Journal of
Applied Remote Sensing

Evaluation of the graphics processing unit architecture for the implementation of target detection algorithms for hyperspectral imagery

Blas Trigueros-Espinosa
Miguel Vélez-Reyes
Nayda G. Santiago
Samuel Rosario-Torres

Evaluation of the graphics processing unit architecture for the implementation of target detection algorithms for hyperspectral imagery

Blas Trigueros-Espinosa, Miguel Vélez-Reyes, Nayda G. Santiago, and Samuel Rosario-Torres
University of Puerto Rico Mayaguez Campus, Laboratory for Applied Remote Sensing and Image Processing, Electrical and Computer Engineering Department, P.O. Box 3535, Mayaguez, Puerto Rico 00681-3535
E-mail: m.velez@ieee.org

Abstract. Recent advances in hyperspectral imaging sensors allow the acquisition of images of a scene at hundreds of contiguous narrow spectral bands. Target detection algorithms try to exploit this high-resolution spectral information to detect target materials present in a scene, but this process may be computationally intensive due to the large data volumes generated by the hyperspectral sensors, typically hundreds of megabytes. Previous works have shown that hyperspectral data processing can significantly benefit from the parallel computing resources of graphics processing units (GPUs), due to their highly parallel structure and the high computational capabilities that can be achieved at relative low costs. We studied the parallel implementation of three target detection algorithms (RX algorithm, matched filter, and adaptive matched subspace detector) for hyperspectral images in order to identify the aspects in the structure of these algorithms that can exploit the CUDA™ architecture of NVIDIA® GPUs. A data set was generated using a SOC-700 hyperspectral imager to evaluate the performance and detection accuracy of the parallel implementations on a NVIDIA® Tesla™ C1060 graphics card, achieving real-time performance in the GPU implementations based on global statistics. © *2012 Society of Photo-Optical Instrumentation Engineers (SPIE).* [DOI: 10.1117/1.JRS.6.061506]

Keywords: hyperspectral imaging; target detection; graphics processing units; parallel processing.

Paper 11171SSP received Sep. 9, 2011; revised manuscript received Mar. 19, 2012; accepted for publication Apr. 25, 2012; published online Jun. 21, 2012; corrected Jun. 25, 2012.

1 Introduction

Remote detection and identification of objects or materials have attracted considerable interest over the last few years and have become a desirable ability in many civilian and military applications. The use of hyperspectral imaging (HSI) techniques for remote detection and classification of materials has been widely studied in many areas like defense and homeland security, biomedical imaging, or Earth sciences.[1–4] Hyperspectral imagers can collect tens or hundreds of images of the same scene, taken at different narrow contiguously spaced spectral bands. This high-resolution spectral information can be used to identify materials by their spectral properties but algorithms that exploit HSI data have usually high computational requirements due to the potentially large volume sizes of these images, typically hundreds of megabytes. This can be an important limitation in remote sensing applications that require real-time processing, such as surveillance or explosive detection. Fortunately, many algorithms designed for hyperspectral data processing show an inherent structure that allows parallel implementations. Previous works have shown that HSI data processing can significantly benefit from parallel computing resources of hardware platforms like computer clusters, field-programmable gate arrays (FPGA), or graphics processing units (GPU).[5–9] Specifically, GPUs have proven to be promising candidates as hardware platforms for accelerating hyperspectral processing tasks due to its highly

0091-3286/2012/$25.00 © 2012 SPIE

parallel structure and the high computational capabilities that can be achieved at relative low costs.[10,11] However, since the GPU architecture is optimized for data-parallel processing, (i.e., tasks where the same computation is repeated many times over different data elements), only hyperspectral algorithms that show this data-parallel structure can significantly benefit from GPU-based implementations.

This work focused on studying different state-of-the-art hyperspectral target detection algorithms in order to analyze the aspects in the structure of these algorithms that can take advantage of the parallel computing resources of GPUs based on the NVIDIA® CUDA™[12] architecture. This paper describes the GPU implementation of the RX algorithm, the matched filter (MF), and the adaptive matched subspace detector (AMSD). Section 2 presents a brief description of the target detection algorithms studied in this work. Section 3 presents an introduction to the CUDA parallel architecture and describes the GPU implementation of each detector. Section 4 describes the methodology used and presents the experimental results. Finally, Sec. 5 presents the conclusions of this research and final remarks for future work.

2 Target Detection Algorithms

In this work, we have studied the GPU implementation of three target detection algorithms: the RX anomaly detector, the matched filter, and the adaptive matched subspace detector. The first two algorithms are full-pixel detectors. These detectors assume that the pixels in the image contain information of only one class (target or background), i.e., the pixel does not contain mixed spectra. In contrast, the adaptive matched subspace detector belongs to the family of sub-pixel detectors which assume that the target may occupy only a portion of the pixel area and the remaining part is filled with background (i.e., a mixed pixel).

Plaza et al. proposed two algorithms for target detection in HSI and their corresponding GPU implementations in Ref. 11 The first algorithm, called automatic target detection and classification algorithm (ATDCA), is based on the orthogonal subspace projection approach.[13] The second algorithm is a GPU-based implementation of the RX algorithm. In this implementation, the inverse of the covariance matrix is computed in parallel on the GPU using the Gauss-Jordan elimination method and is globally estimated using the entire image. We proposed an alternative implementation of this algorithm that computes the Cholesky decomposition of the covariance matrix on the CPU and the resulting triangular systems are solved in parallel on the GPU. We also investigated an adaptive GPU implementation of the RX algorithm that uses a moving window to locally estimate the mean and covariance matrix. A GPU-based implementation of the RX algorithm, recently proposed by Winter et al., can also be found in Ref. 14 Another GPU-based implementation of a target detection algorithm for real-time anomaly detection in HSI has recently been proposed by Tarabalka et al.[15] The proposed anomaly detection algorithm is based on a multivariate normal mixture model of the background.

2.1 *RX Algorithm*

The RX algorithm for anomaly detection, developed by Reed and Yu,[16] is given by

$$D_{\mathrm{RX}}(\mathbf{x}) = (\mathbf{x}-\boldsymbol{\mu}_0)^T\boldsymbol{\Gamma}_0^{-1}(\mathbf{x}-\boldsymbol{\mu}_0). \tag{1}$$

In this detector, the variability of the background is modeled using a multivariate normal distribution, where $\mathbf{x}$ is the pixel spectrum, $\boldsymbol{\mu}_0$ is the mean of the background distribution, and $\boldsymbol{\Gamma}_0$ is the covariance matrix. The output of the RX algorithm corresponds to the *Mahalanobis* distance[17] from the test pixel to the center of the background distribution. The test pixel is considered an anomaly if the resulting distance [Eq. (1)] is greater than a given threshold (the pixel spectrum deviates too much from the background distribution).

