GPU-BASED IMPLEMENTATION OF TARGET DETECTION ALGORITHMS FOR HYPERSPECTRAL IMAGES USING NVIDIA® CUDA™

by

Blas Trigueros Espinosa

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

University of Puerto Rico
Mayagüez Campus

2011

Approved by:

Shawn D. Hunt, Ph.D Member, Graduate Committee	Date
Nayda G. Santiago, Ph.D Co-president, Graduate Committee	Date
Miguel Vélez-Reyes, Ph.D President, Graduate Committee	Date
Dorial Castellanos, Ph.D Representative of Graduate Studies	Date
Erick E. Aponte, Ph.D Chairperson of the Department	Date

Abstract of Dissertation Presented to the Graduate School
of the University of Puerto Rico in Partial Fulfillment of the
Requirements for the Degree of Master of Science

GPU-BASED IMPLEMENTATION OF TARGET DETECTION ALGORITHMS FOR HYPERSPECTRAL IMAGES USING NVIDIA® CUDA™

By

Blas Trigueros Espinosa

June 2011

Chair: Miguel Vélez
Major Department: Electrical and Computer Engineering

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 GPUs, due to their highly parallel structure and the high computational capabilities that can be achieved at relative low costs. In this work, we studied the parallel implementation of target detection algorithms for hyperspectral images in order to identify the aspects in the structure of these algorithms that can exploit the parallel computing resources of GPUs based on the NVIDIA® CUDA™ architecture. A dataset was generated using a SOC-700 hyperspectral imager to evaluate the performance and detection accuracy of the parallel implementations. In addition, a library of target detectors was developed to facilitate the use of the algorithms by future researchers.

Resumen de Disertación Presentado a Escuela Graduada
de la Universidad de Puerto Rico como requisito parcial de los
Requerimientos para el grado de Maestría en Ciencias

IMPLEMENTACIÓN EN GPU DE ALGORITMOS DE DETECCIÓN PARA IMÁGENES HIPERESPECTRALES USANDO NVIDIA® CUDA™

Por

Blas Trigueros Espinosa

Junio 2011

Consejero: Miguel Vélez
Departamento: Ingeniería Eléctrica y Computadoras

Avances recientes en los sensores hiperespectrales permiten la adquisición de imágenes de una escena a cientos de bandas espectrales contiguas y estrechas. Los algoritmos de detección tratan de aprovechar esta alta resolución espectral para detectar materiales de interés en una escena, pero este proceso puede ser computacionalmente intenso debido al gran volumen de datos generado por los sensores hiperespectrales, típicamente cientos de megabytes. Trabajos previos han mostrado que el procesamiento de datos hiperespectrales se puede beneficiar significativamente de los recursos de computación en paralelo de los GPUs, debido a su estructura altamente paralela y las altas capacidades de computación que pueden alcanzar a un precio relativamente bajo. En este trabajo estudiamos la implementación en paralelo de algoritmos de detección para imágenes hiperespectrales con el fin de identificar aspectos en la estructura de estos algoritmos que puedan sacar ventaja de los recursos de computación paralela de GPUs basados en la arquitectura CUDA™ de NVIDIA®. Un conjunto de datos fue generado usando una cámara hiperespectral

SOC-700 para evaluar el rendimiento y la precisión en detección de las implementaciones. En adición, se desarrollo una librería de algoritmos de detección para facilitar el uso de los algoritmos por futuros investigadores.

To my parents Blas and María Jesús.

Acknowledgments

First of all, I would like to express my acknowledgment and gratitude to my advisor Dr. Miguel Vélez-Reyes for his support, patience, and great advice during the development of this research. I would also like to acknowledge my co-advisor Dr. Nayda G. Santiago for her valuable support and contribution, and special thanks to all the members of my Graduate Committee for their guidance and review.

I am also very grateful to all the persons working at LARSIP (Laboratory for Applied Remote Sensing and Image Processing). Special thanks to Victor Asencio and Samuel Rosario for their technical support and to Maribel Feliciano for her help in all the academic and administrative affairs. I also want to acknowledge the academic advisor of the Electrical and Computer Engineering Department, Sandra Montalvo, for her effort, advice, and support during my graduate studies.

Finally, I want to express my very special gratitude to the Rivera-Santiago family for their unconditional support, hospitality and for making my stay in Puerto Rico unforgettable.

This work was primarily 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 was also received from the National Science Foundation under award EEC-0946463.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ACE	Adaptive Cosine/Coherent Estimator.
AMEE	Automated Morphological Endmember Extraction.
AMSD	Adaptive Matched Subspace Detector.
API	Application Programming Interface.
ATDCA	Automatic Target Detection and Classification Algorithm.
AVIRIS	Airborne Visible/Infrared Imaging Spectrometer.
BIL	Band Interleaved by Line.
BIP	Band Interleaved by Pixel.
BLAS	Basic Linear Algebra Subprograms.
BSQ	Band Sequential.
cPMF	Constrained Positive Matrix Factorization.
CUDA	Compute Unified Device Architecture.
DP	Double Precision.
FCLS	Fully Constrained Least Squares.
FLOPS	Floating-point Operations per Second.
GLR	Generalized Likelihood Ratio
GPU	Graphics Processing Unit.
HSD	Hybrid Structured Detector.
HSI	Hyperspectral Imaging.
HUD	Hybrid Unstructured Detector.
HYDICE	Hyperspectral Digital Imagery Collection Experiment
ISRA	Image Space Reconstruction Algorithm.
LAPACK	Linear Algebra Package.
MaxD	Maximum Distance.
MF	Matched Filter.
NNLS	Non Negative Least Squares.
NNSTO	Non Negative Sum to One.
ORASIS	Optical Real-time Adaptive Spectral Identification System.
OSP	Orthogonal Subspace Projector.
PPI	Pixel Purity Index.
SDK	Software Development Kit.
SFU	Special Function Unit.
SIMD	Single-Instruction Multiple-Data.
SIMT	Single-Instruction Multiple-Thread.
SM	Streaming Multiprocessor.

SP	Scalar Processor.
SVD	Singular Value Decomposition.

List of Symbols

H_0	Hypothesis of target absent.
H_1	Hypothesis of target present.
η	Threshold.
$\mathbf{x}$	Pixel spectrum.
$\boldsymbol{\mu}_0$	Background mean vector.
$\boldsymbol{\mu}_1$	Target mean vector.
$\boldsymbol{\Gamma}_0$	Background covariance matrix.
$\boldsymbol{\Gamma}_1$	Target covariance matrix.
N	Number of pixels.
L	Number of spectral bands.
M	Dimensionality of background subspace.
P	Dimensionality of target subspace.
$\mathbf{B}$	Matrix that spans the background subspace.
$\mathbf{S}$	Matrix that spans the target subspace.
$\mathbf{E}$	Matrix that spans the union of background and target subspaces.
$\mathbf{w}$	Noise vector.
$\mathbf{a_S}$	Target abundances.
$\mathbf{a_{b,0}}$	Background abundances under the hypothesis of target absent.
$\mathbf{a_{b,1}}$	Background abundances under the hypothesis of target present.
$\mathbf{P}_{\mathbf{B}}^{\perp}$	Projection matrix onto the subspace orthogonal to the background subspace.
$\mathbf{P}_{\mathbf{E}}^{\perp}$	Projection matrix onto the subspace orthogonal to the union of background and target subspaces.
$\mathbf{L}$	Lower triangular matrix.

Chapter 1

Introduction

1.1 Motivation

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 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.

The research work presented in this thesis 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. Different GPU-based implementations were proposed and evaluated in terms of running time, speedup, and detection accuracy.

1.2 Objectives

The main objectives of this work were to study the parallel implementation of target detection algorithms for hyperspectral imagery using CUDA-capable NVIDIA® GPUs and develop efficient implementations of the algorithms that exploit the parallel hardware architecture. The specific objectives of this work were:

- To study and develop GPU-based parallel implementation of different target detection algorithms for both full-pixel and sub-pixel detection in HSI using the *CUDA C*™ environment.
- To study and identify elements in the structure of the algorithms that can be exploited by the NVIDIA® CUDA architecture to get better performance.
- To develop CPU-based serial and parallel implementations to be used as a baseline for comparison of resulting running times and speedup estimation.

- To evaluate the performance and detection accuracy of the GPU-based parallel implementations and study potential applications in real time detection.
- To develop a CUDA library of target detection algorithms to facilitate future use by researchers.

1.3 Contribution of the Work

This work presents an analysis of the parallel implementation on GPU of three different target detection algorithms: the RX algorithm, the matched filter (MF), and the adaptive matched subspace detector (AMSD). The structure of these algorithms was analyzed to identify the aspects that can be exploited by the CUDA architecture and the aspects that cannot be exploited due to limitations of the GPU.

An implementation approach based on the Cholesky decomposition of the background covariance matrix was proposed for the GPU-based implementation of the full-pixel detectors (RX and MF). For the AMSD, two methods for estimating the background subspace were evaluated in the GPU-based implementation: SVD and MaxD. In addition, the running times of the proposed implementations were measured for different data sizes to analyze the speedup and the real time performance of the GPU-based implementations.

As a final result of this work, a library of target detectors that includes all the proposed GPU-based implementations was developed to facilitate future use by researchers.

1.4 Thesis Outline

Chapter 2 starts presenting background concepts related to hyperspectral imaging and target detection in hyperspectral images. It presents a brief description of the target detection theory and describes different target detection algorithms

proposed in the literature. Chapter 2 also describes the concepts related to the Graphics Processing Units (GPU) and the NVIDIA® CUDA™ parallel computing architecture. Finally, previous works related to hyperspectral data processing using GPUs are presented.

Chapter 3 describes the GPU implementation of three target detection algorithms for hyperspectral images: RX algorithm, matched filter (MF), and adaptive matched subspace detector (AMSD). Two different implementations for the RX algorithm are described. The first one uses global estimates of the background parameters using training samples from the entire image. In the second implementation, the background parameters are locally estimated using a window centered at the test pixel. In the implementation of the AMSD algorithm, two methods for estimating the background subspace were evaluated: SVD and the MaxD algorithm. This chapter also discusses some aspects related to the parallel implementation of the algorithms on the GPU to exploit the CUDA architecture: the task decomposition, the organization of the data in memory, and how to map the computations on the GPU.

Chapter 4 describes the experiments performed to evaluate the running times and the detection accuracy of the algorithms. Finally, the experimental results are presented and analyzed.

Chapter 5 presents the conclusions of this research and final remarks for future work.

Chapter 2

Background and Literature Review

2.1 Hyperspectral Imaging

Spectral imaging is the acquisition of images where every pixel measures the amount of electromagnetic radiation reaching the sensor at different spectral bands, providing both spatial and spectral information of a scene [13]. In hyperspectral imaging, hundreds of images are registered simultaneously at spectrally contiguous narrow bands. This high-resolution spectral information can be used to discriminate between the different materials present in the image, since the spectral distribution of the radiation emitted, absorbed and reflected by the materials depends on their physical and chemical properties.

A hyperspectral image can be viewed as a three-dimensional data cube with two spatial dimensions and a third spectral dimension, so every pixel is a vector corresponding to a single spectrum. The concept of a hyperspectral cube is illustrated in Figure 2–1.

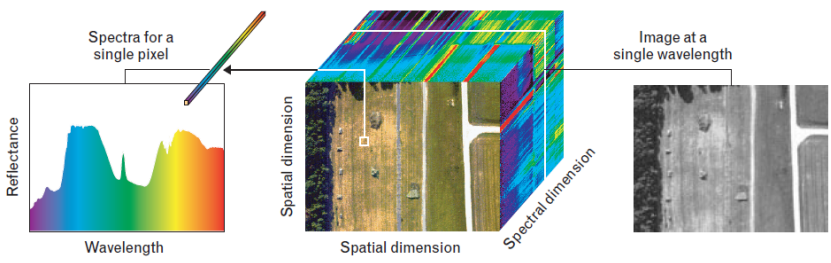


Figure 2–1: Structure of a hyperspectral imaging data cube. From [13].

An example of an airborne hyperspectral imager is the NASA's Airborne Visible/Infrared Imaging Spectrometer (AVIRIS) [14] developed by the NASA Jet Propulsion Laboratory. This sensor is able to record images in the visible and near infrared region (from 0.4 μm to 2.5 μm) using 224 contiguous 10-nm width spectral channels, resulting in hyperspectral data cubes comprising from hundreds of megabytes to several gigabytes. Another example is the HYDICE (HYperspectral Digital Imagery Collection Experiment) sensor [15], which can collect spectral radiance data in 210 bands from 0.4 μm to 2.5 μm. In this case, a hyperspectral data cube consisting of 320 $\times$ 1,280 pixels has a size of 164.1 MB.

2.2 Target Detection in HSI

One of the most important tasks in hyperspectral imaging exploitation is target detection. A target detection algorithm tries to identify the presence of objects or materials in a scene by exploiting the spatial and high spectral resolution information contained in a hyperspectral image. A detection problem can be described as a binary hypothesis test, where two competing hypotheses are generated to differentiate the pixels containing the target of interest from the pixels containing only background spectra [13]:

$$H_0: \ \textbf{target absent}$$

$$H_1: \ \textbf{target present}$$

Based on the measured pixel spectrum, a detection algorithm has to decide which hypothesis is true. A criterion for an optimum design of a detector could be the maximization of the detection probability (the probability of selecting H_1 as true, when H_1 is true) while keeping the probability of false alarm (the probability of selecting H_1 as true, when H_0 is true) as low as possible.

The optimal detection statistic is the likelihood ratio test, which is optimum for a wide range of performance criteria [13]. In this approach, the measured spectrum

is treated as a random vector with a specific probability distribution. If $\mathbf{x}$ is the observed pixel spectrum, the likelihood ratio test is given by:

$$\Lambda(\mathbf{x}) = \frac{p(\mathbf{x}|H_1)}{p(\mathbf{x}|H_0)} \tag{2.1}$$

where $p(\mathbf{x}|H_1)$ and $p(\mathbf{x}|H_0)$ are the conditional probability density functions of $\mathbf{x}$ under the two hypothesis. If the ratio $\Lambda(\mathbf{x})$ exceeds a given threshold η, then the hypothesis H_1 (target present) will be selected as true. Otherwise, the target absent hypothesis H_0 will be selected.

In many practical situations, the conditional probability densities of the likelihood ratio are not known or depend on some unknown parameters (for example, the mean and the covariance matrix of the background class distribution are usually not known a priori). A common approach to handle this scenario is to replace the unknown parameters in the likelihood ratio with their maximum-likelihood estimates. This estimated likelihood ratio is known as Generalized Likelihood Ratio (GLR).

The variability of the target and background spectra can be described by using subspace or statistical models. The target signatures can be estimated from the pixels of the image, from ground truth, or library spectra.

2.2.1 Full-Pixel Target Detectors

Full-pixel target detectors assume that the pixels of the image do not contain mixed spectra, and, therefore, every pixel vector represents the spectral information of only one class (target or background). The spectral variability of every class present in the image can be modeled using some statistical distribution. Multivariate normal distributions are usually chosen since they lead to simple mathematical models and show good performance for a wide range of scenarios. In this case, the

detection problem can be described using the following hypotheses:

$$H_0: \quad \mathbf{x} \sim N(\boldsymbol{\mu}_0, \boldsymbol{\Gamma}_0) \quad \textbf{target absent}$$
$$H_1: \quad \mathbf{x} \sim N(\boldsymbol{\mu}_1, \boldsymbol{\Gamma}_1) \quad \textbf{target present}$$

where $\mathbf{x}$ is the pixel random vector; $\mu_{\mathbf{0}}$, $\mathbf{\Gamma_0}$ are the mean vector and covariance matrix of the multivariate normal distribution that characterize the background class, respectively, and $\mu_{\mathbf{1}}$, $\mathbf{\Gamma_1}$ are the mean vector and covariance matrix of the multivariate normal distribution that characterize the target class, respectively.

Taking the natural logarithm of the likelihood ratio defined in Equation 2.1 and using the multivariate normal distribution as a probability density under each hypothesis, the following detector is derived:

$$y = D(\mathbf{x}) = (\mathbf{x}-\boldsymbol{\mu}_0)^T\boldsymbol{\Gamma}_0^{-1}(\mathbf{x}-\boldsymbol{\mu}_0) - (\mathbf{x}-\boldsymbol{\mu}_1)^T\boldsymbol{\Gamma}_1^{-1}(\mathbf{x}-\boldsymbol{\mu}_1) \underset{H_0}{\overset{H_1}{\gtrless}} \eta \tag{2.2}$$

This detector basically compares the *Mahalanobis distance* [16] of the pixel vector from the centers of the two target and background classes, and makes a decision based on a given threshold.

If we assume that the target and background classes have the same covariance matrix $\mathbf{\Gamma} = \mathbf{\Gamma}_0 = \mathbf{\Gamma}_1$, the detector becomes a linear processor known as *Matched Filter (MF)*, which is given by [13]:

$$y = D(\mathbf{x}) = \kappa(\boldsymbol{\mu}_1-\boldsymbol{\mu}_0)^T\boldsymbol{\Gamma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_0) = \frac{(\boldsymbol{\mu}_1-\boldsymbol{\mu}_0)^T\boldsymbol{\Gamma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_0)}{(\boldsymbol{\mu}_1-\boldsymbol{\mu}_0)^T\boldsymbol{\Gamma}^{-1}(\boldsymbol{\mu}_1-\boldsymbol{\mu}_0)} \underset{H_0}{\overset{H_1}{\gtrless}} \eta \tag{2.3}$$

where κ is a normalization factor that is usually chosen to produce the output $y = 1$ when $\mathbf{x} = \boldsymbol{\mu}_1$. The idea of the matched filter is to project the pixel vector onto the direction that provides the best separability between the background and target classes.

