Parallel Implementation of Polarimetric Synthetic Aperture Radar Data Processing for Unsupervised Classification Using the Complex Wishart Classifier

Sergio Sánchez, *Member, IEEE*, Prashanth R. Marpu, *Senior Member, IEEE*, Antonio Plaza, *Fellow, IEEE*, and Abel Paz-Gallardo

***Abstract*—This work investigates the parallel implementation of target decomposition and unsupervised classification algorithms for polarimetric synthetic aperture radar (POLSAR) data processing. The algorithms are implemented using two different parallel programming models: 1) clusters of CPUs, using message passing interface (MPI), and 2) commodity graphic processing units (GPUs), using the compute device unified architecture (CUDA). POLSAR data processing generally involves a large amount of computations as the full polarimetric information needs to be decomposed and analyzed. Our experiments reveal that GPU architectures provide a good framework for massive parallelization of POLSAR data processing. For instance, it is found that a single GPU can be more efficient than a cluster of 128 nodes with speedups of more than 100× in comparison with the single processor times. The proposed implementation makes the best use of low-level features in the GPU architecture such as shared memories, while also providing coalesced accesses to memory in order to achieve maximum performance.**

***Index Terms*—Clusters of computers, graphic processing units (GPUs), high-performance computing, message passing interface (MPI), polarimetric synthetic aperture radar (POLSAR), target decomposition, unsupervised classification.**

I. Introduction

THE LAST decade has seen the launch of a number of synthetic aperture radar (SAR) satellites such as ALOS/PalSAR, Radarsat-2, and TerraSAR-X which can provide fully polarimetric data, and there are more satellites which are scheduled to be launched in the near future (e.g., ALOS-3). These developments have significantly increased the amount of applications based on the exploitation of polarimetric synthetic aperture radar (POLSAR) data [1], [2]. POLSAR data are often represented using a 3×3 complex coherence matrix and processing such data is computationally very expensive as it involves complex arithmetic operations.

With the recent advances in high-performance computing hardware, there has been a tremendous interest in developing highly efficient parallel algorithms that can make the best possible use of an increasingly growing number of available parallel resources. Commodity graphic processing units (GPUs) have been widely used for high-performance computing applications in remote sensing data processing. Their suitability is mainly due to their low cost and programmability, which makes them a very competitive platform with regard to other architectures with lower power consumption such as multicores or field programmable gate arrays (FPGAs). For instance, in [3], a general overview of remote sensing image processing using GPUs and multicore architectures was provided. A comparison of FPGAs and GPUs in hyperspectral remote sensing applications was given in [4]. Several works on efficient implementations of hyperspectral analysis algorithms have been discussed for different applications, such as spectral unmixing [5]–[7], target detection [8], [9], compressive sensing [10], or segmentation [11]. Acceleration of radiative transfer models in GPUs was discussed in [12]. Change detection was also accelerated in [13] with significant speedups. In [14], a GPU implementation of one-class support vector machine (SVM) was presented with very high computational performance as compared to the CPU version. Very high speedups are achieved in all these examples. There has also been some work on utilizing GPUs for SAR image processing [15]–[20]. However, there is very limited work on using high-performance computing architectures for efficient processing of POLSAR data.

In this work, we study the computational aspects of the combination of Cloude and Pottier's target decomposition technique [21] and the complex Wishart classifier, which is essentially a K-means classifier using the Wishart distance [22]. Target decomposition classifies the image into eight types of scattering, with the categories of surface scattering, dipole scattering, vegetation scattering, and multiple scattering mechanisms. But such classification is not completely exact, as the boundaries of the classes are somewhat arbitrarily chosen. The combination of this target decomposition with the iterative Wishart classifier produces better classification results (in unsupervised fashion) as it uses the full polarimetric information.

Manuscript received January 14, 2015; revised June 03, 2015; accepted July 23, 2015. Date of publication September 08, 2015; date of current version January 18, 2016. This work was supported in part by Masdar Institute of Science and Technology Internal Research Grant; and in part by the computing facilities of Extremadura Research Centre for Advanced Technologies (CETA-CIEMAT), funded by the European Regional Development Fund (ERDF). CETA-CIEMAT belongs to CIEMAT and the Government of Spain.

S. Sánchez and P. R. Marpu are with the Institute Center for Water and Environment (iWater), Masdar Institute of Science and Technology, Abu Dhabi 54224, UAE (e-mail: smartinez@masdar.ac.ae; pmarpu@masdar.ac.ae).

A. Plaza is with the Department of Technology of Computers and Communications, Escuela Politécnica de Cáceres, University of Extremadura, Cáceres 10003, Spain (e-mail: aplaza@unex.es).

A. Paz-Gallardo is with the CETA-CIEMAT, Trujillo, Cáceres 10200, Spain, and also with the Department of Technology of Computers and Communications, Escuela Politécnica de Cáceres, University of Extremadura, Cáceres 10003, Spain (e-mail: abelfrancisco.paz@ciemat.es).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSTARS.2015.2471083

The basic observable in polarimetric data is a complex scattering matrix, which is used to build the covariance and coherence matrices that are further processed to analyze the data. The classification method can be expensive in terms of computation due to the high dimensionality of the data. In this work, we develop two near real-time versions of the method using parallel computing architectures. We first investigate the implementation in a cluster, based on message passing interface (MPI).[1] Also an implementation of the algorithm has been developed using compute device unified architecture (CUDA)[2] and tested on an Nvidia Tesla M2075 GPU. Such an implementation can be used for near real-time on-board processing of POLSAR data. The important aspects of data handling in the GPU and the optimization of the kernel are addressed in greater detail.

This paper is organized as follows. Section II describes the POLSAR data processing chain of the standard target decomposition algorithm. Section III describes the parallel architectures used in this work, along with the parallel implementation of this method in both (cluster and GPU) platforms. Section IV presents an experimental evaluation of the proposed implementations in terms of parallel performance. Section V concludes the paper with some remarks and hints on plausible future research lines.

II. POLSAR Data Processing Chain for Unsupervised Classification

POLSAR data are often represented as a 3×3 complex coherence matrix ($\boldsymbol{T}$) that embeds all the information pertaining to the different polarization combinations that, in turn, characterize the scattering behavior in a pixel. The multilook coherency matrix for n number of looks is given by

$$\langle \boldsymbol{T} \rangle = \frac{1}{n} \sum_{j=1}^{n} \boldsymbol{k}_j \boldsymbol{k}_j^T \tag{1}$$

where $\boldsymbol{k}_j$ is the complex Pauli scattering vector of the jth pixel. The basic observable in POLSAR is a complex scattering matrix with the following coefficients [23]:

$$\mathbf{S} = \begin{bmatrix} S_{HH} & S_{HV} \\ S_{VH} & S_{VV} \end{bmatrix} \tag{2}$$

where the subscript HV, e.g., corresponds to the scattering element of horizontal transmitting and vertical receiving polarizations. For unsupervised analysis of POLSAR data, the coherence matrix is decomposed using an appropriate method to characterize the scattering behavior at the surface. One of the most commonly used decomposition algorithms is the entropy-alpha-anisotropy (H/α/A) decomposition, which can be used to cluster the POLSAR data into 16 classes [21]. Calculating H, α, and A requires eigenvalue decomposition of the complex coherence matrix at every pixel. The target decomposition algorithm is framed by using the eigenvalues and eigenvectors of the multilook coherency matrix as follows:

$$\langle \boldsymbol{T} \rangle = \lambda_1 \boldsymbol{e}_1 \boldsymbol{e}_1^T + \lambda_2 \boldsymbol{e}_2 \boldsymbol{e}_2^T + \lambda_3 \boldsymbol{e}_3 \boldsymbol{e}_3^T \tag{3}$$

where λ_i and $\boldsymbol{e}_i$ are eigenvalues and eigenvectors, respectively. The entropy (H) can be defined as

$$H = \sum_{i=1}^{3} - P_i \log_l P_i \tag{4}$$

where $P_i = \frac{\lambda_i}{\sum_{j=1}^{l} \lambda_j}$.

The eigenvectors can be mathematically written as

$$\boldsymbol{e}_i = e^{i\varphi_1} \left[\cos\alpha_i \;\; \sin\alpha_i \cos\beta_i e^{i\delta_i} \;\; \sin\alpha_i \sin\beta_i e^{i\gamma_i} \right]^T \tag{5}$$

where α is the variation from surface scattering (0°) to dipole scattering (45°) to double bounce scattering (90°), β is twice the polarization orientation angle, δ is the phase difference between the $S_{HH} + S_{VV}$ and $S_{HH} - S_{VV}$ terms, γ is the phase difference between the $S_{HH} + S_{VV}$ and S_{HV} terms, and φ is the phase of the $S_{HH} + S_{VV}$ term.