The parameters of the background distribution, $\boldsymbol{\mu}_0$ and $\boldsymbol{\Gamma}_0$, can be globally estimated using training samples from the data. Other approach commonly used for the RX algorithm is to locally estimate these parameters by using a 2D spatially moving window centered at the test pixel in combination with a guard window as shown in Fig. 1.[18] The mean and covariance matrix can be estimated using the samples from the region between the two windows (the pixels

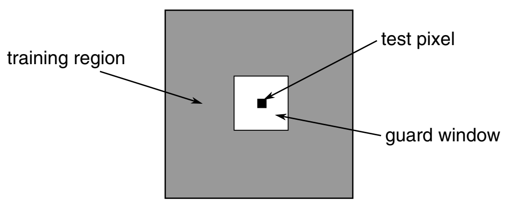

Fig. 1 Structure of the 2D spatially moving window for background parameter estimation in RX algorithm.

contained in the guard window are excluded to avoid bias in the estimates). This is the technique used in this work but other approaches use different regions for estimating the mean and covariance matrix by defining an additional window.[19]

2.2 *Matched Filter Detector*

In the matched filter[20] (MF) detector, both the background and target spectral variability are modeled using a multivariate normal distribution with different means but the same covariance matrix $\mathbf{\Gamma}$. This detector is given by

$$D_{\mathrm{MF}}(\mathbf{x}) = \mathbf{c}_{\mathrm{MF}}^T(\mathbf{x} - \boldsymbol{\mu}_0) = \kappa(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^T\mathbf{\Gamma}^{-1}(\mathbf{x} - \boldsymbol{\mu}_0), \tag{2}$$

where $\boldsymbol{\mu}_0$ and $\boldsymbol{\mu}_1$ are the mean of the background and target classes, respectively, and κ is a normalization constant. The idea of the matched filter is to project the pixel vector onto the direction $\mathbf{c}_{\mathrm{MF}}$ that provides the best separability between the background and target classes. In the implementation described in this work, the normalization constant was selected to produce an output $D_{\mathrm{MF}}(\mathbf{x}) = 1$ when $\mathbf{x} = \boldsymbol{\mu}_1$, as proposed by Manolakis et al.[20] With this selection, the MF detector takes this final form:

$$D_{\mathrm{MF}}(\mathbf{x}) = \frac{(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^T\mathbf{\Gamma}^{-1}(\mathbf{x} - \boldsymbol{\mu}_0)}{(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^T\mathbf{\Gamma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)}. \tag{3}$$

The output of the detector is finally compared to a preselected threshold to decide if the target is present or not. The statistical parameters $\boldsymbol{\mu}_0$, $\boldsymbol{\mu}_1$, and $\mathbf{\Gamma}$ are estimated using training samples taken from the original image.

2.3 *Adaptive Matched Subspace Detector*

The adaptive matched subspace detector[21] (AMSD) is a sub-pixel target detector based on a structured background model, i.e., the spectral variability is described using a linear subspace instead of a statistical distribution. The two competing hypotheses generated to decide if the target is present or not are

$$\begin{array}{lll} H_0: & \mathbf{x} = Ba_{b,0} + \mathbf{w} & \text{(target absent)} \\ H_1: & \mathbf{x} = Sa_s + Ba_{b,1} + \mathbf{w} = Ea + \mathbf{w} & \text{(target present)} \end{array}.$$

$\mathbf{B}$ is a $L \times M$ matrix whose columns span the background subspace and is estimated from the data. $\mathbf{a}_{b,0}$ is a $M \times 1$ vector representing the position of the pixel within the background subspace under the hypothesis of target absent. $\mathbf{S}$ is a $L \times P$ matrix whose columns span the target subspace. The dimensionality P of the target subspace represents the available a priori variability information about the target spectrum. $\mathbf{a} = [\mathbf{a}_s\mathbf{a}_{b,0}]^T$ is a $(M + P) \times 1$ vector representing the position of the pixel within the union of the background and target subspaces, spanned by $\mathbf{E} = [\mathbf{S} \quad \mathbf{B}]$. Finally, $\mathbf{w}$ is an additive Gaussian noise with zero mean and covariance $\mathbf{\Gamma}_w = \sigma_w^2\mathbf{I}$, representing both modeling and measurement errors.

Based on the previous model, the AMSD is given by[21,22]

$$D_{\text{AMSD}}(\mathbf{x}) = \frac{x^T(P_B^{\perp} - P_E^{\perp})x}{x^T P_E^{\perp} x}, \tag{4}$$

where $P_B^{\perp}$ and $P_E^{\perp}$ are the orthogonal projection matrices to the subspaces spanned by $\mathbf{B}$ and $\mathbf{E}$, respectively, and $\mathbf{P}_{\mathbf{A}}^{\perp}$ is defined as $\mathbf{P}_{\mathbf{A}}^{\perp} = \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$.

In the implementation of this detector, two methods for estimating the matrix $\mathbf{B}$ of background basis vectors were evaluated: singular value decomposition (SVD)[23] and *Maximum Distance* (MaxD).[24] In the SVD approach, the basis vectors are selected as first M left singular vectors of the matrix $\mathbf{X}$ representing the image in bands × pixels format. In this case, the number of basis vectors is selected to enclose a given percentage of variability.[23] MaxD is an endmember selection method proposed by Schott et al.[24] The basis vectors selected by this method, which are pixels from the original image, are the set of vectors that best approximate a simplex defining the background subspace. The steps involved in the MaxD method can be summarized as follows:

1. The largest magnitude pixel vector ($\mathbf{v}_1$) and the smallest magnitude pixel vector ($\mathbf{v}_2$) from the image are selected as the first two endmembers.
2. All pixel vectors are projected onto the subspace orthogonal to $\mathbf{v}_1 - \mathbf{v}_2$. Thus, both $\mathbf{v}_1$ and $\mathbf{v}_2$ project to the same point $\mathbf{v}_{12}$.
3. The projected pixel with maximum distance to $\mathbf{v}_{12}$ is selected as the third endmember $\mathbf{v}_3$.
4. All projected points are again projected onto the subspace orthogonal to $\mathbf{v}_{12} - \mathbf{v}_3$.
5. The process is repeated until the desired number of endmembers is selected.

The performance of the SVD and MaxD methods as basis-vector selection techniques for target detection was previously studied by Bajorski et al.[23] Other works comparing different techniques for background subspace generation, including SVD and MaxD, were recently presented by Peña-Ortega and Velez-Reyes.[25,26]

3 GPU-Based Parallel Implementation

All three target detection algorithms have a general structure that shows an inherent parallelism. The detection statistic is calculated independently for every pixel of the image. Therefore, if there are N pixels, the calculation of the detection output for the entire image can be decomposed into N parallel tasks without communication between each other. This algorithm structure, which is known as an embarrassingly parallel problem,[27] can be exploited by data-parallel architectures like the CUDA™ architecture of NVIDIA® GPUs.[12] CUDA, which stands for Compute Unified Device Architecture, is a parallel computing architecture developed by NVIDIA®. CUDA adds a set of extensions to the C programming language that allows the programmer to define portions of the original code to be run in parallel on the GPU. These portions of the code are enclosed in a special type of functions called kernels, which are executed in parallel by many GPU threads. This allows the programmer to offload parallel and compute-intensive sections by moving these computations to the GPU while still making use of the CPU when necessary. Figure 2 shows a typical CUDA program flow. First, the CUDA application allocates the necessary GPU memory and copies the data to be processed from the system memory to the GPU memory. Second, the data is processed in parallel on the GPU by calling one or more kernels functions. Finally, the results are copied back from the GPU memory to the system memory.

