Cluster versus GPU implementation of an Orthogonal Target Detection Algorithm for Remotely Sensed Hyperspectral Images

Abel Paz and Antonio Plaza
Hyperspectral Computing Laboratory
Department of Technology of Computers and Communications
University of Extremadura, Avda. de la Universidad s/n
E-10071 Caceres, Spain
Email: {apazgal, aplaza}@unex.es

***Abstract*—Remotely sensed hyperspectral imaging instruments provide high-dimensional data containing rich information in both the spatial and the spectral domain. In many surveillance applications, detecting objects (targets) is a very important task. In particular, algorithms for detecting (moving or static) targets, or targets that could expand their size (such as propagating fires) often require timely responses for swift decisions that depend upon high computing performance of algorithm analysis. In this paper, we develop parallel versions of a target detection algorithm based on orthogonal subspace projections. The parallel implementations are tested in two types of parallel computing architectures: a massively parallel cluster of computers called Thunderhead and available at NASAs Goddard Space Flight Center in Maryland, and a commodity graphics processing unit (GPU) of NVidia^{TM} GeForce GTX 275 type. While the cluster-based implementation reveals itself as appealing for information extraction from remote sensing data already transmitted to Earth, the GPU implementation allows us to perform near real-time anomaly detection in hyperspectral scenes, with speedups over 50x with regards to a highly optimized serial version. The proposed parallel algorithms are quantitatively evaluated using hyperspectral data collected by the NASAs Airborne Visible Infra-Red Imaging Spectrometer (AVIRIS) system over the World Trade Center (WTC) in New York, five days after the attacks that collapsed the two main towers in the WTC complex.**

***Index Terms*—Hyperspectral data, target detection, clusters of computers, graphics processing units (GPUs).**

I. Introduction

Hyperspectral imaging instruments such as the NASA Jet Propulsion Laboratory's Airborne Visible Infra-Red Imaging Spectrometer (AVIRIS) [1] are now able to record the visible and near-infrared spectrum (wavelength region from 0.4 to 2.5 micrometers) of the reflected light of an area 2 to 12 kilometers wide and several kilometers long using 224 spectral bands. The resulting "image cube" (see Fig. 1) is a stack of images in which each pixel (vector) has an associated spectral signature or *fingerprint* that uniquely characterizes the underlying objects [2]. The resulting data volume typically comprises several GBs per flight [3].

The special properties of hyperspectral data have significantly expanded the domain of many analysis techniques. For instance, automatic target and anomaly detection are considered very important tasks for hyperspectral data exploitation in defense and security applications [4], [5]. Among several developed techniques for this purpose [6], [7], an orthogonal subspace projection (OSP) algorithm [8] has found great success in the task of automatic target detection [9]. Depending on the complexity and dimensionality of the input scene [10], the OSP algorithm may be computationally expensive, a fact that limits the possibility of utilizing the algorithm in time-critical applications [3]. In turn, the wealth of spectral information available in hyperspectral imaging data opens ground-breaking perspectives in many applications, including target detection for military and defense/security deployment [11]. In particular, algorithms for detecting (moving or static) targets, or targets that could expand their size (such as propagating fires) often require timely responses for swift decisions that depend upon high computing performance of algorithm analysis [12].

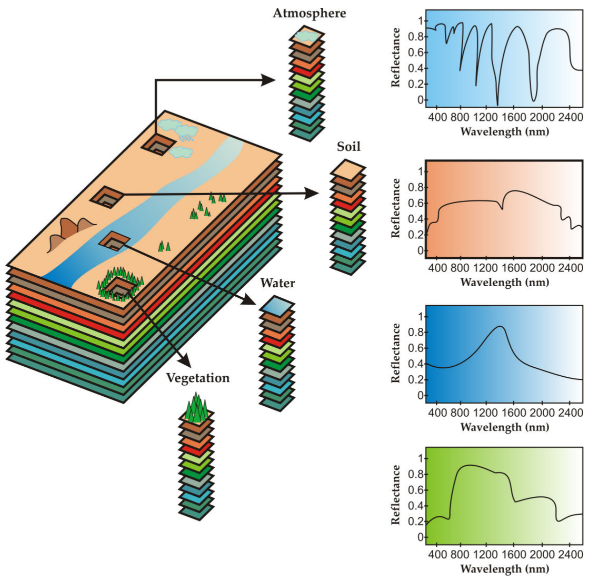

Figure 1. Concept of hyperspectral imaging.

Despite the growing interest in parallel hyperspectral imaging research [13], [14], [15] only a few parallel implementations of automatic target detection algorithms for hyperspectral data exist in the open literature [4]. However, with the recent explosion in the amount and dimensionality of hyperspectral imagery, parallel processing is expected to become a requirement in most remote sensing missions [3]. In the past, Beowulf-type clusters of computers have offered an attractive solution for fast information extraction from hyperspectral data sets already transmitted to Earth [16], [17], [18]. The goal was to create parallel computing systems from commodity components to satisfy specific requirements for the Earth and space sciences community. However, these systems are generally expensive and difficult to adapt to on-board data processing scenarios, in which low-weight and low-power integrated components are essential to reduce mission payload and obtain analysis results in real-time, i.e., at the same time as the data is collected by the sensor. In this regard, an exciting new development in the field of commodity computing is the emergence of commodity graphic processing units (GPUs), which can now bridge the gap towards on-board processing of remotely sensed hyperspectral data [19], [20], [21], [5]. The speed of graphics hardware doubles approximately every six months, which is much faster than the improving rate of the CPUs (even those made up by multiple cores) which are interconnected in a cluster. Currently, state-of-the-art GPUs deliver peak performances more than one order of magnitude over high-end micro-processors. The ever-growing computational requirements introduced by hyperspectral imaging applications can fully benefit from this type of specialized hardware and take advantage of the compact size and relatively low cost of these units, which make them appealing for onboard data processing at lower costs than those introduced by other hardware devices [3], [22].

