The Journal of Supercomputing manuscript No.
(will be inserted by the editor)

A Multi-Device Version of the HYFMGPU Algorithm for Hyperspectral Scenes Registration

Jorge Fernández-Fabeiro · Álvaro Ordóñez · Arturo González-Escribano · Dora B. Heras

Received: date / Accepted: date

Abstract Hyperspectral image registration is a relevant task for real-time applications like environmental disasters management or search and rescue scenarios. Traditional algorithms were not really devoted to real-time performance, even when ported to GPUs or other parallel devices. Thus, the HYFMGPU algorithm arose as a solution to such a lack. Nevertheless, as sensors are expected to evolve and thus generate images with finer resolutions and wider wavelength ranges, a multi-GPU implementation of this algorithm seems to be necessary in a near future. This work presents a multi-device MPI+CUDA implementation of the HYFMGPU algorithm that distributes all its stages among several GPUs. This version has been validated testing it for 5 different real hyperspectral images, with sizes from about 80 MB to nearly 2 GB, achieving speedups for the whole execution of the algorithm from 1.18× to 1.59× in 2 GPUs and from 1.26× to 2.58× in 4 GPUs. The parallelization efficiencies obtained are stable around 86% and 78% for 2 and 4 GPUs respectively, which proves the scalability of this multi-device version.

Keywords Hyperspectral imaging · image registration · Fourier transforms · multi-GPU · CUDA · OpenMP · MPI · remote sensing

Jorge Fernández-Fabeiro
Departamento de Informática, University of Valladolid, Spain
E-mail: jorge@infor.uva.es

Álvaro Ordóñez
Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela, Spain
E-mail: alvaro.ordonez@usc.es

Arturo González-Escribano
Departamento de Informática, University of Valladolid, Spain
E-mail: arturo@infor.uva.es

Dora B. Heras
Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela, Spain
E-mail: dora.blanco@usc.es

1 Introduction

The task consisting on estimating the translation, rotation and scaling parameters of a given image with respect to a second take of the same scene obtained at different times, viewpoints and/or lighting conditions is known as *image registration*. During the last years, different hyperspectral image registration techniques have been proposed, but most of them ignore time performance. However, many real-time applications such as the management of natural disasters or surveillance operations depend on hyperspectral images being processed in real-time. GPUs were used to boost tasks like classification, target detection or segmentation of this kind of images, but few efforts were made to achieve a real-time implementation of a hyperspectral registration algorithm. In [21], Ordóñez et al. introduced HYFM, a Fourier-Mellin algorithm for hyperspectral images registration, and implemented a sequential CPU version of it. That work was followed by HYFMGPU, a single-GPU CUDA-based version whose performance makes it suitable to be used in real-time environments [22]. As hyperspectral sensors technology improve, images will have finer resolutions in both spatial and spectral domains. Because of that, more computational power and more memory space, this latter one being a limited resource in GPUs, will be needed. In this paper we propose a coarse-grained multi-device implementation of HYFMGPU able to satisfy such present and future needs. This proposal results of a thorough study of the inner computing stages performed in each step of the original single-GPU version, conducted in order to assess the feasibility of distributing them among several devices. **This work is a full version of a short paper [3] presented in the CMMSE 2018 conference. That initial approach was a prospective multi-GPU implementation on which only the most embarrasingly parallel stages of the algorithm were distributed, and that was evaluated using a single small image as test case.**

The rest of this paper is organized as follows: Section 2 summarises the HYFMGPU algorithm, describing then in Section 3 our proposal and the implementation process followed to achieve a multi-GPU version of it. The results obtained by this approach are introduced in Section 4, whereas some related research work is discussed in Section 5. Finally, Section 6 presents the conclusions and some feasible research lines for the future.

2 The HYFMGPU algorithm

The HYFMGPU algorithm expects a pair of hyperspectral images (*reference* and *target*) as inputs. Its goal is to register the target image, this is, to compute how it is rotated, shifted and scaled with respect to the reference image. The **procedure** comprises six main stages, which are depicted in Figure 1. These stages are launched to a single GPU by means of a CUDA [13] implementation that relies on some specific-purpose libraries, namely cuBLAS [12], cuFFT [14], cuSOLVER [15], the NVIDIA Performance Primitives (NPP) [18], and Thrust [19]. As Figure 1 shows, this implementation can be roughly decomposed in the following steps:

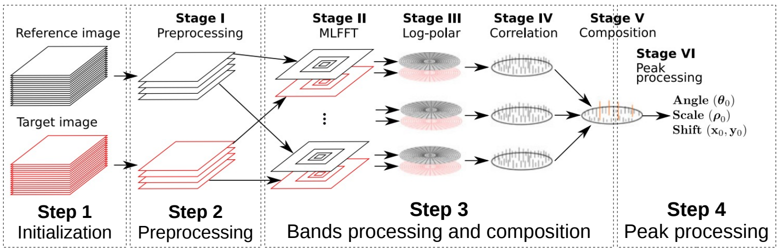

Fig. 1 HYFM scheme for registration of two hyperspectral images

1. **Initialization**: Input images are transferred from the host CPU main memory to the GPU global memory.
2. **Preprocessing**: In Stage I, first a single-element kernel applies a Blackman filter to both inputs to remove higher frequencies, which might be detrimental for the precision of the registration. Those filtered images are centered, and then a principal component analysis (PCA) [24] is applied to each of them in order to extract their most relevant features by retaining a reduced number of principal components (i.e., transformed bands of the original images). Both cuBLAS and cuSOLVER operations are used to implement this analysis. Finally, rows and columns of both band-reduced images are expanded to the nearest common upper power of 2, and data is transformed to complex values.
3. **Selected bands processing and composition**: In this step, pairs composed of the same PCA-extracted band from both inputs are processed by computing a high-pass filter **and a multilayer fractional Fourier transform (MLFFT) [23] (Stage II). Log-polar coordinate maps are extracted from the MLFFT-transformed bands in each pair (Stage III), and then a phase correlation (Stage IV) is applied on them. Stages II and IV are FFT-based,** so that the underlying operations are implemented by means of the cuFFT library. Since this is a single-GPU implementation, the device is commanded to iterate over all the band pairs in order to perform this stage. Finally, a reduction (Stage V) is performed in the global memory of the device in order to average the log-polar correlated maps obtained for each pair of PCA-extracted bands.
4. **Peak processing**: **The peaks contained in the average map of log-polar coordinates** computed in the previous step are sorted in the device using the Thrust library, selecting a given number and processing them one-by-one. This process starts by rotating and scaling the first component of the target image several times using specific functions from the NPP library. Next, a phase correlation and a cuBLAS-based maximum search are performed on the cartesian grid to determine the correct angle and

translation parameters. Finally, the highest peak of all the cartesian grids is selected, as its coordinates determine the *shift* parameters. In turn, their log-polar counterparts decide the *scale* factor and the rotation *angle*. All these values are the expected output of the Stage VI of the **algorithm**.

3 Proposal of multi-GPU parallelization

A glance at the HYFM **scheme** depicted in Figure 1 quickly reveals the workload comprising Stages II, III and IV as a very interesting candidate to a multi-device coarse-grained parallelization. This workload, described in the *Step 3* of the single-GPU CUDA implementation introduced in Section 2, is performed by a loop that iterates over the set of band pairs extracted from reference and target images. **Only this embarrasingly parallel loop was distributed among several GPUs in the initial approach presented in [3]. To implement that parallelization, we prepared first** an OpenMP-threaded version of the HYFMGPU host program. In such a version, a master thread controlled the parts of the multi-device implementation that were kept as single-GPU (Steps 1, 2 and 4), whereas a parallel section was defined to command each GPU to run its stake of the Step 3. So, in this code, once the master thread loads the images, it commands its GPU to apply the Blackman filter, to compute the PCA, and to distribute the extracted components equally among the GPUs available. When the program enters the parallel section, each thread commands its GPU to convert its subset of the PCA-extracted pairs of bands to complex datatype and to perform over them Stages II, III and IV of the **algorithm**. After the program exits this parallel section, the master thread takes the control back, averaging the partial log-polar maps computed by each GPU (Stage V) and processing the peaks (Stage VI) in order to get the final *shift*, *scale* and *angle* outputs expected after this last stage.

3.1 Moving from OpenMP to MPI

As the description of the Step 3 from Section 2 shows, several FFT operations are performed on the PCA-extracted bands. In our multi-GPU version of the algorithm, each GPU performs the same FFT-based operations on a different subset of bands. So, the corresponding cuFFT routines will run on their own in each device, and hence separated sets of FFT execution plans are needed. Despite creating each set of plans separately for each GPU from its corresponding OpenMP thread, we observed that these initialization operations were eventually sequentialized, since the execution time of the Step 3 was always multiplied by the number of selected devices, instead of being divided as expected. To solve this issue, we shifted from forking multiple OpenMP parallel threads to run separated MPI processes. Since MPI programs follow a SPMD (*Single Program,*

***Multiple Data*) model, each of these processes manages an isolated host memory region, instead of sharing all the host memory space as OpenMP parallel threads do. This ensured the independent sets of FFT plans needed by each GPU to be prepared in different processes each with its own memory space, eventually avoiding the sequentialization problem.**

In addition, moving the coarse-grained parallelization support of the initial multi-device implementation from OpenMP to MPI introduced on it some issues related to workflow synchronizations and data communications among processes. Such questions must be taken into account not only to adapt the parallelization of Step 3 introduced in our initial approach but also to distribute Steps 1, 2 and 4 among several GPUs.