The average alpha angle is defined accordingly as

$$\overline{\alpha} = \sum_{i=1}^{3} P_i \alpha_i. \tag{6}$$

Anisotropy is defined as

$$A = \frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2}. \tag{7}$$

The values of H and α are used to define nine classes indicating various types of scattering mechanisms. The different scattering mechanisms that can be classified are as follows:

1) Z9: low entropy surface scatterers (H < 0.5; $\alpha < 42.5$), e.g., scattering surfaces such as water, sea ice, and very smooth land surfaces.
2) Z8: low entropy dipole scatterers (H < 0.5; $42.5 < \alpha < 47.5$), e.g., isolated dipoles.
3) Z7: low entropy multiple scatterers (H < 0.5; $\alpha > 47.5$), e.g., isolated dielectric and metallic dihedral scatterers.
4) Z6: medium entropy surface scattering ($0.5 <$ H < 0.9; $\alpha < 40$), e.g., oblate spheroidal scatterers (leafs and discs).
5) Z5: medium entropy vegetation scattering ($0.5 <$ H < 0.9; $40 < \alpha < 50$), e.g., vegetated surfaces with anisotropic scatterers.
6) Z4: medium entropy multiple scattering ($0.5 <$ H < 0.9; $\alpha > 50$), e.g., forestry and urban areas.
7) Z2: high entropy vegetation scattering (H > 0.9; $40 < \alpha < 52.5$), e.g., forest canopies and highly anisotropic bushes.
8) Z1: high entropy multiple scattering (H > 0.9; $\alpha > 52.5$), e.g., vegetation with well-developed branch and crown structure.

Z3 indicates high entropy surface scattering and is nonexistent. Fig. 1 shows the boundaries of various zones. These

[1] [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/
[2] [Online]. Available: https://developer.nvidia.com/cuda-zone

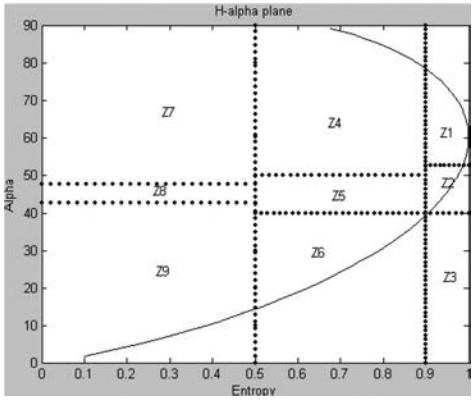

Fig. 1. Classification using target decomposition. The solid line indicates the theoretical boundary of the scatter in the H – α plane.

classes can be further divided into 16 classes by setting a threshold of 0.5 for anisotropy (A). The boundaries are though arbitrarily chosen, some justifications for the choice are given in [12] and [13].

The target decomposition algorithm only uses partial information from the coherence matrix. So, this may result in some of the clusters at the boundaries not being exactly classified or maybe two or more clusters falling in the same region or the elements of the same cluster falling in different classes. This can be avoided by using a combined classification with the Wishart classifier, i.e., a K-means classifier using the Wishart distance.

The classes segmented in the target decomposition algorithm are used as initial clusters for the Wishart classifier, which is iterated till convergence is achieved. The mean of class coherency matrices is found as

$$\boldsymbol{V}_i = \frac{1}{n_i}\sum_{j=1}^{n_i}\langle \boldsymbol{T}\rangle_j \quad \text{for all pixels in a class } \omega_i \tag{8}$$

where n_i is the number of pixels in class i. The Wishart classification is based on the distance measure of the coherency matrix to the cluster center, as defined in (8). The distance measure of a pixel to one of the cluster centers is then given by

$$d\left(\langle \boldsymbol{T}\rangle, \boldsymbol{V}_m\right) = \ln|\boldsymbol{V}_m| + Tr\left(\boldsymbol{V}_m^{-1}\langle \boldsymbol{T}\rangle\right). \tag{9}$$

A pixel is assigned to the class ω_m based on the minimum distance method as

$$d\left(\langle \boldsymbol{T}\rangle, \boldsymbol{V}_m\right)\ d\left(\langle \boldsymbol{T}\rangle, \boldsymbol{V}_j\right) \quad \text{for all } \omega_m \neq \omega_j. \tag{10}$$

The segmentation of the image can be further improved by the iterations. The termination criterion is either the number of pixels changing classes or a prespecified number of iterations.

III. Parallel Implementations

The considered processing chain involves estimation of the complex coherence matrix at every pixel, eigenvalue decomposition of the complex coherence matrix, calculation of H and α values, performing the target decomposition to make a first classification of the pixels into eight classes, and performing several iterations of Wishart classifier to get a better separation of the classes. The main computational aspect of the processing chain is the eigenvalue decomposition of the complex coherence matrix at every pixel. Since the complex-coherence matrix is always a 3×3 matrix for POLSAR data, we simply use the direct formulas derived by solving the eigenvalue decomposition of a 3×3 matrix which is reduced to solving a complex cubic polynomial.

A. MPI Implementation

In the MPI programming model, a computation comprises one or more processes that communicate by calling library routines to send and receive messages to other processes. In most MPI implementations, a fixed set of processes is created at program initialization, and one process is created per processor. The flexibility of MPI allows us to make use of a multiple program multiple data (MPMD) programming model or single program multiple data (SPMD). In this work, SPMD model is used for most of the execution time since the same program is applied to all pixels in the scene. Processes can use *point-to-point* communication operations to send a message from one named process to another; these operations can be used to implement local and unstructured communications. A group of processes can call *collective* communication operations to perform commonly used global operations such as summation and broadcast. MPI's ability to *probe* for messages supports asynchronous communication. A mechanism called a *communicator* allows the MPI programmer to define modules that encapsulate internal communication structures.

The pseudocode for the processing chain is shown in Algorithm 1. We divide the execution into p processes executed in parallel, where each process deals with a part of the data consisting of approximately n/p pixels.

Algorithm 1. Pseudo-code of the MPI parallel implementation of POLSAR unsupervised classification algorithm.

$n \rightarrow$ number of pixels
$\boldsymbol{T}_{9\times n} \rightarrow$ Coherence matrix (each pixel has 9 elements)
$p \rightarrow$ number of processes
$k \rightarrow$ number of classes
$it \rightarrow$ number of iteration in Wishart classifier

parallel for $i = 0$ to $p-1$
  $\boldsymbol{T_i} \leftarrow$ Read_Coherence($\boldsymbol{T}$, i);
  $\boldsymbol{eigvec_i}, \boldsymbol{eigval_i} \leftarrow$ Compute_Eigenv($\boldsymbol{T_i}$);
  $\boldsymbol{H_i}, \boldsymbol{\alpha_i} \leftarrow$ Compute_H_alpha($\boldsymbol{eigvec_i}, \boldsymbol{eigval_i}$);
  $\boldsymbol{Z_i} \leftarrow$ Targ_Decomp($\boldsymbol{H_i}, \boldsymbol{\alpha_i}$);
  for $l = 0$ to it
    $[\boldsymbol{Z_i}]_{\boldsymbol{l}} \leftarrow$ Wishart_Class($\boldsymbol{T_i}$, $[\boldsymbol{Z_i}]_{\boldsymbol{l-1}}$);
  end for
end parallel for

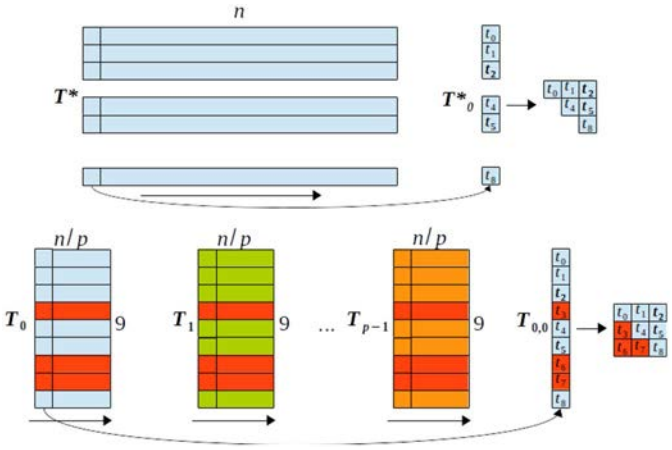

Fig. 2. Data representation for the coherence matrix in the memory.

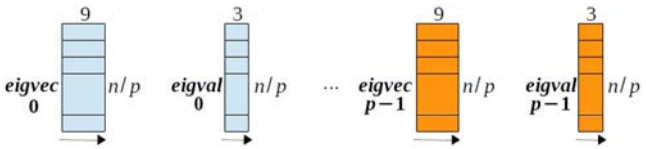

Fig. 3. Illustration of computation of eigenvalues and eigenvectors in each process.

1) Reading of the Coherence Matrix: The process of loading the matrix $\boldsymbol{T}$ from hard disk to main memory is illustrated in Fig. 2. Arrows indicate how the elements are stored in memory. Values in the same pixel are separated by n positions in memory. The coherence matrix is a Hermitian matrix, i.e., the element in the ith row and jth column is equal to the complex conjugate of the element in the jth row and ith column, for all indices i and i. Therefore, we only need the elements of the diagonal and upper triangular matrix to be stored in hard disk, and the lower triangular matrix can be computed and stored in memory. The three diagonal elements are real and the six off-diagonal elements are complex, so we have to store nine values in memory to represent the coherence matrix.

