation (B_P/P), communication times (C_P), total execution times (T_P) and speedups (S_P) achieved by multiprocessor runs of the parallel PPI on Thunderhead.

# CPUs (P)	4	16	36	64	100	144	196	256
A_P	1.63	1.26	1.12	1.19	1.06	0.84	0.91	0.58
B_P/P	292.09	73.24	30.46	15.44	8.76	5.08	3.18	1.91
C_P	2.20	2.41	2.39	2.21	2.46	2.65	2.32	2.49
T_P	295.92	76.91	33.97	18.84	12.38	8.57	6.41	4.98
S_P	3.93	15.12	34.23	61.73	93.89	135.67	181.34	233.45
D_P	1.15	1.10	1.09	1.11	1.07	1.10	1.05	1.04
D_{P-1}	1.04	1.02	1.04	1.03	1.01	1.02	1.03	1.01

For illustrative purposes, load-balancing rates considering all processors (D_P), and considering all processors but the root (D_{P-1}), are also displayed. The total execution time measured for the sequential version of PPI on one Thunderhead processor was $T_1 = 1163.05$ s.

that the size of the partitions to be processed at each local node is very small. On the other hand, although the speedup S_P scores in Table 4 flatten out a little for a large number of processors, they are still close to linear speedup. Finally, the execution times reported reveal that our multiprocessor implementation of PPI can effectively adapt to a massively parallel environment and provide a response below 5 s using a relatively moderate number of processors. Despite the fact that the above results seem promising from the viewpoint of obtaining a highly scalable parallel algorithm, the fact that B_P/P is usually the most significant fraction of the parallel algorithm requires a detailed study of load balance to fully substantiate the parallel properties of the considered algorithm.

To analyze the important issue of load balance in more detail, Table 4 also shows the imbalance scores achieved by the multiprocessor implementation of PPI on Thunderhead. The imbalance is simply defined as $D = R_{max}/R_{min}$, where R_{max} and R_{min} are the maxima and minima processor run times, respectively. Therefore, perfect balance is achieved when $D = 1$. In the table, we display the imbalance considering all processors, D_P, and also considering all processors but the root, D_{P-1}. As we can see from Table 4, the multiprocessor PPI was able to provide values of D_P close to 1 in all considered networks. Further, the algorithm provided almost the same results for both D_P and D_{P-1}, which indicates that the workload assigned to the master node is balanced with regards to that assigned to the workers.

5.3.2 Parallel Performance on Heterogeneous Systems

After evaluating the performance of the proposed cluster-based implementation on a fully homogeneous cluster, a further objective was to evaluate how the proposed heterogeneous implementation performed on the four considered heterogeneous networks. For that purpose, we evaluated its performance by timing the parallel heterogeneous code using four (equivalent) networks of distributed workstations. For that purpose, Table 5 shows the total time spent by the tested algorithms in communications and computations in the four considered networks, as well as the total execution times and load-balancing rates. For comparative purposes, Table 6 reports the measured execution times achieved by the cluster-based (homogeneous) version of the PPI which is the equivalent homogeneous version of the proposed heterogeneous parallel implementation. In all cases, the number of processors available in the heterogeneous network was $P = 16$.

Table 5 Sequential computation (A_{16}), parallel computation (B_{16}), communication time (C_{16}) and total execution time (T_{16}) achieved by the heterogeneous version of PPI on the four considered networks with $P = 16$ workstations.

	Fully heterogeneous	Fully homogeneous	Partially heterogeneous	Partially homogeneous
A_{16}	19.03	16.12	18.87	20.45
B_{16}	58.23	62.04	61.93	60.76
C_{16}	7.15	11.45	8.03	8.24
T_{16}	84,41	89,61	88,93	89.45
D_{16}	1.19	1.16	1.24	1.22
D_{15}	1.05	1.03	1.06	1.03

For illustrative purposes, load-balancing rates considering all processors (D_{16}), and considering all processors but the root (D_{15}), are also displayed.