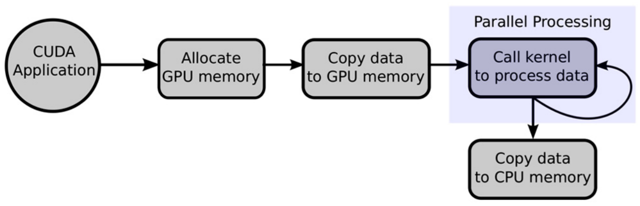

Fig. 2 CUDA program flow for data-parallel processing.

For each detection algorithm, a CUDA kernel function was defined. The kernel functions are configured to be run in parallel by as many GPU threads as pixels in the image, so each thread is responsible for computing the detection output for a different pixel. The image data was transferred to the GPU memory in band sequential format (contiguous words in memory correspond to contiguous pixels in the image for the same band). This storage scheme improves the memory bandwidth when accessing global memory by allowing coalesced memory transaction.[12] The band interleaved by line format may also lead to coalesced memory transaction but the use of the band sequential scheme reduces the pointer arithmetic for indexing data elements when reading the image.

Figure 3 shows the proposed GPU implementation for the RX algorithm and MF detector. First, the statistical parameters of the background and target distributions are estimated from training data samples. These are preprocessing steps, performed on the CPU, which produce the input parameters needed for the RX and MF detectors: $\mathbf{\Gamma}_0$, $\boldsymbol{\mu}_0$, and $\boldsymbol{\mu}_1$. The two algorithms have in common the computation of the inverse of the covariance matrix $\mathbf{\Gamma}_0^{-1}$. In the implementation proposed by Plaza et al.,[11] the inverse of the covariance matrix is computed in parallel on the GPU using the Gauss-Jordan elimination method. This method is parallelized by applying the pivoting operation at the same time to many rows and columns. In our proposed implementation, instead of computing the inverse directly, the covariance matrix is factorized using the Cholesky decomposition $\mathbf{\Gamma}_0 = \mathbf{L}\mathbf{L}^T$, taking advantage that this matrix is symmetric and positive definite. Expressing the output of the RX detector for the pixel i in terms of the lower triangular matrix $\mathbf{L}$, we get

$$\begin{array}{l} y_i^{\mathrm{RX}} = (\mathbf{x}_i - \boldsymbol{\mu}_0)^T \mathbf{\Gamma}_0^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_0) = (\mathbf{x}_i - \boldsymbol{\mu}_0)^T (\mathbf{L}\mathbf{L}^T)^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_0) = \\ (\mathbf{L}^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_0))^T (\mathbf{L}^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_0)) = \mathbf{b}_i^T \mathbf{b}_i \end{array}, \tag{5}$$

where $\mathbf{b}_i = \mathbf{L}^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_0)$ is the solution of the triangular system $\mathbf{L}\mathbf{b}_i = \mathbf{x}_i - \boldsymbol{\mu}_0$. In our proposed implementation, the computation of the Cholesky decomposition is performed on the CPU using the function *SPOTRF* from Intel® MKL library. The main reason is that the dimension (bands × bands) of the covariance matrix does not allow enough amount of parallelism to take advantage of the CUDA architecture. The resulting upper triangular matrix $\mathbf{L}$ is transferred to the GPU global memory to be shared by all GPU threads. Then, each thread computes the value $\mathbf{b}_i = \mathbf{L}^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_0)$ by solving a triangular system through forward substitutions. The background mean $\boldsymbol{\mu}_0$ is stored in the GPU constant memory space. Since this vector does not change its values throughout the computation, storing it in the GPU constant memory improves the memory bandwidth by using the constant memory cache. The matrix $\mathbf{L}$ cannot be stored in the constant memory because this memory space is limited to 64 KB. In order to reduce the latency when reading the values of $\mathbf{L}$ from the GPU global memory in the forward substitutions, these values are temporarily stored in the shared memory space. Since the entire matrix $\mathbf{L}$ does not fit into the GPU shared memory space (it is limited to 16 KB), only one row of the matrix is

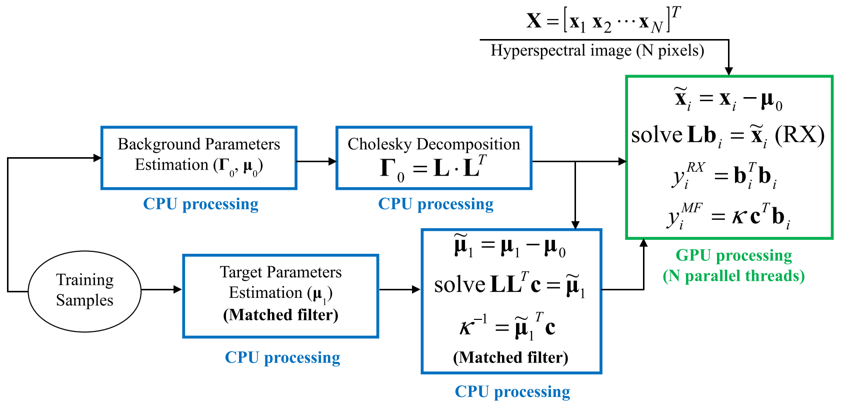
$\mathbf{X} = [\mathbf{x}_1\, \mathbf{x}_2 \cdots \mathbf{x}_N]^T$
Hyperspectral image (N pixels)
Background Parameters Estimation ($\mathbf{\Gamma}_0$, $\boldsymbol{\mu}_0$)
CPU processing
Cholesky Decomposition $\mathbf{\Gamma}_0 = \mathbf{L}\cdot\mathbf{L}^T$
CPU processing
$\tilde{\mathbf{x}}_i = \mathbf{x}_i - \boldsymbol{\mu}_0$
solve $\mathbf{L}\mathbf{b}_i = \tilde{\mathbf{x}}_i$ (RX)
$y_i^{RX} = \mathbf{b}_i^T \mathbf{b}_i$
$y_i^{MF} = \kappa\, \mathbf{c}^T \mathbf{b}_i$
GPU processing (N parallel threads)
Training Samples
Target Parameters Estimation ($\boldsymbol{\mu}_1$) (Matched filter)
CPU processing
$\tilde{\boldsymbol{\mu}}_1 = \boldsymbol{\mu}_1 - \boldsymbol{\mu}_0$
solve $\mathbf{L}\mathbf{L}^T\mathbf{c} = \tilde{\boldsymbol{\mu}}_1$
$\kappa^{-1} = \tilde{\boldsymbol{\mu}}_1^T \mathbf{c}$
(Matched filter)
CPU processing

Fig. 3 Implementation of RX and MF detectors.