We divide the data set into p parts. Each process will read n/p pixels from $\boldsymbol{T}^*$ in the hard disk and store them in memory as shown in Fig. 2, where $\boldsymbol{T_i}$ indicates the data of a group of pixels handled by ith process.

2) Computation of Eigenvalues and Eigenvectors: As the computation in each pixel is independent from the rest, each process computes the eigenvalues and eigenvectors of all its pixels one by one. Each process will read a pixel from its subset, compute the eigenvalues and eigenvectors and store the results in main memory as shown in Fig. 3. For each process, the structure of the eigenvectors is an array of $(n/p) \times 9$ contiguous elements. The three values of each eigenvector are stored in contiguous positions in memory, and the three eigenvectors associated with a coherence matrix are also stored contiguously. The same happens with the structure of the eigenvalues, which is an array of $(n/p) \times 3$ contiguous elements. The three eigenvalues associated with a coherence matrix are stored in contiguous positions. This configuration is efficient because

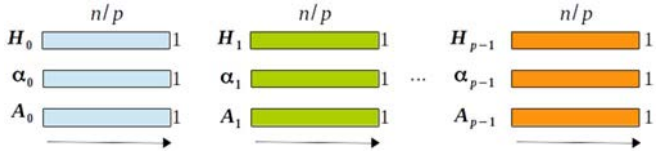

Fig. 4. Illustration of computation of H, α, and A in each process.

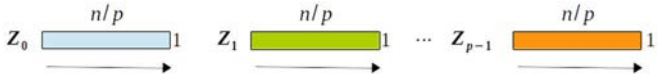

Fig. 5. Illustration of target decomposition computation.

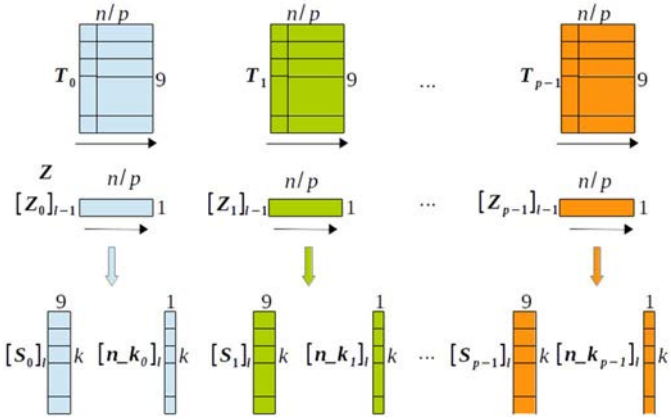

Fig. 6. Illustration of the computation of partial sums and number of pixels in each zone.

it allows us to store and read the values from contiguous positions, thus making a better use of the cache memory.

3) Computation of H, α, and A: Again the computation of the variables of interest (H, α, and A) for each pixel is independent from the rest. Thus, each process will use the eigenvalues and eigenvectors previously computed to compute these variables for all pixels one by one. The results will be stored in main memory as shown in Fig. 4. $\boldsymbol{H}$, $\boldsymbol{\alpha}$, and $\boldsymbol{A}$ structures are arrays of (n/p) contiguous elements.

4) Target Decomposition: Each process will classify its pixels into an $\mathrm{H} - \alpha$ plane divided into nine zones. The result for each process is an array $\boldsymbol{Z}$ with integer values from 1 to 9 (labels). In memory, these values are stored as shown in Fig. 5.

5) Wishart Classifier: The method is divided into three steps.

Using the previous classification, each process performs the sum (partial sum) of all its pixels in each zone, and computes how many pixels are in each zone of H-alpha plane. The results are stored in two structures $\boldsymbol{S_i}$ and $\boldsymbol{n\_k_i}$ as shown in Fig. 6.

The element in row j of $\boldsymbol{S_i}$ corresponds with the addition of all the pixels in $\boldsymbol{T_i}$ that belong to the zone j. Element j of $\boldsymbol{n\_k_i}$ corresponds to the number of pixels in $\boldsymbol{T_i}$ belonging to the zone j. After computing partial sums, a gather operation is performed to combine all $\boldsymbol{S_i}$ and $\boldsymbol{n\_k_i}$ structures into bigger structures in process 0 (we will call them $\boldsymbol{S}^*$ and $\boldsymbol{n\_k}^*$). With these data, process 0 can now compute the mean of each zone ($\boldsymbol{Vm_i}$), which is the centroid of each zone. Also $\boldsymbol{Vm_i}^{-1}$ and $\boldsymbol{det_i}$ are computed in process 0. These results are stored in

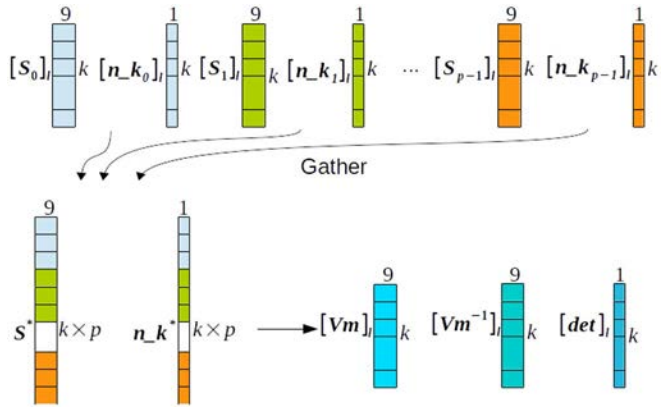

Fig. 7. Illustration of centroids computation on process 0.

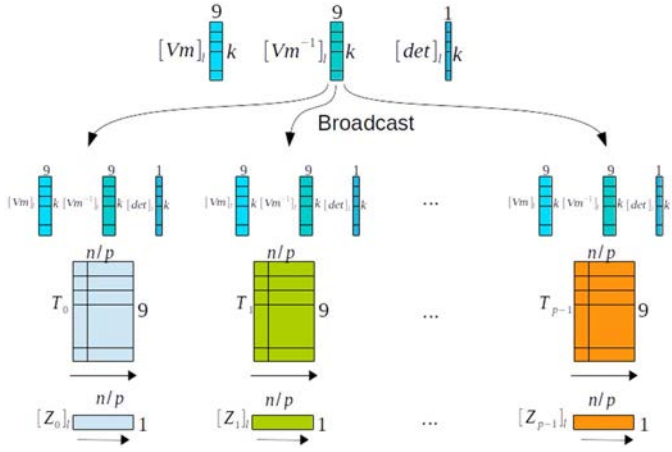

Fig. 8. Illustration of the broadcast operation to send centroids to each process and update new classes.

$[\boldsymbol{Vm}]_l$, $\left[\boldsymbol{Vm}^{-1}\right]_l$, and $[\boldsymbol{det}]_l$, where l means the lth iteration in K-means, as illustrated in Fig. 7.

Finally, a broadcast operation is performed to send $[\boldsymbol{Vm}]_l$, $\left[\boldsymbol{Vm}^{-1}\right]_l$, and $[\boldsymbol{det}]_l$ to each process. Thus, the distance of each pixel to each centroid can be calculated, and the zone associated with each pixel can be updated in $\boldsymbol{Z_i}$ as shown in Fig. 8.

B. GPU Implementation

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

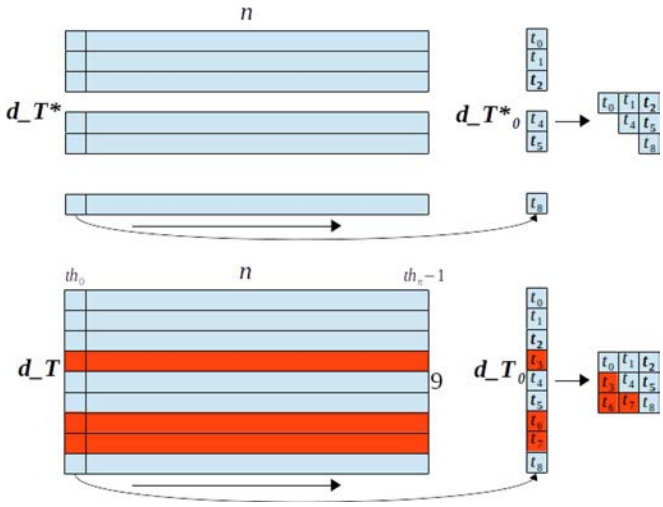

Fig. 9. Data representation of the coherence matrix in the global memory of the GPU.

or more streams as inputs and producing one or more streams as outputs. Thereby, data-level parallelism is exposed to hardware, and kernels can be concurrently applied without any sort of synchronization. The kernels can perform a kind of batch processing arranged in the form of a grid of blocks, where each block is composed by a group of threads that share data efficiently through the shared local memory and synchronize their execution for coordinating accesses to memory.