3.2 Full algorithm with coarse-grained parallelization

In this section we provide a description of the MPI-based parallelization process of all the HYFM algorithm steps, identifying which inner parts of these steps have been just replicated and which ones have been totally distributed among the processes and their corresponding GPUs. In these latter cases, all the particularities introduced by the usage of MPI are also explained. Moreover, let us note that all explicit data transfers among different GPUs referred from now on are made by properly combining CUDA API synchronous memory copy calls and MPI communication primitives in either origin and destination host processes. **Such an explicit management of data transfers allows our implementation to run in any multi-GPU environment, no matter either the specific features of the NVIDIA devices available or how they are connected both to their hosts or among them. Nevertheless, future versions could rely on CUDA-aware MPI implementations [16] and the NVIDIA Collective Communications Library (NCCL) from CUDA 9 [17] to exploit peer-to-peer communications where available or, at least, to avoid intermediate host buffers when data transfers among devices connected to different nodes are needed [9].**

3.2.1 Initialization

Both reference and target images are scattered among the GPUs in equally distributed groups of rows, so given an input image of c columns, r rows and b bands, each GPU will store in its global memory a slice of c columns, r_k rows and b bands, being $k = 0, \dots, n-1$ and n the number of selected GPUs. When this distribution is not exact, the uneven rows are cyclically assigned to each GPU in order to keep the load balanced.

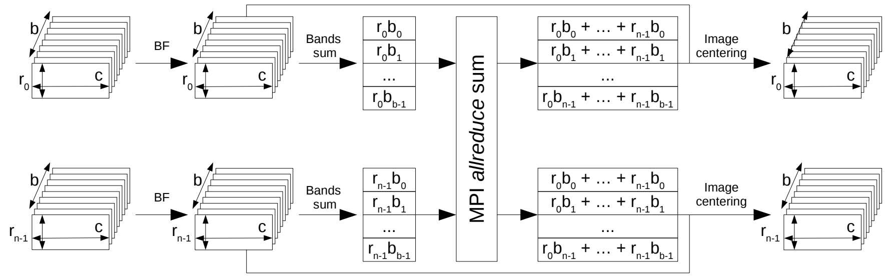

Fig. 2 Multi-GPU workflow for Blackman filter and image centering parts of Step 2

3.2.2 Preprocessing

The preprocessing step of the algorithm is composed of three different parts, on which both reference and target images are first filtered using a Blackman window, then normalized, and finally shrunk to a reduced number of bands by means of a principal component analysis (PCA). In the original HYFM implementation, this analysis is performed by a single-GPU non-iterative procedure based on that one presented in [5].

In this MPI+CUDA approach, the GPU commanded by each process takes the reference and target images slices loaded in Step 1 and applies on them the aforementioned Blackman filter using a single-position kernel. Once the slices of each GPU are filtered, they must be normalized by centering the value of each pixel with relation to the mean value of all the pixels of its band. However, the GPUs only keep for each band the pixel values of their corresponding group of rows. So, at this point, each GPU uses `cublasSgemv` to accumulate band-by-band the pixel values of the slices of both filtered images in 2 arrays of b elements. Then the host process retrieves these arrays and performs on them an `MPI_Allreduce` operation, so after this message exchange, all the processes have the full summation of each band of the input images. Finally, each process transfers both band summations arrays to its GPU, which uses them to compute the mean values needed to center the filtered images slices. Figure 2 depicts the workflow of these two first parts of the input images preprocessing. As the Table 1 provided at the end of this section summarises, we are considering these first parts of the Step 2 as totally distributed among the GPUs, the overload derived from the MPI-based summation and the bidirectional host-GPUs transfers it needed being the cost of such a parallel distribution.

The principal component analysis of a filtered and centered input image is composed of several stages. First, a correlation matrix of the input is calculated. In the single-GPU version, it is calculated as $I \times I^T$, I being a matrix of $r \times c$ columns and b rows that represents the input image. Figure 3 shows how this full correlation matrix can be obtained in parallel from the slice available in each GPU. This is possible thanks to properties of the matrix product operation. So, each GPU computes a partial correlation matrix, all of them being

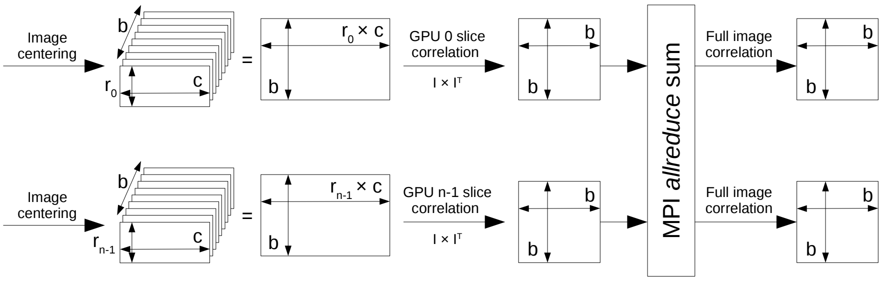

Fig. 3 Workflow of distributed row-group image correlation and full correlation matrix reconstruction