In many practical situations, the statistics of the target class are unknown or are very difficult to estimate when the image only contain a few target pixels. In this case, the detection can be performed using only the statistical information of the background $(\boldsymbol{\mu}_0, \boldsymbol{\Gamma}_0)$ by considering as targets those pixels that differ too much from the background distribution. This approach is known as *anomaly detection* and one of the most used anomaly detectors is the *RX algorithm* [17] given by:

$$y = D(\mathbf{x}) = (\mathbf{x} - \boldsymbol{\mu}_0)^T \boldsymbol{\Gamma}^{-1} (\mathbf{x} - \boldsymbol{\mu}_0) \begin{array}{c} H_1 \\ > \\ < \\ H_0 \end{array} \eta \tag{2.4}$$

The RX algorithm computes the Mahalanobis distance of the pixel vector $\mathbf{x}$ from the mean of the background class. If the distance is larger than the selected threshold, the pixel vector is considered a target.

In practice, these full-pixel targets need to be adaptive because the background statistics are usually unknown and they have to be estimated directly from the image. One common approach is to use kernel-based detectors that use a window centered at the test pixel that defines a surrounding region used to compute the background statistical parameters.

2.2.2 Sub-pixel Target Detectors

Sub-pixel target detection algorithms assume that the target may occupy only a portion of the pixel area and the remaining part is filled with one or more materials,

which constitute the background. Therefore, the observed pixel will be a mixture of the target and background spectra. The most common model used to describe this mixing process is the *linear mixing model* [18], which states that the measured spectrum $\mathbf{x}$ of every pixel is generated by a linear combination of a given number of unique deterministic spectral signatures or *endmembers*, that represent pure materials within the pixel. Every endmember has a corresponding *abundance*, which represents the relative fraction coverage of each endmember within the pixel area. This model assumes that the incident radiation bounces only once when it interacts with the surface, so the radiation reaching the sensor will be a linear combination of the components reflected by all the materials within the field of view of the sensor. The linear mixing model can be stated as follows [18]:

$$\mathbf{x} = \sum_{i=1}^{M} a_i \mathbf{e_i} + \mathbf{w} = \mathbf{E}\mathbf{a} + \mathbf{w} \tag{2.5}$$

where $\mathbf{e_1}, \mathbf{e_2}, \ldots \mathbf{e_M}$, are the M endmember spectra, $a_1, a_2, \ldots a_M$ are their corresponding abundances and $\mathbf{w}$ is an additive noise. The endmember are usually assumed linearly independent (the matrix $\mathbf{E}$ will have full-rank) and the abundance values, since they represent cover material fractions, must satisfied the non-negativity and sum-to-one constraints:

$$\begin{aligned} &a_i \geq 0 \qquad \text{(non-negativity constraint)} \\ &\sum_{i=1}^{M} a_i = 1 \qquad \text{(sum-to-one constraint)} \end{aligned} \tag{2.6}$$

When the incident radiation experiences multiple reflections, the linear mixing model is not appropriate since, in this case, the measured spectrum at the sensor will be a nonlinear mixture of different constituent spectra.

Depending on whether the model used to describe the variability of the background spectrum is a statistical distribution (unstructured background) or a subspace (structured background), the sub-pixel target detectors can be classified into two different classes:

Unstructured Background Models

In unstructured background models, the background spectral variability is modeled using a probability distribution (usually the multivariate normal distribution) and the target variability is represented by a linear subspace. Therefore, the two competing hypotheses are:

$$H_0 : \quad \mathbf{x} = \mathbf{v}$$
$$H_1 : \quad \mathbf{x} = \mathbf{S}\mathbf{a}_\mathbf{S} + \mathbf{v}$$

where $\mathbf{v}$ is a random vector that represents the background variability and the noise, $\mathbf{S}$ is an $L \times P$ matrix whose columns span the target subspace, L is the number of spectral bands, P is the dimensionality of the target subspace and $\mathbf{a_S}$ is a $P \times 1$ abundance vector. The variability of the target increases as the number of columns P of $\mathbf{S}$ increases. When $P = 1$, the shape of the target spectrum is known *a priori* and the spectral variability is restricted to variations in the amplitude of the target spectrum.

One of the most powerful detectors based on unstructured backgrounds is the *Adaptive Cosine/Coherent Estimator* (ACE) [19, 20]. This detector models the entire background as a zero-mean multivariate Normal distribution, yielding the following two hypotheses:

$$H_0 : \quad \mathbf{x} \sim N(0, \sigma_0^2 \mathbf{\Gamma})$$
$$H_1 : \quad \mathbf{x} \sim N(\mathbf{S}\mathbf{a}_\mathbf{S}, \sigma_1^2 \mathbf{\Gamma})$$

Using the maximum likelihood estimates for $\mathbf{a}$, σ_0^2 σ_1^2 and $\mathbf{\Gamma}$, the GLR results in the following detector:

$$D_{ACE}(\mathbf{x}) = \frac{\mathbf{x^T}\hat{\mathbf{\Gamma}}^{-1}\mathbf{S}\hat{\mathbf{a}}_\mathbf{S}}{\mathbf{x^T}\hat{\mathbf{\Gamma}}^{-1}\mathbf{x}} = \frac{\mathbf{x^T}\hat{\mathbf{\Gamma}}^{-1}\mathbf{S}(\mathbf{S^T}\hat{\mathbf{\Gamma}}^{-1}\mathbf{S})^{-1}\mathbf{S^T}\hat{\mathbf{\Gamma}}^{-1}\mathbf{x}}{\mathbf{x^T}\hat{\mathbf{\Gamma}}^{-1}\mathbf{x}} \begin{matrix} H_1 \\ > \\ < \\ H_0 \end{matrix} \eta \tag{2.7}$$

where $\hat{\mathbf{\Gamma}}$, $\hat{\mathbf{a}}_\mathbf{S}$ are the maximum likelihood estimate of the background covariance matrix and the target abundances, respectively.

Structured Background Models

In structured background models, both the background and target spectral variability are modeled using a linear subspace of dimension $M < L$, where L is the number of spectral bands. Therefore, in this model, every pixel vector $\mathbf{x}$ can be represented as a linear combination of M basis vectors. The two competing hypotheses are:

$$H_0 : \quad \mathbf{x} = \mathbf{B}\mathbf{a}_{\mathbf{b,0}} + \mathbf{w}$$
$$H_1 : \quad \mathbf{x} = \mathbf{S}\mathbf{a}_\mathbf{S} + \mathbf{B}\mathbf{a}_{\mathbf{b,1}} + \mathbf{w}$$

where $\mathbf{B}$ is an $L \times M$ matrix whose columns represents the basis vectors for the background subspace and $\mathbf{S}$ is an $L \times P$ matrix whose columns represents the basis vectors for the target subspace. $\mathbf{a}_{\mathbf{b,0}}$, $\mathbf{a}_{\mathbf{b,1}}$ represent the coefficients of the linear combinations of the background basis vectors under each hypothesis, respectively, and $\mathbf{a}_\mathbf{S}$ represents the coefficients of the linear combinations of the target basis vectors. Therefore, in this model, the background pixels and the pixels containing the target are represented using different linear subspaces in the spectral space. The only source of randomness that this model introduces is the $L \times 1$ vector $\mathbf{w}$, which is a random vector that represents the noise and it is usually modeled as a zero-mean multivariate Normal distribution with covariance matrix $\sigma^2\mathbf{I}$.

One example of structured background detector is the *Adaptive Matched Subspace Detector* (AMSD) described in [21]. Following the GLR approach, this detector is derived by computing the maximum-likelihood estimates for $\mathbf{a_{b,0}}$, $\mathbf{a_{b,1}}$, $\mathbf{a_S}$, $\sigma^2\mathbf{I}$ and replacing them into the likelihood ratio:

$$D_{AMSD}(\mathbf{x}) = \frac{\mathbf{x^T}(\mathbf{I} - \mathbf{B}(\mathbf{B^T B})^{-1}\mathbf{B^T})\mathbf{x}}{\mathbf{x^T}(\mathbf{I} - \mathbf{E}(\mathbf{E^T E})^{-1}\mathbf{E^T})\mathbf{x}} = \frac{\mathbf{x^T P_B^{\perp} x}}{\mathbf{x^T P_E^{\perp} x}} \underset{H_0}{\overset{H_1}{\gtrless}} \eta \tag{2.8}$$

where $\mathbf{E} = [\mathbf{S}\ \mathbf{B}]$ is the matrix obtained by the combination of the target and background subspaces, $\mathbf{P_B^{\perp}}$ is the orthogonal projection matrix onto the background subspace and $\mathbf{P_E^{\perp}}$ is the orthogonal projection matrix onto the combined target and background subspaces. The resulting likelihood ratio for the AMSD detector is often transformed into the ratio:

$$D_{AMSD}(\mathbf{x}) = \frac{\mathbf{x^T}(\mathbf{P_B^{\perp}} - \mathbf{P_E^{\perp}})\mathbf{x}}{\mathbf{x^T P_E^{\perp} x}} \underset{H_0}{\overset{H_1}{\gtrless}} \eta \tag{2.9}$$

which produces a detection statistic based on the F-distribution [21].

The AMSD detector requires the estimation of the matrix $\mathbf{B}$ of background basis vectors. The matrix $\mathbf{B}$ can be estimated from the HSI data cube using the eigenvectors of the image correlation matrix as the basis vectors for the background subspace. Other detectors estimate the subspace parameters using the physical constraints of the linear mixing model. In this case, the background basis vectors correspond to physical endmembers present in the scene and the coefficients of the linear combinations correspond to abundances.

The procedure for estimating the endmembers spectral signatures and their corresponding abundances within each pixel of the hyperspectral image is known as *spectral unmixing* [18]. In the unmixing process, the endmembers are usually estimated first and the estimated endmembers signatures are then used to estimate the abundances values for every pixel. However, other approaches, such as the Constrained Positive Matrix Factorization (cPMF) proposed by Masalmah in [22], estimate the endmembers and the abundances simultaneously by solving an optimization problem. This procedure is known as *unsupervised unmixing*.

The estimation of the endmembers is a wide field where many different algorithms have been proposed, such as Pixel Purity Index (PPI) [23], Automated Morphological Endmember Extraction (AMEE) [24], N-FINDR [25], Optical Real-time Adaptive Spectral Identification System (ORASIS) [26], or Maximum Distance (MaxD) [27].

Once the endmembers are known, the estimation of the abundances can be viewed as the problem of minimizing the distance between the observed pixel and the predicted pixel value based on the estimated endmembers. Using the Euclidean norm as a distance measure, the abundance estimation problem can be formulated as a constrained linear least squares problem:

$$\hat{\mathbf{a}} = \arg\min_{\mathbf{a}} \|\mathbf{E}\mathbf{a} - \mathbf{x}\|_2^2 \tag{2.10}$$

$$\text{subject to } \mathbf{a} \geq \mathbf{0}, \sum_{i=1}^{M} a_i = 1$$

where $\hat{\mathbf{a}}$ is the estimated abundance vector, which is also the maximum likelihood estimate if the additive noise $\mathbf{w}$ is white Gaussian noise (which is an acceptable assumption for many applications).

Solving the fully-constrained least square problem for estimating the abundances requires the use of iterative methods. In [28], Velez and Rosario proposed an algorithm that solves the fully-constrained abundance estimation problem called

Non Negative Sum to One (NNSTO). This algorithm is based on a transformation of the original least square problem into an equivalent least distance problem which can be solved using the *Non Negative Least Square* (NNLS) iterative algorithm proposed by Lawson and Hanson [29]. Other approach for solving the fully-constrained abundance estimation problem was proposed by Chang *et al.* in [30]. The Chang's approach, called the *Fully Constrained Least Square* (FCLS) algorithm, enforces the sum-to-one constraint by using a penalty approach.

In [31], Broadway *et al.* proposed two alternative versions of the ACE and AMSD detectors that incorporate the physical constraints of the linear mixing model, where the abundance values are computed using the FCLS algorithm of Chang *et al.* [30]. The proposed new ACE detector is called *Hybrid Unstructured Detector* (HUD) and is derived by replacing the unconstrained least squares estimate of the abundances in equation 2.7 by the fully-constrained least squares estimate of Equation 2.10:

$$D_{HUD}(\mathbf{x}) = \frac{\mathbf{x^T \Gamma^{-1} S \hat{a}}}{\mathbf{x^T \Gamma^{-1} x}} \begin{array}{c} H_1 \\ > \\ < \\ H_0 \end{array} \eta \tag{2.11}$$

In [31] $\hat{\mathbf{a}}$ is computed using the FCLS algorithm. They also proposed a new AMSD detector called *Hybrid Structured Detector* (HSD) and, as in the case of the HUD detector, is derived by replacing the unconstrained least squares estimate of the abundances in equation 2.8 by the fully-constrained least square estimate (Equation 2.10):

$$D_{HSD}(\mathbf{x}) = \frac{(\mathbf{x} - \mathbf{B\hat{a}})^\mathbf{T}(\mathbf{x} - \mathbf{B\hat{a}})}{(\mathbf{x} - \mathbf{E\hat{a}})^\mathbf{T}(\mathbf{x} - \mathbf{E\hat{a}})} \begin{array}{c} H_1 \\ > \\ < \\ H_0 \end{array} \eta \tag{2.12}$$

When including the non-negative and sum-to-one constraints in the estimation of the abundances, the orthogonal projections of the structured detectors become oblique projections, since now the pixel value is restricted to vary on a convex hull (or convex cone when the sum-to-one constraint is omitted).

Target detectors based in oblique projections were also proposed in [32] by Peña-Ortega and Vélez-Reyes. These detectors are variations of the *Orthogonal Subspace Projector* (OSP) proposed by Chang and Harsanyi in [33]. The approach used in the OSP detector is to nullify the interfering background signatures by projecting every pixel vector onto a subspace orthogonal to the background space:

$$D_{OSP}(\mathbf{x}) = \frac{\mathbf{t^T P_B^\perp x}}{\mathbf{t^T P_B^\perp t}} \begin{array}{c} H_1 \\ > \\ < \\ H_0 \end{array} \eta \tag{2.13}$$

where $\mathbf{t}$ is the desired target signature and $\mathbf{P_B^\perp}$ is the orthogonal projection matrix onto the column space of the endmember matrix $\mathbf{B}$. In the approach proposed in [32], the OSP detector is expressed in terms of the projection errors using the idempotent and symmetric properties of the matrix $\mathbf{P_B^\perp}$ as follows:

$$D_{OSP}(\mathbf{x}) = \frac{\mathbf{t^T P_B^\perp x}}{\mathbf{t^T P_B^\perp t}} = \frac{(\mathbf{P_B^\perp t})^\mathbf{T}(\mathbf{P_B^\perp x})}{(\mathbf{P_B^\perp t})^\mathbf{T}(\mathbf{P_B^\perp t})} = \frac{(\mathbf{e_{OSP}^t})^\mathbf{T}(\mathbf{e_{OSP}^x})}{(\mathbf{e_{OSP}^t})^\mathbf{T}(\mathbf{e_{OSP}^t})} \tag{2.14}$$

where $\mathbf{e_{OSP}^t} = \mathbf{t} - \mathbf{Ba^t}$ and $\mathbf{e_{OSP}^x} = \mathbf{t} - \mathbf{Ba^x}$ are the projection errors of the target and the observed pixel $\mathbf{x}$, respectively. Two new detectors are then derived by replacing the unconstrained estimates of $\mathbf{a^t}$, $\mathbf{a^x}$ for the target and the observed pixel abundances by the corresponding positive constrained and fully-constrained estimates computed using the NNLS and NNSTO algorithms [32], respectively. These detectors project the target and pixel spectra onto a convex cone, when only the

positivity constraint is enforced, and onto a convex hull, when both the positivity and sum to one constraints are enforced.

2.3 Graphics Processing Units

A Graphics Processing Unit (GPU) is a specialized many-core processor originally designed for 3D graphics rendering [12]. It is widely used in video cards, motherboards, mobile phones, and games consoles. The increasingly demanding market of computer game has driven a rapid development of the GPU technology in the last few years. Nowadays, a modern GPUs can achieve a computational power on the order of hundreds of giga FLOPS (**FL**oating point **O**perations **P**er **S**econd) (Figure 2–2) at a relative low cost. In addition to the increase in computational power, recent improvements in the programmability of these devices make them an interesting alternative to CPUs for general-purpose computing instead of dedicated devices for graphics rendering. In fact, today some GPUs like the NVIDIA® Tesla™ family are designed exclusively for general-purpose computing and can reach a computational power well above the fastest multi-core CPU.

The big gap between the GPU and the CPU performance is due to the differences in their architecture and physical design. In the design of a GPU chip, most of the transistors are devoted to data processing (arithmetic logic units). A CPU, in contrast, contains more transistors devoted to data caching and flow control, as illustrated in Figure 2–3. Therefore, a CPU is optimized for executing sequential code, whereas, a GPU is optimized for compute-intensive and highly parallel computations (the nature of graphics rendering). This makes the GPU a more suitable platform than the CPU for addressing data-parallel compute-intensive problems, where the same program can be executed on different data elements in parallel. Data-parallelism reduces the need of sophisticated flow control and, if the number

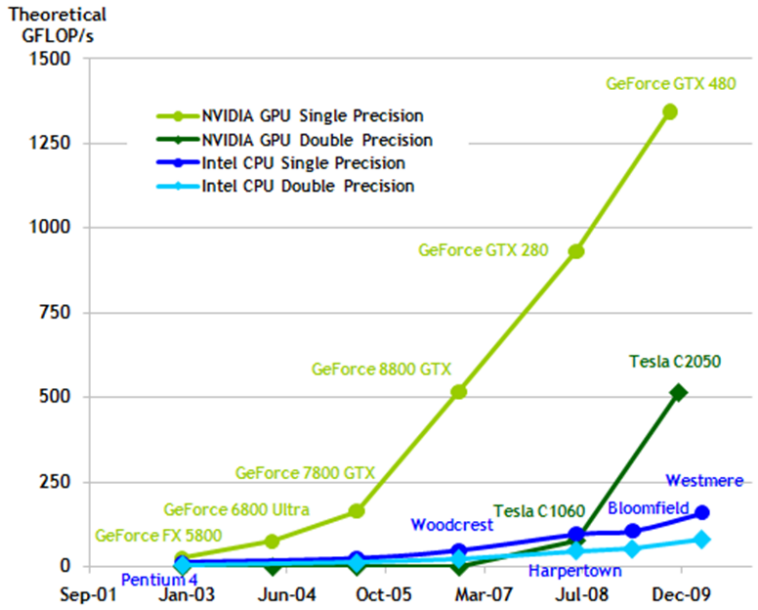


Figure 2–2: Floating-Point Operations per Second: CPU vs. GPU Comparison. From [12].

of arithmetic operations is high enough compared to the number of memory operations, the memory access latency can be hidden with computations instead of large data caches.