With the aforementioned general ideas in mind, the proposed GPU implementation for terrain unsupervised classification algorithm using POLSAR works as shown in Algorithm 2. The same number of threads as the number of pixels is used in our implementation, and the computation of the ith pixel is performed by the ith thread.

1) Read Coherence Matrix: The host first reads the upper triangular coherency matrix from hard disk and sends it to device global memory $\boldsymbol{d\_T}$; then each thread builds each own full coherence matrix as shown in Fig. 9. The arrow indicates how the elements are stored in memory. Values in same pixel are separated by n positions in memory. To improve the transfer bandwidth between host and device, a comparison using pinned and pageable memory was performed. The conclusion of our study was that pinned memory is faster than pageable memory for the size of our dataset (576 MB). Specifically, for pageable transfers, the host to device bandwidth (measured in GB/s) was 4.26 and the device to host bandwidth (measured in GB/s) was 3.98. On the other hand, for pinned transfers, the host to device bandwidth (measured in GB/s) was 6.12 while the device to host bandwidth (measured in GB/s) was 6.54.

2) Computation of Eigenvalues and Eigenvectors: The computation of eigenvalues and eigenvectors in each pixel is independent from the rest. So, the computation of each pixel can be assigned to one thread in the GPU. For that purpose, we maximize the number of threads per block, having a total number of threads th equal to the number of pixels n. The process starts allocating each pixel in local memory in the registers of the MPs. Due to the way the data are stored in memory (values

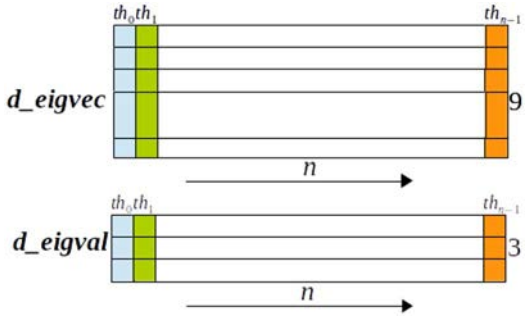

Fig. 10. Illustration of how eigenvalues and eigenvectors are stored in global memory.

in same pixel are separated by n positions in memory), we make use of coalescence while reading the data, i.e., we are parallelizing the access to global memory. Once each pixel is loaded, the ith thread performs the computation with the ith pixel and saves the result back in global memory, as illustrated in Fig. 10. In the calculation of the eigenvalues and eigenvectors, the memory access is strided because each element of the coherence matrix is separated in memory by n positions. This has an impact in performance because, in order to read the complete coherence matrix, we need to jump in the memory and we are not taking advantage of cache memory. Since the access to the data is a small part of the eigenvalue and eigenvector calculation, this impact is small. The contiguous memory access can be more efficient than strided memory, but this is the best configuration for the GPU since it allows us to make use of coalesced access.

Algorithm 2. Pseudo-code of the GPU parallel implementation of POLSAR unsupervised classification algorithm.

$d\_^*$ means data are stored in global memory of the GPU ('d' of device)
$h\_^*$ means data are stored in main memory of CPU ('h' of host)
$n \rightarrow$ number of pixels
$\boldsymbol{T}_{9\times n} \rightarrow$ Coherence matrix (each pixel has 9 elements)
$th \rightarrow$ number of threads
$k \rightarrow$ number of classes
$it \rightarrow$ number of iteration in Wishart classifier

```
1  h_T ← Read_Coherence();
2  d_T ← cudaMemcpy(h_T);
3  th ← n;
4  parallel for i = 0 to th − 1
5     d_eigvec_i, d_eigval_i ← Kernel_Eigenv(d_T);
6     d_H_i, d_α_i ← Kernel_H_alpha(d_eigvec_i, d_eigval_i);
7     d_Z_i ← Kernel_Targ_Decomp(d_H_i, d_α_i);
8     for l = 0 to it
9        [d_Z_i]_l ← Kernel_Wishart_Class(d_T_i, [d_Z_i]_{l−1});
10    end for
11 end parallel for 10
12 h_Z ← cudaMemcpy(s_Z);
```

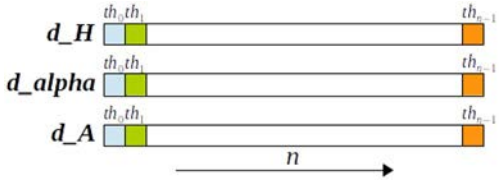

Fig. 11. Illustration of how H, α, and A are stored in global memory.

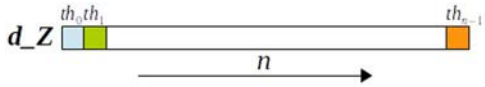

Fig. 12. Illustration of how each element classified by target decomposition is stored in global memory by each thread.

3) Computation of H, α, and A: The number of threads per block is maximized while computing H, α, and A. The ith thread reads the ith eigenvalue from $\boldsymbol{d\_eigval}$ and ith eigenvectors from $\boldsymbol{d\_eigvec}$ and performs the calculation of H, α, and A. Then, it saves the results in ith position of $\boldsymbol{d\_H}$, $\boldsymbol{d\_alpha}$, and $\boldsymbol{d\_A}$ arrays in global memory. Fig. 11 shows how these structures are stored in global memory.

4) Target Decomposition: For the target decomposition step, we again maximize the number of threads per block. ith thread reads ith element from $\boldsymbol{d\_H}$, $\boldsymbol{d\_alpha}$ and the label of the zone in H $-$ α plane that pixel belongs to is recorded in $\boldsymbol{d\_Z}$ (see Fig. 1). Fig. 12 shows how the result is stored in global memory.

5) Wishart Classifier: The computation of the Wishart classifier comprises three kernels: 1) GPU_kmeans1_kernel; 2) GPU_kmeans2_kernel; and 3) GPU_kmeans3_kernel. In the following, we describe these kernels in detail.

a) GPU_kmeans1_kernel: The total number of threads is set to 1024 and the number of blocks to 16 (i.e., 64 threads per block). The use of all the MPs in the GPU (a total of 14) is guaranteed by this configuration, and we will have high occupancy in the MPs because we have 32 cores in each one and we are using 64 threads. Each thread will compute partial sums for each zone of a subset of pixels (in this case, $\frac{n}{1024}$ pixels). Also, each thread will compute the number of pixels in each zone. Each pixel that the ith thread reads is separated in memory by 1024 positions. In this way, the consecutive positions in memory are accessed with each consecutive read operation. The partial sum $\boldsymbol{S_i}$ and the number of pixels in each zone $\boldsymbol{n\_k_i}$ are stored in local memory and updated at every iteration. This process is shown in Fig. 13.

Finally, the partial results computed by each thread are stored in global memory as illustrated in Fig. 14.

b) GPU_kmeans2_kernel: In this kernel, the number of threads is reduced to 128. We use k threads to compute the total number of elements in each zone. The ith thread reads the ith row from $[\boldsymbol{d\_n\_k}]_{\boldsymbol{l}}$ and saves the result in local memory. Then $9 \times k$ threads are used to compute total sum in each zone, reading data from $[\boldsymbol{d\_S}]_{\boldsymbol{l}}$, and saving the results in global memory. At this point, we have the sum of all the pixels in each zone and the total number of pixels in each zone. Finally,

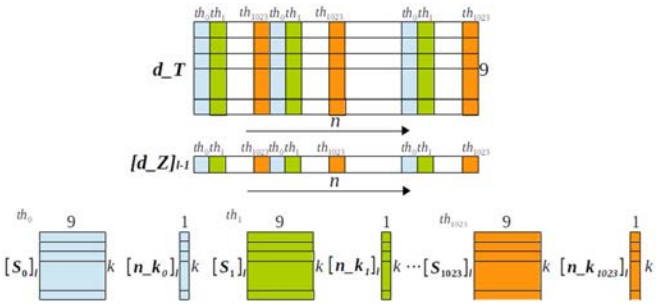

Fig. 13. Illustration of how the partial sums and the number of pixels in each zone are computed by each thread in each iteration.

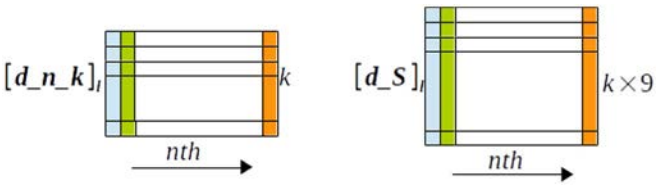

Fig. 14. Illustration of how the partial additions and the number of pixels in each zone are stored in global memory in each iteration.