In this paper, we develop and compare two parallel versions of the OSP algorithm: a cluster version implemented on a massively parallel system called Thunderhead and available at NASA's Goddard Space Flight Center in Maryland, and a GPU version implemented in an NVidia^{TM} card of GeForce GTX 275 type. While the cluster-based implementation reveals itself as appealing for information extraction from remote sensing data already transmitted to Earth, the GPU implementation allows us to perform near real-time anomaly detection in hyperspectral scenes, with speedups over 50x with regards to a highly optimized serial versions. The parallel algorithms are quantitatively evaluated using hyperspectral data collected by the AVIRIS system over the World Trade Center (WTC) in New York, five days after the terrorist attacks that collapsed the two main towers in the WTC complex. The precision of the algorithms is evaluated by quantitatively substantiating their capacity to automatically detect the thermal hot spot of fires (anomalies) in the WTC area. Combined, these parts offer a thoughtful perspective on the potential and emerging challenges in the design of parallel target detection algorithms using high performance computing architectures.

II. Orthogonal Target Detection Algorithm

In this section we briefly describe the target detection algorithm that will be efficiently implemented in parallel (using different high performance computing architectures) in this work. The OSP algorithm [8] was adapted to find potential target pixels that can be used to generate a signature matrix used in an orthogonal approach [9]. Let $\mathbf{x}_0$ be an initial target signature (i.e., the pixel vector with maximum length). The algorithm begins by using an orthogonal projector specified by the following expression:

$$P_{\mathbf{U}}^{\perp} = \mathbf{I} - \mathbf{U}(\mathbf{U}^T\mathbf{U})^{-1}\mathbf{U}^T, \tag{1}$$

which is applied to all image pixels, with $\mathbf{U} = [\mathbf{x}_0]$. It then finds a target signature, denoted by $\mathbf{x}_1$, with the maximum projection in $< \mathbf{x}_0 >^{\perp}$, which is the orthogonal complement space linearly spanned by $\mathbf{x}_0$. A second target signature $\mathbf{x}_2$ can then be found by applying another orthogonal subspace projector $P_{\mathbf{U}}^{\perp}$ with $\mathbf{U} = [\mathbf{x}_0, \mathbf{x}_1]$ to the original image, where the target signature that has the maximum orthogonal projection in $< \mathbf{x}_0, \mathbf{x}_1 >^{\perp}$ is selected as $\mathbf{x}_2$. The above procedure is repeated until a set of target pixels $\{\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_t\}$ is extracted, where t is an input parameter to the algorithm.

III. Implementation for clusters of computers

Clusters of computers are made up of different processing units interconnected via a communication network [23]. In previous work, it has been reported that data-parallel approaches, in which the hyperspectral data is partitioned among different processing units, are particularly effective for parallel processing in this type of high performance computing systems [3], [17], [15]. In this framework, it is very important to define the strategy for partitioning the hyperspectral data. In our implementations, a data-driven partitioning strategy has been adopted as a baseline for algorithm parallelization. Specifically, two approaches for data partitioning have been tested [17]:

- *Spectral-domain partitioning*. This approach subdivides the multi-channel remotely sensed image into small cells or sub-volumes made up of contiguous spectral wavelengths for parallel processing.
- *Spatial-domain partitioning*. This approach breaks the multi-channel image into slices made up of one or several contiguous spectral bands for parallel processing. In this case, the same pixel vector is always entirely assigned to a single processor, and slabs of spatially adjacent pixel vectors are distributed among the processing nodes (CPUs) of the parallel system. Fig. 2 shows two examples of spatial-domain partitioning over 4 processors and over 5 processors, respectively.

Previous experimentation with the above-mentioned strategies indicated that spatial-domain partitioning can significantly reduce inter-processor communication, resulting from the fact that a single pixel vector is never partitioned and communications are not needed at the pixel level [17]. In the following,

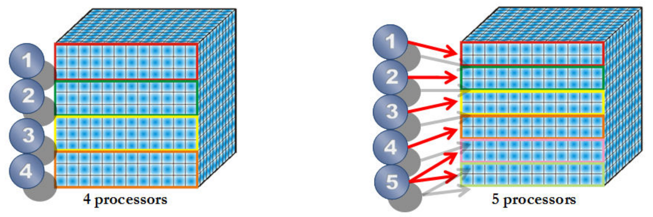

Figure 2. Spatial-domain decomposition of a hyperspectral data set into four (left) and five (right) partitions.

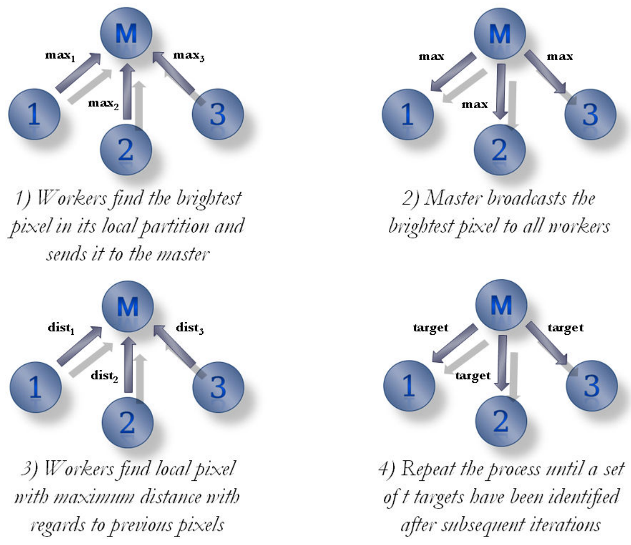
M
max$_1$
max$_2$
max$_3$
1
2
3
1) Workers find the brightest pixel in its local partition and sends it to the master
M
max
max
max
1
2
3
2) Master broadcasts the brightest pixel to all workers
M
dist$_1$
dist$_2$
dist$_3$
1
2
3
3) Workers find local pixel with maximum distance with regards to previous pixels
M
target
target
target
1
2
3
4) Repeat the process until a set of t targets have been identified after subsequent iterations

Figure 3. Graphical summary of the P-OSP algorithm using 1 master processor and 3 slaves.

we assume that spatial-domain decomposition is always used when partitioning the hyperspectral data cube.