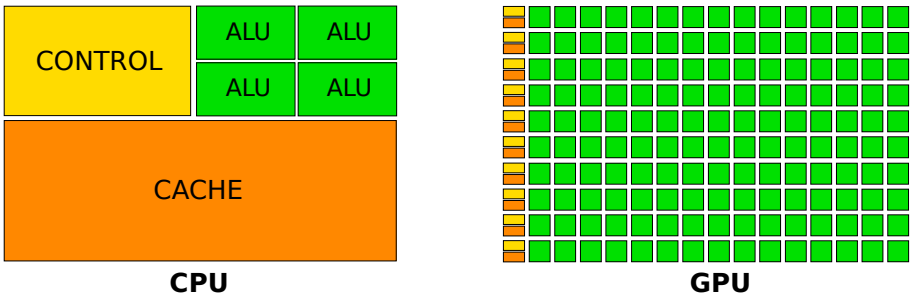


Figure 2–3: Structure of a CPU and a GPU chip.

2.4 NVIDIA® CUDA™ Architecture

CUDA™, which stands for Compute Unified Device Architecture, is a general-purpose parallel computing architecture introduced by NVIDIA® in November 2006 [12]. CUDA provides an easy access to the computing resources of NVIDIA® GPUs

through an application programming interface (API) which is an extension of the C programming language. Basically, these extensions allow the programmer to take an original C program than runs on a CPU and offload some sections of the code to the GPU and take advantage of its high computational power without the need of using a graphics API. This reduces the learning curve and the API overhead for non-graphics applications.

To take advantage of the parallel computing capabilities of GPUs, CUDA allows the definition of a special type of functions called *kernels* that are configured to be executed in parallel on the GPU. Kernels are defined to enclose data-parallel sections of the original code, so the programmer can offload parallel and compute-intensive sections by moving these computations to the GPU while still making use of the CPU when necessary. Therefore, in the CUDA model, the GPU is viewed as a co-processor (device) within a CPU-based computer (host) that is capable of performing intensive computations in parallel and has its own memory space. Figure 2–4 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. Then, 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.

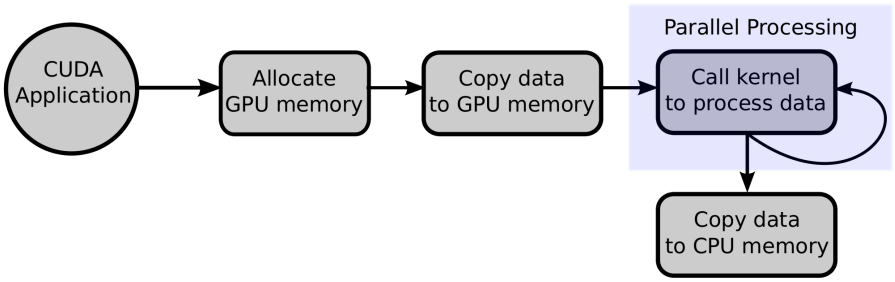


Figure 2–4: CUDA program flow for data parallel processing.

2.4.1 Programming Model

CUDA supports two programming interfaces: *CUDA C* and *CUDA driver API* [12]. CUDA C provides a minimal set of extensions to the C language for defining kernel functions and managing its execution. CUDA driver API is a low-level interface that provides functions to manage CUDA binary and assembly code. Both interfaces come with a runtime API that provides functions to allocate and deallocate GPU memory, transfer data between system and GPU memory, synchronize parallel threads of execution, etc.

CUDA C provides an easy interface for users familiar with the standard C language to define kernel functions. When a kernel is called, it is executed in parallel by different CUDA *threads*. A thread can be defined as the smallest unit of processing scheduled for execution. Unlike CPU threads, CUDA threads are very lightweight, which means that these threads can be created and scheduled in only a few clock cycles. Since all the CUDA threads execute the same kernel code, this programming model follows the Single-Instruction Multiple-Data (SIMD) parallel programming paradigm [34].

CUDA defines a thread hierarchy that allows CUDA applications to transparently scale its parallelism to a widely range of GPUs with varying number of processors. All threads executing the same kernel are grouped into a *grid* of thread *blocks* (Figure 2–5). A *thread block* is a batch of threads that can cooperate together by sharing memory and synchronizing their execution. Threads in different blocks cannot cooperate with each other. A block can be defined as a one, two or three-dimensional array of threads. Therefore, threads can be identified using a one, two or three-dimensional thread index. Similarly, a *grid* can be defined as a one or two-dimensional array of thread blocks, which allows to identify the blocks using a one or two-dimensional block index. The combination of block and thread indices

defines a unique thread ID that allows the threads to distinguish themselves and provides a mechanism for indexing elements in data structures like vectors, matrices or volumes.

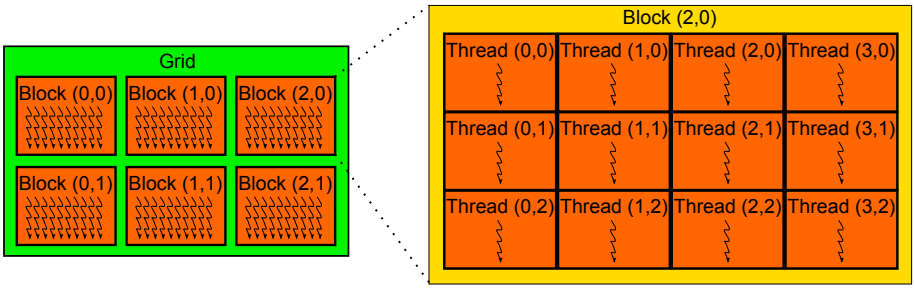


Figure 2–5: Structure of a 3x2 grid of 4x3 thread blocks.

The following code is a sample CUDA C code that adds two vectors $\boldsymbol{A}$ and $\boldsymbol{B}$ of size N and stores the result into vector $\boldsymbol{C}$:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < N)
    C[i] = A[i] + B[i];
}
int main() {
  ...
  // Allocate host memory
  h_A = (float*)malloc(size);
  h_B = (float*)malloc(size);
  h_C = (float*)malloc(size);
  // Initialize input vectors
  ...
  // Allocate device memory
  cudaMalloc((void*)&d_A, size);
  cudaMalloc((void*)&d_B, size);
  cudaMalloc((void*)&d_C, size);
  // Copy data from host to device
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
  // Kernel invocation with 256 threads per block
```

```
    int blocksPerGrid = (N + 255)/256;
    VecAdd<<<blocksPerGrid, 256>>>(d_A, d_B, d_C);
    // Copy back results from device to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    ...
}
```

This code contains some of the keywords that belong to the language extensions introduced by CUDA. For example, the keyword ***__global__*** in the definition of the kernel function indicates that this is a function that will run on the GPU and can be called from the host code only. The keywords ***blockDim***, ***blockIdx*** and ***threadIdx*** are built-in variables that contain the block dimensions, thread block and thread indices, respectively. These variables are used for obtaining the global thread ID that identifies the data element where the current thread is assigned to work on. The code of the main function shows some of the basic steps involved in a CUDA application. The GPU memory is allocated using the function ***cudaMalloc()***, which is defined in the CUDA runtime library. Then, the vectors A and B are copied from the host memory to the GPU memory using the function ***cudaMemcpy()***. The kernel that adds in parallel on the GPU the two vectors is launched using the syntax *<<<GridDimension, BlockDimension >>>*, which is known as the *execution configuration*. In this sample code, the kernel is configured to be launched by a one dimensional grid of 256-thread blocks. Once the kernel finish its execution, the resulting vector ***C*** is copied back from the GPU memory to the host memory using the function ***cudaMemcpy()***. Finally, the GPU memory is freed though the function ***cudaFree()***.

CUDA defines a memory hierarchy consisting of different logical memory spaces that threads can access during their execution:

- ***Global memory:*** All threads have access (read/write) to a global memory space that resides on the device DRAM (off-chip) (Figure 2–6).
- ***Local memory:*** Each thread has access (read/write) to a local memory space that resides on the device DRAM (off-chip) for private data storage.
- ***Shared memory:*** All threads within the same block have access (read/write) to a fast shared memory space that resides on the GPU chip (on-chip) (Figure 2–6).
- ***Registers:*** Each thread has access (read/write) to a limited set of 32-bit registers (on-chip) for storing automatic private variables.
- ***Constant memory:*** All threads have access to a read-only memory region that resides on the device DRAM (off-chip) for constant data storage.
- ***Texture memory:*** All threads have access to a read-only memory region that resides on the device DRAM (off-chip) and is optimized for access patterns with 2D spatial locality.

2.4.2 Hardware Model

From the Geforce™ 8 series onwards, including Quadro™ and Tesla™ families, all NVIDIA® GPUs support CUDA [12]. The different core architectures, also known as *compute capabilites*, are defined by a major revision number and a minor revision number that specifies incremental improvements to the architecture. *Fermi*, the latest CUDA architecture, corresponds to the compute capability **2.x**. All previous CUDA architectures were **1.x** (1.0, 1.1 and 1.3).

The CUDA architecture is built around a fully programmable processor array (cores) organized into ***streaming multiprocessors*** (SM) (Figure 2–6). On devices of compute capability 1.x, each multiprocessor consists of [12]:

- 8 CUDA cores or scalar processors (SP) for integer and single-precision floating-point arithmetic operations.

- 1 double-precision (DP) floating-point unit for double-precision arithmetic.
- 2 special function units (SFU) for transcendentals (sine, cosine, etc.).
- 1 warp scheduler for managing thread concurrent execution.
- A set of 32-bit registers (8,192 registers on devices 1.0 and 1.1, 16,384 registers on devices 1.2 and 1.3).
- 16 KB of shared memory.
- A read-only cache for speeding up reads from constant memory.
- A read-only cache for speeding up reads from texture memory.

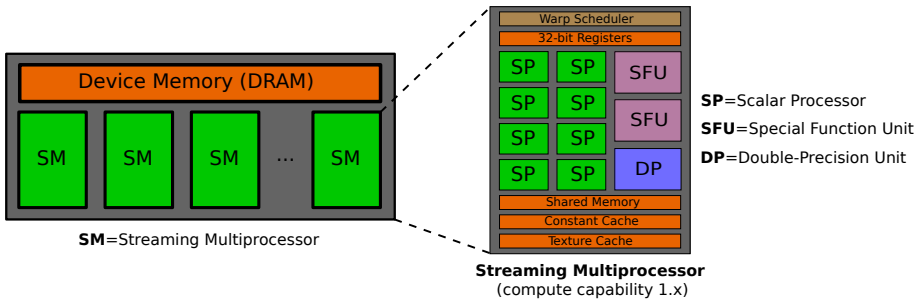


Figure 2–6: CUDA hardware model: array of streaming multiprocessors.

On devices of compute capability 2.x, each multiprocessor consists of [12]:

- 32 CUDA cores on devices 2.0, 48 CUDA cores on devices 2.1.
- 4 SFUs on devices 2.0, 8 SFUs on devices 2.1.
- 2 warp schedulers for managing thread concurrent execution.
- A set of 32,768 32-bit registers.
- 64 KB of L1 cache memory configurable as shared memory.
- A read-only cache for speeding up reads from constant memory.
- A read-only cache for speeding up reads from texture memory.

When a kernel is launched, the thread blocks of the grid are scheduled and distributed to the multiprocessors for execution. Each multiprocessor can manage concurrently a maximum of 8 thread blocks. The thread blocks can be scheduled

in any order, concurrently or sequentially, and, as thread blocks terminate their execution, new blocks are launched on the available multiprocessors. Figure 2–7 shows the execution of a kernel consisting of a 3×2 grid of thread blocks on a device with 3 multiprocessors. In this case, two blocks are assigned to each multiprocessor. If there are enough resources (registers and shared memory), the two blocks will be scheduled to be run concurrently on the multiprocessor. If the available resources are not enough to run at least one block per multiprocessor, the kernel execution will fail.

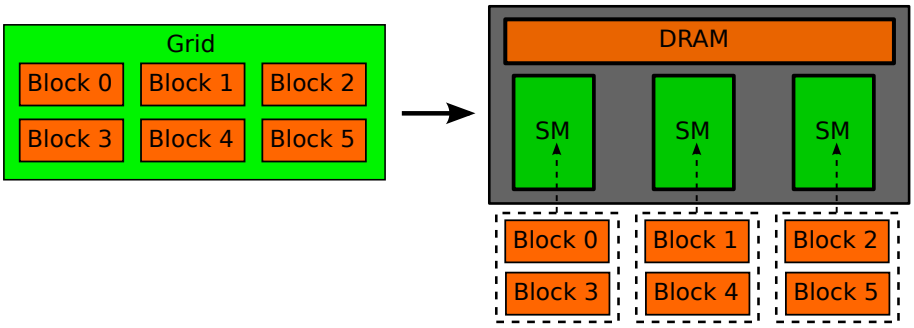


Figure 2–7: Distribution of a 3×2 grid of thread blocks for execution on a device with 2 SMs.

Thread concurrency is managed by the multiprocessors using an architecture called *Single-Instruction Multiple-Thread* (SIMT) [12]. Every thread block is partitioned into groups of 32 threads called *warps* (Figure 2–8). The warps are scheduled and executed independently on the multiprocessor. A *warp scheduler* is responsible for selecting one warp that is ready to execute and issuing the next instruction to all the threads belonging to that warp. All threads within a warp start at the same program address but they are free to follow different execution paths (*branch divergence*). In that case, the different branches are executed serially until all threads within the warp converge to the same path. Since a warp execute one common instruction at a time, maximum performance will be achieved when all 32 threads follow the same execution path.

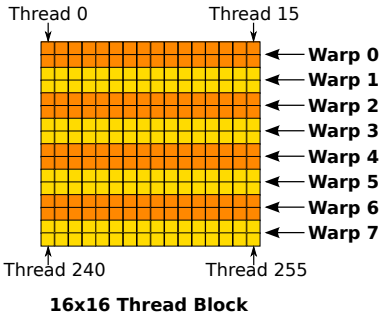


Figure 2–8: A 16x16 thread block partitioned into 8 warps.

2.5 Hyperspectral Data Processing using GPUs

Processing of hyperspectral data requires to deal with large volumes of data in the form of an image cube made up of pixel vectors, which can make this task very time consuming. Fortunately, many algorithms used for hyperspectral data exploitation exhibit an inherent parallelism at multiple levels: pixel-level, spectral level and task-level parallelism [35].

Parallel processing of hyperspectral data has been studied in the works of Plaza *et al.* [7, 36, 37]. In [7], parallel implementations of several HSI algorithms for classification, automatic target detection and spectral unmixing were proposed. Spatial-domain parallelism was used in these implementations by partitioning the image into blocks made up of spatially adjacent pixels and assigning one or more blocks to each processing unit. Plaza *et al.* have recently proposed several GPU-based implementations of different algorithms for hyperspectral data exploitation. They proposed a GPU-based implementation of the Pixel Purity Index (PPI) and the Automated Morphological Endmember Extraction (AMEE) algorithms in [38] and [35], respectively. They also proposed GPU-based implementations of unconstrained, partially constrained and fully constrained abundance estimation algorithms in [10]. In the unconstrained implementation, every pixel vector is multiplied in parallel by the projection matrix $(\mathbf{E}^T\mathbf{E})^{-1}\mathbf{E}^T$. The partially and fully constrained implementations were based on the Image Space Reconstruction Algorithm (ISRA) [39], an iterative

algorithm for solving non-negative least squares problems. The proposed implementation of the fully constrained algorithm enforces the sum-to-one constrain by scaling the abundances obtained by the ISRA algorithm. The GPU implementation of the ISRA algorithm was also previously studied by González *et al.* in [9].

Plaza *et al.* proposed two algorithms for target detection in HSI and their corresponding GPU implementations in [11]. The first algorithm, called ATDCA (automatic target detection and classification algorithm), is based on the OSP approach [33] and works as follows:

1. It selects the maximum length pixel vector $\mathbf{x}_0$ as the initial target signature.
2. It applies an orthogonal subspace projector $\mathbf{P}_U = \mathbf{I} - \mathbf{U}(\mathbf{U}^T\mathbf{U})^{-1}\mathbf{U}^T$ to all image pixels (in parallel on the GPU), where $\mathbf{U} = [\mathbf{x}_0]$. The pixel with maximum projection in the orthogonal complement space linearly spanned by $\mathbf{x}_0$ is selected as the next target signature $\mathbf{x}_1$.
3. Another orthogonal subspace projector $\mathbf{P}_U$ with $\mathbf{U} = [\mathbf{x}_0\ \mathbf{x}_1]$ is applied (in parallel on the GPU) to the original image and the pixel with maximum projection is selected as the next target signature.
4. The process is repeated until a set of targets $\{\mathbf{x_0}, \mathbf{x_1}, ..., \mathbf{x_t}\}$ has been extracted.

The second algorithm is a GPU-based implementation of the RX algorithm (Equation 2.4). 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 [40].

Another GPU-based implementation of a target detection algorithm for real-time anomaly detection in HSI has recently been proposed by Tarabalka *et al.* [41]. The proposed anomaly detection algorithm is based on a multivariate normal mixture model of the background. This model is represented by the probability density function:

$$p(\mathbf{x}) = \sum_{c=1}^{C} \omega_c \phi_c(\mathbf{x}; \mu_\mathbf{c}, \mathbf{\Sigma_c}) \tag{2.15}$$

where C is the number of background spectral classes, ω_c is the mixing portion of class c and $\phi_c(\mathbf{x}; \mu_\mathbf{c}, \mathbf{\Sigma_c})$ is the multivariate normal density function with mean μ and covariance $\mathbf{\Sigma}$. The basic steps of this algorithm are:

1. Estimation of the background parameters model by fitting a multivariate normal mixture model to a spatial subset of the image.
2. Calculation of the pixel probability map based on the estimated model.
3. Detection of anomaly pixels by thresholding low probability values.
4. Image segmentation by merging detected pixels to objects.

The most time-consuming parts of this algorithm are computed on the GPU using a pixel-level parallelism approach. Tarabalka *et al.* proposed two approaches for computing in parallel the covariance matrix of the multivariate normal model. The first approach is called the chunking approach and consists of splitting the image into spatial subsets (chunks) and calculating the covariance sums for all the parts in parallel. The total covariance sum is then calculated by summing in parallel the partial covariance sums. In the second approach, each thread computes a different element of the whole covariance matrix. Therefore, this approach uses spectral-level parallelism and is only interesting when the number of bands is significant.

The results of these previous works suggest that GPUs show promise as a computing device for accelerating hyperspectral data processing tasks like automatic and anomaly target detection. In the work presented in this document, other target detection algorithms, like adaptive RX, matched filter and sub-pixel detectors, were studied for GPU implementation. The structure of the algorithms was analyzed in order to design GPU implementations optimized for the CUDA architecture. In addition, a CUDA library of target detectors was implemented as a result of this work.

Chapter 3

GPU-based Implementation

The previous chapter described different target detection algorithms commonly used in HSI and the architecture of GPUs as massively parallel processors. It also presented previous works on using GPUs as processing hardware to speed up target detectors and other algorithms for hyperspectral image exploitation. This chapter discuses the parallel implementation of three target detection algorithms for both full-pixel (RX algorithm, matched filter) and sub-pixel detection (AMSD), and proposes GPU-based implementations of these algorithms to take advantage of the NVIDIA® CUDA™ architecture.

The detection algorithms were implemented 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 main specifications of the C1060 card are listed in Table 3–1.

The CUBLAS™ [42] and CULA™ [43] libraries were used to perform the GPU-based implementations of the algorithms. CUBLAS™ is a CUDA implementation of the BLAS (Basic Linear Algebra Subprograms) library and is provided by NVIDIA®. CULA[1] is a commercial linear algebra library, developed by EM Photonics®, that

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

provides a subset of the LAPACK[2] (Linear Algebra Package) library functions optimized for the CUDA architecture.

Table 3–1: Specifications of the Tesla C1060 graphics card.

# of Streaming Processor Cores	240
Frequency of Processor Cores	1.3 GHz
Single Precision Floating Point Peak Performance	933
Double Precision Floating Point Peak Performance	78
Total Dedicated Memory	4 GB DDR3
Memory Speed	800 MHz
Memory Interface	512 bit
Memory Bandwidth	102 GB/sec

3.1 Computation Decomposition

All the target detection algorithms described in the previous chapter have a common general structure that shows an inherent parallelism. As shown in Figure 3–1, the detection algorithms map the spectrum $\mathbf{x}_i$ of each pixel into a scalar value y_i that is compared to a threshold to decide if the target is present or not. The output value y_i depends on the spectrum $\mathbf{x}_i$ of the test pixel but does not depend on the other pixel spectra. Therefore, the output values of the detectors can be calculated independently for each pixel of the image. If there are N pixels, the computation of the detection output for the entire image can be decomposed into N parallel tasks without communication between each other. This algorithm structure is known as an embarrassingly parallel problem [34].

In a multiprocessor system, each parallel task can be assigned to a different processor (*worker*), so each processor will be responsible for computing the detection

[2] http://www.netlib.org/lapack/

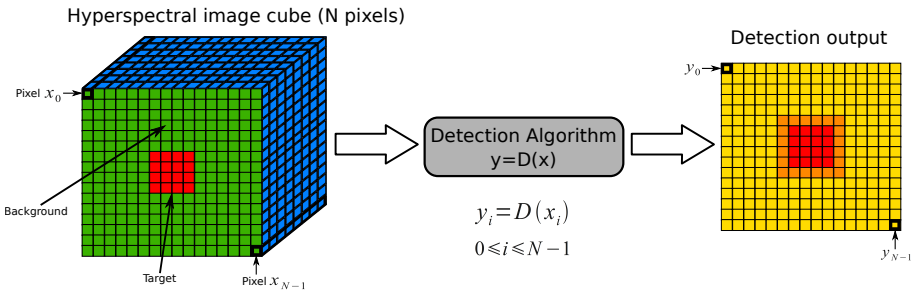


Figure 3–1: Block diagram of a target detection algorithm showing pixel-level parallelism.

output of a different pixel. This is illustrated in Algorithm 1, where the construct ***for all*** ... ***do*** represents a parallel for-loop.

Algorithm 1 Parallel implementation of a hyperspectral target detector

Input: A hyperspectral image $\{\mathbf{x}_i\}$, $1 \leq i \leq N$
Output: Detection output $\{y_i\}$, $1 \leq i \leq N$, where $y_i = D(\mathbf{x}_i)$
  for all $i = 1$ to N **do**
    compute $y_i = D(\mathbf{x}_i)$
  end for

The total amount of parallelism is equal to the number of pixels in the image, N. In a GPU-based implementation, a kernel function is defined containing the code to be executed in parallel by N GPU threads. The thread 0 will be responsible for computing the output of the detector for the pixel $\mathbf{x}_0$, the thread 1 will be responsible for computing the output of the detector for the pixel $\mathbf{x}_1$, and so on. A sample code of a kernel is shown bellow:

```
// Kernel function that implements a target detector y = D(x)
__global__ void ParallelDetector(float* image, float* detect_output, int N, int
    bands, ...)
{
  ...
  float pixel[N];
  if (threadID < N)
  //read pixel value
  for (int i=0;i<bands;i++)
    pixel[i] = image[bands*i+threadID];
  //compute output value and write it to global output matrix
```

```
    detect_output[threadID] = D(pixel);
}
```

Each thread accesses the corresponding pixel data through a pointer (***float**** *image*) to the image data stored in the GPU memory space and save the pixel data into a local variable (*pixel*). Based on the pixel data, the thread computes the detection output value and writes it in an output matrix stored in global memory. The thread ID is used to determine the correct array indices for reading and writing from global memory.

Figure 3–2 shows the structure of thread blocks generated to execute the code of the kernel function. For a given block size, the dimension of the grid is selected to cover the entire hyperspectral image. If the number of samples is not evenly divided by the x block dimension, the smallest multiple of the x block dimension that is greater than the number of samples is selected as the x grid dimension. The same procedure is applied for selecting the y grid dimension if the number of lines is not evenly divided by the y block dimension. The block dimension used in the implementations was 32×16 (512 threads), value that resulted in the best performance (see Chapter 4).

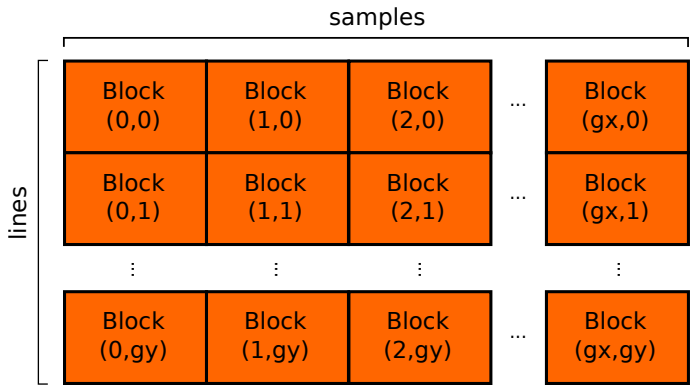


Figure 3–2: Structure of a grid of thread blocks generated to run the code of a kernel function implementing a target detection algorithm.

The next section discusses the details related to memory access on a GPU and analyzes different data storage schemes to maximize GPU memory throughput.

3.2 Data Layout

When accessing GPU global memory, simultaneous memory accesses can be coalesced into one or more memory transactions depending on the size of the data accessed by each thread and the access pattern. The requirements for achieving coalesced memory transaction depend on the device compute capability and are explained in more detail in [12].

As an example, if all the threads of a warp in a device of compute capability 1.3 access consecutive 32-bit words lying in the same 128-byte memory segment, only two 64-byte memory transaction are issued: one for the first 16 threads (*half-warp*) and one for the remaining 16 threads (Figure 3–3). Additionally, the memory segments must be aligned, i.e., the starting address must be a multiple of its size. If the access pattern is not aligned with a memory segment, a memory transaction is issued for every segment accessed by the half-warp (Figure 3–4), thus, reducing the memory throughput. However, the size of the additional transactions can be reduced if only the upper or lower half of a segment is used.

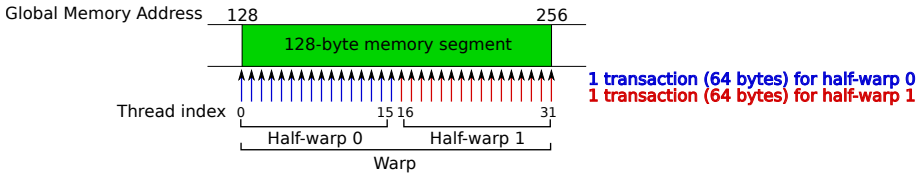


Figure 3–3: Coalesced memory transactions for a sequential aligned access pattern.

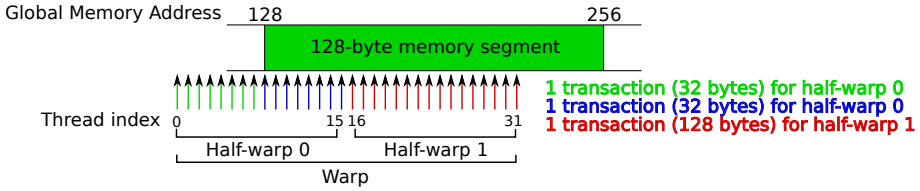


Figure 3–4: Misaligned sequential access patterns produce additional memory transactions.

The global memory throughput is, therefore, determined by how the threads access the data elements and how the global data is stored in memory. There are three storing schemes commonly used in hyperspectral images: *band interleaved by pixel* (BIP), *band interleaved by line* (BIL) and *band sequential* (BSQ). In the BIP scheme, each line of the image is stored sequentially starting from the first pixel for all bands, followed by the second pixel for all bands, etc. Since each thread is working on a different pixel, the BIP scheme do not lead to coalesced memory transactions when reading a value for a single band as shown in Figure 3–5. In the BIL scheme, each line is stored sequentially starting from the first band for all pixels of the first line, followed by the second band for all pixels of the first line, etc. The BIL scheme can lead to coalesced memory transactions as long as all the threads of the same half-warp access the same line, as illustrated in Figure 3–6. Finally, in the BSQ scheme, all the pixels of the image for the band 1 are stored first, followed by all the pixels of the image for band 2, etc. This storage scheme allows coalesced memory transactions as long as every line is aligned to a memory segment (Figure 3–7).

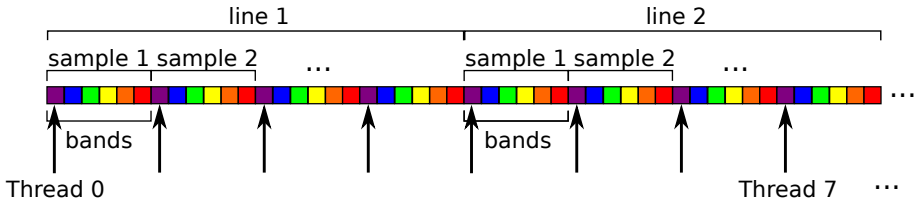


Figure 3–5: Uncoalesced memory transactions when reading band 1 in BIP scheme.

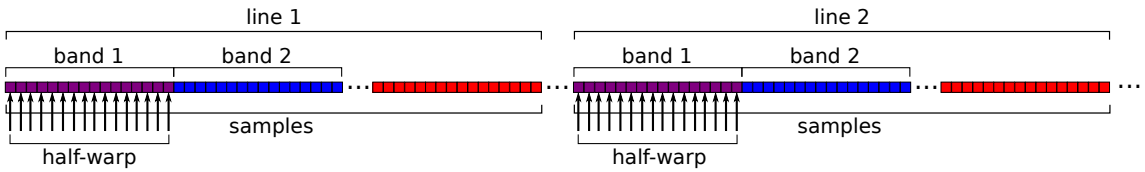


Figure 3–6: Coalesced memory transactions when all threads of a warp access the same line in BIL scheme.

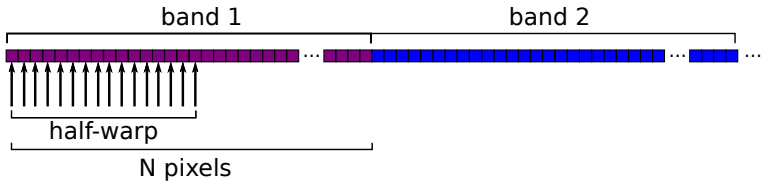


Figure 3–7: Coalesced memory transactions when reading band 1 in BSQ scheme.

In the GPU-based implementations described in this thesis, the BSQ storage scheme was used since it reduces the pointer arithmetic for indexing data elements and the alignment conditions for coalesced accesses can be automatically satisfied by using the run-time CUDA function *cudaMallocPitch* for allocating GPU global memory.

3.3 Implementation of RX and MF detectors

This section describes the GPU-based parallel implementation of the RX algorithm (Equation 2.4) and the matched filter (Equation 2.3) using global statistics. The pseudo-codes of these algorithms are shown bellow (Algorithms 2 and 3, respectively):

Algorithm 2 RX Algorithm for Anomaly Detection

Input: Background mean $\boldsymbol{\mu}_0$, background covariance matrix $\boldsymbol{\Gamma}_0$, hyperspectral image $\mathbf{X} = [\mathbf{x}_1 \ ... \ \mathbf{x}_N]$
Output: Detection output $Y_{RX} = \left[y_1^{RX} \ ... \ y_N^{RX}\right]$
  for $i = 1$ to N **do**
    compute $y_i^{RX} = (\mathbf{x}_i - \boldsymbol{\mu}_0)^T \boldsymbol{\Gamma}_0^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_0)$
  end for

Algorithm 3 Matched Filter Detector

Input: Background mean $\boldsymbol{\mu}_0$, target mean $\boldsymbol{\mu}_1$, background covariance matrix $\boldsymbol{\Gamma}_0$, hyperspectral image $\mathbf{X} = [\mathbf{x}_1 \ ... \ \mathbf{x}_N]$
Output: Detection output $Y_{MF} = \left[y_1^{MF} \ ... \ y_N^{MF}\right]$
  compute $\kappa^{-1} = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^T \boldsymbol{\Gamma}_0^{-1} (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)$
  for $i = 1$ to N **do**
    compute $y_i^{MF} = \kappa (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^T \boldsymbol{\Gamma}_0^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_0)$
  end for

Both algorithms take as inputs a hyperspectral image $\mathbf{X}$ consisting of N pixels, the mean $\boldsymbol{\mu}_0$, and the covariance matrix $\boldsymbol{\Gamma}_0$ of the background distribution. The matched filter also needs the mean of the target signature $\boldsymbol{\mu}_1$. These statistical parameters are estimated previously using training samples from the data. In the proposed implementation, this step is considered a preprocessing step performed on the CPU.

Each iteration of the *for* loops in Algorithms 2 and 3 computes the output of the detectors for a different pixel. The two algorithms have in common the computation of the inverse of the covariance matrix $\boldsymbol{\Gamma}_0^{-1}$. Since a covariance matrix is symmetric and positive definite, instead of computing the inverse directly, the proposed implementation computes the Cholesky decomposition of $\boldsymbol{\Gamma}_0$:

$$\boldsymbol{\Gamma}_0 = \mathbf{L}\mathbf{L}^T \tag{3.1}$$

Expressing the output of the RX detector for the pixel i in terms of the lower triangular matrix $\mathbf{L}$, we get:

$$\begin{aligned} &y_i^{RX} = (\mathbf{x}_i - \boldsymbol{\mu}_0)^T \boldsymbol{\Gamma}_0^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_0) = (\mathbf{x}_i - \boldsymbol{\mu}_0)^T (\mathbf{L}\mathbf{L}^T)^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_0) = \\ &(\mathbf{L}^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_0))^T (\mathbf{L}^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_0)) = \mathbf{b}_i^T \mathbf{b}_i \end{aligned} \tag{3.2}$$

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 $\times$ bands) of the covariance matrix does not allows enough amount of parallelism to take advantage of the CUDA architecture. The resulting upper triangular matrix $\mathbf{L}$ is transfered 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 stored in the shared memory at every iteration of the forward substitution loop. The thread 0 of each block is responsible for transferring the corresponding row of $\mathbf{L}$ to the shared memory to be shared by all the threads of the block. For example, to compute the component j of the vector $\mathbf{b}_i$, the thread 0 of each block transfers the values $L_{j1}, L_{j2},\ \ldots\ , L_{jj}$ from global memory to the shared memory. Then, each thread i belonging to the same block computes its corresponding value $b_{ij} = \frac{1}{L_{jj}} \sum_{k=1}^{j-1} L_{jk} b_{ik}$. Since the values $L_{j1}, L_{j2},\ \ldots\ , L_{jj}$ are shared by all threads of the same block, a synchronization barrier is needed to prevent a thread from reading some value before the thread 0 has finished the transfer to the shared memory. This is accomplished through the CUDA function *__syncthreads*.

The pseudo-code of the GPU-based parallel implementation of the RX detector is shown in Algorithm 4. Each iteration i of the parallel *for all ... do* loop is assigned to the thread i. The steps performed by each thread to compute its corresponding output value are:

- Remove mean from pixel: $\tilde{\mathbf{x}}_i = \mathbf{x}_i - \boldsymbol{\mu}_0$
- Solve a triangular system: $\mathbf{L}\mathbf{b}_i = \tilde{\mathbf{x}}_i$
- Compute the output value: $y_i^{RX} = \mathbf{b}_i^T \mathbf{b}_i$

Following a similar procedure for the MF detector, we get:

$$y_i^{MF} = \kappa(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^T \boldsymbol{\Gamma}_0^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_0) = \kappa(\boldsymbol{\Gamma}_0^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0))^T(\mathbf{x}_i - \boldsymbol{\mu}_0) = \kappa \mathbf{c}^T(\mathbf{x}_i - \boldsymbol{\mu}_0) = \kappa \mathbf{c}^T \tilde{\mathbf{x}}_i \tag{3.3}$$

Algorithm 4 Parallel RX Algorithm

Input: Background mean $\boldsymbol{\mu}_0$, lower triangular matrix $\mathbf{L}$, hyperspectral image $\mathbf{X} = [\mathbf{x}_1 \ ... \ \mathbf{x}_N]$

Output: Detection output $Y_{RX} = \left[y_1^{RX} \ ... \ y_N^{RX}\right]$