As expected, the execution times reported on Table 5 show that the heterogeneous algorithm was able to adapt much better to fully (or partially) heterogeneous environments than the homogeneous version, which only performed satisfactorily on the fully homogeneous network as shown by Table 6. One can see that the heterogeneous algorithm was always several times faster than its homogeneous counterpart in the fully heterogeneous network, and also in both the partially homogeneous and the partially heterogeneous networks. On the other hand, the homogeneous algorithm only slightly outperformed its heterogeneous counterpart in the fully homogeneous network. Table 5 also indicates that the performance of the heterogeneous algorithm on the fully heterogeneous platform was almost the same as that evidenced by the equivalent homogeneous algorithm on the fully homogeneous network (see Table 6). This indicated that the proposed heterogeneous algorithm was always close to the optimal heterogeneous modification of the basic homogeneous one. On the other hand, the homogeneous algorithm performed much better on the par-

Table 6 Sequential computation (A_{16}), parallel computation (B_{16}), communication time (C_{16}) and total execution time (T_{16}) achieved by the homogeneous version of PPI on the considered 16-processor networks of workstations.

	Fully heterogeneous	Fully homogeneous	Partially heterogeneous	Partially homogeneous
A_{16}	19.24	16.93	18.03	20.25
B_{16}	634.12	59.56	611.34	342.50
C_{16}	14.44	6.56	9.03	12.92
T_{16}	667.8	83.05	638.4	375.67
D_{16}	1.62	1.20	1.67	1.41
D_{15}	1.23	1.04	1.26	1.06

For illustrative purposes, load-balancing rates considering all processors (D_{16}), and considering all processors but the root (D_{15}), are also displayed.

tially homogeneous network (made up of processors with the same speed) than on the partially heterogeneous network. This fact reveals that processor heterogeneity has a more significant impact on algorithm performance than network heterogeneity, a fact that is not surprising given our adopted strategy for data partitioning in the design of the parallel heterogeneous algorithm. Finally, Table 6 shows that the homogeneous version only slightly outperformed the heterogeneous algorithm in the fully homogeneous network (see Table 5). This clearly demonstrates the flexibility of the proposed heterogeneous algorithm, which was able to adapt efficiently to the four considered network environments.

Regarding load balance, we can see from Table 5 that the heterogeneous PPI was able to provide values of D_{16} close to 1 in all considered networks. Further, this algorithm provided almost the same results for both D_{16} and D_{15} while, for the homogeneous PPI in Table 6, load balance was much better when the root processor was not included. In addition, it can be seen from Table 6 that the homogeneous algorithm executed on the (fully or partially) heterogeneous networks provided the highest values of D_{16} and D_{15} (and hence the highest imbalance), while the heterogeneous algorithm executed on the homogeneous network resulted in values of D_{15} which were close to 1 (see Table 5). It is our belief that the (relatively high) unbalance scores measured for the homogeneous PPI executed on the fully heterogeneous network are not only due to memory considerations or to an inefficient allocation of spatial-domain partitions to heterogeneous resources, but to the lack of a model of communications in our design of parallel algorithms. As future research, we are planning to include considerations about the heterogeneous communication network in the design of the data partitioning algorithm.

In order to further compare the performance gain of heterogeneous algorithms as compared to their respective sequential versions in more detail, we have conducted a thorough study of scalability on the fully heterogeneous network. For that purpose, Table 7 shows the performance gain of heterogeneous algorithms with regards to their respective sequential versions as the number of processors was increased on the heterogeneous cluster. Here, we assumed that processor p_3 (the fastest) was always the master and varied the number of slaves. The construction of speedup plots in heterogeneous environments is not straightforward, mainly because the workers do not have the same relative speed, and therefore the order in which they are added to plot the speedup curve needs to be further analyzed. In order to evaluate the impact of the order

Table 7 Speedups achieved by the proposed parallel heterogeneous algorithms on the fully heterogeneous network (processor p_3 is used as the master).

# CPUs	Strategy #1	Strategy #2	Strategy #3
2	1.93	1.90	1.87
3	2.91	2.92	2.88
4	3.88	3.89	3.67
5	4.83	4.89	4.72
6	5.84	5.81	5.74
7	6.75	6.83	6.55
8	7.63	7.76	7.61
9	8.81	8.74	7.65
10	9.57	9.68	9.53
11	10.62	10.65	10.44
12	11.43	11.55	11.41
13	12.25	12.42	12.36
14	13.16	13.32	13.29
15	14.22	14.25	14.22
16	15.19	15.22	15.16

of selection of slaves, we have tested three different ordering strategies:

1. *Strategy #1*. First, we used an ordering strategy based on increasing the number of processors according to their processor numbers in Table 1, i.e., the first case study tested (labeled as “2 CPUs” in Table 7 consisted of using processor p_3 as the master and processor p_0 as the slave; the second case tested (labeled as “3 CPUs” in Table 7 consisted of using processor p_3 as the master and processors p_0 and p_1 as slaves, and so on, until a final case (labeled as “15 CPUs” in Table 7 was tested, based on using processor p_3 as the master and all remaining 15 processors as slaves.
2. *Strategy #2*. Second, we used an ordering strategy based on the relative speed of processors in Table 1, i.e., the first case study tested (labeled as “2 CPUs” in Table 7 consisted of using processor p_3 as the master and processor p_{10} (i.e., the one with lowest relative speed) as the slave; the second case tested (labeled as “3 CPUs” in Table 7 consisted of using processor p_3 as the master and processors p_{10} and p_{11} (i.e., the two processors with lowest relative speed) as slaves, and so on, until a final case (labeled as “15 CPUs” in Table 7 was tested, based on using processor p_3 as the master and all remaining 15 processors as slaves.
3. *Strategy #3*. Finally, we also used a random ordering strategy, i.e., the first case study tested (labeled as “2 CPUs” in Table 7 consisted of using processor p_3 as the master and a different processor, selected randomly among the remaining processors (say, processor p_i) as the slave; the second case (labeled

as "3 CPUs" in Table 7 consisted of using processor p_3 as the master, processor p_i as the first slave, and a different processor, selected randomly among the remaining processors, as the second slave, and so on, a final case was tested (labeled as "15 CPUs" in Table 7, based on using processor p_3 as the master and all remaining 15 processors as slaves.

As shown by Table 7, the incorporation of additional processing nodes by means of the three ordering strategies tested provided almost linear performance increase (regardless of the relative speed of the nodes). Although the results presented for homogeneous and heterogeneous clusters above demonstrate that the proposed multiprocessor implementation of the PPI algorithm is satisfactory from the viewpoint of algorithm scalability, code reusability and load balance, there are several restrictions in order to incorporate this type of platform for onboard processing in remote sensing missions. Although the idea of mounting clusters and networks of processing elements onboard airborne and satellite hyperspectral imaging facilities has been explored in the past, the number of processing elements in such experiments has been very limited thus far due to several reasons. For instance, a mini-Beowulf, portable Myrinet cluster (with similar specifications as those of the homogeneous network of 16 workstations used in experiments) was recently developed for the purpose of low power supercomputing at NASA's Goddard Space Flight Center.[4] The portable system, called Proteus and composed of 12 mini-ITX (PC) nodes, was developed for the purpose of spacecraft/satellite data processing and also used as a mobile cluster for field processing of collected data. The cost of the portable cluster was only 3,000 U.S. dollars. Unfortunately, it could still not facilitate real-time performance since the measured processing times were similar to those reported in Table 6, and the incorporation of additional processing elements to the cluster was reportedly difficult due to heat, power and weight considerations which limited its exploitation for onboard processing.

5.3.3 Parallel Performance on the FPGA System

As an alternative to cluster computing, FPGA-based computing provides several advantages for image processing, such as increased computational power, adaptability to different applications via reconfigurability, and compact size [26]. Specifically, the cost of the Xilinx Virtex-II XC2V6000-6 FPGA used for experiments in this work is currently only slightly higher than that of the portable Myrinet cluster mentioned in the previous subsection (3,000 U.S. dollars). However, the mobile cluster required several 9U VME motherboards to accommodate the multiple processors, with an approximate weight of 14 lb and required power of 300 W. On the other hand, the Xilinx Virtex-II FPGA required only one 3U Compact PCI card (weight below 1 lb) and power of approximately 25 W, offering full realtime reconfigurability. These are very important consideration from the viewpoint of remote sensing mission payload requirements, which are widely regarded as a key aspect for sensor design and operation. In particular, it is important to note that electronic components and hardware susceptible of compromising mission payload are often discarded from Earth observation instruments, and therefore evaluating the potential use of FPGAs as an alternative to much heavier computer equipment is of great importance for remote sensing mission design and planning.

In order to fully substantiate the performance of our systolic array-based FPGA implementation, we should first describe the scalability of the systolic array. The peak performance of the array is mainly determined by the dot-product capacity, that is the number of additions/subtractions executed in 1 s. It is expressed (in millions of operations per second) as follows:

$$P_{peak} = \frac{F \times T \times N}{P}, \tag{8}$$

where F is the frequency in MHz, T is the total number of pixel vectors, N is the number of spectral bands in the input scene, and P is the number of processors of the systolic array. The above formula assumes that the array is constantly fed, i.e., on each cycle a new data is available on its input. Unfortunately, this may not be the case, especially if we consider a reconfigurable board plugged trough the IO bus system of the micro-processor. The PPI algorithm proceeds into T/P passes, and each pass requires flushing the hyperspectral image from the main memory to the array. Thus, instead of considering that a data is present every clock cycle, it is better to consider the transfer capacity of the I/O bus for estimating the average performance of the array. The average performance is estimated as follows:

$$P_{avg} = \frac{B_w \times T \times N}{P}, \tag{9}$$

where B_w denotes the bandwidth in Mbytes/second.

