Simplifying the distributed multi-GPU programming of a hyperspectral image registration algorithm

Jorge Fernández-Fabeiro, Arturo Gonzalez-Escribano, Diego R. Llanos
Departamento de Informática
Universidad de Valladolid
Valladolid, Spain
{jorge|arturo|diego}@infor.uva.es

***Abstract*—Hyperspectral image registration is a relevant task for real-time applications like environmental disasters management or search and rescue scenarios. Traditional algorithms for this problem were not really devoted to real-time performance. The HYFMGPU algorithm arose as a high-performance GPU-based solution to solve such a lack. Nevertheless, a single-GPU solution is not enough, as sensors are evolving and then generating images with finer resolutions and wider wavelength ranges. An MPI+CUDA distributed multi-GPU implementation of HYFMGPU was previously presented. However, this solution shows the programming complexity of combining MPI with an accelerator programming model. In this paper we present a new and more abstract programming approach for this type of applications, which provides a high efficiency while simplifying the programming of the distributed code. The solution uses Hitmap, a library to ease the programming of parallel applications based on distributed arrays. It uses a more algorithm-oriented approach than MPI, including abstractions for the automatic partition and mapping of arrays at runtime with arbitrary granularity, as well as techniques to build flexible communication patterns that transparently adapt to the data partitions. We show how these abstractions apply to this application class. We present a comparison of development effort metrics between the original MPI implementation and the one based on Hitmap, with reductions of up to 95% for the Halstead score in specific work redistribution steps. We finally present experimental results showing that these abstractions are internally implemented in a high efficient way that can reduce the overall performance time in up to 37% comparing with the original MPI implementation.**

***Index Terms*—General-Purpose computation on Graphics Processing Units (GPGPU); HPC in Digital Signal and Image Processing and Vision; Libraries and Programming Environments; Partitioning, Mapping, and Scheduling**

I. Introduction

Image registration is the task of 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. 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. Ordóñez et al. introduced in [2] a sequential CPU implementation of HYFM [3], a Fourier-Mellin algorithm for hyperspectral images registration. That work was followed by HYFMGPU, a single-GPU CUDA-based version whose performance makes it suitable to be used in real-time environments [1]. As hyperspectral sensors technology improves, images 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, is needed. We presented in [4] a coarse-grained distributed multi-GPU implementation of HYFM that follows a hybrid MPI+CUDA programming approach and is able to satisfy such present and future needs.

Nevertheless, this approach shows the programming complexities of MPI when it is combined with an accelerator programming model such as CUDA, mixing work redistributions with data movements between host and devices, and calls to external scientific libraries. In this work we present a new and more abstract programming approach for this class of applications. This approach highly simplifies the programming while obtaining a high performance efficiency. The solution is based on Hitmap [5], a library designed to ease the task of programming parallel applications by using distributed arrays. It includes abstractions for the automatic partitioning and mapping of arrays with arbitrary granularity, as well as the automatic construction of flexible communication patterns that are transparently adapted to the partition. In particular, we show how the techniques of automatic partitioning, collective operations and pattern-based communications that Hitmap offers, can be applied to the distributed multi-GPU version of the HYFM image registration algorithm. We present a comparison of the original MPI+CUDA code with the new Hitmap-based approach in terms of development effort metrics, which shows a high reduction of the overall programming complexity of the coordination code of up to 95% for the Halstead development effort score in specific work redistribution steps. The usage of such techniques alleviates the development effort with respect to their MPI counterparts, whose application was proven to be quite tedious and error-prone in some specific steps of the

This work has been funded by 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).

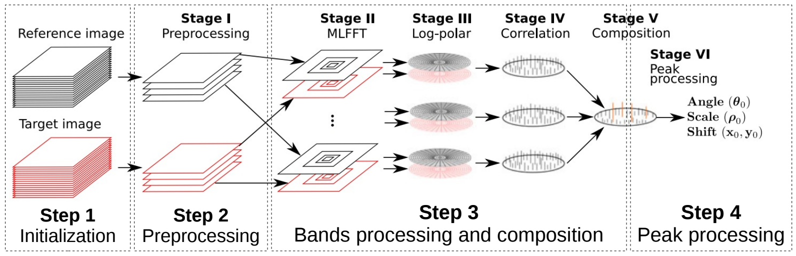

Fig. 1. HYFM scheme for registration of two hyperspectral images. Adapted from [1].

algorithm. When trying to measure the potential overhead that the Hitmap usage may introduce in the performance of our HYFM multi-GPU implementation, we discover that despite introducing an additional code abstraction layer the algorithm decreased its wall time in up to a 37%, namely thanks to the internal use in Hitmap of efficient MPI coding techniques, that can be complex to be expressed manually, including the use of hierarchical derived data types, and highly asynchronous communication patterns.

The rest of this paper is organized as follows: we start discussing some related research in Sect. II. Sect. III recalls the multi-GPU distributed HYFM algorithm, introducing then in Sect. IV the Hitmap distributed programming library. We show in Sect. V how Hitmap techniques are applied to implement the algorithm. The results obtained by this approach are introduced in Sect. VI, and finally Sect. VII presents the conclusions and some feasible research lines for the future.