```
for all i = 1 to N do
    compute x̃_i = x_i − μ_0
    solve system L b_i = x̃_i
    compute y_i^RX = b_i^T b_i
end for
```

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 forwards 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:

$$\kappa^{-1} = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^T \boldsymbol{\Gamma}_0^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0) = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^T \mathbf{c} \tag{3.4}$$

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

The pseudo-code of the GPU-based parallel implementation of the MF detector is shown in Algorithm 5, where each iteration i of the parallel *for all ... do* loop is assigned to the thread i. The steps performed by each thread to compute its corresponding output value are:

- Remove mean from pixel: $\tilde{\mathbf{x}}_i = \mathbf{x}_i - \boldsymbol{\mu}_0$
- Compute the output value: $y_i^{MF} = \kappa \mathbf{c}^T \tilde{\mathbf{x}}_i$

The different processing steps involved in both the RX and MF detectors are summarized in Figure 3–8.

Algorithm 5 Parallel Matched Filter

Input: Background mean $\boldsymbol{\mu}_0$, normalization constant κ, vector $\mathbf{c}$, hyperspectral image $\mathbf{X} = [\mathbf{x}_1 \ ... \ \mathbf{x}_N]$

Output: Detection output $Y_{MF} = \left[y_1^{MF} \ ... \ y_N^{MF}\right]$

for all $i = 1$ to N **do**

compute $\tilde{\mathbf{x}}_i = \mathbf{x}_i - \boldsymbol{\mu}_0$

compute $y_i^{MF} = \kappa \mathbf{c}^T \tilde{\mathbf{x}}_i$

end for

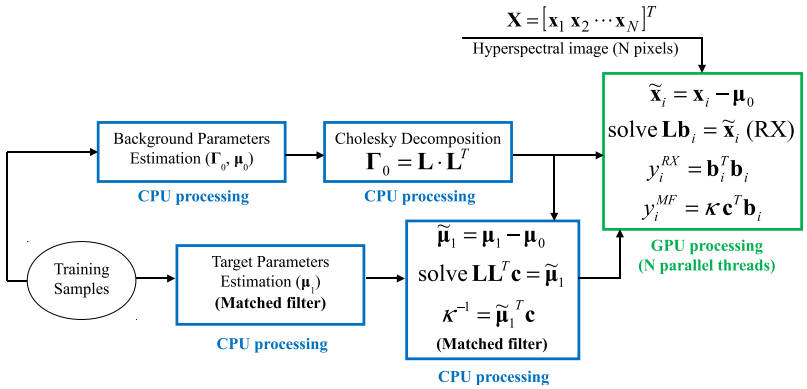


Figure 3–8: Parallel implementation of RX and MF detectors.

3.4 Implementation of an adaptive RX algorithm

This section describes the GPU-based parallel implementation of an adaptive version of the RX algorithm that locally computes the background statistics (mean and covariance) using a 2D sliding window approach [44]. In this approach, the mean and covariance matrix are estimated using the samples from a region between two windows centered at the test pixel, as shown in Figure 3–9. The size of the guard window is selected to enclose the largest target present in the scene, and the pixels contained in this window are excluded to avoid bias in the estimates due to the possible presence of target pixels around the test pixel. The size of the outer window should be small enough to cover an homogeneous background region but large enough to ensure an accurate statistical estimation.

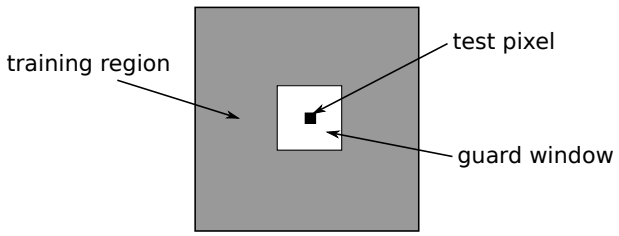


Figure 3–9: Structure of the 2D spatially moving window for background parameter estimation in the adaptive RX algorithm.

In the proposed implementation, the detector is not applied to the regions where the moving window goes out of the image boundaries, approach that sacrifices the detection of possible targets present near the image borders. Other approaches like zero padding or pixel replication were evaluated but they were not finally used due to stability problems related to the invertibility of the covariance matrix.

The general pseudo-code of this algorithm is shown in Algorithm 6, where $R_i^{W,G}$ denotes the set of samples between the two windows of sizes W and G, respectively, and centered at pixel i.

Algorithm 6 RX Algorithm using local statistics

Input: A hyperspectral image $\mathbf{X} = [\mathbf{x}_1 \ ... \ \mathbf{x}_N]$, sliding window size W, guard window size G

Output: Detection output $Y_{RX} = \left[y_1^{RX} \ ... \ y_N^{RX}\right]$

for $i = 1$ to N **do**

estimate $\boldsymbol{\mu}_{0i}$, $\boldsymbol{\Gamma}_{0i}$ from $R_i^{W,G}$

compute $y_i^{RX} = (\mathbf{x}_i - \boldsymbol{\mu}_{0i})^T \boldsymbol{\Gamma}_{0i}^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_{0i})$

end for

This algorithm takes as inputs a hyperspectral image $\mathbf{X}$ consisting of N pixels, the size W of the sliding window, and the size G of the guard window. In this algorithm, both the mean and covariance matrix have to be computed for each pixel using the samples from the region $R_i^{W,G}$. Since this region depends on the pixel location, in this case, the mean $\boldsymbol{\mu}_{0i}$ and covariance $\boldsymbol{\Gamma}_{0i}$ are different for each pixel and cannot be shared. Once the mean and covariance are estimated, the output value

of the detector is computed as $y_i^{RX} = (\mathbf{x}_i - \boldsymbol{\mu}_{0i})^T \boldsymbol{\Gamma}_{0i}^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_{0i})$. This computation is decomposed using the same procedure as in the global implementation:

1. Compute $\boldsymbol{\Gamma}_{0i} = \mathbf{L}_i \mathbf{L}_i^T$
2. Solve system $\mathbf{L}_i \mathbf{b}_i = \mathbf{x}_i - \boldsymbol{\mu}_{0i}$
3. compute $y_i^{RX} = \mathbf{b}_i^T \mathbf{b}_i$

In this implementation, the Cholesky decomposition of $\boldsymbol{\Gamma}_{0i}$ is performed on the GPU by the thread i. 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. Each thread is responsible for computing the corresponding mean $\boldsymbol{\mu}_{0i}$ and covariance $\boldsymbol{\Gamma}_{0i}$ using the pixel values from the neighborhood $R_i^{W,G}$ defined by the moving window. Since the region $R_i^{W,G}$ is different for each thread, the pixel values needed for computing the mean and covariance cannot be stored in the shared memory space. Therefore, they must be read from global memory, which reduces the memory throughput of this algorithm. 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.

The pseudo-code of the GPU-based parallel implementation of the adaptive RX detector is shown in Algorithm 7.

Due to the local structure of this detector, the matrix $\mathbf{L}_i$ has to be stored in the local memory space and cannot be shared among all the threads of the same block as in the global RX implementation. This implementation cannot take advantage of the shared memory cache when solving the triangular system $\mathbf{L}_i \mathbf{b}_i = \tilde{\mathbf{x}}_i$ by forward substitution. Since the local memory has the same low bandwidth as the global

Algorithm 7 Parallel adaptive RX Algorithm

Input: A hyperspectral image $\mathbf{X} = [\mathbf{x}_1 \ ... \ \mathbf{x}_N]$, sliding window size W, guard window size G

Output: Detection output $Y_{RX} = \left[y_1^{RX} \ ... \ y_N^{RX}\right]$

for all $i = 1$ to N **do**
  estimate $\boldsymbol{\mu}_{0i}$, $\boldsymbol{\Gamma}_{0i}$ from $R_i^{W,G}$
  compute $\boldsymbol{\Gamma}_{0i} = \mathbf{L}_i\mathbf{L}_i^T$
  compute $\tilde{\mathbf{x}}_i = \mathbf{x}_i - \boldsymbol{\mu}_{0i}$
  solve system $\mathbf{L}_i\mathbf{b}_i = \tilde{\mathbf{x}}_i$
  compute $y_i^{RX} = \mathbf{b}_i^T\mathbf{b}_i$
end for

memory, the performance of this algorithm is limited by the local memory latency as will be shown in Chapter 4, Section 4.2.3. The different processing steps involved in the adaptive RX detector are summarized in Figure 3–10.

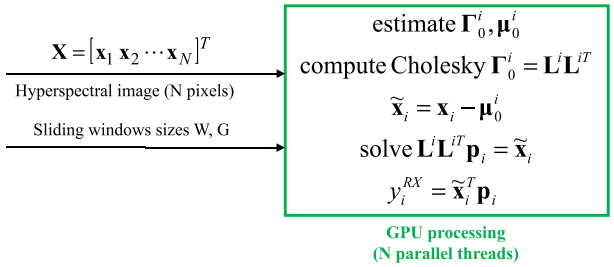


Figure 3–10: Parallel implementation of the adaptive RX algorithm.

3.5 GPU-based Implementation of the Adaptive Matched Subspace Detector (AMSD)

This section describes the GPU-based parallel implementation of the adaptive matched subspace detector (AMSD) (Equation 2.9). The pseudo-code of this algorithm is shown in Algorithm 8.

This algorithm takes as inputs a hyperspectral image $\mathbf{X}$ consisting of N pixels, a matrix $\mathbf{B}$ whose columns span the background subspace, and a matrix $\mathbf{S}$ whose columns span the target subspace. The estimation of the dimensionality of both

Algorithm 8 Adaptive Matched Subspace Detector

Input: A hyperspectral image $\mathbf{X} = [\mathbf{x}_1 \ ... \ \mathbf{x}_N]$, background subspace matrix $\mathbf{B}$, target subspace matrix $\mathbf{S}$

Output: Detection output $Y_{AMSD} = \left[y_1^{AMSD} \ ... \ y_N^{AMSD}\right]$

compute $\mathbf{P}_{\mathbf{B}}^{\perp} = \mathbf{B}(\mathbf{B}^T\mathbf{B})^{-1}\mathbf{B}^T$

compute $\mathbf{P}_{\mathbf{E}}^{\perp} = \mathbf{E}(\mathbf{E}^T\mathbf{E})^{-1}\mathbf{E}^T$, $\mathbf{E} = [\mathbf{S}\ \mathbf{B}]$

for $i = 1$ to N **do**

compute $y_i^{AMSD} = \frac{\mathbf{x}_i^T(\mathbf{P}_{\mathbf{B}}^{\perp} - \mathbf{P}_{\mathbf{E}}^{\perp})\mathbf{x}_i}{\mathbf{x}_i^T\mathbf{P}_{\mathbf{E}}^{\perp}\mathbf{x}_i}$

end for

subspaces and the selection of the basis vectors are preprocessing steps performed using global training samples from the data. In the implementation of this detector, two methods for estimating the matrix $\mathbf{B}$ of background basis vectors were evaluated: singular value decomposition (SVD), and Maximum Distance (MaxD) [27]. 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* $\times$ *pixels* format. In MaxD, the basis vectors selected, which are pixels from the original image, are the set of vectors that try to approximate a simplex defining the background subspace. The steps involved in the MaxD method are 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 process for estimating the background matrix $\mathbf{B}$ can be performed on the GPU depending on the size of the input image $\mathbf{X}$. A GPU-based implementation of the two methods for estimating $\mathbf{B}$ has been evaluated in this work. In the SVD approach, the left singular vectors are computed as the eigenvectors of the image

correlation matrix, which is faster than computing directly the singular value decomposition of $\mathbf{X}$. The correlation matrix $\mathbf{R} = \mathbf{X}\mathbf{X}^T$ is computed on the GPU using the function *SGEMM* for matrix-matrix multiplication from the CULA™ library. The computation of the eigenvectors of $\mathbf{R}$ is performed on the GPU using the function *SSYEV* also from the CULA™ library.

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 CUDA kernel function that computes in parallel the magnitude of each pixel. Then, the largest and smallest pixels are selected using the functions *ISAMAX*, *ISAMIN* from the CUBLAS™ library, respectively. The projection step is performed on the GPU using the the function *SGEMM* from CULA™ and the update of the projection matrix at each iteration is performed through the function *SGER* from CUBLAS™.

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 $\mathbf{P}_{\mathbf{E}}^{\perp} = \mathbf{P}_{\mathbf{B}}^{\perp}\mathbf{P}_{\mathbf{Z}}^{\perp}\mathbf{P}_{\mathbf{B}}^{\perp}$ and $\mathbf{P}_{\mathbf{B}}^{\perp} - \mathbf{P}_{\mathbf{E}}^{\perp} = \mathbf{P}_{\mathbf{B}}^{\perp}\mathbf{P}_{\mathbf{Z}}\mathbf{P}_{\mathbf{B}}^{\perp}$, where $\mathbf{Z} = \mathbf{P}_{\mathbf{B}}^{\perp}\mathbf{S}$, i.e., the part of the target subspace orthogonal to the background subspace. With these equivalences, the output value of the AMSD can be computed as:

$$y_i^{AMSD} = \frac{\left\|\mathbf{P}_{\mathbf{Z}}\mathbf{P}_{\mathbf{B}}^{\perp}\mathbf{x}_i\right\|^2}{\left\|\mathbf{P}_{\mathbf{Z}}^{\perp}\mathbf{P}_{\mathbf{B}}^{\perp}\mathbf{x}_i\right\|^2} \tag{3.5}$$

The matrices $\mathbf{P}_n = \mathbf{P}_{\mathbf{Z}}\mathbf{P}_{\mathbf{B}}^{\perp}$ and $\mathbf{P}_d = \mathbf{P}_{\mathbf{Z}}^{\perp}\mathbf{P}_{\mathbf{B}}^{\perp}$ are computed on the GPU using the function *SGEMM* from CULA™ and stored in the global memory space to be shared by all GPU threads. The pseudo-code of the GPU-based parallel implementation of the AMSD is shown in Algorithm 9.

In the parallel implementation, each GPU thread is responsible for computing the numerator n_i, denominator d_i, and the detection output y_i^{AMSD} of the AMSD for a given pixel $\mathbf{x}_i$. Since the matrices $\mathbf{P}_n$, $\mathbf{P}_d$ are shared by all threads, we can

Algorithm 9 Parallel Adaptive Matched Subspace Detector

Input: A hyperspectral image $\mathbf{X} = [\mathbf{x}_1 \ ... \ \mathbf{x}_N]$, projection matrices $\mathbf{P}_n$, $\mathbf{P}_d$
Output: Detection output $Y_{AMSD} = \left[y_1^{AMSD} \ ... \ y_N^{AMSD}\right]$