The inputs to the parallel algorithm are a hyperspectral image cube $\mathbf{F}$ with n dimensions, where $\mathbf{x}$ denotes the pixel vector of the same scene, and a maximum number of targets to be detected, t. The output in all cases is a set of target pixel vectors $\{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_t\}$. The algorithm has been implemented in the C++ programming language using calls to MPI, the *message passing interface* library commonly available for parallel implementations in multi-processor systems[1]. The parallel implementation, denoted by P-OSP and summarized by a diagram in Fig. 3, consists of the following steps:

1) The master divides the original image cube $\mathbf{F}$ into P spatial-domain partitions. Then, the master sends the partitions to the workers.
2) Each worker finds the brightest pixel in its local partition (local maximum) using $\mathbf{x}_1 = argmax\{\mathbf{x}^T \cdot \mathbf{x}\}$, where the superscript T denotes the vector transpose operation. Each worker then sends the spatial locations of the pixel identified as the brightest one in its local partition back to the master. For illustrative purposes, Fig. 4 shows the piece of C++ code that the workers execute in order to send their local maxima to the master node using the MPI function MPI_send. Here, localmax is the local maximum at the node given by identifier node_id, where node_id $= 0$ for the master and node_id > 0 for the workers. MPI_COMM_WORLD is the name of the *communicator* or collection of processes that are running concurrently in the system (in our case, all the different parallel tasks allocated to the P workers).
3) Once all the workers have completed their parts and sent their local maxima, the master finds the brightest pixel of the input scene (global maximum), $\mathbf{x}_1$, by applying the $argmax$ operator in step 2 to all the pixels at the spatial locations provided by the workers, and selecting the one that results in the maximum score. Then, the master sets $\mathbf{U} = \mathbf{x}_1$ and broadcasts this matrix to all workers. As shown by Fig. 4, this is implemented (in the workers) by a call to MPI_Recv that stops the worker until the value of the global maximum globalmax is received from the master. On the other hand, Fig. 5 shows the code designed for calculation of the global maximum at the master. First, the master receives all the local maxima from the workers using the MPI_Gather function. Then, the worker which contains the global maximum out of the local maxima is identified in the for loop. Finally, the global maximum is broadcast to all the workers using the MPI_Bcast function.
4) After this process is completed, each worker now finds (in parallel) the pixel in its local partition with the maximum orthogonal projection relative to the pixel vectors in $\mathbf{U}$, using a projector given by $P_{\mathbf{U}}^{\perp} = \mathbf{I} - \mathbf{U}(\mathbf{U}^T\mathbf{U})^{-1}\mathbf{U}^T$, where $\mathbf{U}$ is the identity matrix. The orthogonal space projector $P_{\mathbf{U}}^{\perp}$ is now applied to all pixel vectors in each local partition to identify the most distinct pixels (in orthogonal sense) with regards to the previously detected ones. Each worker then sends the spatial location of the resulting local pixels to the master node.
5) The master now finds a second target pixel by applying the $P_{\mathbf{U}}^{\perp}$ operator to the pixel vectors at the spatial locations provided by the workers, and selecting the one which results in the maximum score as follows $\mathbf{x}_2 = argmax\{(P_{\mathbf{U}}^{\perp}\mathbf{x})^T(P_{\mathbf{U}}^{\perp}\mathbf{x})\}$. The master sets $\mathbf{U} = \{\mathbf{x}_1, \mathbf{x}_2\}$ and broadcasts this matrix to all workers.
6) Repeat from step 4 until a set of t target pixels, $\{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_t\}$, are extracted from the input data.

[1]http://www.mcs.anl.gov/research/projects/mpi

```
if ((node_id > 0)&&(node_id < num_nodes)) {
  // Worker sends the local maxima to the master node
  MPI_Send(&localmax,1,MPI_DOUBLE,0,node_id,MPI_COMM_WORLD);
  // Worker waits until it receives the global maximum from the master
  MPI_Recv(&globalmax,1,MPI_INT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
}
```

Figure 4. Portion of the code of a worker in our P-OSP implementation, in which the worker sends a pre-computed local maximum to the master and waits for a global maximum from the master.

```
// The master processor performs the following operations:
max_aux [0] = max;
max_partial = max;
globalmax=0;

// The master receives the local maxima from the workers
MPI_Gather(localmax,1,MPI_DOUBLE,max_aux,1,MPI_DOUBLE,0,
MPI_COMM_WORLD);

// MPI_Gather is equivalent to:
// for(i=1;i<num_nodes;i++)
//    MPI_Recv(&max_aux[i],1,MPI_DOUBLE,i,MPI_ANY_TAG,
//    MPI_COMM_WORLD,&status);

// The worker with the global maximum is identified
for(i=1;i<num_nodes;i++){
      if(max_partial < max_aux[i]){
           max_partial=max_aux[i];
           globalmax=i;}}

// Master sends all workers the id of the worker with global maximum
MPI_Bcast(&globalmax,1,MPI_INT,0,MPI_COMM_WORLD);

// MPI_Bcast is equivalent to:
// for(i=1;i<num_nodes;i++)
//    MPI_Send(&globalmax,1,MPI_INT,i,0,MPI_COMM_WORLD);
```

Figure 5. Portion of the code of the master in our P-OSP implementation, in which the master receives the local maxima from the workers, computes a global maximum and sends all workers the id of the worker which contains the global maximum.