II. Related work

GPU offloading is a quite common approach followed in remote sensing in order to boost the implementation of such algorithms. For example, GPU-accelerated methods for geospatial object detection for both civilian [6] and military [7] fields have been recently presented, as well as for undersea image reconstruction [8] or synthetic aperture radar imaging in marine surfaces [9]. Many of these works also remark the sustained improvement of sensor technology and how it increases both computing and memory needs of any algorithm that manipulates data coming in real-time from sensors embedded in unmanned vehicles or satellites. There are interesting research lines focused on exploiting heterogeneous devices in order to accelerate relevant stages of such tasks. For instance, Martel et al. introduce in [10] some strategies to reach latency-efficient implementations of dimensionality reduction algorithms in a GPU and an FPGA. Other works, in turn, opt to exploit highly-optimized GPU-based deep learning techniques to implement remote sensing tasks like image-based crop health monitoring [11] or cloud segmentation for weather prediction [12].

The mixed MPI+CUDA programming approach is a common solution to scatter among several GPUs data that would require an unreasonably long compute time on a single device or that are too large to fit into its memory. Depending on the properties of the algorithm being distributed, this approach may require to implement complex domain decompositions [13]. Hitmap offers an intermediate abstraction layer to tackle this issue, halfway between the manual programming of distributed data structures on message-passing models, and PGAS languages (Partitioned Global Address Space), like Chapel [14] or UPC [15]. It provides a simple way to create distributed arrays that map to local address spaces, with explicit mechanisms for the construction of reusable communication patterns at runtime. These patterns adapt to the data partition, creating a low number of aggregated communications when moving data across the global space. This leads, for example, to a performance efficiency comparable to UPC, with a reduced programming complexity and development effort [5]. Hitmap extends and generalizes the hierarchy creation and data partition functionalities of other libraries or distributed arrays models, such as HTAs [16] or Parray [17]. It allows the usage of transparent partition policies, either regular or irregular, defined as interchangeable modules with a common interface. This hides to the programmer the decisions about granularity and synchronization across hierarchical levels. Hitmap has also been extended to support data structures such as sparse matrices, or graphs, using the same methodology and interface [18]. Hitmap is also the portable library used to provide a common interface for transparent data management in the Controller model [19], an abstract entity that allows programmers to easily manage the communications, and kernel launching details, on multiple heterogeneous devices, including GPUs and multi-core CPUs [20]. Regarding its application in specific use cases, Hitmap has been presented in [21] as a programming interface to transparently map agents to processes in a multi-agent pedestrian simulator.

III. Distributed HYFMGPU

The HYFM algorithm expects a pair of hyperspectral images (*reference* and *target*) as inputs. The goal is to register the target image, this is, to compute how it is rotated, shifted and scaled with respect to the reference image. This procedure, which is depicted in Fig. 1, was first implemented in CUDA by Ordóñez el al. to be run in single NVIDIA GPUs [1] and then distributed among several devices as we presented in [4]. In this section we present an overview of the coarse-grain parallelization process followed to reach the MPI+CUDA distributed multi-GPU version of the algorithm. This will help the reader to better understand the computation and communication steps and how the new Hitmap approach is introduced.

```
int PCAS = 8 // bands extracted via PCA
int PCAS_BY_GPU = PCAS / world_size;
int g = real_mpi_rank; // from MPI_Comm_rank
float *h_pca1_geosliced = (float*) malloc(sizeof(float)*COLUMNS*im1_gpurows_array[g]*PCAS);
float *h_pca1_bandsliced = (float*) malloc(sizeof(float)*COLUMNS*ROWS*PCAS_BY_GPU);

cudaMemcpy(h_pca1_geosliced,d_pca1_geosliced,sizeof(float)*COLUMNS*im1_gpurows_array[real_mpi_rank]*PCAS,
    cudaMemcpyDeviceToHost);

MPI_Request req_pca_isend[4];
for(g=0;g<world_size;g++) {
    if(g!=real_mpi_rank) {
        int send_tag = 10000 + (real_mpi_rank * 100 + g);
        float* send_ptr = h_pca1_geosliced + (im1_gpurows_array[real_mpi_rank]*COLUMNS*PCAS_BY_GPU*g);
        MPI_Isend(send_ptr, im1_gpurows_array[real_mpi_rank]*COLUMNS*PCAS_BY_GPU, MPI_FLOAT, g, send_tag,
            MPI_COMM_WORLD, &(req_pca_isend[g]));
    }
}
MPI_Datatype pca1_strided_bands_typearray[4];
int gg;
int recv_rows_offset = 0;
for(gg=0;gg<world_size;gg++) {
    g = gg;
    if(g!=real_mpi_rank) {
        int recv_tag = 10000 + (g * 100 + real_mpi_rank);
        float* recv_ptr = h_pca1_bandsliced + (recv_rows_offset*COLUMNS);
        MPI_Type_vector(PCAS_BY_GPU,im1_gpurows_array[g]*COLUMNS,ROWS*COLUMNS,MPI_FLOAT,&(pca1_strided_bands_typearray
            [g]));
        MPI_Type_commit(&(pca1_strided_bands_typearray[g]));
        MPI_Recv(recv_ptr,PCAS_BY_GPU,pca1_strided_bands_typearray[g],g,recv_tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    else {
        int b;
        for(b=0;b<PCAS_BY_GPU;b++) {
            float* d_single_band_ptr = h_pca1_bandsliced + (recv_rows_offset*COLUMNS) + (ROWS*COLUMNS*b);
            float* single_band_ptr = h_pca1_geosliced + COLUMNS*im1_gpurows_array[g]*PCAS_BY_GPU*g + COLUMNS*
                im1_gpurows_array[g]*b;
            memcpy(d_single_band_ptr,single_band_ptr,im1_gpurows_array[g]*COLUMNS*sizeof(float));
        }
    }
    recv_rows_offset += im1_gpurows_array[g];
}
```