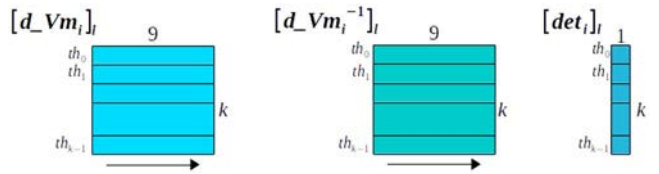

Fig. 15. Illustration of how $[\boldsymbol{d\_Vm_i}]_{\boldsymbol{l}}$, $\left[\boldsymbol{d\_Vm_i}^{-1}\right]_{\boldsymbol{l}}$, and $[\boldsymbol{det_i}]_{\boldsymbol{l}}$ structures are store by the threads in global memory.

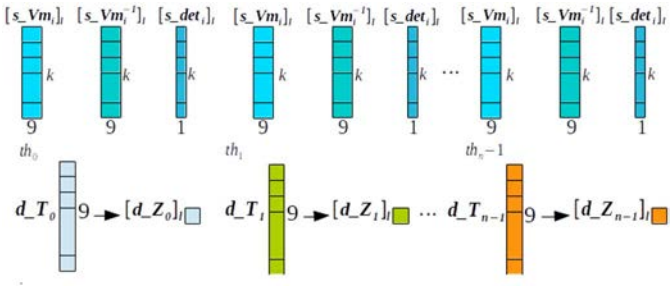

Fig. 16. Illustration of how the class labels of pixels are updated.

we use k threads to perform the computation of $[\boldsymbol{d\_Vm_i}]_{\boldsymbol{l}}$, $\left[\boldsymbol{d\_Vm_i}^{-1}\right]_{\boldsymbol{l}}$, and $[\boldsymbol{det_i}]_{\boldsymbol{l}}$, and save them in global memory as shown in Fig. 15.

c) GPU_kmeans3_kernel: In this kernel, the distance of each pixel to each centroid is computed and the class label of each pixel is updated depending on the closest centroid. We maximize the number of threads per block, and we use as many threads as the number of pixels in the image. First step consists of copying $[\boldsymbol{d\_Vm_i}]_{\boldsymbol{l}}$, $\left[\boldsymbol{d\_Vm_i}^{-1}\right]_{\boldsymbol{l}}$, and $[\boldsymbol{det_i}]_{\boldsymbol{l}}$ to shared memory of each block, as they will be used by all the threads in the kernel. Then, the ith thread reads ith pixel from $\boldsymbol{d\_T}$, measures the distances with each centroid, and updates $[\boldsymbol{d\_Z}]_{\boldsymbol{l}}$. This process is shown in Fig. 16.

TABLE I
HARDWARE SPECIFICATION OF A NODE IN THE CETA-CIEMAT CLUSTER

CPU	2 × Hexacores Core Intel Xeon E5649 2.53 GHz (12 cores, hyperthreading disabled)
Memory	24 GB SDRAM DDR3 1333MHz ECC (6 × 4 GB of 12 slots used, pair positions)
Hard disk (operating system)	1 TB SATA
Storage (data)	Lustre parallel distributed filesystem
GPU	2 TESLA M2075

TABLE II
SPECIFICATIONS OF TESLA M2075 GPU

Form factor	9.75″ PCIe ×16
# of CUDA cores	448
Frequency of CUDA cores	1.15 GHz
Double precision floating point perf.	515 Gflops
Single precision floating point perf.	1.03 Gflops
Total dedicated memory	6 GB GDDR5
Memory speed	1.5 GHz
Memory interface	384-bit
Memory bandwidth	144 GB/s
Power consumption	238 W
System interface	PCIe ×16 Gen2
Compute capability	2.0

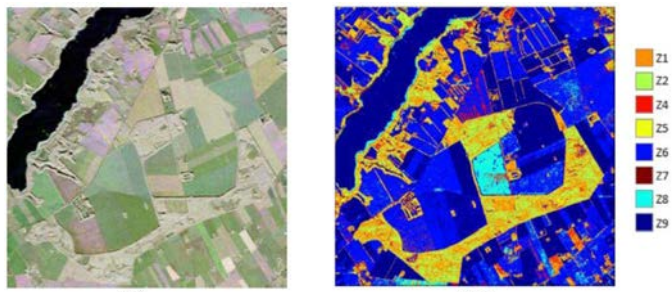

Fig. 17. Foulum agricultural image in Jutland, Denmark, 1024 × 1024 pixels.

C. Hardware Specification

This section describes the hardware resources used for parallel implementation. The high-performance computing cluster at Extremadura Research Centre for Advanced Technologies (CETA-CIEMAT) was employed. The cluster is a GPU cluster with more than 100,000 GPU cores. We used just a portion of it, composed by four bullx R424 E2 (four nodes per bullx), which is a total of 16 nodes. Technical specifications of each node, including CPU model,[3] memory, etc., are shown in Table I. A total of 192 cores are available for processing. All cluster nodes are interconnected by Infiniband QDR (40 Gbps) and mounts a distributed parallel filesystem using Lustre[4] for user data. For the implementation of the CPU version, we use one core of one node and for the MPI version, we use up to 128 cores. The specification of the Tesla M2075 GPU is given in Table II.

[3][Online]. Available: http://ark.intel.com/es-es/products/52581/
[4][Online]. Available: http://opensfs.org/lustre/

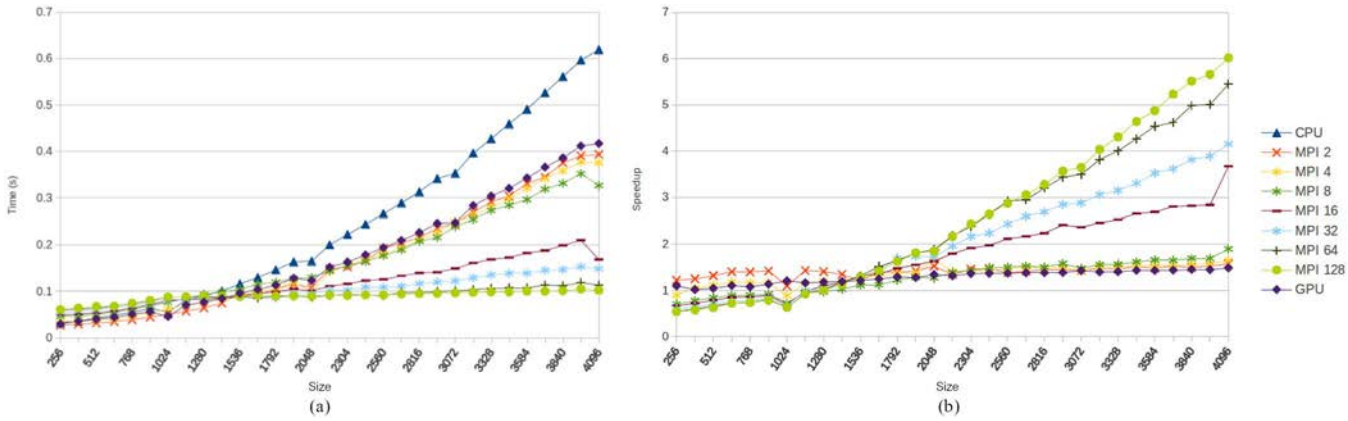

Fig. 18. (a) Execution times for loading the data into the memory. (b) Speedups achieved by the cluster and a single GPU in comparison with the CPU.

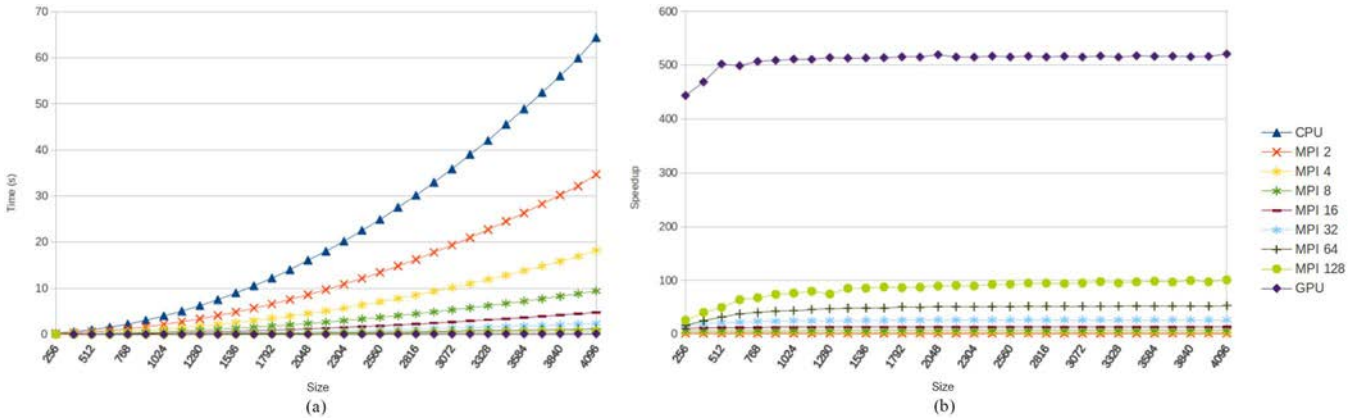

Fig. 19. (a) Execution times for computation of eigenvalues and eigenvectors. (b) Speedups achieved by the cluster and a single GPU in comparison with the CPU.