[4]http://thunderhead.gsfc.nasa.gov/PDF/Low_Power.pdf

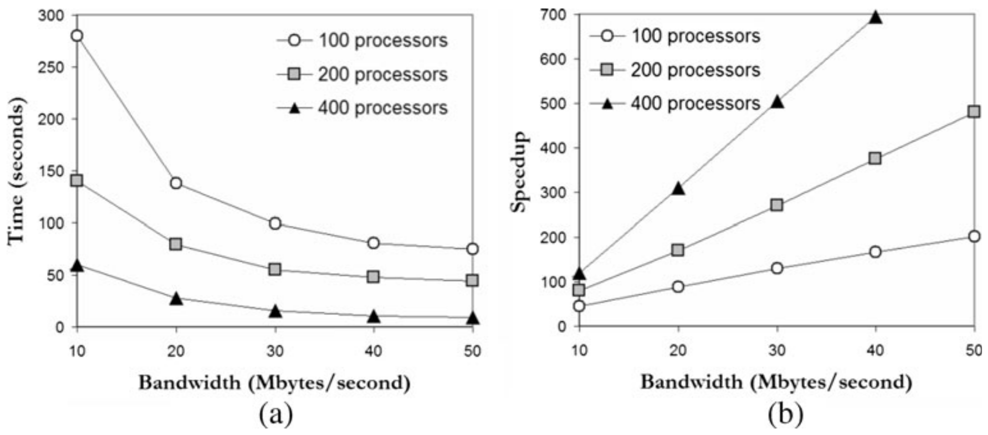


Figure 9 **a** Time, in seconds, for executing the PPI algorithm on a reconfigurable board connected to a PC through the I/O bus. **b** Achieved speedup compared to a single-processor version running on a single Thunderhead node.

Now, if we want to estimate the execution time for computing the PPI algorithm, t_{exec}, the available bandwidth should be taken into consideration as follows:

$$t_{exec} = \frac{P \times T \times N}{B_w}, \tag{10}$$

Using the above rationale [45], we have performed an estimation of the computing time and speedup that can be achieved by the proposed FPGA implementation of the PPI on the considered AVIRIS Cuprite scene using a reconfigurable board connected to a microprocessor through its I/O bus. Figure 9a shows the estimated computing times considering various bandwidths (from $F = 10$ Mbytes/s to $F = 50$ Mbytes/s) and various numbers of processors ($P = 100$, $P = 200$ and $P = 400$). On the other hand, Fig. 9b shows the speedups compared to a single-processor run of the PPI in one of the Thunderhead nodes (which took 1163.05 s), again with a bandwidth ranging from 10 to 50 Mbytes/s and a systolic array with 100, 200 and 400 processors. As it can be seen in Fig. 9, the achieved speedups can be very high, reducing hours of computation to only a few seconds.

In the following, we validate the theoretical estimations given in Fig. 9 on a real Xilinx FPGA architecture. Table 8 shows a summary of resource utilization by the proposed systolic array-based implementation of the PPI on a complete system (systolic array plus PCI interface), implemented on a XC2V6000-6 board, using different numbers of processors. We measured an average PCI bandwidth of 15 Mbytes between the PC and the board, leading to a speed-up of 120 when running the PPI algorithm with 400 processors. It should be noted that, in our experimentation, the performance was seriously limited by the transfer rate between the PC and the board: the array is able to absorb a pixel flow of above 40 Mbytes/s, while the PCI interface can only provide a flow of 15 Mbytes. This experiment, however, demonstrates that the considered board, even with a non-optimized PCI connection (with no DMA), can still yield very good speedup for the PPI algorithm, with a final measured processing time of about 20 s for $P = 400$ processors. This response, although not in real-time, can still be further optimized by increasing the number of processors in the FPGA. However, in our opinion it is very important to leave room in the FPGA for additional algorithms so that dynamic algorithm selection can be performed on the fly. In addition, it is worth noting that full reconfigurability requires additional logic and space in the FPGA [54]. Therefore, we have decided to report realistic experiments by resorting to a moderate amount of resources (gates) in the considered FPGA board. As shown by Table 8, when 400 processors are used, only 36% of the total available resources in the FPGA are consumed. This opens appealing perspectives from an application point of view, such as the possibility of adaptively selecting one out of a pool of algorithms to be applied on-board.