```
for all i = 1 to N do
    compute p_i = P_n x_i
    compute n_i = p_i^T p_i
    compute q_i = P_d x_i
    compute d_i = q_i^T q_i
    compute y_i^AMSD = n_i / d_i
end for
```

take advantage of the shared memory space to perform the products $\mathbf{p}_i = \mathbf{P}_n\mathbf{x}_i$, $\mathbf{q}_i = \mathbf{P}_d\mathbf{x}_i$. Using a similar approach as in the global full-pixel detectors, to compute the component j of the vector $\mathbf{p}_i$, the thread 0 of each block transfers the row j of $\mathbf{P}_n$ from global memory to the shared memory. The same approach is used to compute the component j of the vector $\mathbf{q}_i$. The processing steps involved in the AMSD implementation are summarized in Figure 3–11.

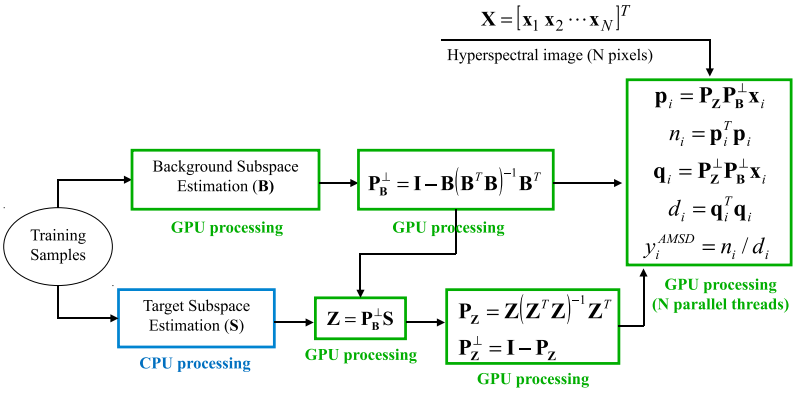


Figure 3–11: Parallel implementation of AMSD algorithm.

3.6 Summary

This chapter described the GPU-based parallel implementation of three target detection algorithms for both full-pixel (RX algorithm, matched filter) and sub-pixel detection (AMSD). The steps followed to design the parallel implementation of the algorithms can be summarized as follows:

- **Computation decomposition**. Identify the most time consuming parts of the algorithms and analyze how can these computations be decomposed into parallel tasks. The particular structure of the target detection algorithms studied supports a pixel-based parallel processing. The tasks associated to the computation of the detection output for each pixel is assigned to a different processing element, i.e., a GPU thread in the CUDA architecture.
- **Data layout**. Select data format that best meet the requirements of the parallel hardware architecture. In the CUDA architecture, the data formats for hyperspectral images that yield the best performance are BSQ and BIL. These two schemes allow coalesced memory transaction from GPU global memory.
- **Mapping the computation to the memory hierarchy**. Identify the memory spaces that are more suitable in terms of memory bandwidth to store the different data elements of the algorithm. Specifically, parameters, like the mean and covariance matrices, that remain constant during the entire computation can take advantage of cache memories to reduce memory latencies. In the CUDA architecture, parameters of small sizes, like the mean of the target and background, can be stored in the constant memory space. The covariance and projection matrices, although they remain constant, cannot be store in the constant memory space of the GPU due to their size. In this case, portions of the matrices can be temporarily stored in the shared memory space to improve the memory bandwidth.

Figure 3–12 shows a diagram that summarizes the GPU implementations described in this chapter. The GPU implementations of two full-pixel detectors studied, the RX anomaly detector and the matched filter (MF), are both based on the Cholesky decomposition of the covariance matrix. In the global RX implementation, the Cholesky decomposition is performed on the CPU and the resulted matrix $\mathbf{L}$ is stored in the GPU memory to be shared by all the threads during the parallel

computations. In this implementation, each threads has to solve a triangular system and perform a dot product in order to compute the output of the RX detector for the pixel the thread is working on. In contrast, in the adaptive RX implementation, each thread has to compute the Cholesky decomposition of the covariance matrix estimated using the samples from the region defined from the moving window. Since each thread is working on a different pixel, the resulted covariance matrix cannot be shared during the GPU parallel computations. Therefore, each thread has to compute a Cholesky decomposition, has to solve a triangular system, and has to perform a dot product in order to compute the output of the adaptive RX detector. In the MF implementation, the normalization constant κ and the vector $\mathbf{c}$ can be precomputed on the CPU. In this case, each thread only has to perform a dot product in order to compute the output of the MF detector for the corresponding pixel. Finally, the implementation of the third algorithm studied, the adaptive matched subspace detector (AMSD) for sub-pixel target detection, is based on orthogonal projections onto the background subspace and the combination of the target and background subspaces. The matrices that span the linear subspaces and the corresponding orthogonal projection matrices are precomputed on the GPU. In order to compute the output of the AMSD, each thread has to perform two matrix-vector products, two dot products, and one scalar division.

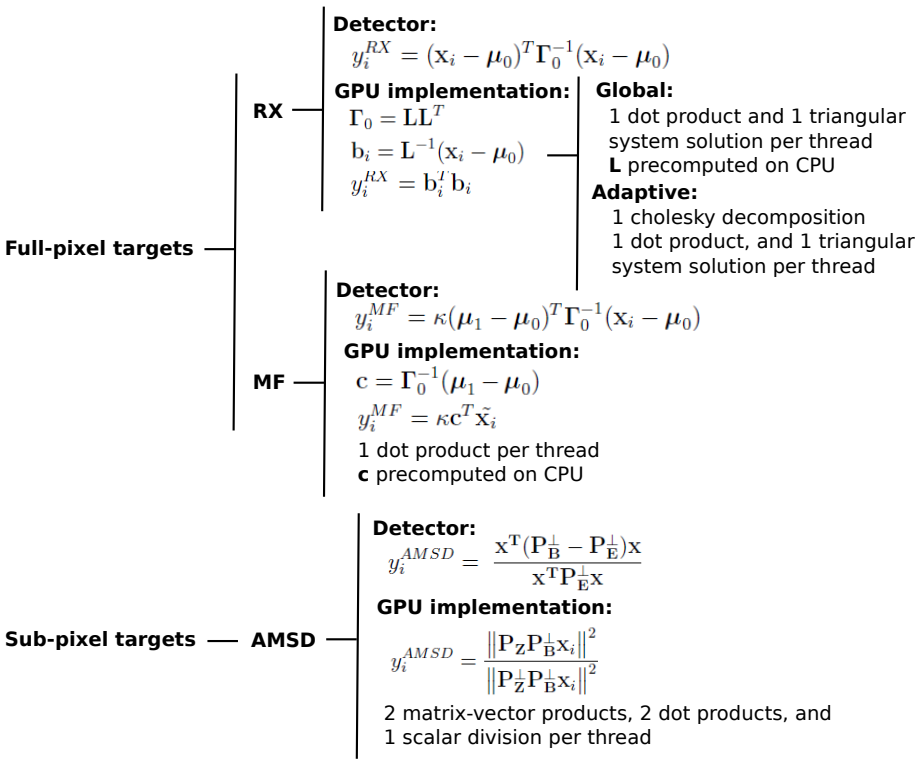


Figure 3–12: Diagram that summarizes the GPU implementations of the detection algorithms.

Chapter 4

Experimental Results

This chapter presents the experimental results obtained using a dataset generated to evaluate the performance of the parallel implementations and the detection accuracy of the algorithms.

4.1 Methodology

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 (Figure 4–1). The image was collected using a SOC-700 visible hyperspectral imager from Surface Optics Corporation®[1] . 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 second to scan 100 lines, thus, the total time needed to complete an image cube is 6.4 seconds.

For the experiments, a 360 $\times$ 360 pixels spatial subset of the original data cube covering a homogeneous background was selected. Figure 4–2 shows a color composite of the region selected using bands 53, 27 and 1 as the red, green, and blue channels, respectively. The scene consists of a T-shirt surface containing traces

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

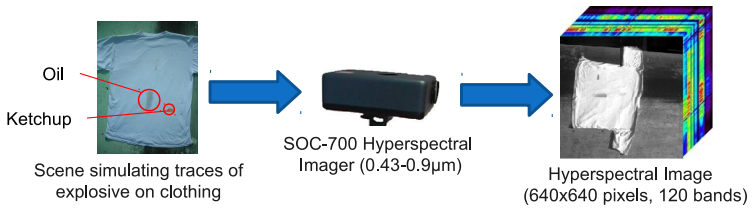


Figure 4–1: Phantom image generation for the experiments: a hyperspectral image was collected using a SOC-700 imager.

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 different image sizes. Table 4–1 shows the sizes in MB of the data cubes generated using this procedure, where the *size 1* corresponds to the size of the initial image subset.

Table 4–1: Sizes in MB of the different data cubes generated by duplicating an original image subset.

Size 1	**Size 2**	**Size 3**	**Size 4**	**Size 5**	**Size 6**
59.3 MB	118.6 MB	237.2 MB	474.4 MB	948.8 MB	1897.6 MB

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 a baseline to estimate the speedups of the GPU-based implementations. The

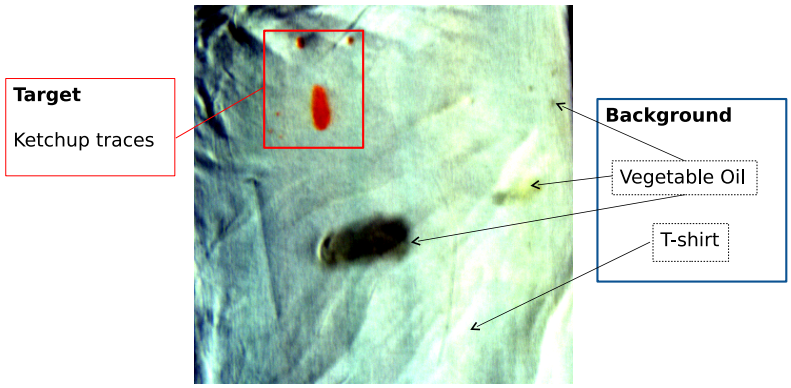


Figure 4–2: Spatial subset selected for the experiments: a scene consists of a T-shirt surface containing traces of vegetable oil and ketchup.

CPU implementation was built with GCC 4.4.5 compiler using C++. In the GPU implementation, the CUBLAS™ [42] and CULA™ [43] 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[2] in combination withthe OpenMP™ [45] interface to exploit CPU parallelism.

4.2 Running Times and Speedups

This section presents the resulting running times of each implementation and the speedups to analyze the performance of the GPU-based implementations with the corresponding CPU-based implementation. The running times were averaged over 10 benchmark executions. The resulting averaged times do not vary significantly if more than 10 executions are used. The function *gettimeofday* from the GNU C header file "sys/time.h" was used for measuring the running times of the algorithms.

[2] http://software.intel.com/en-us/articles/intel-mkl/

The speedups were estimated as the ratio between the averaged running time of the CPU-based and the GPU-based implementations.

In the GPU-based implementations, the number of threads per block was selected as the value that produced the best performance. The CUDA Software Development Kit (SDK) provides a spreadsheet that allows the programmer to choose the number of blocks depending on the amount of shared memory and registers required by the CUDA kernel [12]. The spreadsheet, which is called the CUDA Occupancy Calculator, provides the resulting occupancy for a given resource specification (amount of registers and shared memory per block) and block size selected. The occupancy is defined as the ratio of the number of resident warps per multiprocessor to the maximum number of active warps per multiprocessor that the device allows. A high occupancy helps in reducing memory latencies when accessing global memory by scheduling new warps for execution while another warp is waiting for a memory transaction.

4.2.1 RX algorithm using global statistics

Table 4–2 shows the register, local, shared and constant memory usages required by the CUDA kernel function that implements the RX algorithm using global statistics. Figure 4–3 shows the multiprocessor occupancy for different block sizes, obtained from the CUDA occupancy calculator. There are six block sizes that allow a maximum occupancy of 32 active warps (100 %). The number of threads per block selected was 512, which is the size that produced the best performance.

Table 4–3 shows the running times of the CPU and GPU-based implementations of the global RX algorithm for the different images generated. The last column of this table shows the resulting speedup of the GPU over the CPU implementation.

Table 4–2: Register, local, shared and constant memory usages required by the CUDA kernel function that implements the RX algorithm using global statistics.

Registers	Local Memory	Shared Memory	Constant Memory
9	480 bytes	532 bytes	512 bytes

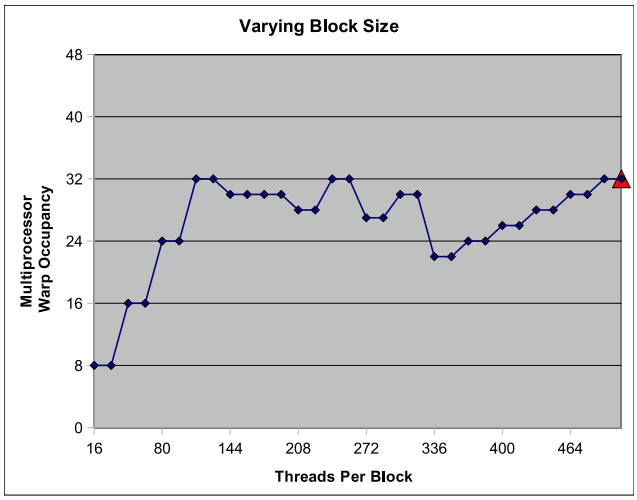


Figure 4–3: Multiprocessor occupancy as a function of the block size (global RX kernel function).

The running times include the computation of the Cholesky decomposition, the memory transfers between the CPU and the GPU, and the kernel execution time. For all the input image sizes, the GPU-based implementation performs faster than the corresponding CPU-based implementation. The speedups achieved vary from 11.25 to 24.76. The evolution of the speedup with the input image size is shown in Figure 4–4. The speedup increases with the image size reaching the maximum value for the largest sized input image. It can be also noticed that for sizes above 237.2 MB the increase in the speedup becomes less significant.

4.2.2 MF algorithm

Table 4–4 shows the resource usage for the CUDA kernel function that implements the matched filter algorithm. This kernel uses more constant memory than the global RX kernel because in the MF implementation both the background mean

Table 4–3: Running times of the the CPU and GPU-based implementations of the global RX algorithm for the different images generated.

Image Size (MB)	CPU Time (sec)	GPU Time (sec)	Speedup
59.3	1.50	0.13	11.25
118.6	2.98	0.19	15.69
237.2	5.94	0.32	18.85
474.4	12.22	0.55	22.16
948.8	24.72	1.03	23.96
1897.6	49.46	1.98	24.76

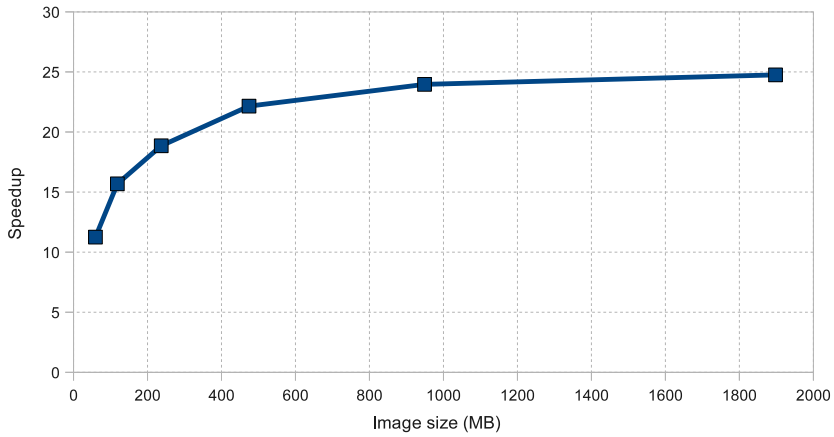


Figure 4–4: Speedup of the global RX GPU-based implementation over the CPU-based implementation for different image sizes.

and the vector **c** were stored in the constant memory space. On the other hand, the amount of local memory used by this kernel is 0 KB. This is another difference with the RX implementation, in which the computation of the solution of the triangular system forces to use the local memory space to store the vector $\mathbf{b}_i$.

Table 4–4: Register, local, shared and constant memory usages required by the CUDA kernel function that implements the MF algorithm.

Registers	Local Memory	Shared Memory	Constant Memory
7	0 bytes	52 bytes	980 bytes

Figure 4–5 shows the multiprocessor occupancy for different block sizes, obtained from the CUDA occupancy calculator. The curve is the same as in the global

RX implementation, since the number of register and the amount of shared memory in both implementations do not limit the maximum number of resident threads per multiprocessor, value which is only limited by the hardware architecture, in this case. In this implementation, a block size of 512 threads was also selected, because was the value that produced the best performance and also guarantee a full occupancy as shown in Figure 4–5.

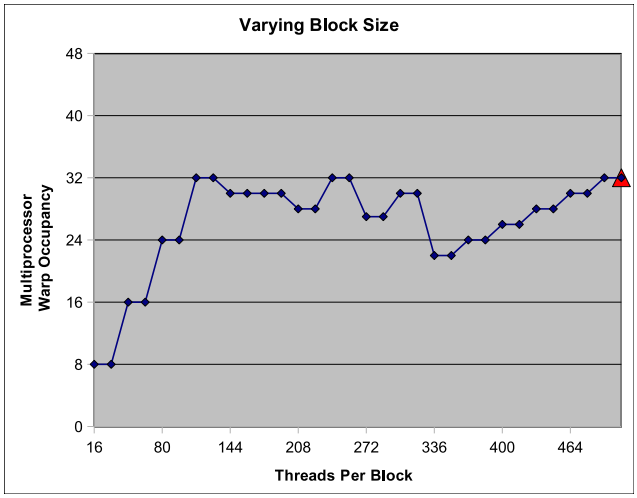


Figure 4–5: Multiprocessor occupancy as a function of the block size (MF kernel function).

Table 4–5 shows the running times of the CPU and GPU-based implementations of the MF algorithm for the different input image sizes, and the resulting speedup of the GPU over the CPU-based implementation. The running times include the computation of the Cholesky decomposition, the computation of the normalization constant κ, the memory transfers between the CPU and the GPU, and the kernel execution time. For the first image size (59.3 MB) the CPU-based implementation is faster (37 milliseconds) than the GPU-based (79 milliseconds). For the rest of the input sizes, the GPU-based 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 limitation in the speedup is due to reduced number

of arithmetic operations performed in the kernel function. Therefore, most of the running time of the GPU-based implementation is spent in the memory transfers between the CPU and GPU.

Table 4–5: Running times of the the CPU and GPU-based implementations of the MF algorithm for the different images generated.

Image Size (MB)	**CPU Time (sec)**	**GPU Time (sec)**	**Speedup**
59.3	0.037	0.079	0.47
118.6	0.094	0.090	1.04
237.2	0.188	0.114	1.65
474.4	0.399	0.161	2.48
948.8	0.879	0.258	3.40
1897.6	1.743	0.448	3.90

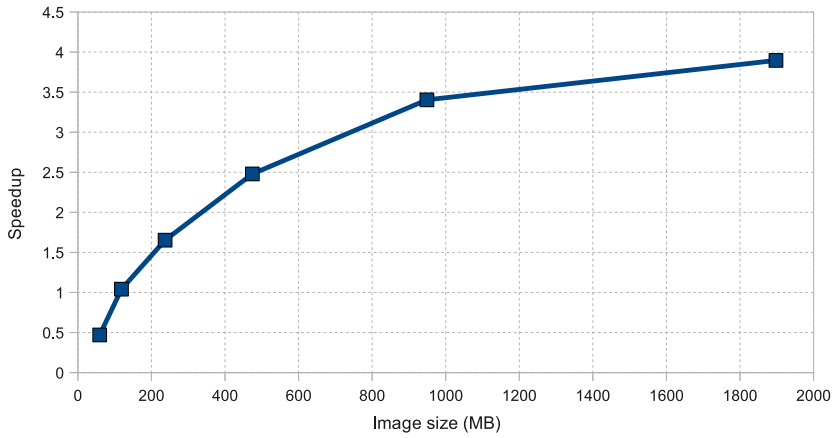


Figure 4–6: Speedup of the MF GPU-based implementation over the CPU-based implementation for different image sizes.

4.2.3 Adaptive RX algorithm

Due to the local memory limitations of the C1060 graphics card, the bands of the input image were downsampled to reduce the number from 120 to 60 in order to fit the requirements of the adaptive RX implementation. The resources used by the CUDA kernel that implements the adaptive RX algorithm are shown in Table 4–6. It

is worth noting the large amount of local memory used by this kernel, value which is close to the physical limit of 16 KB. Figure 4–7 shows the multiprocessor occupancy for different block sizes, obtained from the spreadsheet. In this implementation, the occupancy is limited by the maximum number of registers per multiprocessor. There are eight block sizes that allow a maximum occupancy of 24 warps, which represents a 75 % of occupancy since the maximum number of warps per multiprocessor is 32. However, the best performance was achieved by using a block size of 512 threads, which allows an occupancy of 16 warps (50 %). Since each thread uses 20 registers, only one block can be active per multiprocessor if the block size is 512. But, for this implementation, the maximum occupancy does not result in a better performance.

Table 4–6: Register, local, shared and constant memory usages required by the CUDA kernel function that implements the adaptive RX algorithm.

Registers	Local Memory	Shared Memory	Constant Memory
20	15120 bytes	42 bytes	32 bytes

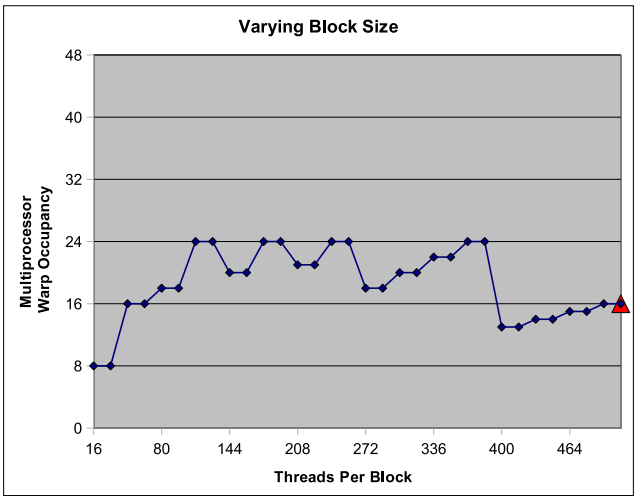


Figure 4–7: Multiprocessor occupancy as a function of the block size (global RX kernel function).

