J Real-Time Image Proc (2018) 14:753–761
https://doi.org/10.1007/s11554-017-0703-6

ORIGINAL RESEARCH PAPER

Parallel real-time virtual dimensionality estimation for hyperspectral images

Emanuele Torti[1] · **Alessandro Fontanella**[1] · **Antonio Plaza**[2]

Received: 30 January 2017 / Accepted: 30 June 2017 / Published online: 6 July 2017

Abstract One of the most important tasks in hyperspectral imaging is the estimation of the number of endmembers in a scene, where the endmembers are the most spectrally pure components. The high dimensionality of hyperspectral data makes this calculation computationally expensive. In this paper, we present several new real-time implementations of the well-known Harsanyi–Farrand–Chang method for virtual dimensionality estimation. The proposed solutions exploit multi-core processors and graphic processing units for achieving real-time performance of this algorithm, together with better performance than other works in the literature. Our experimental results are obtained using both synthetic and real images. The obtained processing times show that the proposed implementations outperform other hardware-based solutions.

Keywords Virtual dimensionality (VD) · Graphics processing units (GPUs) · Multi-core CPUs · Hyperspectral imaging · Real-time processing

1 Introduction

Hyperspectral images comprise hundreds of contiguous spectral bands [1]. Each pixel can be represented as an L-dimensional vector, where L is the number of bands. There are different techniques to determine the content of a hyperspectral image; one of the most popular ones is the linear mixing model. It assumes that each pixel can be modeled as a linear combination of a finite set of spectrally pure constituents (endmembers). In a hyperspectral image, there are generally two kinds of pixels: pure and mixed. The pure pixels contain only one pure signature corresponding to an endmember, while the mixed pixels are given by a superimposition of different endmembers. Typically, the number of endmembers in a scene is significantly lower than the number of bands [2]. The identification of the number of endmembers is a crucial issue in hyperspectral imaging, since it allows representing the data in a lower dimensional subspace. This is a very important aspect, due to the benefits that can be achieved in computational time, data storage and complexity. Moreover, a good estimation is fundamental for different applications, such as classification, target detection and spectral unmixing. In the literature, different algorithms can be found to perform this estimation, including objective function minimization [3], adjacent bands correlation [4], and a minimum squared error approach [5].

A widely used technique is the estimation of the virtual dimensionality (VD) of a hyperspectral image [6]. A commonly used algorithm for this computation is the Harsanyi–Farrand–Chang (HFC) method [7], which is based on the Neyman–Pearson detection theory [8].

The data amount provided by modern hyperspectral sensors makes the development of real-time algorithms a desired goal for hyperspectral imaging. In the literature, several high-performance computing (HPC) architectures have been used for achieving this goal, such as field programmable gate arrays (FPGAs) [9], multi-core CPUs [10], supercomputers and clusters [11], GPUs [12] [13] or even hybrid solutions [14].

✉ Emanuele Torti
emanuele.torti@unipv.it

[1] Department of Electrical, Computer and Biomedical Engineering, University of Pavia, Pavia, Italy

[2] Department of Technology of Computers and Communications, Escuela Politecnica de Caceres, University of Extremadura, Caceres, Spain

In this paper, we propose several new parallel real-time implementations for the computation of the VD through the HFC method. The algorithm is described in Sect. 2, while Sect. 3 contains details about our implementations. The first is based on OpenMP API for multi-core devices, while the second exploits GPU parallelism using the popular NVIDIA compute unified device architecture (CUDA) and its basic linear algebra subroutines (CuBLAS). We also describe an OpenCL implementation that can be executed on both multi-core CPUs and GPUs. Finally, we also evaluated Python, since this language is gaining importance in the scientific community. Section 4 describes the conducted tests and the experimental results. Section 5 gives exhaustive results analysis and provides comparisons with other works in literature. We tested this implementation with both synthetic and real data, obtaining real-time performance, which also outperforms other VD implementations [15] and other previously developed techniques for estimating the number of endmembers [10]. Those issues are discussed in Sect. 6, which concludes this paper and provides hints at plausible future research lines.

2 HFC-VD algorithm

The HFC-VD algorithm has been widely used for computing the number of endmembers within a scene. Let us denote a hyperspectral pixel with $\mathbf{y} = [r^1, r^2, \ldots, r^L]$, where r^i is the reflectance value acquired on the i-th band; a hyperspectral image is given by $\mathbf{Y} = [\mathbf{y}^1, \mathbf{y}^2, \ldots, \mathbf{y}^N]$ where N is the number of acquired samples. This method is based on the eigenvalues computation of the sample data correlation matrix (CM) and the covariance matrix (VM), defined as follows:

$$\mathrm{CM} = \frac{\left(\mathbf{Y}^T * \mathbf{Y}\right)}{N-1} \tag{1}$$

$$\mathrm{VM} = \frac{(\mathbf{Y} - \bar{\mathbf{Y}})^T * (\mathbf{Y} - \bar{\mathbf{Y}})}{N} \tag{2}$$