Fig. 2. Implementation of the rows-to-bands redistribution of one of the two images in the MPI original program.

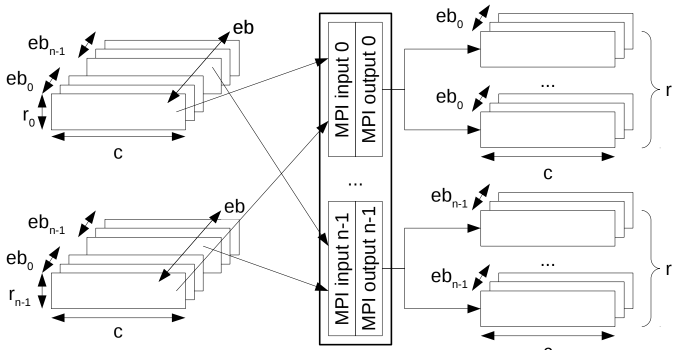

Fig. 3. Workflow of rows-to-bands group redistribution

1) Initialization: Both the reference and the target images are scattered among the GPUs in equally distributed groups of rows. When this distribution is not exact, the uneven rows are cyclically assigned to each GPU in order to keep the load balanced.

2) Preprocessing: This step 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 this distributed approach, the GPU commanded by each process takes the reference and target images slices loaded in Step 1 and filters them. Once 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. Since each GPU only keeps its corresponding group of rows, an `MPI_Allreduce` operation is needed so that all processes could have the full summation of each band of the input images, and then compute the mean values used to center their image slices.

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 [4] we show how this full correlation matrix can be obtained in parallel from the slice available in each GPU. Summarising, each GPU computes a partial correlation matrix, all of them being gathered and summed up into a full correlation matrix and then broadcast to all processes with an `MPI_Allreduce` operation. Second, each GPU uses cuSOLVER [22] to compute the singular value decomposition (SVD) of the full correlation matrix. This decomposition is used to transform the row-sliced input into a band-reduced and principal-component ordered version. The last part of the PCA calculation consists on a MPI-based reconstruction of the band-reduced input images, which is sketched in Fig. 3. Each GPU k expects eb_k full bands as inputs for the Step 3, each band with $r \times c$ elements. The code in Fig. 2 shows how one of the two input images is reconstructed in the original MPI+CUDA version of the program. 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 (lines 7–16), and represented as individual arrows in the left part of Fig. 3. The data received by each process are not retrieved from MPI buffers using as many individual `MPI_Recv` messages as data packages are sent to it. In turn, thanks to a user-defined `MPI_Type_vector` (line 25) the expected slice with eb_k bands of $r \times c$ elements is directly reconstructed using the parts coming from the other processes by means of a single `MPI_Recv` message (lines 22–28). Moreover, there are parts of data that are already available for each process. These parts can be rearranged with no message exchanging, just with proper local `memcpy` operations (lines 29–36).

3) Band processing and composition: In this step, for both input images, each GPU takes the corresponding slice of c columns, r rows and eb_k reduced bands, performs on them a high-pass filter and a multilayer fractional Fourier transform (MLFFT) [23], extracts log-polar coordinate maps and computes a phase correlation (Stages II to IV in Fig. 1). Some of these operations are computed by cuFFT [24] routines that expect `cufftComplex` data as inputs, so a previous data type conversion is needed. Regarding the reduction of the partial log-polar maps computed by each GPU into the final one (Stage V), these maps are first gathered in the root MPI process and then sent to its GPU in order to accumulate them using a specific kernel. These maps are in `cufftComplex` format, so that a suitable `MPI_Type_contiguous` type is defined to simplify these transfers.

4) Peak processing: In the first part of this step, the root MPI process uses Thrust [25] 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 process traverses cyclically the top subset of the ordered indexes array in order to obtain a partial maximum peak. 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 rotation *angle*, *scale* factor and cartesian *shift* shown in Fig. 1.

IV. Overview of the Hitmap library

Hitmap [5] is a library for the partition, mapping, and management of hierarchically distributed data structures at runtime. It was originally designed for dense arrays, and has been also extended to support sparse data structures, such as sparse matrices or graphs, using the same methodology and interface [18]. It is based on an SPMD (Single Program Multiple Data) model and the message-passing paradigm. Hitmap defines several abstractions to write parallel programs using distributed data structures. The functions in the library are grouped in three main modules.