IV. Parallel Implementation for GPUs

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 [17] made up of entire pixel vectors (see Fig. 2), i.e., as in our cluster-based implementations, each spatial-domain partition incorporates all the spectral information on a localized spatial region and is composed of spatially adjacent pixel vectors. 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 [19]. Such partitioning allows us to map four consecutive spectral bands onto the RGBA color channels of a texture element. Once the procedure adopted for data partitioning has been described, our GPU version of the OSP algorithm (G-OSP hereinafter) can be summarized by the following steps:

1) Once the hyperspectral image is mapped onto the GPU memory, a structure (grid) in which the number of blocks equals the number of lines in the hyperspectral image and the number of threads equals the number of samples is created, thus making sure that all pixels in the hyperspectral image are processed in parallel (if this is not possible due to limited memory resources in the GPU, `CUDA` automatically performs several iterations, each of which processes as many pixels as possible in parallel).
2) Using the aforementioned structure, calculate the brightest pixel $\mathbf{x}_1$ in the original hyperspectral scene by means of a `CUDA` kernel which performs part of the calculations to compute $\mathbf{x}_1 = argmax\{\mathbf{x}^T \cdot \mathbf{x}\}$ after computing (in parallel) the dot product between each pixel vector $\mathbf{x}$ in the original hyperspectral image and its own transposed version $\mathbf{x}^T$. For illustrative purposes, Fig. 6 shows a portion of code which includes the definition of the number of blocks `numBlocks` and the number of processing threads per block `numThreadsPerBlock`, and then calls the `CUDA` kernel `BrightestPixel` that computes the value of $\mathbf{x}_1$. Here, `d_bright_matrix` is the structure that stores the output of the computation $\mathbf{x}^T \cdot \mathbf{x}$ for each pixel. Fig. 7 shows the code of the `CUDA` kernel `BrightestPixel`, in which each different thread computes a different value of $\mathbf{x}^T \cdot \mathbf{x}$ for a different pixel (each thread is given by an identification number `idx`, and there are as many concurrent threads as pixels in the original hyperspectral image). Once all the concurrent threads complete their calculations, the G-ATDCA implementation simply computes the value in `d_bright_matrix` with maximum associated value and obtains the pixel in that position, labeling the pixel as $\mathbf{x}_1$.
3) Once the brightest pixel in the original hyperspectral image has been identified as the first target, the pixel (the vector of bands) is allocated as the first column in the U matrix $\mathbf{U} = [\mathbf{x}_1]$.
4) A kernel of *i* blocks and *i* threads is created to calculate $\mathbf{U}^T \cdot \mathbf{U}$, where **i** is the iteration (between 1 and the desired number of targets).
5) The algorithm calculates the inverse of the previous product using the Gauss-Jordan elimination method: $(\mathbf{U}^T\mathbf{U})^{-1}$.
6) A kernel of *i* blocks and *number of bands* threads is created to multiply $\mathbf{U}$ and the inverse of the previous step: $\mathbf{U}(\mathbf{U}^T\mathbf{U})^{-1}$
7) Another kernel of *number of bands* blocks and *number of bands* threads is created to multiply the previous result by $\mathbf{U}^T$: $\mathbf{U}(\mathbf{U}^T\mathbf{U})^{-1}\mathbf{U}^T$
8) The substraction of the identity matrix is calculated by a kernel of *number of bands* blocks and *number of bands* threads, getting the orthogonal subspace projector: $P_{\mathbf{U}}^{\perp} = \mathbf{I} - \mathbf{U}(\mathbf{U}^T\mathbf{U})^{-1}\mathbf{U}^T$ for the current iteration.
9) Once we have $P_{\mathbf{U}_i}^{\perp}$, a kernel in which the number of blocks equals the number of lines in the hyperspectral image and the number of threads equals the number of samples is created to calculate the *Projection Matrix* obtained by applying the OSP-based projection operator $P_{\mathbf{U}_i}^{\perp}$ to each pixel in the image.
10) Last, the algorithm searches for the maximum of the *Projection Matrix*, obtaining $\mathbf{x}_2$, and extends $\mathbf{U}$ matrix with it: $\mathbf{U} = [\mathbf{x}_1, \mathbf{x}_2]$
11) The steps from 4 to 10 are repeated to find the desired number of targets (specified by parameter t provided as input to the algorithm).

V. Experimental Results

This section is organized as follows. In subsection V-A we describe the AVIRIS hyperspectral data set used in our experiments. Subsection V-B describes the parallel computing platforms used for experimental evaluation, which comprise a Beowulf cluster at NASA's Goddard Space Flight Center in Maryland and an NVidia™ GeForce GTX 275 GPU.

```
// Define the number of blocks and the number of processing threads per block
int numBlocks = num_lines;
int numThreadsPerBlock = num_samples;

// Calculate the intensity of each pixel in the original image and store the resulting values in a structure
BrightestPixel<<<numBlocks,numThreadsPerBlock>>>(d_hyper_image,
d_bright_matrix, num_bands, lines_samples);
```

Figure 6. Portion of code which calls the `CUDA` kernel `BrightestPixel` that computes (in parallel) the brightest pixel in the scene in the G-OSP implementation.

```
__global__   void   BrightestPixel(short   int   *d_hyper_image,   float
*d_bright_matrix, int num_bands, long int lines_samples)
{

// The original hyperspectral image is stored in d_hyper_image
int k;
float bright=0, value;

// Obtain the thread id and assign an operation to each processing thread
int idx = blockDim.x * blockIdx.x + threadIdx.x;

for (k = 0; k < num_bands; k++){
            value = d_hyper_image[idx+(k*lines_samples)];
            bright += value;
      }

      d_bright_matrix[idx]=bright;
}
```

Figure 7. `CUDA` kernel `BrightestPixel` that computes (in parallel) the brightest pixel in the scene in the G-OSP implementation.