where $\bar{\mathbf{Y}}$ is the zero-mean hyperspectral image. It is important to notice that those two matrices are $L \times L$-dimensional. The next step is the computation of the eigenvalues of the CM and the VM. Each matrix has L eigenvalues, so it is possible to say that each eigenvalue is related to a specific band. In the following, we will denote the i-th eigenvalue of CM with λ_i^{CM} and the i-th eigenvalue of VM with λ_i^{VM}. The HFC method assumes that the presence of a signal source in the i-th band is indicated by the positive difference between λ_i^{CM} and λ_i^{VM}. The number of signal sources (or endmembers) is computed by counting, for each band, how many times the following condition is verified:

$$\lambda_i^{\mathrm{CM}} - \lambda_i^{\mathrm{VM}} \geq 0 \tag{3}$$

The HFC method tests, for each band, how many times the condition given by Eq. 3 fails for a given false alarm probability (P_{fa}).

Algorithm 1 shows the pseudocode of the HFC-VD algorithm.

In Algorithm 1, steps 1–2 compute the sample data correlation matrix and covariance matrix, respectively. After that, there is the eigenvalues computation (steps 3–4). For each band, the variance is computed (step 7) and the threshold is computed from P_{fa}. This threshold prevents the detection of false positives and is obtained by the normal cumulative distribution function. The solution of the integral equation of step 8 (x) is the tolerance for a false positive. The value is used for the hypothesis test (step 10); if the test is positive, the number of endmember is incremented (step 11).

Algorithm 1 – HFC-VD

INPUT: $\mathbf{Y} = [\mathbf{y}^1, \mathbf{y}^2, \ldots, \mathbf{y}^N], P_{fa}$

1: $\mathrm{CM} = \frac{(\mathbf{Y}^T * \mathbf{Y})}{N-1}$;

2: $\mathrm{VM} = \frac{(\mathbf{Y}-\bar{\mathbf{Y}})^T * (\mathbf{Y}-\bar{\mathbf{Y}})}{N}$;

3: $\lambda^{\mathrm{CM}} = eig(\mathrm{CM})$; {compute the eigenvalues of CM}

4: $\lambda^{\mathrm{VM}} = eig(\mathrm{VM})$; {compute the eigenvalues of VM}

5: $dim = 0$; {initialize the number of endmembers}

6: **for** i:=1 **to** L **do**

7: $\sigma_i \cong \sqrt{\frac{2}{N}(\lambda_i^{\mathrm{CM}})^2 + \frac{2}{N}(\lambda_i^{\mathrm{VM}})^2}$;

8: solve $P_{fa} = \frac{1}{\sigma_i\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{z_i^2}{2\sigma_i^2}} dz_i$ to find x;

9: $diff = \lambda_i^{\mathrm{CM}} - \lambda_i^{\mathrm{VM}}$;

10: **if** $diff > x$ **then**

11: $dim = dim + 1$;

12: **end if**

13: **end for**

OUTPUT: dim

3 Algorithm implementations

In this section, we describe the different implementations of the HFC-VD algorithm, together with some algorithmic choices for carrying out some processing steps, such as the computations in steps 3–4 of Algorithm 1. Moreover, we will motivate our design choices such as the part of the code that must be executed by the CPU and the GPU.

3.1 Code profiling

First, we generate a set of synthetic hyperspectral images for profiling the code and identifying the most time-

consuming computations. Those images size ranges from 1.7 to 427 MB and have been generated using the MATLAB script developed by Nascimento et al. [5] for algorithm testing. The endmember signatures contained in the images are randomly extracted from the United States Geological Survey (USGS) digital spectral library (https://speclab.cr.usgs.gov/spectral-lib.html). We use those images as input for the MATLAB version of HFC-VD contained in the Endmember Induction Algorithms Toolbox for MATLAB. After code profiling, we found that the most computationally expensive parts were the correlation and covariance matrices computation, which took about 90% of the execution time (which is about 25 s for an image of about 140 MB).

It is important to notice that all of those computations can be parallelized. The high data dimensionality suggests exploiting a massively parallel processor, such as the GPUs. Moreover, the data dimensionality, involved into those computations, should be taken into account. The matrices CM and VM are of dimensions $L \times L$ where L is the number of bands. Typically, a hyperspectral image is made up of more than 200 bands, so the CM and VM matrices are about 310 KB each. The amount of data on which the next step of the algorithm will work is not big. The use of a GPU for extracting eigenvalues from those matrices will degrade performance, since there is a low degree of data parallelism. For this reason, the best choice is to transfer the results obtained by the GPU (i.e., VM and CM matrices) to the CPU and identify suitable algorithms for performing the other operations as fast as possible. Another critical issue that we considered is the precision of results. In particular, our tests show that a single precision floating point arithmetic is not sufficient for a correct calculation. Those tests have been conducted with the synthetic dataset, since we have a priori knowledge of the number of endmembers contained in each image. Single precision floating point arithmetic is suitable only for the smallest images (till 50 MB), while it causes an underestimation of the virtual dimensionality with bigger images. Since the double precision floating point arithmetic does not suffer this issue, it has been adopted.