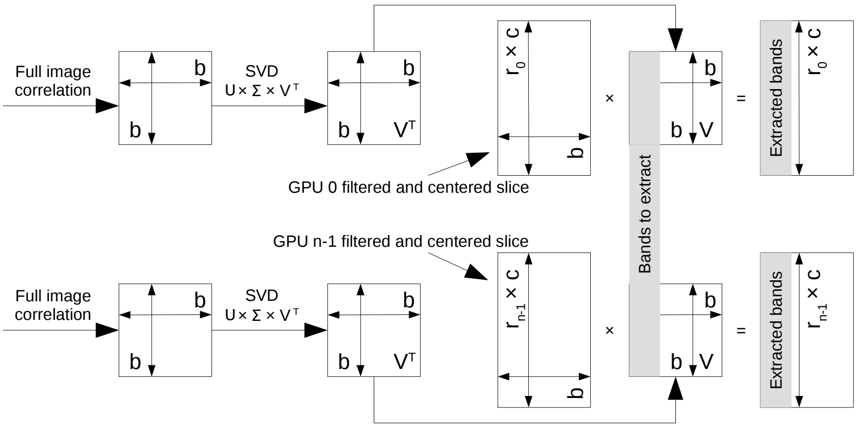

Fig. 4 Workflow of distributed band extraction

gathered and summed up into a full correlation matrix and then broadcast to all processes with an `MPI_Allreduce` operation. Once every GPU has its copy of the full correlation matrix, it can perform its own decomposition of it. Notice how only partial correlation matrices of just $b \times b$ elements are exchanged among the processes, instead of the slices of $c \times r_k \times b$ elements that would be needed to recompose the full input image in each GPU and then making them to compute their own full correlation matrix from scratch.

Second, each GPU computes the singular value decomposition (SVD) of its copy of the full correlation matrix by means of the `cusolverDnSgesvd` function, which returns the $U \times \Sigma \times V^T$ sequence of matrices. The latter one is transposed and then used to transform the corresponding input image slice into a principal-component ordered version of the slice by means of a `cublasSgemm` operation. As Figure 4 shows, by multiplying the input image slice by a submatrix with the eb (extracted bands) first columns of V, the resulting matrix is not only a principal-component ordered version of that slice but also a band-reduced one. However, such versions are still submatrices with just the group of rows initially assigned to each device.

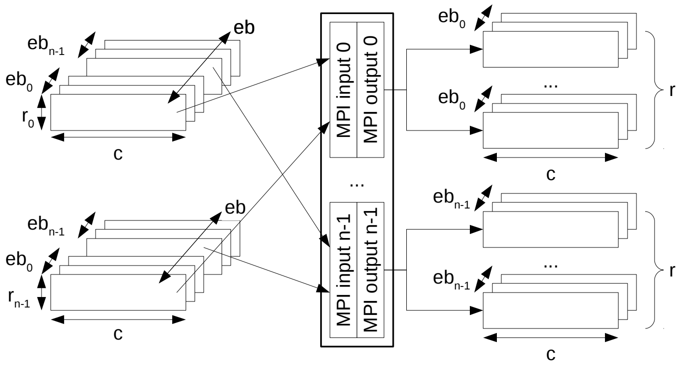

Fig. 5 Workflow of row-to-band group redistribution

The last part of the PCA calculation consists on a MPI-based reconstruction of the band-reduced input images, which is sketched in Figure 5. By this reconstruction, images are rearranged from a row-based distribution to a band-based one, since GPUs are expecting as inputs for the Step 3 slices with eb_k bands, each one with $r \times c$ elements. So first, each process k gets from its GPU the $c \times r_k \times eb$ slice and scatters it among the rest of processes by means of asynchronous `MPI_Isend` messages, represented as individual arrows in the left part of the figure. The data received by each process is not retrieved from MPI buffers using as many individual `MPI_Recv` messages as data packages are sent to it, but thanks to a user-defined `MPI_Type_vector` the expected slice with eb_k bands of $r \times c$ elements is directly reconstructed using the parts coming from every process by means of a single `MPI_Recv` message. Let us also note that a band-based distribution of input images was evaluated too. In this alternative distribution, every GPU stored in its global memory an slice of c columns $\times$ r rows $\times$ b_k bands for each image. However, in that case the workflow would force the GPUs to recompose the filtered and centered input images in order to compute the full correlation matrices, which leads to a much more massive data exchange during the PCA calculation part. These inconveniences were experimentally confirmed, the PCA computation resulting to be up to 9 times slower than when applying the row-based distribution strategy.

We classify the whole PCA calculation as distributed among the GPUs, in the same way we did for the Blackman filtering and image centering parts. The cost of such a parallel distribution is the overloads introduced by the MPI-based operations needed to compute the full correlation matrices, by the SVD decomposition needed to perform the band reduction, and by the rearrangement of the reduced image row-based slices in the GPUs as groups of full image reduced bands. Moreover, the creation of the cuBLAS and cuSOLVER handlers needed by some inner operations of both the normalization and the PCA parts must be replicated in all the processes. The creation of these handlers is consuming a fixed time of about 430 ms, no matter the input image sizes or the number of GPUs available. Table 1 also includes these considerations.