stored in the shared memory at every iteration of the forward substitution loop. Following a similar procedure for the MF detector, we get

$$\begin{aligned} y_i^{\mathrm{MF}} &= \kappa(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^T \boldsymbol{\Gamma}_0^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_0) = \kappa(\boldsymbol{\Gamma}_0^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0))^T(\mathbf{x}_i - \boldsymbol{\mu}_0) = \\ \kappa\mathbf{c}^T(\mathbf{x}_i - \boldsymbol{\mu}_0) &= \kappa\mathbf{c}^T\tilde{\mathbf{x}}_i \end{aligned}, \tag{6}$$

where $\mathbf{c} = \boldsymbol{\Gamma}_0^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)$ is the solution of the linear system $\boldsymbol{\Gamma}_0\mathbf{c} = \boldsymbol{\mu}_1 - \boldsymbol{\mu}_0$. By performing the Cholesky decomposition of $\boldsymbol{\Gamma}_0$, as in the RX implementation, the linear system can be solved through forward and back substitutions. Since the vector $\mathbf{c} = \boldsymbol{\Gamma}_0^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)$ does not depend on the pixel value $\mathbf{x}_i$, it can be computed on the CPU and transferred to the constant GPU memory to be shared by all threads. The MF detector also needs the computation of the normalization constant κ, which can be computed as

$$\kappa^{-1} = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^T \boldsymbol{\Gamma}_0^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0) = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^T\mathbf{c}. \tag{7}$$

This step is also computed on the CPU since it has to be performed only once and its computation is relatively fast.

Figure 4 shows a GPU implementation of the adaptive RX algorithm. In this approach, the statistical parameters of the background distribution are locally estimated using the double sliding window technique illustrated in Fig. 1. In this implementation, the parameter estimation, the Cholesky decomposition and the detection output computation steps are all performed within the CUDA kernel function. Each GPU thread will compute the mean $\boldsymbol{\mu}_0^i$ and covariance matrix $\boldsymbol{\Gamma}_0^i$ by using training samples from the region between the two windows centered at the working pixel $\mathbf{x}_i$. Then, each thread computes the Cholesky decomposition $\boldsymbol{\Gamma}_0^i = \mathbf{L}^i\mathbf{L}^{iT}$, solves the triangular system $\mathbf{L}^i\mathbf{L}^{iT}\mathbf{p}_i = \tilde{\mathbf{x}}_i$, and computes the detection output y_i^{RX}. The Cholesky decomposition is performed in a kernel function using a CUDA-adapted version of the C algorithm proposed by Teukolsky et al.[28] Since each thread has its own copy of the mean $\boldsymbol{\mu}_{0i}$ and covariance $\boldsymbol{\Gamma}_{0i}$, these parameters are stored in the thread local memory space. The amount of local memory per thread is limited to 16 KB in devices of compute capability 1.x and 512 KB in devices of compute capability 2.x (Fermi). This imposes a limitation in the number of spectral bands in the original hyperspectral image, which determines the size of the covariance matrix. This matrix can be stored on the local memory if the number of bands is less than 64 for devices 1.x and less than 362 for devices 2.x. The limitation in devices 1.x forces the use of a band reduction step on the input data before RX processing.

Figure 5 shows the proposed GPU implementation of the AMSD algorithm. The structure of the background subspace (dimensionality and basis vectors) is estimated from the data using two different approaches: singular value decomposition (SVD) and MaxD. In the SVD implementation, the left singular vectors are computed as the eigenvectors of the image correlation matrix $\mathbf{R} = \mathbf{X}\mathbf{X}^T$. The matrix $\mathbf{R}$ and the corresponding eigenvectors are computed on the GPU using the function *SGEMM* for matrix-matrix multiplication and the function *SSYEV* for eigenvalues/eigenvectors computation of a real symmetric matrix, both from the CULA™ library, respectively. In the MaxD method, the process of selecting the largest and the smallest magnitude pixel vector from the image is performed on the GPU through a kernel that computes in parallel the magnitude of each pixel. The projection steps are performed on the GPU using the function *SGEMM* from CULA™ and the update of the projection matrix at each iteration is performed through the function *SGER* from CUBLAS™, which performs the operation $\mathbf{A} = \alpha\mathbf{x}\mathbf{y}^T + \mathbf{A}$. Once the matrix of basis vectors $\mathbf{B}$ has been computed, the rest of the computations are based on an implementation approach of this detector proposed by Manolakis et al.[21] that uses the identities $P_E^\perp = P_B^\perp P_Z^\perp P_B^\perp$ and $P_B^\perp - P_E^\perp = P_B^\perp P_Z P_B^\perp$, where $Z = P_B^\perp S$, i.e., the part of the target subspace orthogonal to the background subspace. In the kernel that computes the output of the AMSD, each GPU thread is responsible for computing the numerator $\|P_Z P_B^\perp \mathbf{x}_i\|^2$, denominator $\|P_Z^\perp P_B^\perp \mathbf{x}_i\|^2$, and the detection output y_i^{AMSD} of the AMSD for a given pixel $\mathbf{x}_i$. Since the projection matrices are shared by all threads, we can take advantage of the shared memory space to perform the matrix products using a similar approach as in the global full-pixel detectors.

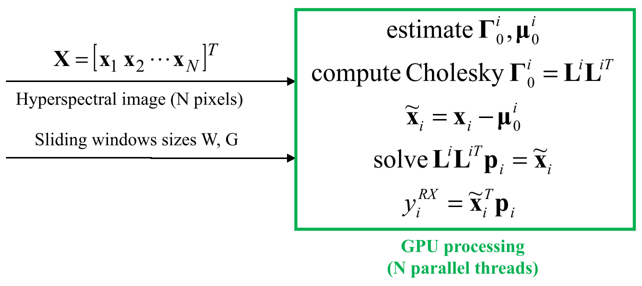
$\mathbf{X}=[\mathbf{x}_1\,\mathbf{x}_2\cdots\mathbf{x}_N]^T$
Hyperspectral image (N pixels)
Sliding windows sizes W, G
estimate $\mathbf{\Gamma}_0^i, \mathbf{\mu}_0^i$
compute Cholesky $\mathbf{\Gamma}_0^i = \mathbf{L}^i\mathbf{L}^{iT}$
$\tilde{\mathbf{x}}_i = \mathbf{x}_i - \mathbf{\mu}_0^i$
solve $\mathbf{L}^i\mathbf{L}^{iT}\mathbf{p}_i = \tilde{\mathbf{x}}_i$
$y_i^{RX} = \tilde{\mathbf{x}}_i^T\mathbf{p}_i$
GPU processing
(N parallel threads)

Fig. 4 Implementation of the adaptive RX algorithm.