3.2 OpenMP implementation

OpenMP is a paradigm for exploiting shared memory systems. It is based on a set of directives that indicates to the compiler the code parts to parallelize. In our work, we chose to parallelize the routines for computing the CM and the VM matrices, which are the most time-consuming steps. In particular, we used the *#pragma omp parallel for* statement on the outer *for* cycle for the matrix–matrix multiplication. We also specified which variables must be stored in the shared memory and which ones in the private memory, using suitable options. Finally, we also chose to divide loop iterations into equal-size chunks, by specifying the *static* schedule mode.

As already discussed, the next steps work on reduced data size, so it is more convenient to perform them serially. For achieving better performances, we considered different approaches for eigenvalues computation, including SVD-based techniques and Householder reductions. We conducted simulation with different techniques, identifying the one with the best tradeoff between precision and processing speed.

In particular, we used the method described in [16]. The computation is performed by two subroutines: *tred2* and *tqli*. The first one performs a Householder reduction of the input matrix. The result is a tridiagonal matrix. This matrix is given as input to the *tqli* function, which performs a QL algorithm with implicit shifts for determining eigenvalues and eigenvectors. Since we need only the eigenvalues, we removed the instruction, from both the subroutines, for eigenvectors computation.

3.3 CUDA implementation

For the GPU implementation, we chose to use NVIDIA CUDA. In this paradigm, the GPU (called *device*) is seen as a coprocessor, with private memory, capable of executing hundreds or even thousands of threads in parallel. The execution of a CUDA application always starts from a traditional CPU (called *host*). When the execution reaches a parallel code section, the host transfers data to the device memory, through the PCI-express bus. After that, the device starts its computation and, when its elaboration ends, results are transferred back to the host.

As already said, the most intensive calculations are the computation of Eqs. (1) and (2), which are the first steps of the algorithm. The host transfers the input image to the device memory. To compute Eqs. (1) and (2), we used the CuBLAS library [17], which provides highly optimized kernels for linear algebra routines. In particular, we used the *cublasSgemm* routine, which computes the expression:

$$C = \alpha op(A) op(B) + \beta op(C) \tag{4}$$

where A and B are the input matrices, C is an input/output matrix, and α and β are scalar values. The notation $op()$ indicates the possibility of reading the matrix as transpose or not. It is important to notice that if the scalar value β is equal to zero, the matrix addition is not performed and the matrix C is considered only as an output parameter. To implement Eqs. (1) and (2), we do not need matrix addition, so we set β to zero.

The scalar value α has been set to $1/(N-1)$ and to $1/N$ to compute CV and VM, respectively. An important issue to deal with when using CuBLAS library is that it uses column-major indexing instead of row-major

indexing. In our implementation, developed in CUDA C language, we use the standard array storing of C language, which is row-major. Thus, in order to convert a row-major matrix into a column-major matrix, the transposition operation is used. The input matrices are converted into the column-major format by the *cublasDgemm* function by properly setting the *op*() parameters. On the other hand, the results are also stored into column-major format, but the next steps require row-major format. We do not need to convert the storage format this time, since the output matrices are both symmetric and, in this specific case, it is not necessary to perform the conversion. Results are now transferred back to the host for the next computations. Before computing Eq. (2), it is necessary to compute the zero-mean image. This is done by using a CuBLAS function for summing up all the samples of each band (*cublasDasum*). It is important to notice that this function allows storing the result in the host memory or in the device memory. This choice is specified by the function *cublasSetPointerMode*; in our case, we choose to store the results in the device memory, to avoid unnecessary memory transfers. At this point, in the GPU memory, there is an array that stores the sum of each element of each band. After that, a custom CUDA kernel computes the mean of each band and another custom kernel carries out the computation of the zero-mean image. Figure 1 depicts the steps of the CUDA implementation, clearly pointing out memory transfers.

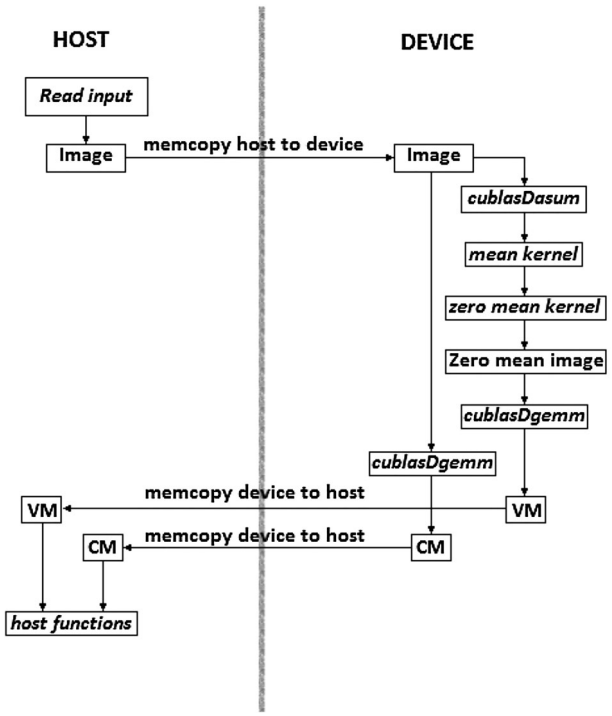

Fig. 1 CUDA implementation diagram. Functions are indicated in italic, while data instances are indicated in normal font