Table 1 Summary of replicated and distributed operations performed by each step of the MPI+CUDA multi-GPU HYFM implementation

Step 1: Initialization	Replicated	-
	Distributed	Input images GPU loading
Step 2: Preprocessing	Replicated	cuBLAS and cuSOLVER handlers creation
	Distributed	Blackman filter, image centering, PCA calculation
Step 3: Band processing and composition	Replicated	cuFFT plans creation, auxiliary arrays calculation
	Distributed	Complex conversion, processing, composition
Step 4: Peak processing	Replicated	-
	Distributed	Peak sorting, highest peak search

3.2.3 Band processing and composition

In this step, for each input image, every GPU takes its corresponding band-grouped slice of c columns, r rows and eb_k reduced bands and performs on it the high-pass filter, the MLFFT, the log-polar map calculation and the phase correlation, just as it is introduced in the algorithm description from Section 2 and following the same parallelization approach **described in our preliminary work presented in [3]**. Some of these operations are computed by cuFFT routines that expect `cufftComplex` data as inputs and that also need the previous creation of several FFT execution plans. In this parallel implementation, the conversion of the band-grouped slices to that complex type, the FFT plans creation, and the calculation of some auxiliary coefficient arrays needed by those operations, have been moved to this step from their original location at the end of the *Preprocessing* step. Regarding the reduction of the partial log-polar maps computed by each GPU into the final one, these maps are first gathered in a root MPI process and then sent to its GPU in order to average them using a specific kernel. Let us recall that these maps are in `cufftComplex` format, so a suitable `MPI_Type_contiguous` type is defined to simplify their transfer. Finally, as it is applied also on both the reference and the target slices processed by each device, the complex conversion is considered as a distributed part of the algorithm. In turn, the FFT plans creation and the calculation of the auxiliary arrays are both just replicated, as every GPU need its own copies of them.

3.2.4 Peak processing

In the first part of this step, the root MPI process uses Thrust to generate a host-side ordered indexes vector from the GPU-stored final average map. Both arrays are broadcast to the rest of processes along with the first PCA component from both reference and target images, since all these data structures are needed to process the peaks pointed by a number of the top elements of the ordered indexes vector. Then, every MPI process traverses cyclically that top subset of the ordered indexes array, applying to each peak the operations described in Step 4 from Section 2 and obtaining a partial maximum peak by

Table 2 Dimensions and size in MB of hyperspectral images used as test cases

Name	Width	Height	Bands	Image size
Pavia University	340	610	103	81 MB
Pavia Centre	715	1096	102	305 MB
Jasper Ridge	588	1286	224	646 MB
Indian North South	614	2678	220	1447 MB
Bay Area Box Line	512	4096	224	1879 MB

means of a `cublasIcamax` operation. The information of these partial peaks is on structs that are packed as `MPI_Type_contiguous` messages and then gathered in the root process. Finally, this process inspects the partial peaks received and computes the expected outputs of cartesian *shift*, *scale* factor and rotation *angle*. For this step, all the operations are being considered as distributed among the available devices, as Table 1 shows.

4 Experimental work

The performance obtained by this multi-GPU implementation has been evaluated by registering rotated and scaled variations of five real hyperspectral images used as references, and whose main relevant properties are described in Table 2. The images named *Pavia University*, *Pavia Centre* and *Jasper Ridge* are commonly used for testing in remote sensing [20]**. The smallest one, *Pavia University*, was the only test case used in the prospective implementation from [3].** In turn, the images *Indian North South* and *Bay Area Box Line* are obtained from Purdue Research Foundation MultiSpec [2] and NASA AVIRIS [11] repositories, respectively. Let us note that the number of operations performed by the algorithm depends only on the sizes of both input images, being independent of scale, rotation and shift differences among them. Moreover, this is a multi-GPU distribution of the original HYFMGPU algorithm, hence it has the same registration precision as the reference implementation from [22]. All the tests were run in a node equipped with a dual-socket host CPU composed of 2 Intel Xeon E5-2609v3 (1.9 GHz, 6 cores each) with 64 GB of RAM, and 4 GPUs NVIDIA GeForce GTX TITAN Black (GK110B architecture, compute capability 3.5, 15 SMs with 192 CUDA cores each up to 2880, **6 GB RAM**) controlled by the 384.59 driver. The CUDA code has been compiled under Linux using nvcc from CUDA Toolkit 9.0, as well as the libraries used. Multi-threaded preliminary tests were supported by linking libraries from OpenMP 3.1, whereas the MPI support was provided by mpich-3.2. The values of the parameters that control the registration algorithm are the same as those specified in [22], this is: 8 bands to be extracted in the PCA, vector $\alpha = \{1, 1/4, 1/16, 1/64\}$ for the FFTs performed to obtain the log-polar maps, and 50 highest peaks to be examined in the peak processing stage.