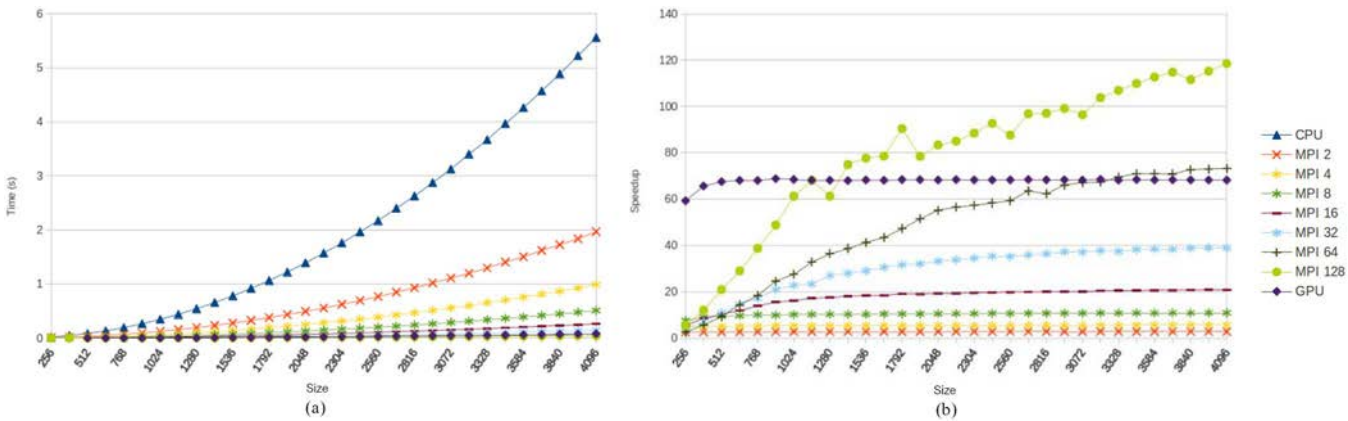

Fig. 20. (a) Execution times for computation of H, α, and A. (b) Speedups achieved by the cluster and a single GPU in comparison with the CPU.

IV. Experimental Results

In this study, we use a real POLSAR data set acquired by the fully polarimetric Danish airborne SAR system, EMISAR, which operates at two frequencies, C-band (5.3 GHz/5.7 cm wavelength) and L-band (1.25 GHz/24 cm wavelength), respectively. The data are obtained from the POLSAR Pro website.[5] The data are acquired over Foulum, Denmark. The nominal one-look spatial resolution is 2 m $\times$ 2 m; the ground range swath is approximately 12 km and typical incidence angles range from 35° to 60°. The processed data from this system are fully calibrated by using an advanced internal calibration system. The effective spatial resolution is approximately 8 m $\times$ 8 m at midrange. Fig. 17(a) shows a false color composite of the image.

In our experiments, we execute all the steps for terrain unsupervised classification algorithm (i.e., eigenvalue and eigenvector computation, H, α, and A computation, target

[5][Online]. Available: https://earth.esa.int/web/polsarpro/airborne-data-sources

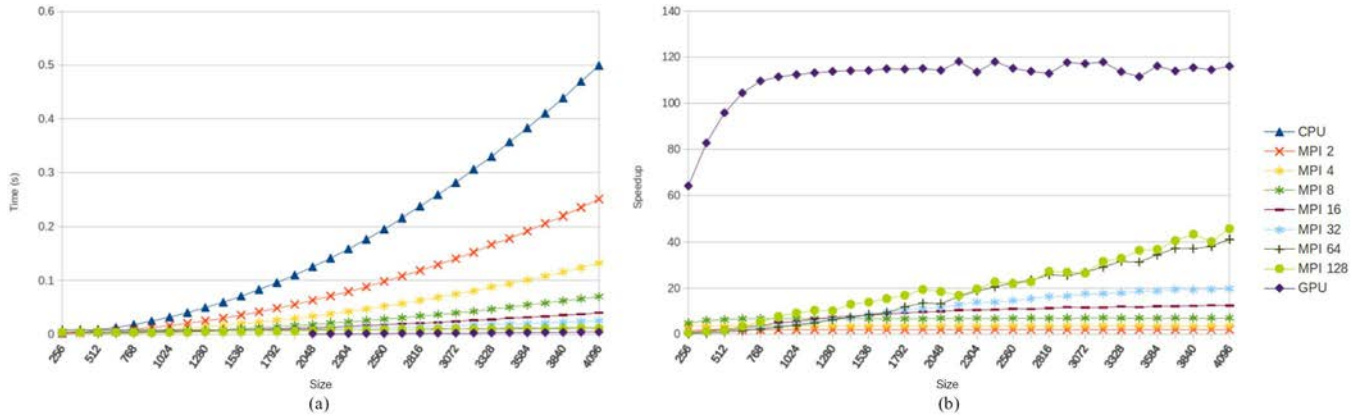

Fig. 21. (a) Execution times for computation of target decomposition. (b) Speedups achieved by the cluster and a single GPU in comparison with the CPU.

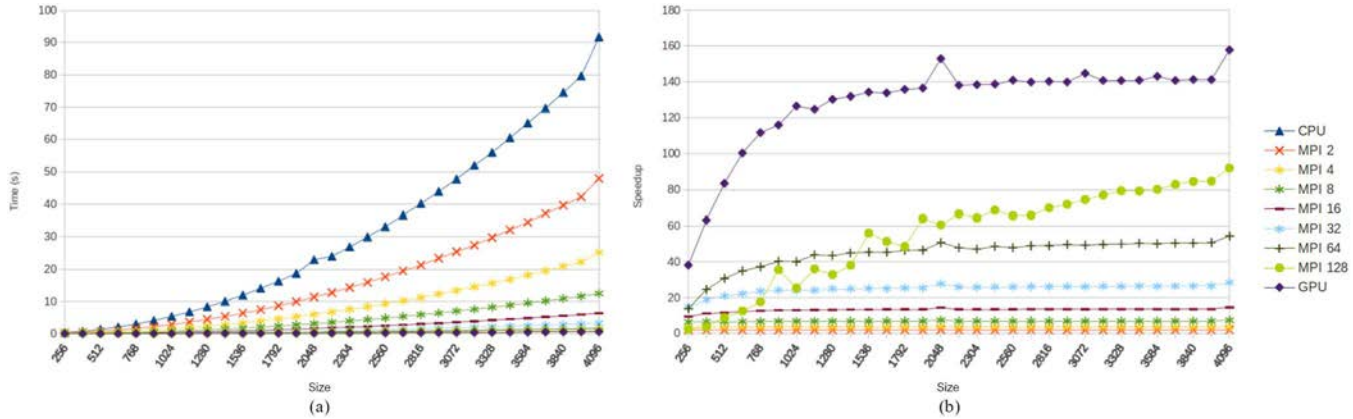

Fig. 22. (a) Execution times for computation of Wishart classifier. (b) Speedups achieved by the cluster and a single GPU in comparison with the CPU.

decomposition, and complex Wishart classifier) using the three different architectures mentioned previously (CPU, cluster, and GPU). For the MPI version, we execute the processing chain using different number of cores (2, 4, 8, 16, 32, 64, and 128). Also, we measure the execution times for different sizes of our dataset (from 256×256 up to 4096×4096 by increasing the size of width and height by 256 each time). For sizes bigger than 1024×1024, we replicate the image to create a big image. To measure the time without bias, each individual experiment was repeated 50 times and the average of the times was computed. We used profiling tools such as NVIDIA Visual Profiler to measure the performance of the different GPU kernels and discover possible performance or memory issues. All the implementations were compiled in the CentOS 6.5.

Operating system using gcc compiler version 4.4.7 and release 6.0, V6.0.1 of CUDA compilation tools.

A. Results

The efficiency of the parallel implementation is estimated in terms of the speedup achieved in comparison with the processing using a single node. The computational time of loading the data in the memory, eigenvalue decomposition, computation of H, α, and A values, target decomposition, and Wishart classification is recorded separately and the speedups achieved at all these stages are estimated. To achieve a consistent comparison, all the reported times are calculated as the average of 50 executions of the same process. To measure the time of a stage using more than one process, we keep the time stamp of all the processes at the beginning and after the end, and then we compute the difference between the maximum and the minimum of all the recorded time stamps.

The results of the time taken for loading the data in the cluster, the CPU, and the GPU for different configurations are shown in Fig. 18. It can be seen that no speedup is achieved for small data sets. However, very high speedups are achieved for large datasets.