Tiling functions. They allow the definition and management of hierarchically tiled data structures. These functionalities can be used independently of the rest of the library to improve locality on sequential code. They define classes to represent domains of indexes in a compact form. A class named `HitTile` represents the association between the elements of the indexes-domain space and the actual data, allowing the accesses to data with the same efficiency as manually developed codes that do not use the tile abstraction. A process can declare and allocate a subspace of the original domain, in order to create a distributed data structure.

Mapping functions. They include interchangeable modules that implement policies to automatically part and map domains in terms of the processes of a virtual topology. The virtual topologies are also generated by another class of policy modules at runtime. Neighbor relations across processes are established by these policies. The partitions are represented by objects named `HitLayout` that can be queried to obtain the indexes subdomain mapped to the local, a neighbor, or any other remote virtual process. New distributed arrays constructors have been added to the latest Hitmap version to directly map and allocate distributed arrays in terms of the selected topology and layout function names.

Communication functions. They are an abstraction of the message-passing model for tiles or tiles parts across virtual processes. They allow the creation of `HitCom` objects that store the information needed to marshall/unmarshall and exchange selected tile data across processes. Several interfaces for different types of point-to-point and collective communications are available. More complex patterns composed of multiple communication operations involving one or more tiles (several `HitCom` objects), are implemented as `HitPattern` objects. The constructor functions use the `HitLayout` objects associated to the distributed arrays to automatically determine who communicates and what. Thus, these objects are transparently adapted on construction time to the target platform details and the actual data distribution selected. These communication objects have a method that can be called at any time, and as many times as needed, to execute the communications. Internally, these objects exploit efficient MPI techniques such as derived data types, asynchronous communications, etc.

V. Applying Hitmap to Distributed HYFMGPU

In this section we describe the main changes we apply to the original MPI+CUDA distributed HYFMGPU implementation in order to hide the MPI layer, by exploiting the more abstract Hitmap funcionalities described in Sect. IV.

A. Environment initialization

First of all, the whole Hitmap environment must be initialized by invoking `hit_comInit()` (line 1 of Fig. 4). This

```
1  hit_comInit( &argc, &argv );
2
3  hitNewType( float );
4  HitTile_float im1_tileRows = hitDistribTile( float,
       hitShape( (BANDS), (ROWS), (COLUMNS) ),
       ArrayDimProjection(HIT_ROWSDIM), Blocks );
5  HitTile_float im2_tileRows = hitDistribTile( float,
       hitShape( (BANDS2), (ROWS2), (COLUMNS2) ),
       ArrayDimProjection(HIT_ROWSDIM), Blocks );
```

Fig. 4. Hitmap initialization and distributed tile creation code

```
1  HitTile_float ht_corr = hitDistribTile( float, hitShape
       ( (numBands*numBands) ), Plain, Copy );
2  cudaMemcpy(ht_corr.data,d_correlacion,sizeof(float)*
       numBands*numBands,cudaMemcpyDeviceToHost);
3  HitTile_float ht_corr_sum = hitDistribTile( float,
       hitShape( (numBands*numBands) ), Plain, Copy );
4  hit_comDoReduce(ht_corr, ht_corr_sum, HIT_RANKS_NULL,
       HIT_FLOAT, HIT_OP_SUM_FLOAT);
5  hit_tileFree(ht_corr);
6  cudaMemcpy(d_correlacion,ht_corr_sum.data,sizeof(float)
       *numBands*numBands,cudaMemcpyHostToDevice);
7  hit_tileFree(ht_corr_sum);
```

Fig. 5. All-reduce sum example in Hitmap

function internally calls `MPI_Init()` and should be paired at the end of the program with a call to `hit_comFinalize()`, which internally invokes `MPI_Finalize()`.

B. *Distributed memory allocation for images*

Hitmap provides a `hitDistribTile()` constructor to create tiles which are automatically distributed among the processors. This function receives the input native data type, the shape of the global buffer to distribute, the name of a topology function that groups the processors to create neighbor relations, and the name of a layout defining the desired type of distribution. Lines 4-5 in Fig. 4 show how both reference and target input images of the algorithm are distributed by rows following a `Blocks` layout on top of a topology that projects the processors along the rows dimension of the image (`ArrayDimProjection(HIT_ROWSDIM)`).

C. *All-reduce operations*

There are three points along the distributed HYFMGPU algorithm on which an all-reduce collective operation is needed: the first two ones are the centering of filtered images and the summation of the full correlation matrix, both in Step 1; the last one is the composition of the final correlation map at the end of Step 3. The code in Fig. 5 shows how the full correlation matrix from Step 1 is all-reduced in Hitmap. First, each process allocates two local copies of a matrix. The first one (line 1) is used to store a partial correlation matrix retrieved from the GPU using `cudaMemcpy()` (line 2). The second one is used to save the resulting all-reduced full matrix (line 3). A `Copy` layout function and a `Plain` non-projected topology are used to replicate both tiles across all the processes. The reduction operation is invoked using `hit_comDoReduce()` (line 4). This function receives the input and output tiles, the root process of the operation (or `HIT_RANKS_NULL` if it is an *all-reduce* operation), the data type, and the reduction operator (in this case, a single-precision float summation). Finally, each process copies the result to its GPU and deallocates the tiles (lines 5–7). Code excerpts for the other two aforementioned reductions are not shown, since they follow the same scheme.