Table 8 Summary of resource utilization for the FPGA-based implementation of the PPI algorithm (operation frequency is given in MHz and processing time in seconds).

Number of processors	Total gates	Total slices	% of total	Operation frequency	Processing time
100	97,443	1,185	3%	29,257	69.91
200	212,412	3,587	10%	21,782	35.36
400	526,944	12,418	36%	18,032	20.48

5.3.4 Parallel Performance on the GPU

In this subsection we compare the performance of the CPU and GPU-based implementations by measuring the execution time as a function of the image size, where the largest one corresponds to the full hyperspectral scene (1,939 × 677 pixels and 224 bands) whereas the others correspond to cropped portions of the same image. Table 9 shows the execution times measured for different image sizes by the CPU and GPU-based

Table 9 Processing time (in milliseconds) for the CPU and GPU-based implementations of the PPI.

Size (MB)	Processing time (CPU)	Processing time (GPU)
68	81.5	6.3
136	162.7	10.7
205	244.2	15.9
273	325.2	21.5
410	489.6	32.1
574	685.4	45.2

implementations, respectively. To determine the program execution time, the C function *clock()* was used for the CPU implementation and the CUDA timer was used to measure time for the GPU implementation. The total time measurement is started right after the hyperspectral image file is read to the CPU memory and stopped right after the resulting PPI map is obtained and stored in the CPU memory.

From Table 9, it is worth noting that the AVIRIS data cube was processed in about 45 s, in spite of the overheads involved in data transfer between the main memory and the GPU. Although the processing time measured in the GPU is about two times higher than the one measured in the FPGA, the cost in U.S. dollars of the GPU device is about ten times lower than the cost of the FPGA device. Results in Table 9 further demonstrate that the complexity of the implementation scales linearly with the problem size, i.e., doubling the image size doubles the execution time. It can also be seen from the tables that the speedups achieved by the GPU implementations over their CPU counterparts remained close to 15 for all considered image sizes, which confirms our intuition that GPUs are indeed suitable for parallel processing of hyperspectral data cubes. The above results are significant, in particular, taking into account that one single GPU device was used in experiments (networks of GPUs may significantly improve the speedup factors).

5.4 Discussion

Through the detailed analysis of the PPI algorithm, we have explored different systems and strategies to increase computational performance of the algorithm (which can take up to several hours of computation to complete its calculations in latest-generation desktop computers). Two of the considered techniques, i.e., commodity cluster-based parallel computing and heterogeneous parallel computing seem particularly appropriate for efficient information extraction from very large hyperspectral data archives. In this regard, we have provided a detailed discussion on the effects that platform heterogeneity has on degrading parallel performance of hyperspectral image and signal processing algorithms. The evaluation strategy conducted in this work was based on experimentally assessing heterogeneous algorithms by comparing their efficiency on (fully or partially) heterogeneous networks of workstations with the efficiency achieved by their homogeneous versions on equally powerful homogeneous networks. Our study reveals that the combination of the (readily available) computational power offered by heterogeneous computing with the recent advances in sensor technology is likely to introduce substantial changes in the (mostly homogeneous and expensive) parallel systems currently used by NASA and other agencies for exploiting the sheer volume of Earth and planetary remotely sensed data which is already available in data repositories.

To fully address the time-critical constraints introduced by many remote sensing applications, we have also developed FPGA and GPU-based implementations of the hyperspectral data processing chain for on-board analysis (before the hyperspectral data is transmitted to Earth). A major goal is to overcome an existing limitation in many remote sensing and observatory systems: the bottleneck introduced by the bandwidth of the downlink 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 provides a response in (near) real-time which is believed to be acceptable in most remote sensing applications. 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 real-time in applications that certainly demand so, such as military target detection, wildland fire monitoring and tracking, oil spill quantification, etc. Although the experimental results presented in this paper are encouraging, further work is still needed to arrive to optimal parallel design and implementations for the PPI and other hyperspectral image and signal processing algorithms.