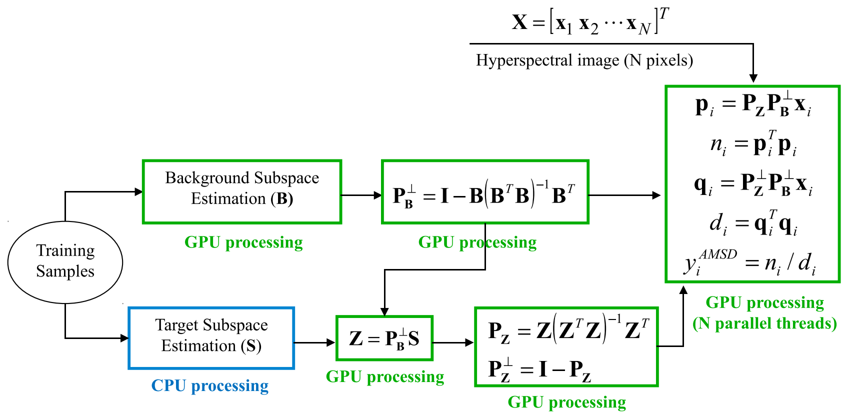
$\mathbf{X}=[\mathbf{x}_1\,\mathbf{x}_2\cdots\mathbf{x}_N]^T$
Hyperspectral image (N pixels)
Training Samples
Background Subspace Estimation (B)
GPU processing
$\mathbf{P}_\mathbf{B}^\perp = \mathbf{I}-\mathbf{B}(\mathbf{B}^T\mathbf{B})^{-1}\mathbf{B}^T$
GPU processing
Target Subspace Estimation (S)
CPU processing
$\mathbf{Z}=\mathbf{P}_\mathbf{B}^\perp\mathbf{S}$
GPU processing
$\mathbf{P}_\mathbf{Z}=\mathbf{Z}(\mathbf{Z}^T\mathbf{Z})^{-1}\mathbf{Z}^T$
$\mathbf{P}_\mathbf{Z}^\perp=\mathbf{I}-\mathbf{P}_\mathbf{Z}$
GPU processing
$\mathbf{p}_i=\mathbf{P}_\mathbf{Z}\mathbf{P}_\mathbf{B}^\perp\mathbf{x}_i$
$n_i=\mathbf{p}_i^T\mathbf{p}_i$
$\mathbf{q}_i=\mathbf{P}_\mathbf{Z}^\perp\mathbf{P}_\mathbf{B}^\perp\mathbf{x}_i$
$d_i=\mathbf{q}_i^T\mathbf{q}_i$
$y_i^{AMSD}=n_i/d_i$
GPU processing
(N parallel threads)

Fig. 5 Parallel implementation of AMSD algorithm.

4 Experimental Results

The GPU-based implementations of the algorithms were developed using CUDA 3.2 and tested on a NVIDIA® Tesla™ C1060 graphics card. The Tesla® C1060 card contains 240 processor cores and 4 GB of DDR3 memory. The theoretical single-precision peak performance and memory bandwidth for this GPU are 933 Gflops and 102 GB/ sec, respectively. The Tesla™ C1060 is installed on a workstation equipped with an Intel® Xeon® E5520 2.27 GHz CPU, 12 GB of RAM memory and running Ubuntu™ 10.10 64 bits as operating system.

For each detection algorithm, a CPU-based implementation was developed to use as baseline to estimate the speedups of the GPU-based implementations. The CPU implementation was built with GCC 4.4.5 compiler using C++. In the GPU implementation, the CUBLAS™[29] and CULA™[30] R10 libraries were used for linear algebra computations (matrix multiplications, Cholesky decomposition, and eigenvectors computation). In the CPU-based implementations, these computations are performed using the Intel®MKL 10.3 library (http://software.intel.com/en-us/articles/intel-mkl/) in combination with the OpenMP™[31] interface to exploit CPU parallelism.

In order to evaluate the running times and detection accuracy of the implemented algorithms, a phantom image simulating traces of different materials on clothing was generated (Fig. 6). The image was collected using a SOC-700 visible hyperspectral imager from Surface Optics Corporation® (http://www.surfaceoptics.com). The SOC-700 imager acquires a 640 by 640 pixel image, 120 bands deep, in the visible-near infrared region (0.43 to 0.9 μm). This instrument takes 1 s to scan 100 lines, thus, the total time needed to complete an image cube is 6.4 s.

For the experiments, a 360×360 pixels spatial subset of the original data cube was selected. The scene consists of a T-shirt surface containing traces of vegetable oil and ketchup. The ketchup was considered as the target material in the algorithms and the remaining pixels, representing the T-shirt surface and the oil traces, were considered as the background clutter. For the evaluation of the running time and speedup of the implemented algorithms, the image subset was duplicated in a tiled fashion in order to generated six different image sizes: 59.3, 118.6, 237.2,

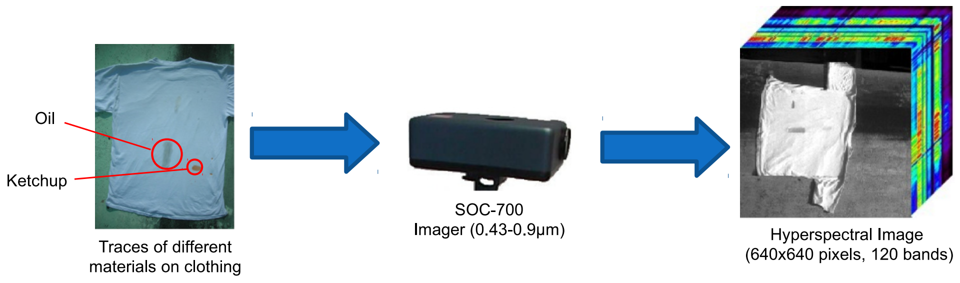
Oil
Ketchup
Traces of different materials on clothing
SOC-700 Imager (0.43-0.9µm)
Hyperspectral Image (640x640 pixels, 120 bands)

Fig. 6 Phantom image generation for the experiments: a hyperspectral image of a scene simulating traces of different materials on clothing was collected using a SOC-700 imager.

474.4, 948.8, and 1897.6 MB. In the adaptive RX implementation, the bands of the input image were downsampled to reduce the number from 120 to 60 due to the local memory limitations of the C1060 graphics card, as mentioned in Sec. 3.

Figure 7 shows the speedup of the GPU implementations over the corresponding CPU implementations for the different data sizes and Table 1 shows the resulting running times for the largest data size (1897.6 MB). The running times were measured using the function *gettimeofday* from the GNU C header file "sys/time.h" and averaged over 10 benchmark executions. The speedups were estimated as the ratio between the averaged running time of the CPU-based and the GPU-based implementations. In the RX implementation, the speedups achieved vary from 11.25 to 24.76. In the MF implementation, the CPU implementation is faster (37 ms) than the GPU-based (79 ms) for the first image size (59.3 MB). For the rest of the input sizes, the GPU implementation runs faster than the CPU-based but the differences in the running times are not significant, reaching a maximum speedup of 3.9 for the largest input size. This

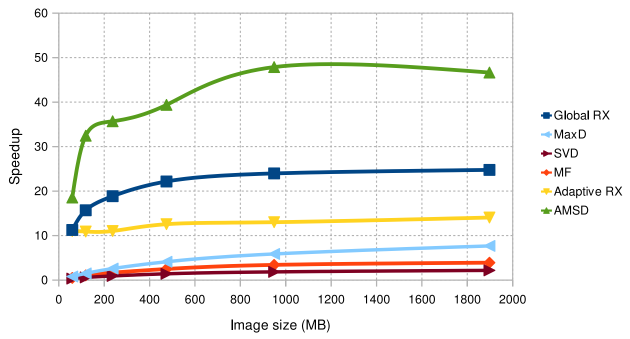

Fig. 7 Speedup of the GPU implementations over the CPU implementations for different image sizes.

