Journal of Computational and Applied Mathematics 273 (2015) 414–427

Contents lists available at ScienceDirect

IranPaper.blog.ir

journal homepage: www.elsevier.com/locate/cam

An MPI–CUDA library for image processing on HPC architectures

Antonella Galizia *, Daniele D'Agostino, Andrea Clematis

Institute of Applied Mathematics and Information Technologies, National Research Council of Italy, Genoa, Italy

A R T I C L E I N F O

Article history:
Received 4 October 2013
Received in revised form 3 May 2014

Keywords:
Image processing
Parallel computing
GPU

A B S T R A C T

Scientific image processing is a topic of interest for a broad scientific community since it is a mean of gaining understanding and insight into the data for a growing number of applications. Furthermore, the technological evolution permits large data acquisition, with sophisticated instruments, and their elaboration through complex multidisciplinary applications, resulting in datasets that are growing at an extremely rapid pace. This results in the need of huge computational power for the processing. It is necessary to move towards High Performance Computing (HPC) and to develop proper parallel implementations of image processing algorithms/operations. Modern HPC resources are typically highly heterogeneous systems, composed of multiple CPUs and accelerators such as Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs). The actual barrier posed by heterogeneous HPC resources is the development and/or the performance efficient porting of software on such complex architectures. In this context, the aim of this work is to enable image processing on cluster of GPUs, through the use of PIMA(GE)^2 Lib, the Parallel IMAGE processing GEnoa Library. The library is able to exploit traditional clusters through MPI, GPU device through CUDA and a first experimentation is aimed to explore the use of GPU-clusters. Library operations are provided to the users through a sequential interface defined to hide the parallelism of the computation. The parallel computation, at each level, is managed employing specific policies designed to suitably coordinate the parallel processes/threads involved in the elaboration and their use is tightly coupled with the PIMA(GE)^2 Lib interface. In this paper, we present the incremental approach adopted in the development of the library and the performance gains in each implementations: quite linear speedup is achieved on cluster architecture, about a 30% improvement in the execution time on a single GPU and the first results on cluster of GPUs are promising.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

During the last decades image processing has become an important topic of interest for a broad scientific community since it is a mean of gaining understanding and insight into the data for a growing number of applications. The variety of problems arising with the different application domains leads to the evolution of several specific analysis and processing techniques that mainly differ in the type of images to manage and/or in the task to perform. More specifically, these techniques mainly could be divided into three categories [1]:

* Corresponding author. Tel.: +39 3381823073.
E-mail addresses: antonella@ge.imati.cnr.it, galizia@ge.imati.cnr.it (A. Galizia), dagostino@ge.imati.cnr.it (D. D'Agostino), clematis@ge.imati.cnr.it (A. Clematis).

http://dx.doi.org/10.1016/j.cam.2014.05.004
0377-0427/© 2014 Elsevier B.V. All rights reserved.

- *Image processing*, also called low-level image processing: when the manipulation input and output data are images. This kind of elaboration has a local nature and the computation is performed applying an operation for each pixel in an image. Examples are: basic operations (e.g. point arithmetic, binary operations), basic filter operations (e.g., smoothing, edge enhancement) and image transformations (e.g., rotation, scaling).
- *Image analysis*, also called intermediate level image processing: when starting from an image, the elaboration extracts measurements. In this case the image is reduced into segments and the manipulation produces a more compact structure to represent the image. Examples are: region labelling and object tracking.
- *Image understanding*, also called high-level image processing: when the manipulation starts with an image and extracts high-level descriptions. It primarily concerns the output obtained from the Image Analysis, the operations try to imitate human cognition and decision-making. Examples are: object recognition and semantic scene interpretation.

Nowadays, the technological evolution permits large data acquisition, through sophisticated instruments, to be elaborated by complex multidisciplinary applications. This results in the need of high computational power for the processing. There is the need to move towards High Performance Computing (HPC) and to develop proper parallel implementations of image processing algorithms [2,3]. Modern HPC resources are typically highly heterogeneous systems, composed of multiple CPUs and accelerators [4]; latest generation CPUs are hierarchical, since they are made up of multicore processors that can also access powerful specialized hardware, as Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs) [5]. Such resources can offer actually significant computational power, in the order of TFlops, that can be achieved only with a massive exploitation of such huge parallelism. The actual barrier posed to the exploitation of modern heterogeneous HPC resources are the difficulties in the development and the porting of software on such architectures, a careful restructuring of existing algorithms is required to achieve performance improvement [6,7]. The development of efficient software has to take into account key points as the efficient data management since performance can be dominated by memory traffic, rather than floating-point operations; moreover, in terms of memory management, the different kinds of processors pose new and major issues. In the image processing field, valuable contributions have been provided [8,9].