Table 3 shows, for each test case, the wall time consumed by Steps 1 to 4 from Section 2 individually, and by the whole algorithm, when run in a single-GPU using the original CUDA implementation of HYFM and in 2 and 4 GPUs with the MPI+CUDA version. Moreover, the global speedups achieved by these two latter experiments are shown in a separated column. Those speedups range from 1.18× to 1.59× for 2 GPUs and from 1.26× to 2.58× for 4 GPUs, which could seem quite away from the expected peaks of 2× and 4× respectively. By decomposing the wall times of each algorithm step according to the parallelization process summarised in Table 1 we can weigh the influence of every inner part of each step in the global performance of this

Table 3 Wall times in seconds for each step and the whole algorithm, and global speedups obtained for tests run in *NVIDIA GeForce Titan Black* GPUs

Test case	Version	Wall time in seconds					Speedup
		Step 1	Step 2	Step 3	Step 4	Total	
Pavia University	HYFM	0.031	0.557	0.924	0.303	1.815	-
	2-GPU	0.025	0.566	0.719	0.227	1.537	1.18×
	4-GPU	0.015	0.585	0.630	0.212	1.442	1.26×
Pavia Centre	HYFM	0.114	0.662	2.309	0.949	4.035	-
	2-GPU	0.089	0.673	1.426	0.611	2.800	1.44×
	4-GPU	0.047	0.646	1.000	0.440	2.133	1.89×
Jasper Ridge	HYFM	0.241	1.061	2.327	0.933	4.562	-
	2-GPU	0.186	0.971	1.441	0.631	3.230	1.41×
	4-GPU	0.097	0.840	0.994	0.440	2.371	1.92×
Indian North South	HYFM	0.838	2.188	8.319	3.172	14.518	-
	2-GPU	0.445	1.440	4.560	2.957	9.402	1.54×
	4-GPU	0.231	1.036	2.578	1.833	5.678	2.56×
Bay Area Box Line	HYFM	0.912	2.622	8.330	2.845	14.709	-
	2-GPU	0.706	1.662	4.553	2.345	9.265	1.59×
	4-GPU	0.281	1.140	2.585	1.704	5.710	2.58×

Table 4 Disgregated wall times in seconds for replicated and distributed parts of Steps 2 and 3 obtained for tests run in *NVIDIA GeForce Titan Black* GPUs

Test case	Version	Step 2			Step 3		
		Replicated	Distributed		Replicated	Distributed	
Pavia University	HYFM	0.431	0.126	-	0.463	0.461	-
	2-GPU	0.431	0.135	0.93×	0.473	0.247	1.87×
	4-GPU	0.431	0.154	0.82×	0.495	0.135	3.42×
Pavia Centre	HYFM	0.428	0.234	-	0.476	1.833	-
	2-GPU	0.428	0.246	0.95×	0.472	0.954	1.92×
	4-GPU	0.428	0.219	1.07×	0.498	0.501	3.66×
Jasper Ridge	HYFM	0.424	0.636	-	0.496	1.831	-
	2-GPU	0.424	0.547	1.16×	0.493	0.947	1.93×
	4-GPU	0.424	0.416	1.53×	0.497	0.497	3.69×
Indian North South	HYFM	0.432	1.757	-	0.559	7.761	-
	2-GPU	0.432	1.008	1.74×	0.583	3.977	1.95×
	4-GPU	0.432	0.604	2.91×	0.519	2.059	3.77×
Bay Area Box Line	HYFM	0.434	2.188	-	0.590	7.740	-
	2-GPU	0.434	1.228	1.78×	0.588	3.965	1.95×
	4-GPU	0.434	0.706	3.10×	0.530	2.056	3.77×

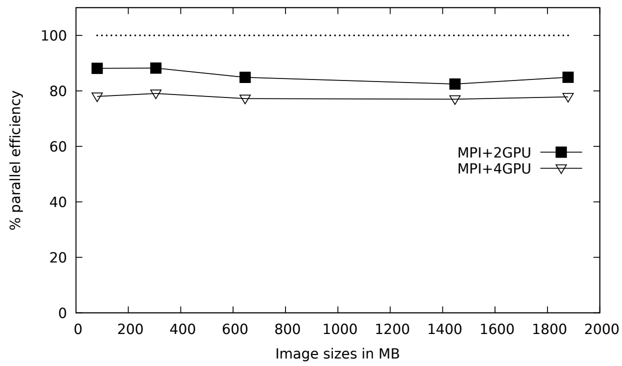

Fig. 6 Evolution of parallelization efficiencies against input images size in MB for the tests run in *NVIDIA GeForce Titan Black* GPUs

implementation. Table 4 shows such times decomposed in replicated and distributed parts along with the speedup obtained for this latter portion for Steps 2 and 3. Regarding the Step 2, the distributed speedups range from 0.93× to 1.78× for 2 GPUs and from 0.82× to 3.10× for 4 GPUs, something expectable having in mind that the distributed part of this step is quite intensive on MPI data transfers and synchronizations, as explained in Section 3.2.2. Speedups for the distributed part of the Step 3 range from 1.87× to 1.95× for 2 GPUs and from 3.42× to 3.77× for 4 GPUs, all being very close to the theoretical peaks. Only the final gathering and averaging of the partial maps makes this distributed part not to be embarrassingly parallel. Times for Steps 1 and 4 are not disgregated in Table 4, as they are considered as fully distributed. Nevertheless, the wall times from the *Step 4* (column in Table 3) are consistent with the overload expected as a consequence of the input data broadcast and the final peak reduction performed in this step and described in Section 3.2.4.