D. *Data redistributions*

One of the most powerful abstractions offered by Hitmap is the `hit_patternDoRedistribute()` operation. It provides a quick and transparent mechanism to rearrange the data of a distributed `HitTile` into another tile with a different mapping. It is general enough to support both typical MPI collective operations, such as broadcast or gather, and more complex algorithm-dependent data redistributions.

Custom data redistributions. The rows-to-bands data rearrangement introduced in Step 2 of the distributed HYFMGPU algorithm (see Sect. III-2) is a representative example of the kind of custom data redistributions that can be smoothly implemented by means of this feature. The code presented in Fig. 6 shows how this redistribution is implemented with Hitmap for one image. Each process declares and allocates space for a global image tile distributed by blocks of rows (see line 1). However, the algorithm only works with the first bands of the image, more specifically the number of them indicated by the `PCAS` variable. This shrinking in the bands dimension is obtained adding an optional `HitShpView` parameter when creating the distributed tile. The data of the `PCAS` number of bands is retrieved from the associated GPU in line 3. Then, a second distributed tile is declared and allocated to store the same data but distributed by blocks of bands (see line 5). The rearrangement of the data from the first distributed tile to the second one is issued with a call to the `hit_patternDoRedistribute()` function. It receives as parameters the input and output tiles, which contain all the mapping information needed to issue the communications needed (see line 6). Finally, once the data reach the destination tile, the origin tile can be freed (see line 7).

A visual comparison of this code with the equivalent code in the original MPI version presented in Fig. 2, shows that the programmer needs about five times less lines of code to implement the same image rearrangement with Hitmap, and using a much clearer programming interface. Now the redistribution can be written focusing on algorithmic terms (from *image rows* to *image bands*) rather than on the MPI low-level and error-prone details such as custom vector types definitions or offset calculations needed to unpack data following the proper layout. A more comprehensive evaluation of these advantages is presented in Sect. VI.

Alternative to MPI collectives. Programmers can also exploit the pattern-based redistribution functionalities of Hitmap to abstract from MPI collective primitives like `MPI_bcast` or `MPI_gather`. For example, after the peak processing performed in Step 4, the partial peaks obtained should be gathered and examined by a single process. The code in Fig. 7 shows how this specific stage is implemented in Hitmap. Peaks

```
HitTile_float ht_im1_pca_geosliced = hitDistribTile( float, hitShape( (BANDS), (ROWS), (COLUMNS) ), ArrayDimProjection
    (HIT_ROWSDIM), Blocks, hitShpView( (HIT_BANDDIM, HIT_SHAPE_FIRST, PCAS) ) );

cudaMemcpy(ht_im1_pca_geosliced.data,d_pca1_geosliced,sizeof(float)*COLUMNS*im1_tileRows_rows*PCAS,
    cudaMemcpyDeviceToHost);

HitTile_float im1_tileBands = hitDistribTile( float, hitShape( (PCAS), (ROWS), (COLUMNS) ), ArrayDimProjection(
    HIT_BANDDIM), Blocks );
hit_patternDoRedistribute(ht_im1_pca_geosliced, im1_tileBands, HIT_FLOAT);
hit_tileFree(ht_im1_pca_geosliced);
```

Fig. 6. Implementation of the rows-to-bands redistribution of one of the two images using Hitmap.

```
1  HitType Hit_Peak;
2  hit_comTypeStruct(&Hit_Peak, CPeak, 6, peak_id, 1,
       HIT_INT, max2, 1, HIT_DOUBLE, scal, 1, HIT_DOUBLE,
       rot, 1, HIT_DOUBLE, indx, 1, HIT_INT, indy, 1,
       HIT_INT);
3  HitTile_CPeak allPeaksTile_Gather = hitDistribTile(
       CPeak, hitShape( (world_size) ), Plain, Blocks );
4  hit(allPeaksTile_Gather, 0) = local_peak;
5  HitTile_CPeak allPeaksTile_InLeader = hitDistribTile(
       CPeak, hitShape( (world_size) ), Plain, InLeader );
6  hit_patternDoRedistribute(allPeaksTile_Gather,
       allPeaksTile_InLeader, Hit_Peak);
7  if(hit_layImActive(hit_tileLayout(allPeaksTile_InLeader
       ))) {
8    CPeak* local_peaks = allPeaksTile_InLeader.data;
9    /*
10    * Host-side peak selection code, omitted
11    */
12 }
13 hit_tileFree(allPeaksTile_Gather);
14 hit_tileFree(allPeaksTile_InLeader);
```

Fig. 7. Partial peaks gather in Hitmap

have their own structured type, so first we must create its equivalent in Hitmap (lines 1–2). Since every process returns one partial peak, a distributed tile with as many elements as processes is needed. This tile is declared and allocated in line 3, using a `Plain` topology and a `Blocks` layout with a shape containing one index per process. Then, each process saves its peak value in the single tile position it physically owns (line 4). Another tile is needed to gather all the partial peaks in the *leader* process. Hitmap uses the term *leader* to identify a special process that can be used as the equivalent of an MPI's *root* process. This tile is created in line 5 using the same `Plain` topology but an `InLeader` layout. Let us note that although every process is calling this instruction, the `InLeader` layout makes the tile to be physically allocated only in the leader of the topology. The gather is effectively performed in line 6, by means of the corresponding `hit_patternDoRedistribute()` call. Once the leader have all the partial peaks, it can process them to select the final candidate (see lines 7–12). The condition in line 7 restricts this operation to the leader process. As usual, tiles are freed when they are not needed anymore (lines 13–14).