Subsection V-C discusses the target and anomaly detection accuracy of the considered parallel algorithm when analyzing the hyperspectral data set described in subsection V-A. Subsection V-D describes the parallel performance results obtained after implementing the P-OSP on the Beowulf cluster. Subsection V-E describes the parallel performance results obtained after implementing the G-OSP on the GPU.

A. Data description

The image scene used for experiments in this work was collected by the AVIRIS instrument, which was flown by NASA's Jet Propulsion Laboratory over the World Trade Center (WTC) area in New York City on September 16, 2001, just five days after the terrorist attacks that collapsed the two main towers and other buildings in the WTC complex. The full data set selected for experiments consists of 614×512 pixels, 224 spectral bands and a total size of (approximately) 140 MB. The spatial resolution is 1.7 meters per pixel. The leftmost part of Fig. 8 shows a false color composite of the data set selected for experiments using the 1682, 1107 and 655 nm channels, displayed as red, green and blue, respectively. Vegetated areas appear green in the leftmost part of Fig. 8, while burned areas appear dark gray. Smoke coming from the WTC area (in the red rectangle) and going down to south Manhattan appears bright blue due to high spectral reflectance in the 655 nm channel.

Extensive reference information, collected by U.S. Geological Survey (USGS), is available for the WTC scene[2]. In this work, we use a U.S. Geological Survey thermal map[3] which

[2]http://speclab.cr.usgs.gov/wtc
[3]http://pubs.usgs.gov/of/2001/ofr-01-0429/hotspot.key.tgif.gif

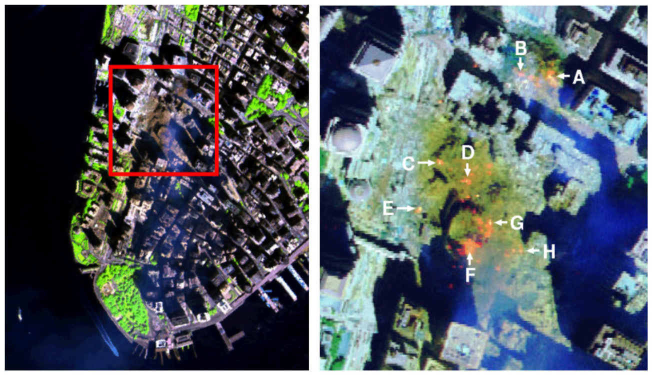

Figure 8. False color composition of an AVIRIS hyperspectral image collected by NASA's Jet Propulsion Laboratory over lower Manhattan on Sept. 16, 2001 (left). Location of thermal hot spots in the fires observed in World Trade Center area, available online: http://pubs.usgs.gov/of/2001/ofr-01-0429/hotspot.key.tgif.gif (right).

Table I
PROPERTIES OF THE THERMAL HOT SPOTS REPORTED IN THE RIGHTMOST PART OF FIG. 8.

Hot spot	**Latitude (North)**	**Longitude (West)**	**Temperature (Kelvin)**	**Area (Square meters)**
‘A’	$40^{o}42‘47.18“$	$74^{o}00‘41.43“$	1000	0.56
‘B’	$40^{o}42‘47.14“$	$74^{o}00‘43.53“$	830	0.08
‘C’	$40^{o}42‘42.89“$	$74^{o}00‘48.88“$	900	0.80
‘D’	$40^{o}42‘41.99“$	$74^{o}00‘46.94“$	790	0.80
‘E’	$40^{o}42‘40.58“$	$74^{o}00‘50.15“$	710	0.40
‘F’	$40^{o}42‘38.74“$	$74^{o}00‘46.70“$	700	0.40
‘G’	$40^{o}42‘39.94“$	$74^{o}00‘45.37“$	1020	0.04
‘H’	$40^{o}42‘38.60“$	$74^{o}00‘43.51“$	820	0.08

shows the target locations of the thermal hot spots at the WTC area, displayed as bright red, orange and yellow spots at the rightmost part of Fig. 8. The map is centered at the region where the towers collapsed, and the temperatures of the targets range from 700F to 1300F. Further information available from USGS about the targets (including location, estimated size, and temperature) is reported on Table I. As shown by Table I, all the targets are sub-pixel in size since the spatial resolution of a single pixel is 1.7 square meters. The thermal map displayed in the rightmost part of Fig. 8 will be used in this work as ground-truth to validate the target detection accuracy of the proposed parallel algorithms and their respective serial versions.

B. Parallel computing platforms

The parallel computing architectures used in experiments are the Thunderhead Beowulf cluster at NASA's Goddard Space Flight Center (NASA/GSFC) and a NVidia^{TM} GeForce GTX 275 GPU:

- The Thunderhead Beowulf cluster is composed of 2.4 GHz Intel Xeon nodes, each with 1 GB of memory and a scratch area of 80 GB of memory shared among the different processors[4]. The total peak performance of the system is 2457.6 Gflops. Along with the 256-processor computer core (out of which only 32 were available to us at the time of experiments), Thunderhead has several

[4]http://thunderhead.gsfc.nasa.gov

nodes attached to the core with 2 GHz optical fibre Myrinet [16]. The parallel algorithms tested in this work were run from one of such nodes, called `thunder1` (used as the master processor in our tests). The operating system used at the time of experiments was Linux RedHat 8.0, and `MPICH` was the message-passing library used[5].