3.4 OpenCL implementation

OpenCL is a framework for heterogeneous computing. Similarly to CUDA, it adopts a model where a host processor executes the sequential code parts, while the intrinsically parallel code sections are processed by a device. In this scenario, a device is, typically, a GPU (not necessarily developed by NVIDIA) or a multi-core CPU. The approach adopted for this implementation is the same of CUDA, since we initially transfer the image to the device memory and perform the computation for obtaining the CM and VM matrices. It is important to notice that we cannot use the CuBLAS library in this case and we chose to write OpenCL kernels by hand. This choice is related to the fact that we want to develop a code that is as portable as possible to different OpenCL platform. However, for obtaining better performance on different devices, we needed to develop two different OpenCL codes. The first one targets GPUs platform and uses OpenCL APIs that allow data transfers from the host memory to the device memory. The latter targets multi-core devices and GPUs that share memory with the host. In this scenario, it is not necessary to transfer data, since it is possible to share data stored in the unified memory. Therefore, we chose to use OpenCL APIs that do not cause a data transfer, but automatically share data pointers between host and device. We highlight that the steps performed on the device are conceptually the same described for the CUDA implementation. OpenCL is a framework that makes of portability the major point of strength, so we decide not performing device-specific optimizations.

3.5 Python implementation

Finally, we carried out parallel versions through the popular Python language. The first one exploits the NumPy package [18], while the latter exploits NVIDIA GPUs through the PyCuda package [19]. NumPy contains routines for both matrix multiplication and eigenvalues computation. NumPy is based on BLAS library, so it automatically exploits multi-core processors. This drastically improves code development times, since our code simply calls pre-built functions.

For what concerns PyCuda, the Reikna package [20], which contains highly optimized GPU algorithms, has been used to perform the matrix multiplication. We used these functions for carrying out the most expensive code parts. In addition, in this case, the code development was faster, since all the necessary functions are available in the packages.

However, it is important to highlight that, as discussed in the next section, Python and PyCuda performance are lower than CUDA.

4 Experimental results

The set of synthetic images, used for profiling the MATLAB code of HFC-VD, has also been used to validate the code. In particular, we developed a serial C version, to serve as a fair comparison with the OpenMP, CUDA, OpenCL and Python applications. It is important to highlight that the MATLAB reference code and all our versions produce the same results for all the images (both real and synthetic ones). The real images used in our testing phase are a Hydice image collected over a forest (referred to as Hydice in the following), five AVIRIS scenes, two ROSIS images and one EO-1 scene. The five AVIRIS images are: the well-known Cuprite mining district, the World Trade Center, the Kennedy Space Center, the Indian Pines over the test site of Indiana and a Salinas Valley scenes (referred to in the following as Cuprite, WTC, KSC, Indian Pines and Salinas, respectively). The ROSIS images are acquired over the center of the Pavia city in Italy and over the Pavia University (referenced as PaviaC and PaviaU in the following). The EO-1 image was acquired over Okavango Delta, Botswana (referenced as Botswana in the following). Moreover, we used two images, obtained by joining multiple Indian Pines images (indicated in the following as Indian Pine_EW-line_R and Indian Pine_NS-line).

The number of pixels in the synthetic images ranges from 2,000 to 500,000 with the number of bands fixed to 224. The number of pixels in the real images ranges from 4096 to 1,024,000 with a variable number of bands (from 102 to 224).

Our tests have been conducted on two different PCs. The first one (PC1) is equipped with an Intel i7 3770 processor (4 physical cores) working at 3.40 GHz, with 8 GB of DDR3 RAM. The processor is connected to an NVIDIA Tesla K40 GPU through a PCI-express 2.0 bus. The GPU is equipped with 2880 CUDA cores working at 875 MHz, with 12 GB of DDR5 RAM and a bandwidth of 288 GB/s. The second PC (PC2) is equipped with an Intel i7 6700 processor working at 3.40 GHz, with 32 GB of DDR4 RAM. The processor also integrates a graphics module, called Intel HD Graphics 530 Skylake GT2 equipped with 192 shade units working at 300 MHz. The processor is also connected to an NVIDIA GTX 1060 GPU through a PCI-express 3.0 bus. The GPU has 1152 cores working at 1.5 GHz, with 3 GB of DDR5 RAM and a bandwidth of 192 GB/s. All the features of those two systems are summarized in Table 1.

We developed and tested all the versions (both serial and parallel) under 64-bit Microsoft Windows 10. We compiled the code using the *vc140* compiler under Microsoft Visual Studio 2015 integrated development environment. The CUDA environment 8.0 is used for the GPU version. Concerning Python, we used the 3.5 version.