Figs. 19 and 20 show the execution time and speedups achieved for the computation of eigenvalues and eigenvectors, and the computation of H, α, and A, respectively. Figs. 21 and 22 show the execution time and speedups achieved for the computation of target decomposition and Wishart classifier methods. The total execution time for the processing chain and the speedups achieved are shown in Fig. 23. It can be clearly seen that a significant speedup is achieved in the computation of the eigenvalues and eigenvectors. The speedups are not necessarily equal to number of cores due to the fact that different processes start and end at different time stamps. It can be observed that near real-time performance is achieved in the cluster even for large datasets. For a dataset of size 4096×4096 pixels, the estimated computation time for the entire processing chain is around 1.79 s for the MPI version

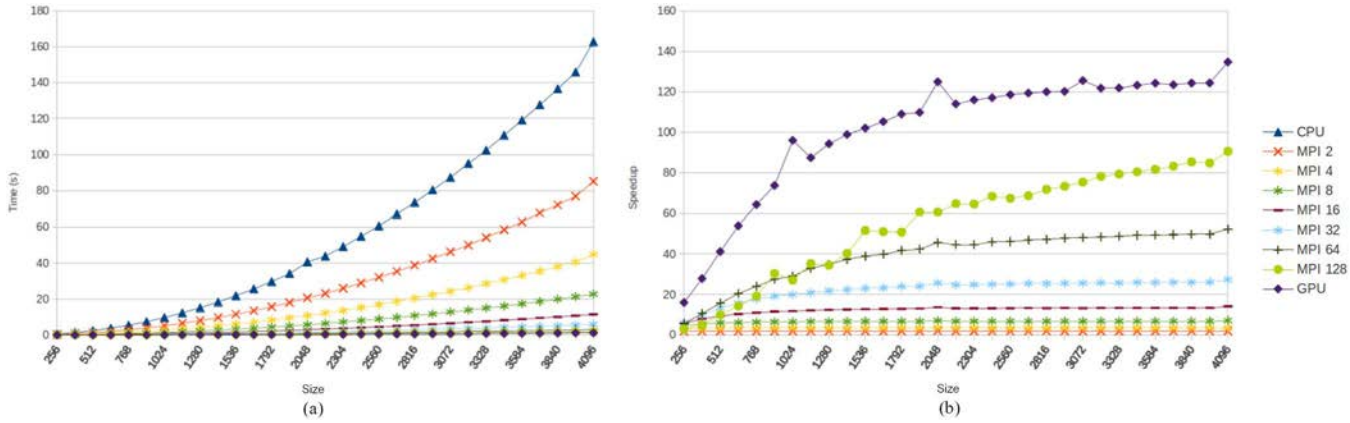

Fig. 23. (a) Total execution times for the complete chain. (b) Speedups achieved by the cluster and a single GPU in comparison with the CPU.

TABLE III
TOTAL EXECUTION TIME AND SPEEDUPS FOR IMAGE SIZES OF 256×256, 512×512, 1024×1024, 2048×2048, AND 4096×4096 ACHIEVED WITH THE THREE IMPLEMENTATIONS (CPU, MPI 128, AND GPU)

SIZE	CPU	MPI 128	GPU
	TIME (S)		
256	0.63	0.21	0.03
512	2.46	0.24	0.05
1024	9.73	0.35	0.10
2048	40.67	0.67	0.32
4096	162.76	1.79	1.20
	SPEEDUP		
256	1	2.91	16.04
512	1	10.04	41.25
1024	1	27.11	96.00
2048	1	60.54	125
4096	1	90.56	134.79

using 128 cores and 1.20 s for the GPU version. Table III shows the total execution time and speedups for image sizes of 256×256, 512×512, 1024×1024, 2048×2048, and 4096×4096 achieved with the three implementations (CPU, MPI 128, and GPU).

It is very interesting to note that a single GPU performs better or similar when compared with MPI execution using 128 cores. However, GPUs cannot handle very large datasets due to memory limitations. It can also be noted that there are local peaks in performance for image sizes of 1024, 2048, and 4096, which represent powers of 2. Additionally, Fig. 24 shows a pie chart for the three different implementations in which the percentage of the total execution time achieved in each step can be seen. It can be seen that most of the time, the algorithm is performed computation rather than memory operations. In this sense, the algorithm is compute-bound. The shared memory size configuration was set to cache L1 (as a preference) for all the kernels. This means that we have a larger cache size (49 152 bytes) and smaller shared memory size (16 384 bytes). Different kernels have different configurations and different usage of resources. According to this, the occupancy of the GPU for each kernel can be summarized in Table IV. However, it has to be noted that higher occupancy does not necessarily mean high performance.

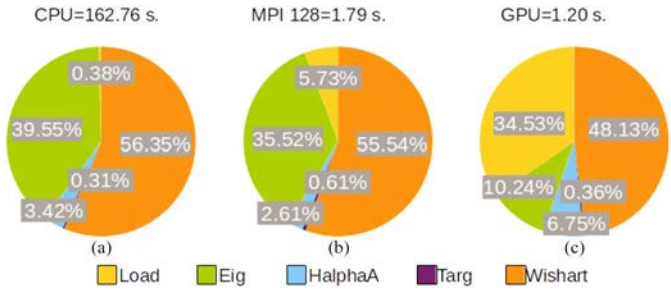

Fig. 24. Percentage of time of each step in the processing chain for the three different implementations. (a) CPU. (b) MPI 128. (c) GPU.

TABLE IV
PERCENTAGE OF OCCUPANCY FOR EACH KERNEL IN THE GPU

Kernel name	Occupancy (%)
buildCoherence	67
GPU_eig3x3	33
GPU_makeHAlphaA	67
GPU_targDecomp	100
GPU_kMeans1	33
GPU_kMeans2	50
GPU_kMeans3	33

The CPU and cluster versions produce exactly the same result and the results are exact (except for 10 pixels) in the GPU version. NVIDIA GPUs differ from the $\times 86$ architecture in the fact that rounding modes are encoded within each floating-point instruction instead of dynamically using a floating-point control word. As a result, different math libraries cannot be expected to compute exactly the same result for a given input. This applies to GPU programming as well: functions compiled for the GPU will use the NVIDIA CUDA math library implementation while functions compiled for the CPU will use the host compiler math library implementation (e.g., *glibc* on Linux). Since these implementations are independent and none of them guaranteed to be correctly rounded, the results may differ slightly.

As the result of the implemented method is a classification map, some of the pixels may lay very near to the border of two classes. A small change in the values produced by the precision may result in the classification of a pixel to a different

class. However, this may only happen for a very small number of pixels as compared to the total number of pixels in the scene.

V. Conclusion and Future Work

In this work, we have developed a near real-time parallel implementation of the target decomposition algorithm for processing POLSAR data. Two parallel versions of the algorithm were developed. The first one using MPI on a cluster, and the second one using the CUDA on a GPU. The performance achieved for a single GPU is comparable in terms of processing time with regard to using 128 nodes in a cluster of computers for large datasets, and is also significantly better for small datasets. The bottleneck of the GPU implementation is the loading time, as the data need to be stored in the CPU memory first and then moved to the global GPU memory. In the future, we are planning on extending this work by developing a FPGA implementation and comparing the obtained results in terms of power consumption to fully calibrate the possibility to exploit these implementations onboard remote sensing instruments for Earth observation.

Acknowledgment

The authors would like to thank the Associate Editor and the two Anonymous Reviewers who evaluated this paper by their outstanding comments and suggestions, which greatly helped to improve the quality of this paper.

References

[1] S. R. Cloude, *Polarisation: Applications in Remote Sensing*. London, U.K.: Oxford Univ. Press, 2010.

[2] J. S. Lee and T. L. Ainsworth, "An overview of recent advances in polarimetric SAR information extraction: Algorithms and applications," in *Proc. IEEE Int. Geosci. Remote Sens. Symp. (IGARSS)*, 2010, vol. 1, pp. 851–854.

[3] E. Christophe, J. Michel, and J. Inglada, "Remote sensing processing: From multicore to GPU," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 4, no. 3, pp. 643–652, Sep. 2011.

[4] C. Gonzalez, S. Sanchez, A. Paz, J. Resano, D. Mozos, and A. Plaza, "Use of FPGA or GPU-based architectures for remotely sensed hyperspectral image processing," *Integr. VLSI J.*, vol. 46, no. 2, pp. 89–103, Mar. 2013.

[5] S. Sanchez, R. Ramalho, L. Sousa, and A. Plaza, "Real-time implementation of remotely sensed hyperspectral image unmixing on GPUs," *J. Real-Time Image Process.*, vol. 10, no. 3, pp. 469–483, 2012 [Online]. Available: http://link.springer.com/article/10.1007%2Fs11554-012-0269-2

[6] S. Sanchez and A. Plaza, "Fast determination of the number of endmembers for real-time hyperspectral unmixing on GPUs," *J. Real-Time Image Process.*, vol. 9, no. 3, pp. 397–405, Sep. 2014.

[7] J. M. P. Nascimento, J. M. Bioucas-Dias, J. M. Rodriguez Alves, V. Silva, and A. Plaza, "Parallel hyperspectral unmixing on GPUs," *IEEE Geosci. Remote Sens. Lett.*, vol. 11, no. 3, pp. 666–670, Mar. 2014.