E. Integration with specific GPU libraries

Both the original MPI+CUDA program, and this new Hitmap+CUDA implementation, make an intensive use of GPU libraries such as cuBLAS [26], cuFFT [24], cuSOLVER [22], the NVIDIA Performance Primitives (NPP) [27], and Thrust [25]. Programming interfaces of such libraries generally expect device-side buffers as input/output arguments, so explicit `cudaMemcpy()` calls are still needed. Hitmap smoothly provides host-side pointers to be passed as arguments to such calls via the `data` attribute of `HitTile` objects, as it is shown in multiple source lines in Figs. 5 to 7.

TABLE I
MEASUREMENTS OF DEVELOPMENT EFFORT METRICS FOR THE DISTRIBUTED HYFMGPU ALGORITHM

Operation	Version	LOC	TOC	CCN	HAL
All-reduces	Hitmap	22	354	0	14694
	MPI	24	362	0	13393
Collective-based redistributions	Hitmap	45	501	4	42546
	MPI	48	397	6	36818
Rows-to-bands rearrangement	Hitmap	14	239	0	12282
	MPI	79	768	10	239745
Full tool code	Hitmap	1451	14930	267	71948436
	MPI	1595	15990	305	75663635

VI. EXPERIMENTAL STUDY

This section describes an experimental study to assess the advantages of using Hitmap on the distributed multi-device version of HYFMGPU described in Sect. V. The study is focused on two aspects: the code complexity or development effort, and the performance effects that Hitmap may introduce due to its additional abstraction layer.

A. Development effort measures

First we analyze the differences in development effort between the Hitmap version and the baseline implementation using pure MPI primitives. We measure four classical development effort metrics:

- Lines of code (LOC) and number of tokens (TOC), which offer plain figures about the volume of code that the programmer should develop.
- McCabe's cyclomatic complexity (CCN), which intends to assess the rational effort needed in terms of code divergences and potential issues that should be considered to develop, test, and debug the program [28].
- Halstead development effort (HAL), which uses both code complexity and volume indicators to obtain a comprehensive measure of the development effort [29].

Table I shows the values measured for the aforementioned metrics in both Hitmap and MPI versions of the program. The

TABLE II
DIMENSIONS AND SIZE IN GB OF
SYNTHETIC HYPERSPECTRAL IMAGES USED AS TEST CASES

Input	Width	Height	Bands	Image size
C1	614	2678	220	1.45 GB
C2	512	4096	224	1.88 GB
C3	1228	2678	220	2.89 GB
C4	1024	4096	224	3.76 GB

table present measurements for the whole program (bottom), and for code snippets of the coordination code that respectively comprise: (1) The all-reduce operations; (2) The collective-based redistributions; and (3) The rows-to-bands rearrangement. For both the all-reduce operations and the collective-based redistributions the numbers of lines and tokens are similar among MPI and Hitmap versions. However, the Halstead metrics indicates that the Hitmap interface is slightly more complex. This is due to the creation and use of auxiliary tiles, shapes, layouts or topologies that are not directly related with the creation of the distributed tiles. The power of Hitmap abstractions gets revealed when implementing more complex data movements. For the rows-to-bands rearrangement, the Hitmap version can express the same operation in less than a fifth of the number of source lines than MPI. The cyclomatic complexity disappears and the Halstead value drops in one order of magnitude. These are the effects of all the message exchange logic being delegated to Hitmap. Moreover, in this case not only is the development effort reduced in quantitative terms, but also an important amount of time in debugging tasks is saved. Finally, the metrics for the full tool code show the overall development effort reduction obtained when using Hitmap, which affects only the part of code devoted to coordination and data management in the distributed multi-GPU version of the HYFM algorithm.

B. Performance impact of Hitmap

In this section we study the performance impact of adding the additional abstraction layer provided by Hitmap. These tests have been run using four pairs of randomly generated matrices as synthetic hyperspectral images. Table II shows the main properties of a single image of each pair. The HYFM algorithm properties allow the usage of such inputs for performance evaluation purposes, as its performance is insensitive to translation, rotation, scale, and noise in the input images [3].

The testbed was 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 410.48 driver. The code has been compiled under Linux using nvcc and other CUDA libraries provided with CUDA Toolkit 10.0. The MPI support was provided by mpich-3.2.1.