We use optimization flags such as*/Ox* to perform advanced optimization and we included the SSE2 advanced instruction dataset for maximizing serial code execution speed. Moreover, we enable the OpenMP 2.0 support for the parallel CPU version using the flag*/openmp*. We measured execution times using suitable functions such as *ompgetwtime* for the parallel version of the C code. We tested the serial, OpenMP, CUDA, OpenCL, Phyton and PyCuda versions on PC2, while we did not test OpenCL on PC1 since it has an old hardware that does not fully support this paradigm.

Tables 2 and 3 report the processing time for real images obtained with PC1 and PC2, respectively. The speedup compared to the serial implementation is indicated between brackets.

Table 4 shows the processing times obtained by the OpenCL implementation considering different devices. It must be noted that the reported times are obtained considering the best OpenCL implementation for the considered device (the Intel i7 6700 CPU referenced as CPU, the Intel HD Graphics 530 referenced as GPU1 and the NVIDIA GTX 1060 referenced as GPU2).

For what concerns synthetic images, the processing times are shown in Fig. 2 using a logarithmic scale. For clarity of interpretation, we only show results related to the best implementations, compared with the serial one. In all the cases, the reported times are obtained as the mean of multiple execution, with a standard deviation less than 1%.

5 Discussion

In this section, we conduct an exhaustive analysis and comparisons between the proposed versions, together with the works available in the literature. We also describe a

Table 1 Systems features

	PC1		PC2		
	Intel i7 3770	Tesla K40	Intel i7 6700	HD graphics 530	GTX 1060
Cores number	4	2880	4	192	1152
RAM (GB)	8	12	32	–	3
Working frequency (GHz)	3.40	0.875	3.40	0.300	1.5

Table 2 Processing times for real images on PC1

Image	Size (MB)	Serial (s)	OpenMP (s)	CUDA (s)	Python (s)	PyCuda (s)
Hydice	2.64	0.197	0.098 (2.01×)	0.023 (8.52×)	0.032 (6.17×)	0.103 (1.91×)
Indian pines	17.64	1.596	0.711 (2.24×)	0.062 (25.74×)	0.270 (5.9×)	0.133 (12.0×)
Cuprite	34.24	2.605	1.035 (2.52×)	0.062 (40.01×)	0.499 (5.22×)	0.180 (14.47×)
PaviaU	81.49	3.401	2.954 (1.51×)	0.077 (44.17×)	0.624 (5.45×)	0.244 (13.93×)
Salinas	94.94	8.639	4.528 (1.90×)	0.165 (52.36×)	1.304 (6.625×)	0.296 (29.18×)
Botswana	209	13.628	11.677 (1.17×)	0.200 (68.14×)	2.419 (5.63×)	0.474 (28.75×)
KSC	211.06	16.734	13.689 (1.22×)	0.233 (71.82×)	2.642 (6.33×)	0.510 (32.81×)
WTC	268.63	27.071	23.082 (1.17×)	0.358 (75.62×)	4.878 (5.54×)	0.671 (40.34×)
PaviaC	398.44	22.542	17.654 (1.28×)	0.345 (65.34×)	3.431 (6.57×)	0.802 (28.11×)
Indian pines-EW_lineR	952.25	97.290	91.154 (1.07×)	2.740 (35.51×)	16.693 (5.82×)	3.240 (30.03×)
Indian pines-NS_line	1379.94	141.184	127.080 (1.11×)	7.631 (18.50×)	26.485 (4.95×)	9.182 (15.38×)

Table 3 Processing times for real images on PC2

Image	Size (MB)	Serial (s)	OpenMP (s)	CUDA (s)	Phyton (s)	PyCuda (s)
Hydice	2.64	0.289	0.082 (3.52×)	0.035 (8.26×)	0.041 (7.05×)	0.120 (2.41×)
Indian pines	17.64	2.360	0.627 (3.76×)	0.119 (19.83×)	0.363 (6.50×)	0.208 (11.35×)
Cuprite	34.24	3.880	0.857 (4.53×)	0.130 (29.85×)	0.506 (7.67×)	0.239 (16.23×)
PaviaU	81.49	4.941	2.083 (2.37×)	0.226 (21.86×)	0.734 (6.73×)	0.354 (13.96×)
Salinas	94.94	13.077	3.124 (4.19×)	0.389 (33.62×)	1.853 (7.06×)	0.480 (27.24×)
Botswana	209	17.548	7.635 (2.30×)	0.534 (32.86×)	2.568 (6.83×)	0.756 (23.21×)
KSC	211.06	21.235	8.961 (2.37×)	0.520 (40.84×)	2.952 (7.19×)	1.183 (17.95×)
WTC	268.63	36.068	16.013 (2.25×)	0.995 (36.25×)	5.288 (6.82×)	1.107 (32.58×)
PaviaC	398.44	24.133	11.783 (2.05×)	0.838 (28.80×)	3.628 (6.65×)	1.258 (19.18×)
Indian pines-EW_lineR	952.25	121.513	65.889 (1.84×)	2.991 (40.63×)	18.558 (6.55×)	3.522 (34.50×)
Indian pines-NS_line	1379.94	176.023	91.472 (1.92×)	7.838 (22.46×)	27.508 (6.40×)	8.203 (21.46×)

Table 4 OpenCL processing times

Image	GPU2 (s)	GPU1 (s)	CPU (s)
Hydice	0.194	0.147	0.170
Indian pines	0.573	0.386	0.595
Cuprite	0.976	0.937	0.924
PaviaU	1.258	2.441	3.546
Salinas	3.280	3.441	3.539
Botswana	4.514	4.585	5.357
KSC	5.585	5.632	6.181
WTC	8.310	8.632	9.091
PaviaC	6.225	6.800	7.240
Indian pines-EW_lineR	30.622	31.504	37.420
Indian pines-NS_line	49.247	52.821	54.380

careful profiling carried out for the CUDA version, which is the one that reaches the best performances in terms of processing times.