With these ideas in mind, the aim of this work is to enable image processing on heterogeneous resources adopting suitable programming tools to exploit specific levels of complex architectures and in particular clusters of GPUs, i.e. a cluster in which each node is equipped with a GPU. We develop PIMA(GE)^2 Lib, the Parallel IMAGE processing GEnoa Library, to provide the most common low level image processing operations. The development of the library reflects an incremental approach: the first version of the library has been developed to run on cluster-computing environments using MPI [10] the standard de facto for such architectures; the adoption of suitable parallelization policies permits to obtain interesting values of performance and scalability. A second parallel implementation of the library is oriented to GPU devices; the parallelism at this level has been implemented through CUDA [11], also in this case a suitable strategy enables to speed up the computation on such devices. As final third step, a new study is aimed to explore the use of GPUs cluster through a proper integration of the previous implementations, i.e. MPI is used to distribute the computation and CUDA as the main execution engine. The parallelism of the library, at each level, is completely transparent to the users thanks to a sequential interface suitably defined.

The paper is organized as follow: a brief overview of the heterogeneous architectures and parallel tools used in the development of the library is given in Section 2. In Section 3, we present the general structure of the library that can be considered independent from the different implementations, together with a real bioinformatics application, i.e. the analysis of images obtained through the Tissue MicroArray (TMA) technology, used to test speedup values of the library. In Section 4, the approach adopted in the development the MPI implementation of PIMA(GE)^2 Lib and code scalability are detailed. In Section 5, the CUDA implementation of PIMA(GE)^2 Lib is discussed (again) in terms of parallelization strategy and performance achieved. Section 6 describes the combination of both implementations with experimental results collected on a set of operations. Section 7 concludes the work.

2. Heterogeneous HPC architectures

In modern complex computing systems the computational cores, memories and communication bandwidth can be extremely heterogeneous. This means that to get the expected level of performance is mandatory to manage effectively such intrinsic architectural complexity. For example, data movement among the memories of the different kinds of processors represents a new and major issue and restrictions on memory and interprocessor bandwidth make locality of data an important consideration. The view of such heterogeneous computational systems corresponds to different types of parallel cooperation among parallel processes: distributed memory for cooperation among nodes; shared memory for core cooperation; SIMD parallelism inside CPUs and accelerators. The true challenge in using a similar system is the programming of parallel applications that are able to exploit in an effective way the different levels and capabilities. This leads to the development of a large number of software libraries supporting the parallel programming; in particular, in this work we use MPI and CUDA. MPI (Message Passing Interface) is a language-independent communications library used to program parallel computers. It supports explicit communications among processes that constitute a parallel program running on a distributed memory system. Communications can be both point-to-point and collective. MPI's goals are high performance, scalability and portability. MPI implementation can be clever enough to spot that it is being used in a shared memory environment and therefore optimizes its behaviour accordingly. Designing programs around the MPI model (contrary to explicit shared memory models) may have advantages over NUMA architectures since MPI encourages memory locality.

The Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model created by nVidia that gives developers access to the instruction set and memory of the parallel computational elements in nVidia GPUs. CUDA is accessible to software developers through CUDA-accelerated libraries, compiler directives, extensions to programming languages such as C/C++ and Fortran and other interfaces, including OpenCL, Microsoft's Direct Compute and C++ AMP. It performs at a lower level compared with the previous tools and thus it requires more programming skills.

To test the implementations proposed in this paper, we use a heterogeneous cluster. It is composed by three blades, each one is equipped with two Intel Xeon E5645 processors (six cores each) 32 GB of RAM and an nVidia GTX 580 GPU with a 1.5 GB of GPU memory and interconnected with a Gigabit Ethernet, thus proving 36 cores for MPI computations and a cluster of GPUs with 3 nodes.

3. The PIMA(GE)^2 Lib general behaviour

The work is focused on low-level image processing; usually images could be photographs or frames of video or images acquired using specific instruments such as biomedical machines and more in general imaging instruments. We considered the Image Algebra theory [1] to identify a set of consistent and well defined operators for image processing.

Furthermore, we decided to implement a software library since the use of sequential and parallel libraries is well assessed for numerical applications. Software evolution history provides examples in this direction through the diffusion of different packages, in particular for numerical applications [12], but also for the field of image processing [13,14]. A sequential library represents a user-friendly tool that provides optimized and ready-to-use operations in a specific domain. A parallel library is a more complex tool, since it has to manage in a proper way the burdens related to set up the parallel environment, the management of communication, data exchange, input/output handling, etc. Their exploitation to develop demanding applications is a common practice, which provides optimized performance on a wide range of target architectures [15,16].