The running times of the adaptive RX implementations were measured using a windows size of 21 × 21. The running times of both the CPU-based and GPU-based

implementations with the corresponding speedups are shown in Table 4–7. The running times includes the memory transfers from the CPU to the GPU and the kernel execution time. The resulting speedups vary from 10.99 for the smallest input size to 14.05 for the largest input size. For the smallest input size, the implementation on the CPU takes 183.58 seconds and the implementation on the GPU takes 16.69 seconds. For the largest input size, the implementation on the CPU takes 8,029.90 seconds (2.23 hours) to complete the execution, whereas the implementation on the GPU takes 571.65 seconds (9.52 minutes). These running times show the high computational complexity of this algorithm. Figure 4–8 shows the resulting speedups as a function of the input size. 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.

Table 4–7: Running times of the the CPU and GPU-based implementations of the adaptive RX algorithm for the different images generated.

Image Size (MB)	**CPU Time (sec)**	**GPU Time (sec)**	**Speedup**
59.3	183.58	16.69	10.99
118.6	385.80	35.38	10.91
237.2	795.24	72.16	11.02
474.4	1,762.27	140.36	12.56
948.8	3,709.10	285.27	13.00
1897.6	8,029.90	571.65	14.05

4.2.4 AMSD algorithm

Table 4–8 shows the resource usage for the CUDA kernel function that implements the adaptive matched subspace algorithm. It can be noticed, as in the MF kernel, that the implementation of this detector does not use local memory. The best performance was achieved for a block size of 512 threads, as in the other three implementations.

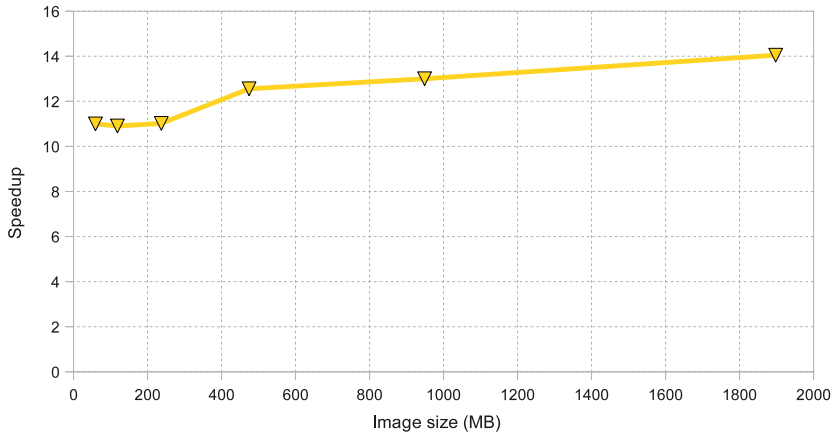


Figure 4–8: Speedup of the adaptive RX GPU-based implementation over the CPU-based implementation for different image sizes.

Table 4–8: Register, local, shared and constant memory usages required by the CUDA kernel function that implements the AMSD algorithm.

Registers	**Local Memory**	**Shared Memory**	**Constant Memory**
12	0 bytes	540 bytes	28 bytes

Table 4–9 shows the running times of the CPU and GPU-based implementations of the preprocessing step that estimates the background subspace basis vectors using the SVD approach and the resulting speedups. The CPU-based implementation outperforms the GPU-based implementation for the first three image sizes: 59.3, 118.6, and 237.2 MB. For the size 474.4 MB and above, the GPU-based implementation becomes slightly faster but the speedup is very limited, reaching a maximum value of 2.17 for the largest image size. This shows that the computation of the autocorrelation matrix and the corresponding eigenvectors do not take advantage of the GPU parallel architecture.

Table 4–9 shows the running times of the CPU and GPU-based implementations of the preprocessing step that estimates the background subspace basis vectors using the MaxD algorithm and the resulting speedups. In this case, the CPU-based

Table 4–9: Running times of the the CPU and GPU-based implementations of the background subspace basis vector estimation step using the SVD approach.

Image Size (MB)	CPU Time (sec)	GPU Time (sec)	Speedup
59.3	0.26	0.80	0.33
118.6	0.50	0.86	0.58
237.2	0.98	1.07	0.92
474.4	1.95	1.39	1.40
948.8	3.85	2.10	1.83
1897.6	7.71	3.56	2.17

implementation outperforms the GPU-based implementation only for the first image sizes of 59.3 MB. For the other sizes, the GPU-based implementation becomes faster, reaching speedups from 1.49 to 7.67. The MaxD algorithm achieves better performance on the GPU than the SVD approach, although the speedup is still limited specially for image sizes bellow 237.2 MB.

Table 4–10: Running times of the the CPU and GPU-based implementations of the background subspace basis vector estimation step using the MaxD algorithm.

Image Size (MB)	CPU Time (sec)	GPU Time (sec)	Speedup
59.3	0.52	0.65	0.80
118.6	1.03	0.70	1.49
237.2	2.02	0.78	2.57
474.4	4.03	0.98	4.1
948.8	8.13	1.39	5.86
1897.6	16.27	2.12	7.67

Table 4–11 shows the running times of the CPU and GPU-based implementations of the AMSD algorithm for the different input image sizes, and the resulting speedup of the GPU over the CPU-based implementation. The running times include the memory transfers between the CPU and the GPU, and the kernel execution time. The GPU-based implementation outperforms the corresponding CPU-based implementation for all input image sizes. The speedups achieved vary from 18.54 to

46.64. The evolution of the speedup with the input image size is shown in Figure 4–9. The GPU-based implementation of the AMSD algorithm is the only implementation that could achieve speedups larger than 30, reaching a maximum value of 47.87.

Table 4–11: Running times of the the CPU and GPU-based implementations of the AMSD algorithm for the different images generated.

Image Size (MB)	CPU Time (sec)	GPU Time (sec)	Speedup
59.3	5.32	0.29	18.54
118.6	16.22	0.50	32.43
237.2	32.55	0.91	35.69
474.4	68.31	1.74	39.37
948.8	161.11	3.37	47.87
1897.6	322.62	6.92	46.64

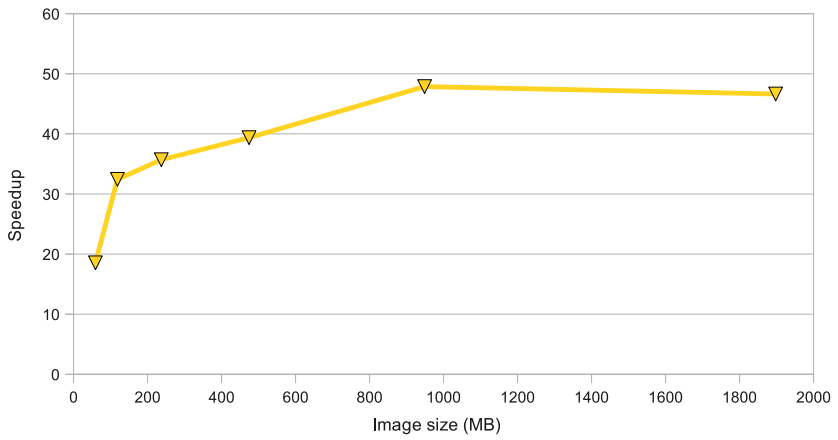


Figure 4–9: Speedup of the AMSD GPU-based implementation over the CPU-based implementation for different image sizes.

4.2.5 Processing Rate

Since the scanning rate of the SOC-700 imager is 15 megabytes per second, 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 were estimated as the ratio of the input image size in MB to the total execution time in seconds needed to process the data.

Table 4–12 and Figure 4–10 shows the resulting processing rates of each implementation as a function of the input size. 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 for the input data sets evaluated.

Table 4–12: Resulting processing rates (MB/sec) of each implementation for the different input image sizes.

Image Size (MB)	**Global RX**	**MF**	**Adaptive RX**	**AMSD**
59.3	442.41	744.92	3.55	206.78
118.6	622.61	1,311.58	3.35	237.10
237.2	752.85	2,083.15	3.29	260.12
474.4	859.87	2,945.36	3.38	273.42
948.8	919.75	3,670.93	3.33	281.91
1897.6	949.72	4,239.81	3.32	274.31

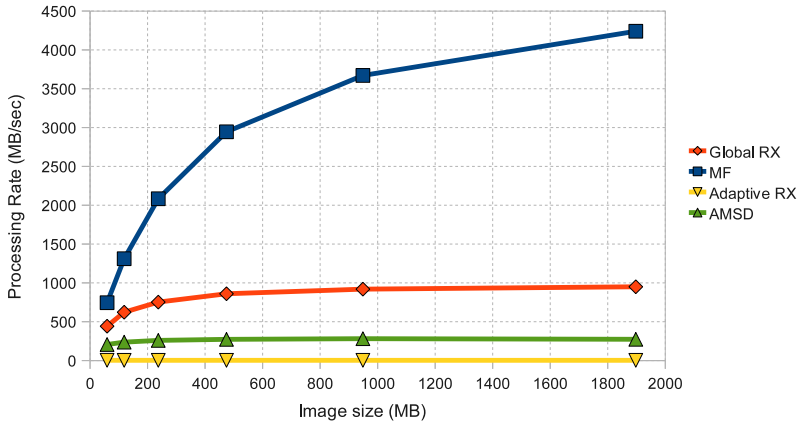


Figure 4–10: Resulting processing rates (MB/sec) of each implementation as a function of the input size.

4.3 Detection Results

This section presents the detection results obtained from each detection algorithm using the image subset shown in Figure 4–2. The ground truth for the target traces is shown in Figure 4–11, where the full-pixels are represented in red, the sub-pixels in yellow, and the guard pixels in green. The target contains 568 full-pixels, 197 sub-pixels and 255 guard-pixels.

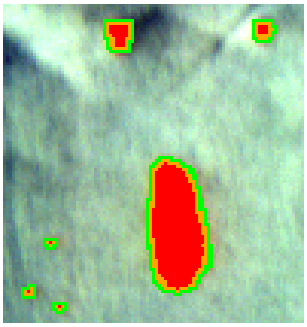
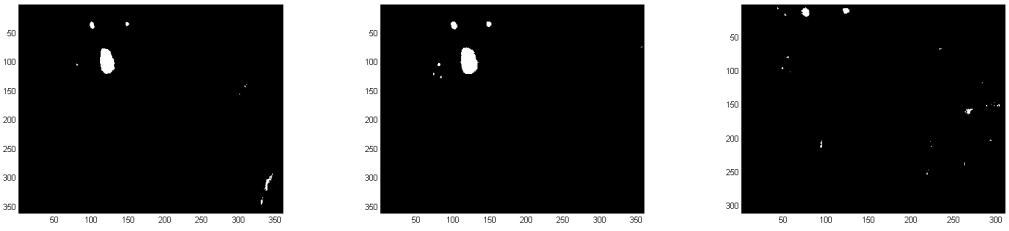


Figure 4–11: Ground truth for the target traces showing the full-pixels (red), sub-pixels (yellow) and guard-pixels (green).

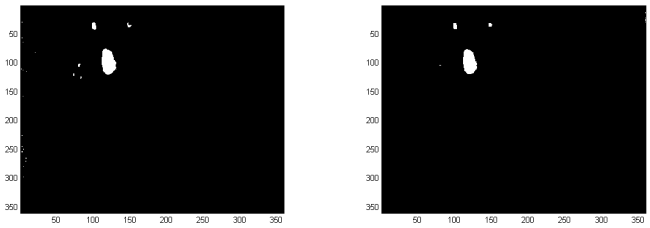
Figures 4–12 and 4–13 show the resulting detection maps for each detector algorithm. The threshold values used for generating the detection maps were selected as the least value that allows the detection of all the full-pixels of the two small traces located on the top of Figure 4–11. Table 4.3 shows the detection statistics (detection accuracy and percentage of false alarms). The best detection accuracy 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 51x51, only 5 small targets were detected (8 % of 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.



(a) (b) (c)

Figure 4–12: Detection Results, (a) RX with global statistics, (b) MF detector, (c) adaptive RX



(a) (b)

Figure 4–13: Detection Results, (a) AMSD with SVD as background subspace estimation method, (b) AMSD with MaxD as background subspace estimation method.

Table 4–13: 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

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this work, the GPU-based parallel implementation of three target detection algorithms for hyperspectral images has been analyzed. 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 first implementation, the statistical parameters of the background distribution (mean and covariance matrix) are globally estimated from training samples as a preprocessing step on the CPU, approach also used in the MF implementation. In contrast, the other implementation estimates these parameters locally 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. In the implementation of this algorithm, two methods for estimating the set of background basis vectors were evaluated, SVD and MaxD, both implemented on the GPU.

In the design of the GPU implementations, we have analyzed three important aspects:

- **Computation decomposition**: The particular structure of the target detectors studied allows a task decomposition in terms of a pixel-level parallelism. 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. Therefore, the output of the detectors for the entire image can be computed in parallel by N threads, being N the total number of pixels in the image. Since the GPU architecture is optimized for data-parallel computations, this type of algorithm structure can be efficiently implemented on the GPU.

- **Data layout**: How the input image is stored in the GPU memory is an important aspect when designing a GPU-based implementation of the detectors, since it can affect the performance considerably. Accessing the GPU global memory generates latencies of hundreds of clock cycles, but the memory throughput can be increased by coalesced memory transactions if the access pattern to global memory satisfied specific requirements that depend on the core architecture of the GPU device. In the GPU-based implementations described in this document, the BSQ storage scheme was used since it leads to coalesced memory transactions because consecutive memory positions correspond to consecutive pixels for the same band. Therefore, threads with consecutive ID numbers will access contiguous memory positions when reading or writing a single band, thus, satisfying the requirements for the core architecture of the GPU graphics card employed in this work.
- **Mapping the computation to the memory hierarchy**: In the design of the GPU-based parallel implementations, 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, they 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 since all the threads of the same block can share the values and read them from the shared memory cache.

In the GPU implementations, linear algebra computations like matrix-matrix multiplications or eigenvectors computations were performed using routines from the CUBLAS™ and CULA™ libraries, since they provide implementations of the BLAS and LAPACK routines optimized for the CUDA architecture. In addition, a CPU-based implementation of each target detector was developed to be used as a baseline to estimate the speedups of the GPU-based implementations. The CPU-based implementations were developed in C++ using the Intel® MKL™ library.

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. The images were spatially duplicated in a tiled fashion in order to evaluate the running times of the implementations for different input data sizes.

The most important results can be summarized as follows:

- The maximum speedup was achieved for the largest data size in all the implementations. Processing larger data sets keeps the multiprocessors more occupied, which helps in reducing the memory latencies.
- The GPU-based 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 slow number of arithmetic operations performed by this detector, achieving speedups bellow 5. 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 this implementation, each thread has to compute the mean and covariance matrix using the samples from the neighborhood defined by the moving window. Since this neighborhood is different for each pixel, both the mean and covariance matrix has to be stored in the local memory space and calculated by reading from global memory. The low bandwidth of this memory space is the many factor that limits the performance of the adaptive RX implementation.
- 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 dataset evaluated.

From the previous results, we can identify some important aspects that should be present in the structure of an algorithm for hyperspectral image exploitation in order to take advantage of the CUDA architecture:

- The algorithm allow a data-parallel decomposition of the computations.
- The portion of the computations performed on the GPU is computationally intensive.
- The structure of the algorithm allows the use of fast GPU memory caches like the shared memory or constant memory spaces.

5.2 Future Work

- Study the incorporation of techniques for the automatic estimation of the background statistical parameters as part of the GPU-based implementations of the full-pixel detectors.
- Study other approaches to implement on the GPU the adaptive RX algorithm in order to take more advantage of the CUDA architecture and study the implementation of other approaches for handling the image borders.
- Study the implementation on the GPU of other algorithms for estimating the background basis vectors for the AMSD.
- Analyze further optimizations of the implementations 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.
- Evaluate the use of other GPU libraries, like MAGMA [1] or LibJacket [2], as alternatives to the CULA library for linear algebra computations.

[1] http://icl.cs.utk.edu/magma/

[2] http://www.accelereyes.com/products/libjacket/

References

[1] H. C. Schau and B. D. Jennette. Hyperspectral requirements for detection of trace explosives agents. In *Proceedings of SPIE: Algorithms and Technologies for Multispectral, Hyperspectral, and Ultraspectral Imagery XII*, volume 6233, page 62331Y, 2006.

[2] C. M. Bachmann, C. R. Nichols, M. J. Montes, R. A. Fusina, Rong-Rong Li, C. Gross, J. Fry, C. Parrish, J. Sellars, S. A. White, C. A. Jones, and K. Lee. Coastal characterization from hyperspectral imagery: An intercomparison of retrieval properties from three coast types. In *2010 IEEE International Geoscience and Remote Sensing Symposium*, pages 138–141, 2010.

[3] H. Burke, S. Hsu, M. Griffin, C. Upham, and K. Farrar. EO-1 hyperion data analysis applicable to cloud detection, coastal characterization and terrain classification. In *2004 IEEE International Geoscience and Remote Sensing Symposium*, volume 2, pages 1483–1486, 2004.

[4] M. Martin, M. Wabuyele, K. Chen, P. Kasili, M. Panjehpour, M. Phan, B. Overholt, G. Cunningham, D. Wilson, R. DeNovo, and T. Vo-Dinh. Development of an advanced hyperspectral imaging (HSI) system with applications for cancer detection. *Annals of Biomedical Engineering*, 34(6):1061–1068, 2006.

[5] A. Paz and A. Plaza. Cluster versus GPU implementation of an orthogonal target detection algorithm for remotely sensed hyperspectral images. In *2010 IEEE International Conference on Cluster Computing*, pages 227–234, 2010.

[6] A. Plaza and C-I. Chang. Clusters versus FPGA for parallel processing of hyperspectral imagery. *Int. J. High Perform. Comput. Appl.*, 22:366–385, November

2008.