TABLE III
PERFORMANCE RESULTS FOR
HITMAP VERSION OF DISTRIBUTED HYFMGPU

Test case			$N = 10$ runs each version		
			$\bar{x}$ (wall time)	σ	Overhead
C1	2 GPU	Hitmap	8.07 s	0.012	−36.48 %
		MPI	12.71 s	0.028	
	4 GPU	Hitmap	5.60 s	0.174	−35.52 %
		MPI	8.69 s	0.705	
C2	2 GPU	Hitmap	8.24 s	0.017	−37.24 %
		MPI	13.13 s	0.016	
	4 GPU	Hitmap	6.27 s	0.041	−31.51 %
		MPI	9.15 s	1.011	
C3	2 GPU	Hitmap	11.43 s	0.014	−23.09 %
		MPI	14.86 s	0.110	
	4 GPU	Hitmap	7.43 s	0.124	−29.59 %
		MPI	10.56 s	1.411	
C4	2 GPU	Hitmap	13.08 s	0.015	−17.89 %
		MPI	15.93 s	0.283	
	4 GPU	Hitmap	8.31 s	0.146	−21.55 %
		MPI	10.59 s	1.538	

Table III contains, for each test case, the average ($\bar{x}$) and the standard deviation (σ) of the wall time after 10 runs of both versions of the code, distributing the work among 2 or 4 GPUs. Moreover, the performance overhead introduced by Hitmap when comparing it with the pure MPI implementation is shown in a separated column. Overhead figures are negative for all the test cases, ranging from -17.89% to -37.48%. In other words, the Hitmap abstraction layer put on top of the MPI version of distributed multi-device HYFMGPU did not introduce performance overhead at all, but it yielded a systematic performance gain for all the test cases run. In addition, the application performance is in general more stable when using Hitmap, as it is shown in the deviation column. Squeezing all the potential of MPI when implementing distributed versions of complex algorithms is an arduous task, and it usually needs a thorough knowledge of asynchronous and non-blocking functionalities. If they are not used properly, faulty scenarios that are quite difficult to debug tend to appear. To avoid such situations programmers commonly write their codes having in mind a trade-off among performance and complexity. For instance, in the MPI version of the rows-to-bands rearrangement shown in Fig. 2 each process sends data asynchronously, but receives them with a blocking version of the receive operation. The internal logic of `hit_patternDoRedistribute()` is based on fully asynchronous and skewed communication patterns, deriving in a performance improvement when porting this part of the algorithm to Hitmap.

VII. CONCLUSIONS

In this paper we present a Hitmap version of the distributed multi-GPU MPI+CUDA implementation of HYFM, a hyperspectral image registration algorithm. We review the main functionalities offered by Hitmap to manage distributed arrays, and recall how the multi-device version of HYFMGPU divides up the work among several GPUs. Then, we present a detailed explanation about how Hitmap can simplify the programming tasks of such a distributed multi-device code, focusing on some interesting parts of the algorithm. Four different development effort metrics have been measured for both Hitmap and MPI versions. In general terms, thanks to Hitmap, less development effort is needed. The development of the rows-to-bands rearrangement specially takes advantage of Hitmap usage, with a Halstead score one order of magnitude lower than the previous MPI implementation. To assess the impact of Hitmap in the performance of the overall application, tests with both 2 and 4 GPUs have been run using four pairs of synthetic images with sizes up to 3.8 GB. Hitmap did not introduced performance overhead but it yielded gains for all the test cases run, with wall times reductions between 17.89% and 37.48%. These gains are attributed to the efficient exploitation that Hitmap internally does of global communication information and MPI asynchronous and non-blocking communications, which can improve suboptimal user algorithm implementations.

Finally, some future research lines are proposed. In order to decrease even more the effort needed to implement all-reduces and other collective operations, improvements of the Hitmap interface can be introduced. The performance of the Hitmap implementation of the distributed multi-device HYFMGPU algorithm has been only assessed inside a single node equipped with a number of identical GPUs. Scenarios with GPUs in distributed nodes, and mixing GPUs with different capabilities should be tested to wholly assess the performance of the approach. The integration of Hitmap with the Controllers model [20] for generic accelerators programming could also simplify the programming and portability of the GPU code.

References

[1] A. Ordóñez, F. Argüello, and D. B. Heras, "GPU accelerated FFT-based registration of hyperspectral scenes," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 10, no. 11, pp. 4869–4878, Nov 2017.

[2] A. Ordóñez, F. Argüello, and D. B. Heras, "Fourier–Mellin registration of two hyperspectral images," *International Journal of Remote Sensing*, vol. 38, no. 11, pp. 3253–3273, 2017.

[3] B. S. Reddy and B. N. Chatterji, "An FFT-based technique for translation, rotation, and scale-invariant image registration," *IEEE Transactions on Image Processing*, vol. 5, no. 8, pp. 1266–1271, Aug 1996.

[4] J. Fernández-Fabeiro, A. Ordóñez, A. Gonzalez-Escribano, and D. B. Heras, "A multi-device version of the HYFMGPU algorithm for hyperspectral scenes registration," *The Journal of Supercomputing*, 2018, Online First article. Last access: 2019-03-25. [Online]. Available: https://rdcu.be/bbLZW

[5] A. Gonzalez-Escribano, Y. Torres, J. Fresno, and D. R. Llanos, "An extensible system for multilevel automatic data partition and mapping," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1145–1154, May 2014.