It is important to highlight that a typical hyperspectral sensor acquires about 140 MB of data in <5 s, so this is the threshold for real-time processing.

As shown in Tables II, III and IV, the CUDA implementation on the NVIDIA Tesla K40 board achieves the best performance. On this board, the PyCuda implementation achieves higher processing times than the CUDA one. This is due to the fact that the Reikna library is not as optimized as CuBLAS; moreover, Python has a higher level of abstraction and memory transfers are managed by the functions and not directly by the programmer. The Hydice image is an exception for what concerns Python and PyCuda performance. This is the only image where Python is faster than PyCuda. This is due to the fact that NumPy has different optimizations with respect to Reikna, so it is more performant on smaller images. It is important to notice that this behavior is only shown with the Hydice image, which is the smallest one of the dataset.

The OpenMP version does not take advantage of the multi-core CPU as NumPy does. However, it is important

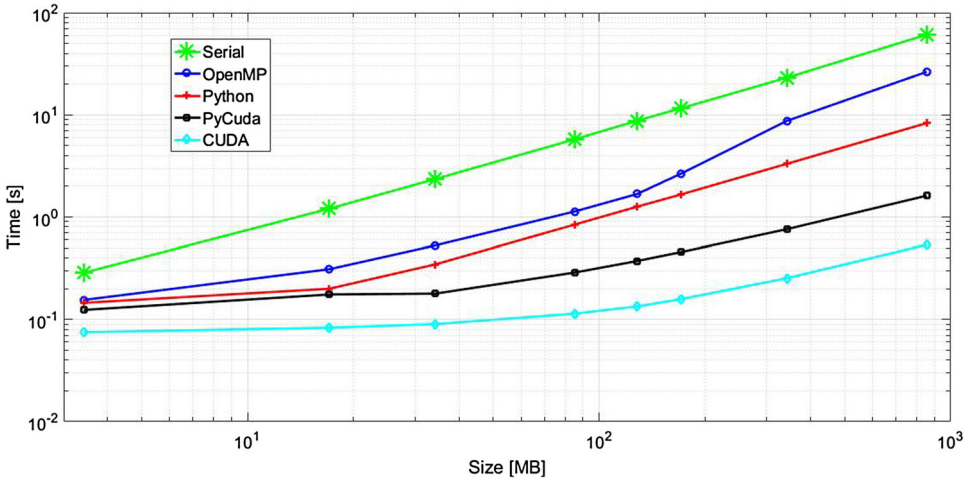

Fig. 2 Processing times for the synthetic images

to highlight that NumPy is based on highly optimized parallel functions; instead, the OpenMP version uses routines written by hand. Moreover, higher computational times of OpenMP are because the parallelization of the code is performed through directives, so at a high level. The parallelization is then left to the compiler. In our case, we are using vc140 compiler, which supports OpenMP 2.0, so it is not possible to use advanced instructions such as nested loops. Concerning Python, we used NumPy which is an optimized library for linear algebra operations.

The OpenCL version performs worse than the other, except OpenMP. The explanation is that the kernel is the same for all the devices, without device-specific optimizations. This choice is due to the fact that we want to develop an OpenCL code as portable as possible.

By analyzing Table II, it is possible to notice that OpenMP, CUDA, and Python implementations show lower computation time for the PaviaC image than for WTC image. However, PyCuda implementation shows opposite. This behavior is due to the different optimization of Reikna package used for the PyCuda version. The two images have a different number of bands and number of pixels (102×1024000 for PaviaC and 224×314368 for WTC) so their geometry is different.

Our time execution analysis highlights that the proposed CUDA implementation outperforms the other versions. It is important to notice that this version is also real-time compliant. For example, WTC acquisition time is about 5 s, while we process it in about 0.358 s. On the other hand, the two synthetic Indian Pines images are five and six times bigger than WTC, respectively, so it is possible to assume that they are acquired in about 30 s. Also under this hypothesis the proposed implementation is real-time compliant. Moreover, for the two biggest images, the speedup is of about one order of magnitude with respect to serial and OpenMP implementation. CUDA also outperforms Python based versions.

We also perform code profiling using NVIDIA Visual Profiler, for characterizing the different activity performed while using the GPU. For both considered GPUs, we found that the memory transfers took a time that ranges, in percentage, from 30 to 10%, while the effective computation took from 70 to 90%. Memory transfers have bigger impact on the total processing time when considering smaller images, since the time needed to transfer the data is close to the effective computation time.