Table 1 Processing times (ms) and speedups of the GPU implementations over the corresponding CPU implementations for an image size of 1897.6 MB. The table also shows the processing times and speedups for the two methods for basis selection (SVD and MaxD).

Algorithms	CPU processing time (s)	GPU processing time (s)	Speedup
RX	49.46	1.98	24.76
MF	1.74	0.45	3.90
Adaptive RX	8029.90	571.65	14.05
AMSD	322.62	6.92	46.64
SVD	7.71	3.56	2.17
MaxD	16.27	2.12	7.67

limitation in the speedup of the MF implementation is due to reduced number of arithmetic operations performed in the kernel. Therefore, most of the running time of this GPU implementation is spent in the memory transfers between the CPU and GPU. In the adaptive RX implementation, the speedups vary from 10.99 for the smallest input size to 14.05 for the largest input size. For the largest input size, the implementation on the CPU takes 8029.90 s (2.23 h) to complete the execution, whereas the implementation on the GPU takes 571.65 s (9.52 min). In this implementation, the speedup does not increase considerably with the input size, which may be due to the high local memory dependency of this algorithm resulting in a poor memory throughput. In the AMSD implementation, the speedups achieved vary from 18.54 to 47.87. This was the kernel that achieved the best speedup improvement. The GPU implementation of the two methods evaluated for estimating the background subspace in the AMSD, SVD and MaxD, only outperforms the corresponding CPU implementations for large data sizes. The GPU implementation of SVD is faster than the CPU implementation for image sizes larger than 474.4 MB, reaching a maximum speedup of 2.17, as shown in Table 1. This shows that the computation of the autocorrelation matrix and the corresponding eigenvectors do not take advantage of the GPU parallel architecture. The GPU implementation of the MaxD algorithm is faster than the CPU implementation for image sizes larger than 118.6 MB, reaching a maximum speedup of 7.67 for the largest image size. The MaxD algorithm achieves better performance on the GPU than the SVD approach, although the speedup is still limited specially for image sizes below 237.2 MB.

Since the scanning rate of the SOC-700 imager is 15 MB/ sec, we can analyze the real-time performance of the GPU-based implementations by comparing their processing rates to this value. The processing rate of the implementations was estimated as the ratio of the input image size in MB to the total execution time in seconds needed to process the data. All the implementations exceed the processing rate of 15 MB/ sec except for the adaptive RX algorithm, which achieves a processing rate of around 3.3 MB/ sec. Therefore, the GPU-based implementation of the adaptive RX algorithm was the only implementation that does not achieve a real-time processing rate.

The ground truth used for estimating the detection statistics is shown in Fig. 8, where the full-pixels are represented in red, the sub-pixels in yellow, and the guard-pixels in green. Figures 9 and 10 show the resulting detection maps for the RX, the MF, and the AMSD algorithms. Table 2 shows the detection statistics (detection accuracy and percentage of false alarms). The target contains 568 full-pixels, 197 sub-pixels, and 255 guard-pixels. The best detection accuracy

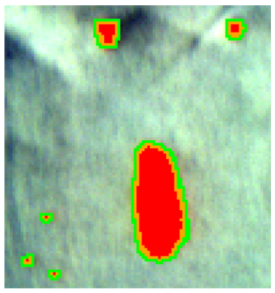

Fig. 8 Ground truth for the target traces showing the full-pixels (red), sub-pixels (yellow), and guard-pixels (green).

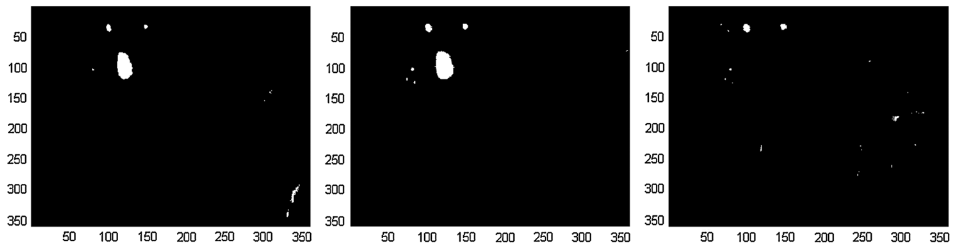

Fig. 9 Detection results: RX with global statistics (left), MF detector (center), adaptive RX (right).

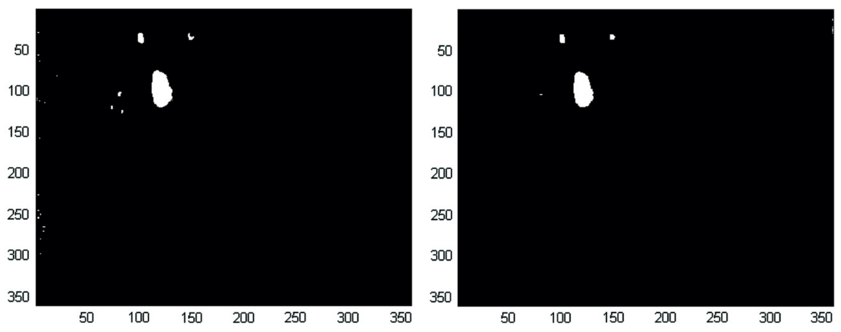

Fig. 10 Detection results: AMSD with SVD as background subspace estimation method (left), AMSD with MaxD as background subspace estimation method (right).

Table 2 Detection accuracy of different target detection algorithms.

Target detectors	Detection accuracy (%)	False alarms (%)
RX	93.3	0.09
MF	98.4	0.07
Adaptive RX	8.4	1.1
AMSD (SVD)	93.3	0.03
AMSD (MaxD)	90.2	0.01

was achieved by the matched filter (98.4% of targets detected for a false alarm rate of 0.07%). The detection accuracy of the adaptive RX algorithm is very limited by the size of the 2D moving window. For a window size of 51×51, only five small targets were detected (8% detection accuracy). The adaptive RX assumes small targets, hence the reason for this poor performance. The percentage of detected targets for the AMSD algorithm, using SVD as background subspace estimation method, and the RX algorithm, using global background statistics, were both 93.3%, but the percentage of false alarms in the RX algorithm was slightly higher. In addition, the detection accuracy of the AMSD algorithm was reduced when using MaxD as background subspace estimation method.