6 Conclusions

Remotely sensed hyperspectral data processing exemplifies a subject area that has drawn together an eclectic collection of participants. However, a common requirement in most available hyperspectral image and signal processing applications is the need to develop efficient systems and architectures able to cope with the extremely high dimensionality of the data. In this paper, we have taken a necessary first step towards the understanding and assimilation of advanced parallel and distributed systems and architectures in remote sensing applications. We have also discussed some of the problems that need to be addressed in order to translate the tremendous advances in our ability to gather and store remotely sensed hyperspectral data into fundamental, application-oriented scientific advances through the design of efficient data processing algorithms. Specifically, four innovative parallel implementations have been introduced and evaluated from the viewpoint of both algorithm accuracy and parallel performance. Techniques discussed include a commodity cluster-based implementation, a heterogeneity-aware parallel implementation developed for distributed networks of workstations, an FPGA-based hardware implementation, and a GPU-based implementation. The discussed techniques provide a snapshot of the state-of-the-art in current remote sensing research which, despite the enormous computational demands and potential societal impact, has not yet developed standardized parallel-solution algorithms able to process high-dimensional data sets. Regarding our future research avenues, we are currently working towards the implementation of the proposed parallel techniques on other massively parallel computing architectures, such as NASA's Project Columbia and Grid/heterogeneous computing environments.

Acknowledgements 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). Funding from Junta de Extremadura (local government) under project PRI09A110 is also gratefully acknowledged. The authors gratefully thank Profs. Manuel Prieto and Franciscto Tirado and their group at Complutense University of Madrid for many fruitful discussions and for his collaboration in the generation of some of the figures presented in the paper.

References

1. Goetz, A.F.H., Vane, G., Solomon, J.E., & Rock, B.N. (1985). Imaging spectrometry for Earth remote sensing. *Science, 228*, 1147–1153.
2. Schowengerdt, R.A. (2007). *Remote sensing*, 3rd edn. New York: Academic.
3. Green, R.O., et al. (1998). Imaging spectroscopy and the airborne visible/infrared imaging spectrometer (AVIRIS). *Remote Sensing of Environment, 65*, 227–248.
4. Chang, C.-I. (2003). *Hyperspectral imaging: Techniques for spectral detection and classification*. New York: Kluwer Academic.
5. Crookes, D. (1999). Architectures for high performance image processing: The future. *Journal of Systems Architecture, 45*, 739–748.
6. Jeon, J., Kim, H.-S., Choi, G., & Park, H. (2000). KAIST image computing system (KICS): A parallel architecture for real-time multimedia data processing. *Journal of Systems Architecture, 46*, 1403–1418.
7. Chen, L., Fujishiro, I., & Nakajima, K. (2003). Optimizing parallel performance of unstructured volume rendering for the Earth simulator. *Parallel Computing, 29*, 355–371.
8. Aloisio, G., & Cafaro, M. (2003). A dynamic earth observation system. *Parallel Computing, 29*, 1357–1362.
9. Hawick, K.A., Coddington, P.D., & James, H.A. (2003). Distributed frameworks and parallel algorithms for processing large-scale geographic data. *Parallel Computing, 29*, 1297–1333.
10. Wang, P., Liu, K.Y., Cwik, T., & Green, R.O. (2002). MODTRAN on supercomputers and parallel computers. *Parallel Computing, 28*, 53–64.
11. Tehranian, S., Zhao, Y., Harvey, T., Swaroop, A., & McKenzie, K. (2006). A robust framework for real-time distributed processing of satellite data. *Journal of Parallel and Distributed Computing, 66*, 403–418.
12. Plaza, A., Valencia, D., Plaza, J., & Martinez, P. (2006). Commodity cluster-based parallel processing of hyperspectral imagery. *Journal of Parallel and Distributed Computing, 66*(3), 345–358.
13. Landgrebe, D.A. (2003). *Signal theory methods in multispectral remote sensing*. Hoboken: Wiley.
14. Dorband, J., Palencia, J., Ranawake, U. (2003). Commodity computing clusters at Goddard Space Flight Center. *Journal of Space Communication, 1*(3), 113–123. Available online: http://satjournal.tcom.ohiou.edu/pdf/Dorband.pdf.
15. Brightwell, R., Fisk, L.A., Greenberg, D.S., Hudson, T., Levenhagen, M., Maccabe, A.B., et al. (2000). Massively parallel computing using commodity components. *Parallel Computing, 26*, 243–266.
16. Kalluri, S., Zhang, Z., JaJa, J., Liang, S., & Townshend, J. (2001). Characterizing land surface anisotropy from AVHRR data at a global scale using high performance computing. *International Journal of Remote Sensing, 22*, 2171–2191.
17. Tilton, J.C. (2005). Method for implementation of recursive hierarchical segmentation on parallel computers. *U.S. Patent Office, Washington, DC, U.S. Pending Published Application 09/839147*, 2005. Available online: http://www.fuentek.com/technologies/rhseg.htm.
18. Le Moigne, J., Campbell, W.J., & Cromp, R.F. (2002). An automated parallel image registration technique based on the correlation of wavelet features. *IEEE Transactions on Geoscience and Remote Sensing, 40*, 1849–1864.
19. Achalakul, T., & Taylor, S. (2003). A distributed spectral-screening PCT algorithm. *Journal of Parallel and Distributed Computing, 63*, 373–384.
20. Lastovetsky, A. (2003). *Parallel computing on heterogeneous networks*. Hoboken: Wiley-Interscience.
21. Dhodhi, M.K., Saghri, J.A., Ahmad, I., & Ul-Mustafa, R. (1999). D-ISODATA: A distributed algorithm for unsupervised classification of remotely sensed data on network of