We chose to evaluate two different platforms (PC1 and PC2) in order to characterize the performance of two different kinds of GPUs. The NVIDIA Tesla K40 is based on the Kepler architecture and has been designed for scientific computations. Instead, the NVIDIA GTX 1060 is a more modern GPU, based on the Pascal architecture, and is a general purpose device, not optimized for scientific applications.

In the literature, to the best of our knowledge, the most recent works about real-time estimation of the number of endmembers are [10, 15, 21].

Authors of [15] proposed a FPGA-based implementation of HFC-VD algorithm. They used a Xilinx Virtex 7 XC7VX690T FPGA. The designed architecture works at the frequency of 75 MHz. Experiments have been conducted using the Cuprite and the WTC images. For Cuprite, the processing took about 1.64 s, while for WTC it took 4.26 s. Our implementation outperforms this one, since, for the same computation, we need 0.062 and 0.358 s, respectively. It is important to notice that NVIDIA Tesla K40 was released in 2013, while Xilinx Virtex 7 in 2011. Moreover, they both are high-end devices. Finally, nowadays, Virtex 7 is still the top performing FPGA produced by Xilinx, while Tesla K40 has been replaced by more modern GPUs (such as NVIDIA P100). For what concerns the Virtex 7 implementation, details are provided by the authors of reference [15] together with the description of the optimizations performed on this device. In our case, we

chose different optimizations; since GPU architecture is different from FPGA one, so optimization performed in [15] are not suitable in our case.

The speedup of our implementation with respect to the work proposed in [15] is about 26 for the Cuprite and 4.5 for the WTC. It is important to notice that both works are real-time compliant. However, for what concerns the WTC image, the constraint of processing the image in less than 5 s (required for real-time performance) is respected but with a smaller margin than our implementation. This is a critical issue, since the estimation of the number of endmembers is often the first step for further processing steps such as hyperspectral unmixing or hyperspectral data compression.

Regarding the work proposed in [10], the number of endmembers estimation is performed using hyperspectral subspace identification with minimum error (HySime). Authors evaluated different technologies: multi-core CPUs and DSPs and GPUs. The best performance is obtained using the Automatically Tuned Linear Algebra Software (ATLAS) library, which exploited multi-core CPUs. They reported results for the Cuprite image, whose processing took about 0.549 s. Our implementation is about 8 times faster than the one proposed in [10]. This is due to the fact that HFC-VD algorithm has lower complexity than HySime, but also to the solutions carried out in our implementation for achieving real-time elaboration (i.e., CuBLAS library and eigenvalues computation method).

For what concerns the work reported in [21], authors introduced both VD and HySime implementation on an NVIDIA GTX 580 GPU. Experimentation has been conducted on the Cuprite and WTC images. The reported times are of 0.43 s and 1.07 for the VD algorithm and of 1.29 and 2.81 s for HySime. In both cases, we outperform the implementations proposed in [21]. Moreover, it is important to highlight that authors of [21] adopted single precision floating point arithmetic, while we adopted double precision floating point arithmetic, in order to provide accurate results even using bigger images. As already said in Sect. 3.1, our tests show that single precision floating point arithmetic is not adequate for images bigger than 50 MB.

6 Conclusion

In this paper, we present several new parallel implementations of the well-known HFC-VD algorithm for estimation of endmembers number. The best version exploits NVIDIA CUBLAS using CuBLAS library for performing general matrix multiplication operations, which are the most time-consuming code parts. We presented a code profiling that motivates our design choices, i.e., which computations to perform on the GPU and which ones to perform on CPU. Our tests have been conducted using both synthetic and real images.

Our obtained results show the effectiveness of our implementation, that is real-time compliant.

This is a very important contribution, since it allows executing the estimation of endmembers (an important step for hyperspectral unmixing) in computationally efficient form.

We also compared our work with recent ones [10], [15], [21], showing that our implementation is faster than those proposed in these works. In particular, we showed that our GPU implementation outperforms other implementations based on FPGAs and multi-core CPUs concerning the estimation of the number of endmembers.

Future works will be focused on developing further algorithms on GPUs and building fully customizable analysis chains, such as the one proposed in [22]. Moreover, we will exploit multi-GPU systems for allowing fast and complex analysis of hyperspectral images, such as unmixing, classification and compression.

Acknowledgements The authors gratefully thank NVIDIA Corporation for the donation of the GPU Tesla K40 used for this research.