5 Conclusions

In this work, the GPU implementation of three target detection algorithms for hyperspectral images was studied. The first two algorithms were detectors for full-pixel targets: the RX algorithm and the MF detector. Two different implementations were studied for the RX detector. In the global implementation, the mean and covariance matrix were globally estimated from training samples as a preprocessing step on the CPU. In the adaptive implementation, these parameters were locally estimated using a moving window centered at the test pixel. The third algorithm studied was the AMSD, a detector for sub-pixel targets based on structured modeling of the background. The detection algorithms were implemented on a NVIDIA® Tesla™ C1060 graphics card. In addition, a CPU implementation of each target detector was developed to be used as a baseline to estimate the speedups of the GPU implementations. The computational performance of the implementations and the detection accuracy of the algorithms were evaluated using a set of phantom images of a scene simulating traces of different materials on clothing and collected using a SOC-700 hyperspectral imager. In the design of the GPU implementations, we have analyzed three important aspects: computation decomposition, data layout, and the computation mapping to the GPU memory hierarchy. The computation is decomposed in a one-thread-per-output basis. Since the output value of the detectors is computed independently for each pixel of the image, the task of computing the output value for a single pixel can be assigned to a single processing unit, i.e., a GPU thread in the CUDA architecture. The data layout is related to how input image is stored in the GPU memory. In the GPU-based implementations described in this document, the band sequential storage scheme was used since it leads to coalesced memory transactions since threads with consecutive ID numbers will access contiguous memory positions when reading or writing a single band. Finally, different memory spaces were used for storing the data elements of the algorithms in order to exploit the GPU architecture. Parameters that do not change their values throughout the computation, like the background mean $\boldsymbol{\mu}_0$, are stored in the GPU constant memory space to improve the memory throughput. Other parameters, like the covariance matrix of the full-pixels detectors and the projection matrices of the AMSD algorithm, although they remain constant, cannot be stored in the constant memory space due to their size. In this case, the rows of the matrices are temporarily stored in the shared memory space in order to increase the memory throughput. The GPU implementations of the global RX and AMSD algorithms showed best performance improvement achieving maximum speedups of 24.76 and 46.64 respectively. The performance of the MF algorithm was limited by the low number of arithmetic operations performed by this detector in the kernel, achieving speedups below five. The parallel portion of this algorithm only consists of a dot product, which is relatively fast. Therefore, most of the total running time is spent in transferring data from the CPU to the GPU and vice versa. The performance of the adaptive RX algorithm was also limited, but in this case, due to high dependency on local data which limits the memory throughput. In addition, in the adaptive RX implementation the number of bands had to be reduced to 60 since the local memory space per thread is limited to 16 KB in the C1060 card. Experimental results also showed that the method evaluated for estimating the background subspace, SVD and MaxD, are only accelerated on the GPU for large data sizes. In terms of detection accuracy, the MF showed the best detection results for the data set evaluated.

Future research plans would be the analysis of further optimizations in order to take advantage of the new Fermi architecture of NVIDIA® GPUs. Fermi provides new features like configurable memory caches, more amount of local memory per thread, more amount of shared memory, concurrent kernel executions, etc. These new features open new possibilities in the optimization of the algorithms, specially, in the adaptive RX algorithm which is limited by the local memory throughput. Since the local memory per thread in Fermi is 512 KB, this will allow the use of a number of bands up to 362. In addition, other GPU libraries, like MAGMA (http://icl.cs.utk.edu/magma/) or LibJacket (http://www.accelereyes.com/products/libjacket/), could be evaluated as alternatives to the CULA library for linear algebra computations.

Acknowledgments

This material is based upon work supported by the U.S. Department of Homeland Security under Award Number 2008-ST-061-ED0001 and used facilities of the Bernard M. Gordon Center for Subsurface Sensing and Imaging Systems sponsored by the Engineering Research Centers Program of the National Science Foundation under Award EEC-9986821. Partial support also received from NSF under grant EEC-0946463. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Department of Homeland Security or the National Science Foundation.

References

1. H. C. Schau and B. D. Jennette, "Hyperspectral requirements for detection of trace explosives agents," *Proc. SPIE*, **6233**, 62331Y (2006).http://dx.doi.org/10.1117/12.665139
2. C. M. Bachmann et al., "Coastal characterization from hyperspectral imagery: an intercomparison of retrieval properties from three coast types," in *Proc. 2010 IEEE Int. Geosci. Remote Sens. Symp. (IGARSS)*, 25–30 July 2010, Honolulu, HI, pp. 138–141 (2010), http://dx.doi.org/10.1109/IGARSS.2010.5650660.
3. H. Burke et al., "Eo-1 hyperion data analysis applicable to cloud detection, coastal characterization and terrain classification," in *Proc. 2004 IEEE Int. Geosci. Remote Sens. Symp.*, 20–24 September 2004, Anchorage, AK, Vol. **2**, pp. 1483–1486 (2004), http://dx.doi.org/10.1109/IGARSS.2004.1368701.

4. M. Martin et al., "Development of an advanced hyperspectral imaging (HSI) system with applications for cancer detection," *Ann. Biomed. Eng.* **34**(6), 1061–1068 (2006), http://dx.doi.org/10.1007/s10439-006-9121-9.
5. A. Paz and A. Plaza, "Cluster versus GPU implementation of an orthogonal target detection algorithm for remotely sensed hyperspectral images," in *Proc. 2010 IEEE Int. Conf. Cluster Comput.*, 20–24 September 2010, Heraklion, Crete, Greece, pp. 227–234, IEEE Computer Society (2010), 10.1109/CLUSTER.2010.28.
6. A. Plaza and C.-I. Chang, "Clusters versus FPGA for parallel processing of hyperspectral imagery," *Int. J. High Perform. C.* **22**(4), 366–385 (2008), http://dx.doi.org/10.1177/1094342007088376.
7. A. Plaza et al., "Commodity cluster and hardware-based massively parallel implementations of hyperspectral imaging algorithms," *Proc. SPIE* **6233**(1), 623316 (2006).
8. C. González et al., "FPGA implementation of the pixel purity index algorithm for remotely sensed hyperspectral image analysis," *EURASIP J. Adv. Signal Process.* **2010**, 13 (2010).http://dx.doi.org/10.1155/2010/969806.
9. D. González et al., "Abundance estimation algorithms using NVIDIA CUDA technology.," *Proc. SPIE* **6966**, 69661E (2008).
10. S. Sanchez et al., "GPU implementation of fully constrained linear spectral unmixing for remotely sensed hyperspectral data exploitation," *Proc. SPIE* **7810**, 78100G (2010).
11. A. Paz and A. Plaza, "GPU implementation of target and anomaly detection algorithms for remotely sensed hyperspectral image analysis," *Proc. SPIE* **7810**, 78100R (2010).
12. *NVIDIA CUDA Programming Guide*, http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf Version 3.2. (November 2010).
13. J. Harsanyi and C.-I. Chang, "Hyperspectral image classification and dimensionality reduction: an orthogonal subspace projection approach," *IEEE Trans. Geosci. Remote Sens.* **32**(4), 779–785 (1994), http://dx.doi.org/10.1109/36.298007.
14. M. E. Winter and E. Winter, "Hyperspectral processing in graphical processing units," *Proc. SPIE* **8048**, 80480O (2011).http://dx.doi.org/10.1117/12.884668
15. Y. Tarabalka et al., "Real-time anomaly detection in hyperspectral images using multivariate normal mixture models and GPU processing," *J Real-Time Image Proc.* **4**(3), 287–300 (2009), http://dx.doi.org/10.1007/s11554-008-0105-x. (print version) (electronic version).
16. I. Reed and X. Yu, "Adaptive multiple-band CFAR detection of an optical pattern with unknown spectral distribution," *IEEE Trans. Acoust. Speech Signal Process.* **38**(10), 1760–1770 (1990), http://dx.doi.org/10.1109/29.60107.
17. R. J. Muirhead, *Aspects of Multivariate Statistical Theory*, Wiley, New York (1982).
18. N. Acito, G. Corsini, and M. Diani, "Adaptive detection algorithm for full pixel targets in hyperspectral images," *IEE Proc. Vis. Image Signal Process.* **152**(6), 731–740 (2005), http://dx.doi.org/10.1049/ip-vis:20045025.
19. J. E. West, Capt., D. W. Messinger, and J. R. Schott, "Comparative evaluation of background characterization techniques for hyperspectral unstructured matched filter target detection," *J. Appl. Remote Sens.* **1**, 013520 (2007), http://dx.doi.org/10.1117/1.2768621.
20. D. Manolakis, D. Marden, and G. Shaw, "Hyperspectral image processing for automatic target detection applications," *Lincoln Lab. J.* **14**(1), 79–116 (2003).
21. D. Manolakis, C. Siracusa, and G. Shaw, "Hyperspectral subpixel target detection using the linear mixing model," *IEEE Trans. Geosci. Remote Sens.* **39**(7), 1392–1409 (2001), http://dx.doi.org/10.1109/36.934072.
22. J. Broadwater, R. Meth, and R. Chellappa, "A hybrid algorithm for subpixel detection in hyperspectral imagery," in *Proc. 2004 IEEE Int. Geosci. Remote Sens. Symp.*, 20–24 September 2004, Anchorage, AK, Vol. **3**, pp. 1601–1604 (Sept. 2004), http://dx.doi.org/10.1117/12.542460.
23. P. Bajorski, E. J. Ientilucci, and J. R. Schott, "Comparison of basis-vector selection methods for target and background subspaces as applied to subpixeltarget detection," *Proc. SPIE* **5425**, 97 (2004).http://dx.doi.org/10.1117/12.542460