[8] S. Bernabe, S. Lopez, A. Plaza, and R. Sarmiento, "GPU implementation of an automatic target detection and classification algorithm for hyperspectral image analysis," *IEEE Geosci. Remote Sens. Lett.*, vol. 10, no. 2, pp. 221–225, Mar. 2013.

[9] X. Li, B. Huang, and K. Zhao, "Massively parallel GPU design of automatic target generation process in hyperspectral imagery," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 8, no. 6, pp. 2862–2869, Jun. 2015.

[10] G. Martin, J. Bioucas-Dias, and A. Plaza, "HYCA: A new technique for hyperspectral compressive sensing," *IEEE Trans. Geosci. Remote Sens.*, vol. 53, no. 5, pp. 2819–2831, May 2015.

[11] M. A. Hossam, H. M. Ebied, and M. H. Abdel-Aziz, "Hybrid cluster of multicore CPUs and GPUs for accelerating hyperspectral image hierarchical segmentation," in *Proc. Int. Conf. Comput. Eng. Syst. (ICCES)*, 2013, vol. 1, pp. 262–267.

[12] J. Mielikainen, B. Huang, and H. A. Huang, "GPU-accelerated multi-profile radiative transfer model for the infrared atmospheric sounding interferometer," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 4, no. 3, pp. 691–700, Sep. 2011.

[13] H. Zhu, Y. Cao, Z. Zhou, and M. Gong, "Parallel multi-temporal remote sensing image change detection on GPUs," in *Proc. IEEE Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, 2012, vol. 1, pp. 1898–1904.

[14] F. Giannesini and B. Le Saux, "GPU-accelerated one-class SVM for exploration of remote sensing data," in *Proc. IEEE Geosci. Remote Sens. Symp. (IGARSS)*, 2012, vol. 1, pp. 7349–7352.

[15] T. Chi, H. Chen, X. Zhang, Q. Wang, C. Chen, and Y. Lu, "Agent communication based SAR image parallel processing," in *Proc. IEEE Int. Geosci. Remote Sens. Symp. (IGARSS)*, Sep. 20–24, 2004, vol. 6, pp. 3923–3925, doi: 10.1109/IGARSS.2004.1369984.

[16] M. Cafaro, I. Epicoco, S. Fiore, D. Lezzi, S. Mocavero, and G. Aloisio, "Near real-time parallel processing and advanced data management of SAR images in grid environments," *J. Real-Time Image Process.*, vol. 4, no. 3, pp. 219–227, 2009.

[17] H. Zhili, H. Chao, and L. Hongping, "Parallel processing imaging algorithm for synthetic aperture radar based on pipeline," in *Proc. Nat. Conf. Inf. Technol. Comput. Sci. (CITCS'12)*, Oct. 2012, pp. 516–519.

[18] A. Goller, "Parallel processing strategies for large SAR image data sets in a distributed environment," *Computing*, vol. 62, no. 4, pp. 277–291, 1999.

[19] W. Chapman *et al.*, "Parallel processing techniques for the processing of synthetic aperture radar data on GPUs," in *Proc. IEEE Int. Symp. Signal Process. Inf. Technol. (ISSPIT)*, Dec. 2011, pp. 573–580.

[20] A. Castillo Atoche, R. Carrasco Alvarez, J. Ortegón Aguilar, and J. Vázquez Castillo, "A new tool for intelligent parallel processing of radar/SAR remotely sensed imagery," *Math. Prob. Eng.*, vol. 2013, 10 p., 2013, doi: 10.1155/2013/405372.

[21] S. R. Cloude and E. Pottier, "An entropy based classification scheme for land applications of polarimetric SAR," *IEEE Trans. Geosci. Remote Sens.*, vol. 35, no. 1, pp. 68–78, Jan. 1997.

[22] J. S. Lee, M. R. Grunes, T. L. Ainsworth, L. J. Du, D. L. Schuler, and S. R. Cloude, "Unsupervised classification using polarimetric decomposition and the complex Wishart classifier," *IEEE Trans. Geosci. Remote Sens.*, vol. 37, no. 5, pp. 2249–2258, Sep. 1999.

[23] J. J. van Zyl and F. T. Ulaby, "Scattering matrix representation for simple targets," in *Radar Polarimetry Geoscience Applications*, F. T. Ulaby and C. Elachi, Eds. Norwood, MA, USA: Artech House, 1990, ch. 2, pp. 17–52.

Sergio Sánchez (M'14) received the Ph.D. degree in computer engineering from the University of Extremadura, Cáceres, Spain, in 2013.

He is currently a Ph.D. Research Associate with the Department of Chemical and Environmental Engineering, Masdar Institute of Science and Technology, Abu Dhabi, UAE. His research interests include hyperspectral image analysis and efficient implementations of large-scale scientific problems on commodity graphical processing units (GPUs).

Prashanth R. Marpu (M'04–SM'14) received the M.Sc. degree in wireless engineering from the Technical University of Denmark, Kongens Lyngby, Denmark, in 2006 and the Ph.D. degree in interdisciplinary from the Freiberg University of Mining and Technology, Freiberg, Germany, in 2009.

He was a Marie Curie Fellow with the University of Pavia, Pavia, Italy, and was working as a Senior Researcher with the University of Iceland, Reykjavik, Iceland. He is currently an Assistant Professor with Masdar Institute of Science and Technology, Abu Dhabi, UAE. His research interests include machine learning, image processing, environmental data analysis, high-performance computing, remote sensing, and GIS.

Antonio Plaza (M'05–SM'07–F'15) was born in Cáceres, Spain, in 1975. He received the B.Sc., M.Sc., and the Ph.D. degrees in computer engineering from the University of Extremadura, Cáceres, Spain, in 1997, 1999, and 2002, respectively.

He is an Associate Professor (with accreditation for Full Professor) with the Department of Technology of Computers and Communications, University of Extremadura, where he is the Head of the Hyperspectral Computing Laboratory (HyperComp), one of the most productive research groups working on remotely sensed hyperspectral data processing worldwide. He has been the advisor of 12 Ph.D. dissertations and more than 30 M.Sc. dissertations. He was the Coordinator of the Hyperspectral Imaging Network. He has authored more than 500 publications, including 152 journal papers (more than 100 in IEEE journals), 22 book chapters, and over 240 peer-reviewed conference proceeding papers (94 in IEEE conferences). He has edited a book on *High-Performance Computing in Remote Sensing* (CRC Press/Taylor and Francis) and guest edited 9 special issues on hyperspectral remote sensing for different journals. He has reviewed more than 500 manuscripts for over 50 different journals. His research interests include hyperspectral data processing and parallel computing of remote sensing data.

Dr. Plaza is an Associate Editor of the IEEE Access. He was a member of the Editorial Board of the IEEE GEOSCIENCE AND REMOTE SENSING NEWSLETTER (2011–2012) and the IEEE GEOSCIENCE AND REMOTE SENSING MAGAZINE (2013), and also a member of the steering committee of the IEEE JOURNAL OF SELECTED TOPICS IN APPLIED EARTH OBSERVATIONS AND REMOTE SENSING (JSTARS). He has served as an Associate Editor for the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING from 2007 to 2012, the Director of the Education Activities for the IEEE Geoscience and Remote Sensing Society (GRSS) from 2011 to 2012, and is currently serving as a President of the Spanish Chapter of the IEEE GRSS (since November 2012) and Editor-in-Chief of the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING JOURNAL. He has also served as a Proposal Evaluator for the European Commission, the National Science Foundation, the European Space Agency, the Belgium Science Policy, the Israel Science Foundation, and the Spanish Ministry of Science and Innovation. He was the recipient of the recognition of the Best Reviewers of the IEEE GEOSCIENCE AND REMOTE SENSING LETTERS (in 2009) and the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING (in 2010), the Best Paper Award of the JSTARS journal, and also the recipient of the most highly cited paper (2005–2010) in the *Journal of Parallel and Distributed Computing*, Best Paper Awards at the IEEE INTERNATIONAL CONFERENCE ON SPACE TECHNOLOGY and the IEEE Symposium on Signal Processing and Information Technology, and the Best Ph.D. Dissertation Award at the University of Extremadura. Additional information: http://www.umbc.edu/rssipl/people/aplaza

Abel Paz-Gallardo received the B.S. degree in computer engineering in 2007 and the M.Sc. and Ph.D. degrees in computer science from the University of Extremadura, Cáceres, 2009 and 2011, respectively.

In 2010, he was working with the Bull Spain S.A. to deploy at CETA-CIEMAT, Trujillo, Spain. Since November 2010, he has been an Associate Professor with the Department of Computer Architecture, University of Extremadura. Since October 2011, he has been the IT Manager (CIO) of CETA-CIEMAT. His research interests include remotely sensed hyperspectral image analysis, signal processing, and efficient implementations of large-scale scientific problems on high-performance computing architectures such as clusters and graphical processing units (GPUs).