- The NVidia™ GeForce GTX 275 GPU features 240 processor cores operating at 1550 MHz, 80 texture processing units, a 448-bit memory interface, and a 1792 MB GDDR3 framebuffer at a 2520 MHz. It is based on the GT200 architecture, and is commercially available from April 2009[6]. The GPU is connected to a CPU Intel Q9450 with 4 cores, which uses a motherboard ASUS Striker II NSE (with NVidia™ 790i chipset) and 4 GB of RAM memory at 1333 MHz. Hyperspectral data are moved to and from the host CPU memory by DMA transfers over a PCI Express bus.

C. Analysis of target detection accuracy

It is first important to emphasize that our parallel versions of OSP (implemented both for clusters and GPUs) provide exactly the same results as the serial versions of the same algorithms, implemented using the Intel C/C++ compiler and optimized via compilation flags to exploit data locality and avoid redundant computations. As a result, in order to refer to the target and anomaly detection results provided by the parallel versions, we will refer to both of them as PG-OSP in order to indicate that the same results were achieved by the `MPI`-based and `CUDA`-based implementations for clusters and GPUs, respectively. At the same time, these results were also exactly the same as those achieved by the serial implementation and, hence, the only difference between the considered algorithms (serial and parallel) is the time they need to complete their calculations, which varies depending on the computer architecture in which they are run.

Table II shows the spectral angle distance (SAD) values (in degrees) between the most similar target pixels detected by PG-OSP and the pixel vectors at the known target positions, labeled from 'A' to 'H' in the rightmost part of Fig. 8. The lower the SAD score, the more similar the spectral signatures associated to the targets. In all cases, the number of target pixels to be detected was set to $t = 30$ after calculating the virtual dimensionality (VD) of the data [24]. As shown by Table II, the PG-OSP extracted targets were similar, spectrally, to the known ground-truth targets (this method was able to perfectly detect the targets labeled as 'C' and 'D', and had more difficulties in detecting very small targets).

D. Parallel performance in the Thunderhead cluster

In this subsection we evaluate the parallel performance of P-OSP in a Beowulf cluster. Table III shows the processing times in seconds the multi-processor version of P-OSP using different numbers of processors (CPUs) on the Thunderhead Beowulf cluster at NASA's Goddard Space Flight Center. As shown by Table III, when 32 processors were used the P-OSP was able to finalize in about 50 seconds, thus clearly outperforming the sequential version which takes 4 minutes of computation in one Thunderhead processor.

Table IV reports the speedups (number of times that the parallel version was faster than the sequential one as the number of processors was increased) achieved by multi-processor runs of the P-OSP algorithm. For illustrative purposes, the speedups are also graphically illustrated in Fig. 9, which indicates that P-OSP resulted in speedups close to linear.

Finally, Table V shows the load balancing scores for all considered parallel algorithms. The imbalance is defined as D = Max/Min, where Max and Min are the maxima and minima processor run times, respectively. Therefore, perfect balance is achieved when D = 1. As we can see from Table V, the P-OSP algorithm was able to provide values of D very close to optimal in the considered cluster, indicating that our parallel implementation achieved satisfactory load balance.

[5]http://www.mcs.anl.gov/research/projects/mpi/mpich1

[6]http://www.nvidia.com/object/product_geforce_gtx_275_us.html

Table II
SPECTRAL ANGLE VALUES (IN DEGREES) BETWEEN TARGET PIXELS AND KNOWN GROUND TARGETS FOR PG-OSP.

A	B	C	D	E	F	G	H
9,17°	13,75°	0,00°	0,00°	20,05°	28,07°	21,20°	21,77°

Table III
PROCESSING TIMES IN SECONDS MEASURED FOR P-OSP USING DIFFERENT NUMBERS OF CPUS ON THE THUNDERHEAD BEOWULF CLUSTER.

1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs	32 CPUs
1263,21	879,06	447,47	180,94	97,90	49,54

Table IV
SPEEDUPS FOR THE P-OSP ALGORITHM USING DIFFERENT NUMBERS OF CPUS ON THE THUNDERHEAD BEOWULF CLUSTER.

2 CPUs	4 CPUs	8 CPUs	16 CPUs	32 CPUs
1,437	2,823	6,981	12,902	25,498

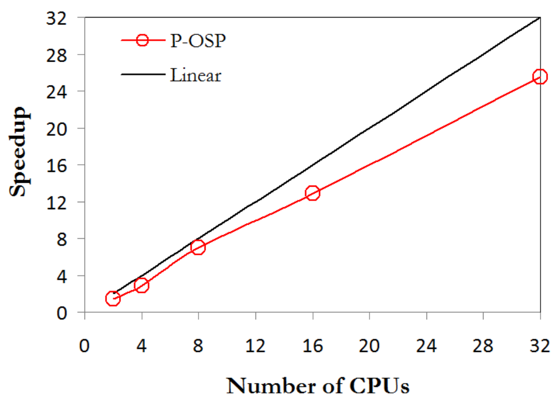

Figure 9. Speedups achieved by P-OSP on the Thunderhead Beowulf cluster at NASA's Goddard Space Flight Center in Maryland.

Table V
LOAD BALANCING RATIOS FOR THE P-OSP ON THE THUNDERHEAD BEOWULF CLUSTER AT NASA'S GODDARD SPACE FLIGHT CENTER IN MARYLAND.

Imbalance	2 CPUs	4 CPUs	8 CPUs	16 CPUs	32 CPUs
Max	879,06	447,47	180,94	97,90	49,54
Min	878,94	447,01	180,23	97,06	48,95
D	1,00013	1,0010	1,00393	1,00865	1,0120

Table VI
PROCESSING TIME (SECONDS) AND SPEEDUPS MEASURED FOR THE CPU AND GPU IMPLEMENTATIONS.