24. J. R. Schott et al., "A subpixel target detection technique based on the invariance approach," *Proc. the AVIRIS Workshop*, Pasadena CA, Jet Propulsion Laboratory Publication (February 2003).
25. C. Peña-Ortega and M. Velez-Reyes, "Comparison of basis-vector selection methods for structural modeling of hyperspectral imagery," *Proc. SPIE* **7457**(1), 74570C (2009).
26. C. Peña-Ortega and M. Velez-Reyes, "Evaluation of different structural models for target detection in hyperspectral imagery," *Proc. SPIE* **7695**, 76952H (2010).
27. B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, New Jersey (August 1998).
28. W. H. Press et al., *Numerical Recipes in C. The Art of Scientific Computing*, 2nd ed., Cambridge University Press, New York (1998).
29. "CUDA CUBLAS library," http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUBLAS_Library.pdf Version 3.2. (August 2010).
30. J. R. Humphrey et al., "CULA: hybrid GPU accelerated linear algebra routines," *Proc. SPIE* **7705**, 770502 (2010).
31. R. Chandra et al., *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers Inc., San Francisco (2001).

Blas Trigueros-Espinosa received the BS degree in telecommunication engineering from the Miguel Hernández University of Elche, Spain, in 2005; and the MS degree in computer engineering from the University of Puerto Rico at Mayaguez in 2011. He was a research assistant at the Bioengineering Institute of the Miguel Hernández University from 2006 to 2007, working on the development of image enhancement algorithms and visual diagnostic software. From 2008 to 2011 he was working at the Laboratory for Applied Remote Sensing and Image Processing (LARSIP) under a graduate assistantship conducting research on the implementation of hyperspectral target detection algorithms using NVIDIA GPUs. His research interests include image processing, remote sensing, and parallel programming.

Miguel Vélez-Reyes received the BSEE degree from the University of Puerto Rico at Mayagüez (UPRM), in 1985, and the MS and PhD degrees from the Massachusetts Institute of Technology, Cambridge, in 1988, and 1992, respectively. In 1992, he joined the University of Puerto Rico at Mayaguez, where he is currently a professor at the electrical and computer engineering department. He has held faculty internship positions with AT&T Bell Laboratories, Air Force Research Laboratories, and the NASA Goddard Space Flight Center. His teaching and research interests are in physics-based signal processing and signal and image processing algorithms for remote sensing using hyperspectral imaging (or imaging spectroscopy). He has over 120 publications in journals and conference proceedings and has contributed to three books. He is the director of the Institute for Research in Integrative Systems and Engineering at UPRM and associate director of the NSF Center for Subsurface Sensing and Imaging Systems a NSF engineering research center led by Northeastern University. He was director for the UPRM Tropical Center for Earth and Space Studies, a NASA University research center. In 1997, Dr. Vélez-Reyes was one of 60 recipients from across the United States and its territories of the Presidential Early Career Award for Scientists and Engineers from the White House. In 2005, Dr. Vélez-Reyes was inducted in the Academy of Arts and Sciences of Puerto Rico. In 2010, Dr. Velez-Reyes was elected fellow of SPIE for his contributions to hyperspectral image processing. He is a senior member of the IEEE where he has held many posts such as president of the IEEE Western Puerto Rico Section, and Latin America representative to the IEEE PELS AdCom. He is a member of the Tau Beta Pi, Sigma Xi, and Phi Kappa Phi honor societies.

Nayda G. Santiago received the BSEE degree from University of Puerto Rico, Mayaguez campus, in 1989, the MEEE degree from Cornell University in 1990, and the PhD degree in electrical engineering from Michigan State University in 2003. Since 2003, she has been a faculty member of the University of Puerto Rico, Mayaguez campus at the electrical and computer engineering department where she holds a position as associate professor. She has been recipient of the 2008 Outstanding Professor of Electrical and Computer Engineering Award, 2008 Distinguished Computer Engineer Award of the Puerto Rico Society of Professional Engineers and Land Surveyors, the 2008 HENAAC (Hispanic Engineer National Achievement Awards Conference) Education Award, the 2009 Distinguished Alumni Award of the University of Puerto Rico, Mayaguez campus, and the 2011 Women on the Forefront of the Puerto Rico Society of Professional Engineers and Land Surveyors. She is a member of SANCAS, Latinas in Computing, IEEE, and the ACM. She is one of the founding members of CAHSI and Femprof. Her areas of research include use of novel architectures for hyperspectral image analysis and low power software design. She teaches computer architecture and capstone in computer engineering.

Samuel Rosario-Torres is graduated with a bachelor degree in computational mathematics from University of Puerto Rico (UPR) at Humacao (1995 to 1999) and graduated with a master degree in computer engineering from UPR at Mayaguez (2001 to 2004). Working in the Laboratory of Applied of Remote Sensing and Image Analysis at UPR-Mayaguez as software developer and research associate from 2004 to the present developing, enhancing and optimizing image analysis algorithms.