Besides the aforementioned data transfer and synchronization overloads, let us also recall that Steps 2 and 3 perform some replicated parts whose time cost can be considered as fixed no matter the input images size or the number of GPUs available. Amdahl efficiencies for each test case have been calculated computing these fixed times as not parallelizable. The line chart of Figure 6 represents these efficiencies in relation to the size in MB of the input images, and grouped by the number of GPUs used. Moreover, the ideal efficiency is represented as a constant horizontal dotted line at the 100% value. All the tests cases obtained quite high efficiencies which are stable around 86% and 78% for 2 and 4 GPUs respectively.

5 Related work

Spatial unmixing algorithms [8] are another kind of hyperspectral analysis **procedures** that perform PCAs based on SVD decompositions. For example, Jiménez et al. [7] implement a single-GPU version of the spatial–spectral endmember extraction algorithm [25] that, unlike HYFMGPU, computes SVDs by means of a CPU LAPACK routine. However, they still rely on cuBLAS for matrix multiplications.

Lončar et al. cover in [10] how multiple FFT operations are integrated in both OpenMP- and MPI-based solvers for the dipolar Gross–Pitaevskii equation, a condensed matter physics problem. They present first a couple of multithreaded multi-CPU implementations that are based on the aforementioned parallel programming standards. Both versions use the FFTW3 [4] library for FFT routines, since it is tailored to their specific needs. Moreover, they also present an MPI-based implementation that distributes the work among several nodes in a cluster, each with a single NVIDIA GPU. In this case the solver must call cuFFT routines, as there is no CUDA version of FFTW3. They also state that such a combination of MPI processes and cuFFT routines should work properly in a single cluster node with several GPUs, something that our work confirms.

CUDA-aware MPI implementations are commonly exploited in different multi-device versions of algorithms from multiple applied science scopes. For instance, Glaser et al. [6] implement strong scaling versions of general-purpose molecular dynamics simulations on GPUs, and Lončar et al. use it too in the aforementioned solver from [10]. Deep learning and data analytics are other scopes that are taking advantage of CUDA-aware MPI implementations, for example by exploiting it to support efficient large message broadcast operations [1]. That work also exploits NCCL in order to optimize intra-node communications among directly-connected GPUs.

6 Conclusions and future work

In this paper we present a coarse-grained distributed, multi-device version of the original HYFMGPU CUDA implementation, starting with a thorough study of all the steps that compose this registration algorithm. In a very first approach, we opted for OpenMP to support the multi-device distribution of the *Band Processing* step, but some sequentialization issues related to the cuFFT routines called in this step led us to develop the full distributed version using MPI. Due to the properties of the different parts of the algorithm, we had to deal with multiple communication and synchronization points along the MPI-based implementation, as well as with the fixed latencies introduced by some specific-purpose libraries used by the GPUs. Nevertheless, this multi-device version of HYFMGPU is able to boost the registration process of the test cases up to 1.59× using 2 GPUs and up to 2.59× for 4 GPUs, and with quite high and stable parallelization efficiencies around 86% and 78%, respectively. It must be noted that these test cases are real hyperspectral images that cover a wide range of sizes from about 80 MB to nearly 2 GB, which proves this implementation to have a robust scalability. **The results have shown that GPUs are adequate platforms to perform efficient registration even for large images. The computational efficiency achieved have**

practical applications especially for environments where real-time registration is required, for example, in the case of disaster and damage control or surveillance.

In relation to feasible future research lines, the development of a version based in both NCCL and a CUDA-aware MPI environment stands up as a promising option to improve the data communication and synchronization operations among GPUs. **On the one hand, such an implementation of the algorithm should be less error-prone and easier to maintain, as all the transfer operations now explicitly coded will be replaced by their equivalent library calls. On the other hand, important performance improvements can be expected thanks to the transparent exploitation of peer-to-peer data transfers among GPUs, if supported by the underlying hardware.** Moreover, there are some CUDA-related fine-grain optimizations taking into account several GPU architectures that by now are being inherited from the original single-GPU implementation. Further optimizations for both current and upcoming GPU architectures could be studied. A survey of such details will help to exploit, for instance, the asynchronous CUDA API. The usage of this interface might lead to discover delay slots on which the fixed latencies derived from the cuFFT plans and cuBLAS/cuSOLVER handlers creations could be hidden.

Acknowledgements This work has been partially supported by:

Universidad de Valladolid – Consejería de Educación of Junta de Castilla y León, Ministerio de Economía, Industria y Competitividad of Spain, and European Regional Development Fund (ERDF) program: Project PCAS (TIN2017-88614-R), Project PROPHET (VA082P17) and CAPAP-H6 network (TIN2016-81840-REDT).

Universidade de Santiago de Compostela – Consellería de Cultura, Educación e Ordenación Universitaria of Xunta de Galicia (grant numbers GRC2014/008 and ED431G/08) and Ministerio de Economía, Industria y Competitividad of Spain (grant number TIN2016-76373-P), all co-funded by the European Regional Development Fund (ERDF) program. The work of Álvaro Ordóñez was supported by the Ministerio de Educación, Cultura y Deporte under an FPU Grant (grant number FPU16/03537).