CPU	GPU	Speedup
2486,79	48,17	51,6

E. Parallel performance in the NVidia GPU

In this subsection we evaluate the parallel performance of G-OSP in the NVidia^{TM} GeForce GTX 275 GPU. Table VI shows the execution times measured after processing the full hyperspectral scene (614×512 pixels and 224 spectral bands) on the CPU (the Intel Core 2 Quad Q9450) and on the GPU, along with the speedup achieved in each case. The C function `clock()` was used for timing the CPU implementation, and the `CUDA` timer was used for the GPU implementation. The time measurement was started right after the hyperspectral image file was read to the CPU memory and stopped right after the results of the target/anomaly detection algorithm were obtained and stored in the CPU memory.

It is clear from Table VI that the speedup result obtained when implementing the G-OSP was quite significant. Specifically, we measured a speedup of $51,6$ when comparing the processing time measured in the GPU with the processing time measured in the CPU. It should be noted that the implementations of all the kernels developed were validated using the CUDA occupancy calculator[7] with the goal of maximizing the occupancy of the multiprocessors. This is a helpful tool to predict how efficient an implemented kernel will be on the GPU. For illustrative purposes, Fig. 10 shows an example of the output given by the occupancy calculator for one of our kernels: a plot where the red triangle is the chosen resource usage, and the other points represent the occupancy for a range of block sizes, register counts and shared memory allocation. In our particular case, Fig. 10 reveals that the occupancy of the multiprocessors is 100% for 512 (red triangle) and 256 (middle of the plot) threads per block. These two values represent the number of threads per block used by the kernel during the whole algorithm, depending on the amount of calculations to be done (which increases in each iteration). Hence this kernel achieves maximum occupancy of the multiprocessors, obtaining a very good performance. The green line limits the maximum occupancy that the kernel can achieve using three parameters: the number of threads per block, the number of registers per thread, and the shared memory used per block. Note that a maximum occupancy does not necessarily mean a very high performance. For instance, increasing the occupancy if a kernel is not bandwidth bound will not necessarily increase performance.

[7]http://developer.download.nvidia.com

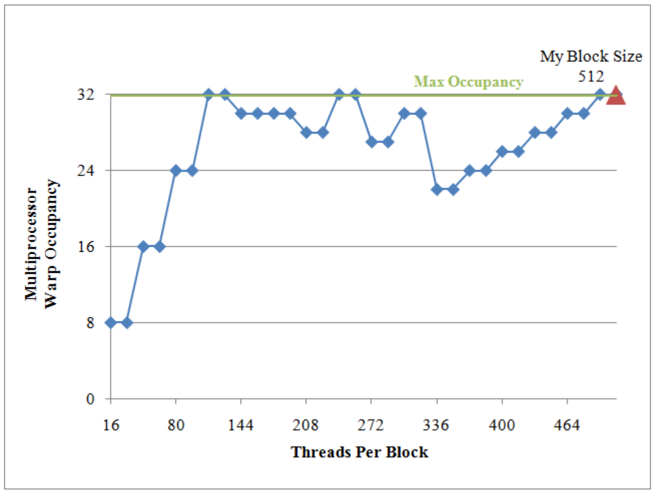

Figure 10. Maximum occupancy for one kernel

Although the speedup and multiprocessor occupancy values achieved by our implementation were significant, the final processing time for the G-OSP is still above 48 seconds after parallelization. This response time is not in real-time since the cross-track line scan time in AVIRIS, a push-broom instrument [1], is quite fast (8.3 msec). This introduces the need to process the full image cube (614 lines, each made up of 512 pixels with 224 spectral bands) in about 5 seconds to achieve fully achieve real-time performance. Although the proposed implementations can still be optimized, Table VI indicates that significant speedups can be obtained in most cases using only one GPU device, with very few on-board restrictions in terms of cost, power consumption and size, which are important when defining mission payload (defined as the maximum load allowed in the airborne or satellite platform that carries the imaging instrument).

VI. CONCLUSIONS AND FUTURE RESEARCH

With the ultimate goal of drawing a comparison of clusters versus GPUs as high performance computing architectures in the context of remote sensing applications, this paper described new parallel implementations of an orthogonal algorithm for target detection in hyperspectral images. Two types of parallel computing platforms: a Beowulf cluster at NASA's Goddard Space Flight Center in Maryland and an NVidia^{TM} GeForce GTX 275 GPU, are investigated and inter-compared. Experimental results have been focused on the analysis of hyperspectral data collected by NASA's AVIRIS instrument over the World Trade Center (WTC) in New York, five days after the terrorist attacks that collapsed the two main towers in the WTC complex. Our experimental assessment of clusters versus GPUs in the context of this particular application indicates that commodity clusters represent a source of computational power that is both accessible and applicable to obtaining results quickly enough and with high reliability in target detection applications in which the data has already been transmitted to Earth. However, GPU hardware devices may offer important advantages in defense and security applications that demand a response in real-time, mainly due to the low weight and compact size of these devices, and to their capacity to provide high performance computing at very low costs.

Although the results reported in this work are encouraging, further experiments should be conducted in order to increase the parallel performance of the proposed parallel algorithms by resolving memory issues in the cluster-based implementations and optimizing the parallel design of the algorithms in the GPU-based implementations. Experiments with additional scenes are also highly desirable. Finally, experiments with radiation-hardened GPU devices will be required in order to evaluate the possibility of adapting the proposed parallel algorithms to hardware devices which have been already certified by international agencies and are mounted on-board satellite platforms for Earth and planetary observation from space.

Acknowledgment

This work has been supported by the European Community's Marie Curie Research Training Networks Programme under reference MRTN-CT-2006-035927 (HYPER-I-NET). Funding from the Spanish Ministry of Science and Innovation (HYPERCOMP/EODIX project, reference AYA2008-05965-C04-02) is gratefully acknowledged.