[6] H. Jiang, S. Chen, D. Li, C. Wang, and J. Yang, "Papaya tree detection with UAV images using a GPU-accelerated scale-space filtering method," *Remote Sensing*, vol. 9, no. 7, 2017, paper 721.

[7] S. Zhuang, P. Wang, B. Jiang, G. Wang, and C. Wang, "A single shot framework with multi-scale feature fusion for geospatial object detection," *Remote Sensing*, vol. 11, no. 5, 2019, paper 594.

[8] M. Rossi, P. Trslić, S. Sivčev, J. Riordan, D. Toal, and G. Dooly, "Real-time underwater stereofusion," *Sensors*, vol. 18, no. 11, 2018, paper 3936.

[9] L. Linghu, J. Wu, Z. Wu, and X. Wang, "Parallel computation of EM backscattering from large three-dimensional sea surface with CUDA," *Sensors*, vol. 18, no. 11, 2018, paper 3656.

[10] E. Martel, R. Lazcano, J. López, D. Madroñal, R. Salvador, S. López, E. Juarez, R. Guerra, C. Sanz, and R. Sarmiento, "Implementation of the Principal Component Analysis onto high-performance computer facilities for hyperspectral dimensionality reduction: Results and comparisons," *Remote Sensing*, vol. 10, no. 6, 2018, paper 864.

[11] F. Rançon, L. Bombrun, B. Keresztes, and C. Germain, "Comparison of SIFT encoded and deep learning features for the classification and detection of Esca disease in Bordeaux vineyards," *Remote Sensing*, vol. 11, no. 1, 2018, paper 1.

[12] J. Drönner, N. Korfhage, S. Egli, M. Mühling, B. Thies, J. Bendix, B. Freisleben, and B. Seeger, "Fast cloud segmentation using convolutional neural networks," *Remote Sensing*, vol. 10, no. 11, 2018, paper 1782.

[13] F. Bonelli, M. Tuttafesta, G. Colonna, L. Cutrone, and G. Pascazio, "An MPI-CUDA approach for hypersonic flows with detailed state-to-state air kinetics using a GPU cluster," *Computer Physics Communications*, vol. 219, pp. 178–195, 2017.

[14] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[15] D. A. Mallón, A. Gómez, J. C. Mouriño, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, R. Doallo, and B. Wibecan, "UPC performance evaluation on a multicore system," in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, ser. PGAS '09. New York, NY, USA: ACM, 2009, pp. 9:1–9:7.

[16] B. B. Fraguela, G. Bikshandi, J. Guo, M. J. Garzarán, D. Padua, and C. von Praun, "Optimization techniques for efficient HTA programs," *Parallel Computing*, vol. 38, no. 9, pp. 465 – 484, 2012.

[17] Y. Chen, X. Cui, and H. Mei, "Parray: A unifying array representation for heterogeneous parallelism," *SIGPLAN Not.*, vol. 47, no. 8, pp. 171–180, Feb. 2012.

[18] J. Fresno, A. Gonzalez-Escribano, and D. R. Llanos, "Blending extensibility and performance in dense and sparse parallel data management," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2509–2519, Oct 2014.

[19] A. Moreton–Fernandez, H. Ortega–Arranz, and A. Gonzalez–Escribano, "Controllers: An abstraction to ease the use of hardware accelerators," *The International Journal of High Performance Computing Applications*, vol. 32, no. 6, pp. 838–853, 2018.

[20] A. Moreton-Fernandez, A. Gonzalez-Escribano, and D. R. Llanos, "Multi-device controllers: A library to simplify parallel heterogeneous programming," *International Journal of Parallel Programming*, vol. 47, no. 1, pp. 94–113, Feb 2019.

[21] E. Rodriguez-Gutiez, F. Martinez-Gil, J. M. Orduña, and A. Gonzalez-Escribano, "MARL-Ped+Hitmap: Towards improving agent-based simulations with distributed arrays," in *Algorithms and Architectures for Parallel Processing*. Springer International Publishing, 2016, pp. 212–225.

[22] "NVIDIA Corporation: cuSOLVER Library User's Guide," Last access: 2019-03-25. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUSOLVER_Library.pdf

[23] W. Pan, K. Qin, and Y. Chen, "An Adaptable-Multilayer Fractional Fourier Transform Approach for Image Registration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 3, pp. 400–414, 2009.

[24] "NVIDIA Corporation: cuFFT Library User's Guide," Last access: 2019-03-25. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf

[25] "NVIDIA Corporation: Thrust Quick Start Guide," Last access: 2019-03-25. [Online]. Available: https://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf

[26] "NVIDIA Corporation: cuBLAS Library User's Guide," Last access: 2019-03-25. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf

[27] "NVIDIA Corporation: NVIDIA Performance Primitives (NPP)," Last access: 2019-03-25. [Online]. Available: https://docs.nvidia.com/cuda/npp/index.html

[28] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[29] M. H. Halstead, *Elements of software science*, ser. Operating and programming systems. Elsevier, 1977.