References

1. Awan, A.A., Hamidouche, K., Venkatesh, A., Panda, D.K.: Efficient Large Message Broadcast Using NCCL and CUDA-Aware MPI for Deep Learning. In: Proceedings of the 23rd European MPI Users' Group Meeting, EuroMPI 2016, pp. 15–22. ACM, New York, NY, USA (2016)
2. Baumgardner, M.F., Biehl, L.L., Landgrebe, D.A.: 220 Band AVIRIS Hyperspectral Image Data Set: June 12, 1992 Indian Pine Test Site 3. URL https://purr.purdue.edu/publications/1947/1
3. Fernández-Fabeiro, J., Álvaro Ordóñez, González-Escribano, A., Heras, D.B.: Towards a multi-device version of the HYFMGPU algorithm for hyperspectral scenes registration (2018). DOI 10.5281/zenodo.1475157. URL http://dx.doi.org/10.5281/zenodo.1475157
4. Frigo, M., Johnson, S.G.: Massachusetts Institute of Technology (MIT) FFTW3 Library. URL http://www.fftw.org

5. Garea, A.S., Heras, D.B., Argüello, F.: GPU classification of remote-sensing images using kernel ELM and extended morphological profiles. International Journal of Remote Sensing **37**(24), 5918–5935 (2016)
6. Glaser, J., Nguyen, T.D., Anderson, J.A., Lui, P., Spiga, F., Millan, J.A., Morse, D.C., Glotzer, S.C.: Strong scaling of general-purpose molecular dynamics simulations on GPUs. Computer Physics Communications **192**, 97 – 107 (2015)
7. Jiménez, L.I., Sánchez, S., Martín, G., Plaza, J., Plaza, A.J.: Parallel Implementation of Spatial Spectral Endmember Extraction on Graphic Processing Units. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing **10**(4), 1247–1255 (2017)
8. Keshava, N., Mustard, J.F.: Spectral unmixing. IEEE Signal Processing Magazine **19**(1), 44–57 (2002)
9. Kraus, J.: An Introduction to CUDA-Aware MPI, NVIDIA Developer Blog. URL https://devblogs.nvidia.com/introduction-cuda-aware-mpi/
10. Lončar, V., Young-S., L.E., Škrbić, S., Muruganandam, P., Adhikari, S.K., Balaž, A.: OpenMP, OpenMP/MPI, and CUDA/MPI C programs for solving the time-dependent dipolar Gross–Pitaevskii equation. Computer Physics Communications **209**, 190–196 (2016)
11. NASA Jet Propulsion Laboratory: Airborne Visible/Infrared Imaging Spectrometer (AVIRIS) Database. URL https://aviris.jpl.nasa.gov/data/index.html
12. NVIDIA Corporation: cuBLAS Library User's Guide. URL https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf
13. NVIDIA Corporation: CUDA C Programming Guide. URL https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
14. NVIDIA Corporation: cuFFT Library User's Guide. URL https://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf
15. NVIDIA Corporation: cuSOLVER Library User's Guide. URL https://docs.nvidia.com/cuda/pdf/CUSOLVER_Library.pdf
16. NVIDIA Corporation: MPI Solutions for GPUs. URL https://developer.nvidia.com/mpi-solutions-gpus
17. NVIDIA Corporation: NVIDIA Collective Communications Library (NCCL). URL https://developer.nvidia.com/nccl
18. NVIDIA Corporation: NVIDIA Performance Primitives (NPP). URL https://docs.nvidia.com/cuda/npp/index.html
19. NVIDIA Corporation: Thrust Quick Start Guide. URL https://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf
20. Ordóñez, A., Argüello, F., Heras, D.B.: Repository of hyperspectral images for 'GPU Accelerated FFT-Based Registration of Hyperspectral Scenes'. URL https://gitlab.citius.usc.es/hiperespectral/RegistrationRepository
21. Ordóñez, A., Argüello, F., Heras, D.B.: Fourier–Mellin registration of two hyperspectral images. International Journal of Remote Sensing **38**(11), 3253–3273 (2017)
22. Ordóñez, A., Argüello, F., Heras, D.B.: GPU Accelerated FFT-Based Registration of Hyperspectral Scenes. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing **10**(11), 4869–4878 (2017)
23. Pan, W., Qin, K., Chen, Y.: An Adaptable-Multilayer Fractional Fourier Transform Approach for Image Registration. IEEE Transactions on Pattern Analysis and Machine Intelligence **31**(3), 400–414 (2009)
24. Richards, J.: Remote Sensing Digital Image Analysis, chap. Feature Reduction, pp. 343–380. Springer, Berlin, Heidelberg (2013)
25. Rogge, D., Rivard, B., Zhang, J., Sanchez, A., Harris, J., Feng, J.: Integration of spatial–spectral information for the improved extraction of endmembers. Remote Sensing of Environment **110**(3), 287 – 303 (2007)