References

1. Lillesand, T.M., Kiefer, R.W., Chipman, J.W.: Remote Sensing and Image Interpretation, 5th edn. Wiley, Hoboken (2004)
2. Keshava, N., Mustard, J.: Spectral unmixing. IEEE Signal Process. Mag. **19**(1), 44–57 (2002)
3. Scharf, L.L.: Statistical Signal Processing, Detection Estimation and Time Series Analysis. Addison-Wesley, Boston (1991). **Reading**
4. Chang, C.-I., Wang, S.: Constrained band selection for hyperspectral imagery. IEEE Trans. Geosci. Remote Sens. **44**(6), 1575–1585 (2006)
5. Nascimento, J.M.P., Bioucas-Dias, J.M.: Hyperspectral subspace identification. IEEE Trans. Geosci. Remote Sens. **46**(8), 1445–2435 (2008)
6. Chang, C.-I.: Hyperspectral Imaging: Techniques for Spectral Detection and Classification. Kluwer/Plenum Academic Publishers, New York (2003)
7. Harsanyi, J.C., Farrand, W., Chang, C.-I.: Detection of subpixel spectral signatures in hyperspectral image sequences, In: Proceedings of the American Society for Photogrammetry and Remote Sensing, pp. 236–247. Reno (1994)
8. Patel, A., Kosko, B.: Optimal noise benefits in Neyman-pearson signal detection. In: IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 3889–3892. Las Vegas (2008)
9. Lopez, S., Vladimirova, T., Gonzales, C., Resano, J., Mozos, D., Plaza, A.: The promise of reconfigurable computing for hyperspectral imaging onboard systems: a review and trends. Proc. IEEE **101**(3), 698–722 (2013)
10. Torti, E., Acquistapace, M., Danese, G., Leporati, F., Plaza, A.: Real-time identification of hyperspectral subspaces. IEEE J. Sel. Topics Appl. Earth Obs. Remote Sens. **7**(6), 2680–2687 (2014)

11. Plaza, A., Valencia, D., Plaza, J., Chang, C.-I.: Parallel implementation of endmember extraction algorithms for hyperspectral data. IEEE Geosci. Remote Sens. Lett. **3**(3), 334–338 (2006)
12. Barberis, A., Danese, G., Leporati, F., Plaza, A., Torti, E.: Real-time implementation of the vertex component analysis algorithm on GPUs. IEEE Geosci. Remote Sens. Lett. **10**(2), 251–255 (2013)
13. Wu, X., Huang, B., Wang, L., Zhang, J.: GPU-based parallel design of the hyperspectral signal subspace identification by minimum error (HySime). IEEE J. Sel. Topics Appl. Earth Obs. Remote Sens. **9**(9), 4400–4406 (2016)
14. Torti, E., Danese, G., Leporati, F., Plaza, A.: A hybrid CPU-GPU real-time hyperspectral unmixing chain. IEEE J. Sel. Topics Appl. Earth Obs. Remote Sens. **10**(2) 945–951 (2016)
15. Gonzales, C., Lopez, S., Mozos, D., et al.: A novel FPGA-based architecture for the estimation of the virtual dimensionality in remotely sensed hyperspectral images. J. Real-Time Imag. Proc. 1–12 (2015). doi:10.1007/s11554-014-0482-2
16. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical recipes in C: the art of scientific computing, 2nd edn. Cambridge University Press, Cambridge (2007)
17. http://docs.nvidia.com/cuda/cublas/. Last access: Jan 2017
18. http://www.numpy.org/. Last access: Jan 2017
19. https://documen.tician.de/pycuda/. Last access: Jan 2017
20. https://pypi.python.org/pypi/reikna. Last access: Jan 2017
21. Sanchez, S., Plaza, A.: Fast determination of the number of endmembers for real-time hyperspectral unmixing on GPUs. J. Real-Time Image Process. **9**(3), 397–405 (2014)
22. Sanchez, S., Ramalho, R., Sousa, L., Plaza, A.: Real-time implementation of remotely sensed hyperspectral image unmixing on GPUs. J. Real-Time Image Process. **10**(3), 469–483 (2015)

Emanuele Torti (M'13) was born in Voghera, Italy, in 1987. He received the Ph.D. degree in electronics and computer science engineering from University of Pavia, Pavia, Italy, in 2014. He received the Bachelor's Degree in Electronic Engineering and Master's Degree in Computer Science Engineering (cum laude) from University of Pavia in 2009 and 2011, respectively. He is a post-doc researcher at the Engineering Faculty of the University of Pavia. His research is focused on high-performance architectures for real-time image processing and signal elaboration.

Alessandro Fontanella was born in Pavia, Italy, in 1991. He received the Bachelor's degree in computer science engineering and Master's degree in computer science engineering (cum laude) from the University of Pavia, Pavia, Italy, in 2013 and 2015, respectively, where he is currently pursuing the Ph.D. degree in computer science engineering. His research is focused on high-performance architectures for real-time image processing.

Antonio Plaza (SM'07) 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). He was elevated to IEEE Senior Member status in 2007. Prof. Plaza is an Associate Editor for IEEE Access and the IEEE Geoscience and Remote Sensing Magazine, and was a member of the Editorial Board of the IEEE Geoscience and Remote Sensing Newsletter (2011–2012) and a member of the steering committee of the IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing (2012). He served as the Director of Education Activities for the IEEE Geoscience and Remote Sensing Society (GRSS) in 2011–2012, and is currently serving as President of the Spanish Chapter of IEEE GRSS (since November 2012). Currently, he is serving as the Editor-in-Chief of the IEEE Transactions on Geoscience and Remote Sensing journal (since January 2013).