References

[1] R. O. Green, M. L. Eastwood, C. M. Sarture, T. G. Chrien, M. Aronsson, B. J. Chippendale, J. A. Faust, B. E. Pavri, C. J. Chovit, M. Solis *et al.*, "Imaging spectroscopy and the airborne visible/infrared imaging spectrometer (AVIRIS)," *Remote Sensing of Environment*, vol. 65, no. 3, pp. 227–248, 1998.

[2] C.-I. Chang, *Hyperspectral Imaging: Techniques for Spectral Detection and Classification.* Norwell, MA: Kluwer, 2003.

[3] A. Plaza and C.-I. Chang, *High performance computing in remote sensing*. Boca Raton: CRC Press, 2007.

[4] A. Paz, A. Plaza, and S. Blazquez, "Parallel implementation of target and anomaly detection algorithms for hyperspectral imagery," *Proc. IEEE Geosci. Remote Sens. Symp.*, vol. 2, pp. 589–592, 2008.

[5] Y. Tarabalka, T. V. Haavardsholm, I. Kasen, and T. Skauli, "Real-time anomaly detection in hyperspectral images using multivariate normal mixture models and gpu processing," *Journal of Real-Time Image Processing*, vol. 4, pp. 1–14, 2009.

[6] C.-I. Chang and H. Ren, "An experiment-based quantitative and comparative analysis of hyperspectral target detection and image classification algorithms for hyperspectral imagery," *IEEE Trans. Geosci. Remote Sens.*, vol. 2, p. 1044.

[7] D. Manolakis, D. Marden, and G. A. Shaw, "Hyperspectral image processing for automatic target detection applications," *MIT Lincoln Laboratory Journal*, vol. 14, pp. 79–116, 2003.

[8] J. C. Harsanyi and C.-I. Chang, "Hyperspectral image classification and dimensionality reduction: An orthogonal subspace projection approach," *IEEE Trans. Geosci. Remote Sens.*, vol. 32, pp. 779–785, 1994.

[9] H. Ren and C.-I. Chang, "Automatic spectral target recognition in hyperspectral imagery," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 39, pp. 1232–1249, 2003.

[10] A. Plaza, P. Martinez, J. Plaza, and R. Perez, "Dimensionality reduction and classification of hyperspectral image data using sequences of extended morphological transformations," *IEEE Trans. Geosci. Remote Sens.*, vol. 43, no. 3, pp. 466–479, 2005.

[11] C.-I. Chang and D. Heinz, "Constrained subpixel target detection for remotely sensed imagery," *IEEE Trans. Geosci. Remote Sens.*, vol. 38, pp. 1144–1159, 2000.

[12] N. Acito, M. Diani, and G. Corsini, "Computational load reduction for anomaly detection in hyperspectral images: An experimental comparative analysis," *Proc. IEEE Geosci. Remote Sens. Symp.*, vol. 1, pp. 3206–3209, 2009.

[13] K. Itoh, "Massively parallel fourier-transform spectral imaging and hyperspectral image processing," *Optics and Laser Technology*, vol. 25, p. 202, 1993.

[14] T. El-Ghazawi, S. Kaewpijit, and J. L. Moigne., "Parallel and adaptive reduction of hyperspectral data to intrinsic dimensionality," *Cluster Computing*, vol. 1, pp. 102–110, 2001.

[15] A. Plaza, J. Plaza, and D. Valencia, "Impact of platform heterogeneity on the design of parallel algorithms for morphological processing of high-dimensional image data," *Journal of Supercomputing*, vol. 40, pp. 81–107, 2007.

[16] J. Dorband, J. Palencia, and U. Ranawake, "Commodity computing clusters at goddard space flight center," *J. Space Commun.*, vol. 3, p. 1, 2003.

[17] A. Plaza, D. Valencia, J. Plaza, and P. Martinez, "Commodity cluster-based parallel processing of hyperspectral imagery," *Journal of Parallel and Distributed Computing*, vol. 66, pp. 345–358, 2006.

[18] A. Plaza and C.-I. Chang, "Clusters versus FPGA for parallel processing of hyperspectral imagery," *International Journal of High Performance Computing Applications*, vol. 22, no. 4, pp. 366–385, 2008.

[19] J. Setoain, M. Prieto, C. Tenllado, A. Plaza, and F. Tirado, "Parallel morphological endmember extraction using commodity graphics hardware," *IEEE Geosci. Remote Sens. Lett.*, vol. 43, pp. 441–445, 2007.

[20] A. Paz and A. Plaza, "Gpu implementation of target and anomaly detection algorithms for remotely sensed hyperspectral image analysis," *SPIE Optics and Photonics, Satellite Data Compression, Communication, and Processing Conference*, 2010.

[21] A. P. A. Paz and J. Plaza, "Comparative analysis of different implementations of a parallel algorithm for automatic target detection and classification of hyperspectral images," *Proc. SPIE*, vol. 7455, pp. 1–12, 2009.

[22] A. P. y. J. P. A. Paz, "Parallel implementation of target detection algorithms for hyperspectral imagery," *IEEE International Geoscience and Remote Sensing Symposium*, 2008.

[23] R. Brightwell, L. Fisk, D. Greenberg, T. Hudson, M. Levenhagen, A. Maccabe, and R. Riesen, "Massively parallel computing using commodity components," *Parallel Comput.*, vol. 26, p. 243266, 2000.

[24] C.-I. Chang and Q. Du, "Estimation of number of spectrally distinct signal sources in hyperspectral imagery," *IEEE Trans. Geosci. Remote Sens.*, vol. 42, no. 3, pp. 608–619, 2004.