workstations. *Journal of Parallel and Distributed Computing, 59*, 280–301.
22. Hawick, K., James, H., Silis, A., Grove, D., Pattern, C., Mathew, J., et al. (1999). DISCWorld: An environment for service-based meta-computing. *Future Generation Computer Systems, 15*, 623–635.
23. Foster, I., & Kesselman, C. (1999). *The grid: Blueprint for a new computing infrastructure*. San Francisco: Morgan Kaufman.
24. Vladimirova, T., & Wu, X. (2006). On-board partial run-time reconfiguration for pico-satellite constellations. *First NASA/ESA Conference on Adaptive Hardware and Systems*, AHS.
25. El-Araby, E., El-Ghazawi, T., & Le Moigne, J. (2004). Wavelet spectral dimension reduction of hyperspectral imagery on a reconfigurable computer. In *Proceedings of the 4th IEEE international conference on field-programmable technology*.
26. Valencia, D., & Plaza, A. (2006). FPGA-based compression of hyperspectral imagery using spectral unmixing and the pixel purity index algorithm. *Lecture Notes in Computer Science, 3993*, 24–31.
27. Setoain, J., Prieto, M., Tenllado, C., Plaza, A., & Tirado, F. (2007). Parallel morphological endmember extraction using commodity grahics hardware. *IEEE Geoscience and Remote Sensing Letters, 43*, 441–445.
28. Boardman, J., Kruse, F.A., & Green, R.O. (1995). *Mapping target signatures via partial unmixing of AVIRIS data*. Summaries of the NASA/JPL Airborne Earth Science Workshop, Pasadena, CA.
29. ITT Visual Information Solutions. (2008). ENVI users guide. Boulder: ITTVIS. Online: http://www.ittvis.com.
30. Plaza, A., & Chang, C.-I. (2007). *High performance computing in remote sensing*. Boca Raton: CRC.
31. Plaza, A., Martinez, P., Perez, R., & Plaza, J. (2002). Spatial-spectral endmember extraction by multidimensional morphological operations. *IEEE Transactions on Geoscience and Remote Sensing, 40*, 2025– 2041.
32. Plaza, A., Plaza, J., & Valencia, D. (2007). Impact of platform heterogeneity on the design of parallel algorithms for morphological processing of high-dimensional image data. *Journal of Supercomputing, 40*, 81–107.
33. Plaza, A., Plaza, J., & Valencia, D. (2006). AMEEPAR: Parallel morphological algorithm for hyperspectral image classification in heterogeneous networks of workstations. *Lecture Notes in Computer Science, 3391*, 888–891.
34. Plaza, A., Martinez, P., Perez, R., & Plaza, J. (2004). A quantitative and comparative analysis of endmember extraction algorithms from hyperspectral data. *IEEE Transactions on Geoscience and Remote Sensing, 42*, 650–663.
35. Chang, C.-I., & Plaza, A. (2006). A fast iterative implementation of the pixel purity index algorithm. *IEEE Geoscience and Remote Sensing Letters, 3*, 63–67.
36. Plaza, A., & Chang, C.-I. (2006). Impact of initialization on design of endmember extraction algorithms. *IEEE Transactions on Geoscience and Remote Sensing, 44,* 3397–3407.
37. Keshava, N., & Mustard, J.F. (2002). Spectral unmixing. *IEEE Signal Processing Magazine, 19*, 44–57.
38. Seinstra, F.J., Koelma, D., & Geusebroek, J.M. (2002). A software architecture for user transparent parallel image processing. *Parallel Computing, 28*, 967–993.
39. Veeravalli, B., & Ranganath, S. (2003). Theoretical and experimental study on large size image processing applications using divisible load paradigm on distributed bus networks. *Image and Vision Computing, 20*, 917–935.
40. Gropp, W., Huss-Lederman, S., Lumsdaine, A., & Lusk, E. (1999). In *MPI-Ů-the complete reference, the MPI extensions* (Vol. 2). Cambridge: MIT.
41. Lastovetsky, A., & Reddy, R. (2006). HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. *Journal of Parallel and Distributed Computing, 66*, 197–220.
42. MPICH Team. (2005). MPICH: A portable implementation of MPI. Available online: http://www-unix.mcs.anl.gov/mpi/mpich.
43. Kung, S.Y. (1988). *VLSI array processors*. Upper Saddle River: Prentice-Hall.
44. Petkov, N. (1992). *Systolic parallel processing*. The Netherlands: North Holland.
45. Lavernier, D., Fabiani, E., Derrien, S., & Wagner, C. (1999). Systolic array for computing the pixel purity index algorithm on hyperspectral images. In *Proceedings of SPIE* (Vol. 4480, pp. 130–138).
46. Celoxica Ltd., (2003). *Handel-C language reference manual*.
47. Celoxica Ltd. (2003). *DK design suite user manual*. Available online: http://www.celoxica.com.
48. Xilinx Inc. (2010). Available online: http://www.xilinx.com (last check: January 2010).
49. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A.E., et al. (2005). A survey of general purpose computation on graphics hardware. In *Proceedings of Eurographics: State of the art reports* (pp. 21–51).
50. Montrym, J., Moreton, H. (2005). The GeForce 68000. *IEEE Micro, 25*, 41–51.
51. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., et al. (2004). Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics, 23*, 777–786.
52. Lastovetsky, A., & Reddy, R. (2004). On performance analysis of heterogeneous parallel algorithms. *Parallel Computing, 30*, 1195–1216.
53. Hennessy, J.L., & Patterson, D.A. (2002). *Computer architecture: A quantitative approach*, 3rd ed. San Mateo: Morgan Kaufmann.
54. Jyotheswar, J., & Mahapatra, S. (2007). Efficient FPGA implementation of DWT and modified SPIHT for lossless image compression. *Journal of Systems Architecture, 53*, 369–378.