[7] A. Plaza, C-I. Chang, J. Plaza, and D. Valencia. Commodity cluster and hardware-based massively parallel implementations of hyperspectral imaging algorithms. In *Proceedings of SPIE: Algorithms and Technologies for Multispectral, Hyperspectral, and Ultraspectral Imagery XII*, volume 6233, page 623316, 2006.

[8] C. González, J. Resano, D. Mozos, A. Plaza, and D. Valencia. FPGA implementation of the pixel purity index algorithm for remotely sensed hyperspectral image analysis. *EURASIP J. Adv. Signal Process*, 2010:68:1–68:13, February 2010.

[9] D. González, C. Sánchez, R. Veguilla, N. G. Santiago, S. Rosario-Torres, and M. Vélez-Reyes. Abundance estimation algorithms using NVIDIA CUDA technology. In *Proceedings of SPIE, Algorithms and Technologies for Multispectral, Hyperspectral, and Ultraspectral Imagery XIV*, volume 6966, page 69661E, 2008.

[10] S. Sanchez, G. Martin, A. Plaza, and C-I. Chang. GPU implementation of fully constrained linear spectral unmixing for remotely sensed hyperspectral data exploitation. In *Proceedings of SPIE: Satellite Data Compression, Communications, and Processing VI*, volume 7810, page 78100G, 2010.

[11] A. Paz and A. Plaza. GPU implementation of target and anomaly detection algorithms for remotely sensed hyperspectral image analysis. In *Proceedings of SPIE: Satellite Data Compression, Communications, and Processing VI*, volume 7810, page 78100R, 2010.

[12] NVIDIA CUDA Programming Guide. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf, November 2010. Version 3.2.

[13] D. Manolakis, D. Marden, and G.A Shaw. Hyperspectral image processing for automatic target detection applications. *Lincoln Laboratory Journal*, 14(1):79–116, 2003.

[14] R. O. Green. Imaging spectroscopy and the Airborne Visible/Infrared Imaging Spectrometer (AVIRIS). In *Remote Sensing of Environment*, volume 68, pages 227–248, 1998.

[15] F. Kruse. Visible-infrared sensors and case studies. In *Remote Sensing for the Earth Scienses*, 1999.

[16] R. J. Muirhead. *Aspects of Multivariate Statistical Theory*. Wiley, New York, 1982.

[17] I.S. Reed and X. Yu. Adaptive multiple-band CFAR detection of an optical pattern with unknown spectral distribution. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 38(10):1760–1770, October 1990.

[18] N. Keshava and J.F. Mustard. Spectral unmixing. *IEEE Signal Processing Magazine*, 19(1):44–57, January 2002.

[19] S. Kraut, L.L. Scharf, and R.W. Butler. The adaptive coherence estimator: a uniformly most-powerful-invariant adaptive detection statistic. *IEEE Transactions on Signal Processing*, 53(2):427–438, February 2005.

[20] D. Manolakis and G. Shaw. Detection algorithms for hyperspectral imaging applications. *IEEE Signal Processing Magazine*, 19(1):29–43, January 2002.

[21] D. Manolakis, C. Siracusa, and G. Shaw. Hyperspectral subpixel target detection using the linear mixing model. *IEEE Transactions on Geoscience and Remote Sensing*, 39(7):1392–1409, July 2001.

[22] Y. M. Masalmah and M. Vélez-Reyes. Unsupervised unmixing of hyperspectral imagery using the constrained positive matrix factorization. In *Proceedings of SPIE: Independent Component Analyses, Wavelets, Unsupervised Smart Sensors, and Neural Networks IV*, 2006.

[23] J. Boardman, F. A. Kruse, and R. O. Green. Mapping target signatures via partial unmixing of aviris data. In *Proceedings of JPL Airborne Earth Science Workshop*, pages 23–26, 1995.

[24] A. Plaza, P. Martinez, R. Perez, and J. Plaza. Spatial/spectral endmember extraction by multidimensional morphological operations. *IEEE Transactions on Geoscience and Remote Sensing*, 40(9):2025–2041, September 2002.

[25] M. E. Winter. N-FINDR: an algorithm for fast autonomous spectral endmember determination in hyperspectral data. In *Proceedings of SPIE: Imaging Spectrometry V*, volume 3753, pages 266–275, 1999.

[26] J.M. Grossmann, J.H. Bowles, D. Haas, J.A. Antoniades, M.R. Grunes, P.J. Palmadesso, D. Gillis, K.Y. Tsang, M. Baumback, M. Daniel, J. John Fisher, and I.A. Triandaf. Hyperspectral analysis and target detection system for the adaptive spectral reconnaissance program (ASRP). In *Proceedings of SPIE: Algorithms for Multispectral and Hyperspectral Imagery IV*, volume 3372, pages 2–13, 1998.

[27] J. R. Schott, K. Lee, R. V. Raqueno, G. D. Hoffmann, and G. Healey. A subpixel target detection technique based on the invariance approach. *Proceedings of the AVIRIS Workshop*, February 2003.

[28] M. Velez-Reyes and S. Rosario. Solving adundance estimation in hyperspectral unmixing as a least distance problem. In *2004 IEEE International Geoscience and Remote Sensing Symposium*, volume 5, pages 3276–3278, September 2004.

[29] C. L. Lawson and R. J. Hanson. *Solving Least Squares Problems*. Prentice-Hall, 1974.

[30] C.-I. Chang and D.C. Heinz. Constrained subpixel target detection for remotely sensed imagery. *IEEE Transactions on Geoscience and Remote Sensing*, 38(3):1144–1159, May 2000.

[31] J. Broadwater and R. Chellappa. Hybrid detectors for subpixel targets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(11):1891–1903, November 2007.

[32] C. Peña-Ortega and M. Velez-Reyes. Comparison of basis-vector selection methods for structural modeling of hyperspectral imagery. In *Proceedings of SPIE: Imaging Spectrometry XIV*, volume 7457, page 74570C, 2009.

[33] J.C. Harsanyi and C.-I. Chang. Hyperspectral image classification and dimensionality reduction: an orthogonal subspace projection approach. *IEEE Transactions on Geoscience and Remote Sensing*, 32(4):779–785, July 1994.

[34] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, August 1998.

[35] J. Setoain, M. Prieto, C. Tenllado, A. Plaza, and F. Tirado. Parallel morphological endmember extraction using commodity graphics hardware. *IEEE Geoscience and Remote Sensing Letters*, 4(3):441 –445, July 2007.

[36] A. Plaza, D. Valencia, J. Plaza, and P. Martinez. Commodity cluster-based parallel processing of hyperspectral imagery. *J. Parallel Distrib. Comput.*, 66(3):345–358, 2006.

[37] A. Plaza, D. Valencia, J. Plaza, and Chein-I Chang. Parallel implementation of endmember extraction algorithms from hyperspectral data. *IEEE Geoscience and Remote Sensing Letters*, 3(3):334–338, July 2006.

[38] A. Plaza, J. Plaza, and S. Sanchez. Hyperspectral unmixing on NVidia GPUs, 2009. NVidia CUDA Zone.

[39] M. E. Daube-Witherspoon and G. Muehllehner. An iterative image space reconstruction algorthm suitable for volume ECT. *IEEE Transactions on Medical Imaging*, 5(2):61–66, 1986.

[40] M. E. Winter and E. Winter. Hyperspectral processing in graphical processing units. In *In Proceedings of SPIE: Algorithms and Technologies for Multispectral, Hyperspectral, and Ultraspectral Imagery XVII*, volume 8048, To be published, 2011.

[41] 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. *J Real-Time Image Proc*, 4(3):287–300, August 2009.

[42] CUDA CUBLAS Library. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUBLAS_Library.pdf, August 2010. Version 3.2.

[43] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis. CULA: Hybrid GPU Accelerated Linear Algebra Routines. In *Proceedings of SPIE: Modeling and Simulation for Defense Systems and Applications V*, volume 7705, page 770502, 2010.

[44] N. Acito, G. Corsini, and M. Diani. Adaptive detection algorithm for full pixel targets in hyperspectral images. *Vision, Image and Signal Processing, IEE Proceedings*, 152(6):731 – 740, 2005.

[45] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, 2001.

APPENDICES

Appendix A

Library of Target Detection Algorithms

This appendix documents the functions provided by the compiled library that includes the GPU-based implementations of the target detection algorithms described in this work.

A.1 List of Functions

The library implements the following set of functions:

- RXdetector
- MFdetector
- RXdetector_adaptive
- getSubspace
- AMSD

A.2 Function Description

Since all functions perform the computations using single-precision, they only accept pointers to *float* data types. The functions return an integer value that can be used for error handling. The possible returned values are:

- $= 0$: Success exit. No errors occurred.
- < 0: A returned value of $-i$ means that the i-th argument had an illegal value.

The first argument of all functions is a pointer to the hyperspectral image data. The image must be stored in the CPU memory as a linear array in BSQ format, in order to be read successfully by these functions.

A.2.1 RXdetector

Description:

This function computes the output values of the RX detection algorithm for all the pixels of a hyperspectral image. The mean and covariance matrix that characterize the background statistical distribution are input parameters and, therefore, they have to be estimate previously.

Function prototype:

int RXdetector(float image, int lines, int samples, int bands, float* mean, float* covar, float* RXoutput)*

Arguments:

- **image:** a pointer to the input image data.
- **lines:** number of lines of the hyperspectral image.
- **samples:** number of samples of the hyperspectral image.
- **bands:** number of bands of the hyperspectral image.
- **mean:** pointer to the background mean vector.
- **covar:** pointer to the background covariance matrix.
- **RXoutput:** pointer to the RX output values.

A.2.2 MFdetector

Description:

This function computes the output values of the MF detection algorithm for all the pixels of a hyperspectral image. The target, background mean and background covariance matrix are input parameters and, therefore, they have to be estimate previously.

Function prototype:

int MFdetector(float image, int lines, int samples, int bands, float* mean, float* covar, float* target, float* MFoutput)*

Arguments:

- **image:** a pointer to the input image data.
- **lines:** number of lines of the hyperspectral image.
- **samples:** number of samples of the hyperspectral image.
- **bands:** number of bands of the hyperspectral image.
- **mean:** pointer to the background mean vector.
- **covar:** pointer to the background covariance matrix.
- **target:** pointer to the target mean vector.
- **MFoutput:** pointer to the MF output values.

A.2.3 RXdetector_adaptive

Description:

This function computes the output values of the RX detection algorithm for all the pixels of a hyperspectral image. The mean and covariance matrix that characterize the background statistical distribution are locally estimated by this function using a double moving window centered at the test pixel. The user has to specify the parameters that define the sizes of the moving windows.

Function prototype:

int RXdetector_adaptive(float image, int lines, int samples, int bands, int W, int G, float* RXoutput)*

Arguments:

- **image:** a pointer to the input image data.
- **lines:** number of lines of the hyperspectral image.
- **samples:** number of samples of the hyperspectral image.
- **bands:** number of bands of the hyperspectral image.

- **W:** parameter that defines the size of the outer window as $(2W+1) \times (2W+1)$.
- **G:** parameter that defines the size of the inner guard window as $(2G+1) \times (2G+1)$.
- **RXoutput:** pointer to the RX output values.

A.2.4 getSubspace

Description:

This function estimates a set of basis vectors to characterize a background subspace model. The user has to specify the estimation method and the desired number of background basis vectors. Two method are implemented in this function: SVD and MaxD. In the SVD method, the basis vectors are estimated as the eigenvectors of the image correlation matrix.

Function prototype:

int getSubspace(float image, int lines, int samples, int bands, const char* method, int M, float* B)*

Arguments:

- **image:** a pointer to the input image data.
- **lines:** number of lines of the hyperspectral image.
- **samples:** number of samples of the hyperspectral image.
- **bands:** number of bands of the hyperspectral image.
- **method:** character string defining the method used for estimating the background subspace. Two values are allowed: 'SVD' and 'MaxD'.
- **M:** desired number of basis vectors.
- **B:** resulting matrix of background basis vectors.

A.2.5 AMSD

Description:

This function computes the output values of the AMSD detection algorithm for all the pixels of a hyperspectral image. The user has to specify the dimensionality of

the background subspace, the matrix of background basis vectors, the dimensionality of the target subspace, and the matrix of target basis vectors.

Function prototype:

int AMSD(float image, int lines, int samples, int bands, int M, float* B, int P, float* S, float* AMSDoutput)*

Arguments:

- **image:** a pointer to the input image data.
- **lines:** number of lines of the hyperspectral image.
- **samples:** number of samples of the hyperspectral image.
- **bands:** number of bands of the hyperspectral image.
- **M:** dimensionality of background subspace.
- **B:** pointer to the matrix of background basis vectors.
- **P:** dimensionality of target subspace.
- **S:** pointer to the matrix of target basis vectors.
- **AMSDoutput:** pointer to the AMSD output values.

Appendix B

List of LAPACK functions

This appendix presents the description of the LAPACK functions used in this work as part of the CUBLAS, CULA, and MKL libraries.

ISAMAX Finds the smallest index of the maximum magnitude element of a vector. That is, given an input vector $\mathbf{x} = [x_1\ x_2 \cdots x_n]$, this function finds the first i that maximizes $|x_{1+i \cdot incx}|$, where $incx$ is the storage spacing between the elements of $\mathbf{x}$ and i is an integer between 0 and n-1.

ISAMIN Finds the smallest index of the minimum magnitude element of a vector. That is, given an input vector $\mathbf{x} = [x_1\ x_2 \cdots x_n]$, this function finds the first i that minimizes $|x_{1+i \cdot incx}|$, where $incx$ is the storage spacing between the elements of $\mathbf{x}$ and i is an integer between 0 and n-1.

SGEMM It performs the following matrix-matrix operation:

$$\mathbf{C} = \alpha\ op(\mathbf{A}) \cdot op(\mathbf{B}) + \beta\mathbf{C}$$

where $\cdot$ denotes matrix multiplication and $op(\mathbf{X}) = \mathbf{X}$ or $op(\mathbf{X}) = \mathbf{X}^T$. α and β are scalar constants. $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are matrices stored in column-major format, with $op(\mathbf{A})$ an $m \times k$ matrix, $op(\mathbf{B})$ an $k \times n$ matrix, and $\mathbf{C}$ an $m \times n$ matrix.

SGER It performs the symmetric rank 1 operation:

$$\mathbf{A} = \alpha\ \mathbf{x} \cdot \mathbf{y}^T + \mathbf{A}$$

where $\cdot$ denotes vector multiplication, α is a scalar constant, $\mathbf{x}$ is an m-element vector, $\mathbf{y}$ is an n-element vector, and $\mathbf{A}$ is an $m \times n$ matrix stored in column-major format.

SPOTRF It computes the Cholesky factorization of a real symmetric positive definite matrix $\mathbf{A}$. The factorization has the form $\mathbf{A} = \mathbf{U}^T \cdot \mathbf{U}$ or $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^T$, depending on the option selected by the user, where $\mathbf{U}$ is an upper triangular matrix and $\mathbf{L}$ is lower triangular.

SSYEV It computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix $\mathbf{A}$.