Antonio Plaza is currently an Associate Professor with the Department of Technology of Computers and Communications, University of Extremadura, Spain, where he received his PhD degree in 2002.

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). Dr. Plaza has been visiting researcher/professor at several institutions, including the Computational and Information Sciences and Technology Office (CISTO) at NASA's Goddard Space Flight Center, Greenbelt, Maryland; the Remote Sensing, Signal and Image Processing Laboratory (RSSIPL) at the Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County; the Microsystems Laboratory at the Department of Electrical & Computer Engineering, University of Maryland, College Park (UMCP); and the AVIRIS group at NASA's Jet Propulsion Laboratory, Pasadena, California. 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. Dr. Plaza is the project coordinator of the Hyperspectral Imaging Network (Hyper-I-Net), a four-year Marie Curie Research Training Network designed to build an interdisciplinary European research community focused on hyperspectral imaging activities. He is author or co-author of more than 200 publications including journal papers, book chapters and peer-reviewed conference proceedings. Additional information about the research activities of Dr. Plaza is available at http://www.umbc.edu/rssipl/people/aplaza.

Javier Plaza is currently an Assistant Profesor with the Department of Technology of Computers and Communications, University of Extremadura, Spain, where he received his PhD degree in 2008. His current research work is focused on the development of efficient implementations of neural network-based algorithms for analysis and classification of hyperspectral scenes. He is also involved in the design and configuration of homogeneous and fully heterogeneous parallel computing architectures for high-performance scientific applications. Other major research interests of Dr. Plaza include networking configuration and training of neural network architectures architectures for specific applications. He has served as a reviewer for several different journals in the aforementioned areas and is author or co-author of more than 80 publications including journal papers, book chapters and peer-reviewed conference proceedings. Additional information about the research activities of Dr. Plaza is available at http://www.umbc.edu/rssipl/people/jplaza.

Hugo Vegas obtained his Computer Engineer degree in 2008 and is currently a Research Associate with the ArTeCS Group, Department of Computer Architecture at the Complutense University of Madrid, Spain. His current research interests cover the development of parallel algorithms in different types of computer architectures, including clusters of computers, commodity graphics processing units (GPUs) and ditributed computing using Kernel for Adaptative, Asynchronous Parallel and Interactive programming (KAAPI).