PIMA(GE)^2 Lib elaborates bidimensional and tridimensional datasets and may perform operations both in sequential and in parallel on the architectures already specified. In the second case, the parallelism is transparent to the users thanks to the definition of a sequential interface [17]. A proper management of parallel computations is obtained with a set of functionalities; the specific parallelization policies applied on the exploited architecture are presented in the remaining of the paper. In Section 3.1, we introduce the image processing operations offered by the library and their classification in groups, in Section 3.2 the sequential operation APIs (Application Programming Interface) are presented and Section 3.3 introduces a real application, the analysis of TMA images, used to test library performance.

In terms of code design, i.e. organization and modularity, PIMA(GE)^2 Lib follows the directions indicated by significant and successful examples software library as ScaLAPACK [16], in this way it is possible to derive many important features, such as maintenance, extensibility and efficiency. The great efforts spent on this point during the development of the first version of the library, i.e. MPI implementation, actually paid during the development of the second and the experimentations of third version of the library, i.e. CUDA and the mix of CUDA and MPI implementations, and result in a quite smooth integration of the different implementations behind a unique interface.

3.1. Image processing operations

PIMA(GE)^2 Lib provides parallel implementations of the most common low-level image processing operations. During the implementation of the library, we started from two simple considerations:

- a large class of image processing operations requires a spatially localized portion of the input image in order to compute the required output image. In the simplest case, an output image is computed by processing single pixels of the input image, more in general, only neighbourhood pixels are used to compute an output pixel. For this reason it is possible to compute the output data in largely independent way, e.g. in parallel;
- a large number of image processing routines uses a relatively small number of computational kernels, where each kernel uses a similar methodology to implement computational engine. Therefore, it is possible to consider each image processing operation as belonging to a group and define image operation classes.

Such considerations mean that most of the image processing functions are easily parallelizable in a data parallel fashion. Grouping together the operations, we can outline their computational behaviour and derive a few schemes, acting as guide in the parallelization strategy. Thus we group together similar image processing operations, defining an *algorithmic pattern* that characterizes each group. The algorithmic patterns may correspond to the execution (also in parallel) of sequential operations and may require one or more communication phases. They provide a restricted number of parallel behaviours and results useful to avoid redundant data communications/movement phases in more complex application developed as a sequence of library operations, as testified by successful experiences in the field [14].

Thus, the first step in the development of PIMA(GE)^2 Lib has been the definition of conceptual classes of image processing operations grouped according to their algorithmic pattern. This represents a well-established code arrangement, already presented in the literature [13,14,18].

Operations classes/groups have been structured in logical levels, obtaining a hierarchy of image processing operation classes; operations of the same class have the same "sort", i.e. act on the same data types, have the same input/output data type and elaborate images following the same schema (again, algorithmic pattern). Abstract data types, i.e. IMAGE,

MATRIX and KERNEL, together with a set of basic functions/methods for their management, have been defined to support the computation; we can mention functionalities to acquire IMAGEs, to define specific KERNELs for convolutions and MATRIXs for the Geometric operations. We consider five conceptual Image operation classes [17]:

1. *Point operations* that take in input one IMAGE and apply a unary operation, such as square root, absolute value, and threshold;
2. *Reduction operations* that take in input one IMAGE and a collective value, i.e. the maximum, minimum pixel value;
3. *Image arithmetic operations* that take in input two IMAGE and apply a binary operation, i.e. an operation that combines two elements, e.g. addition, product, etc.;
4. *Geometric operations* that take in input one IMAGE and one MATRIX and apply a Region Of Interest (ROI) operation as restriction, or a domain transformation as a scaling;
5. *Convolution operations* that take in input one IMAGE and one KERNEL and apply a convolution, as Gaussian convolution, erosion, and dilation.

We developed further two fundamental methods: OpenLib and CloseLib, mandatory to use the library functionalities. The former is aimed to start-up the parallel environment and the information related to the parallel processes involved in the computation. In a specular manner, the latter terminates parallel processing and library operations. Any library calls made after this function raise an error.

3.2. A user friendly API

To allow a simple and coherent use of PIMA(GE)^2 Lib, the most relevant element is the definition of a effective interface, that is coupled to the parallelization policies hidden by such interface, i.e. the call of operations (through the API) activates proper functionalities. The design of the library operation classes directly provides a model to derive interfaces with the aforementioned properties; in box code Listing 1, we provide a few examples of operation interfaces encoded in C programming language.

Listing 1 Example of library interfaces.

```
// Functionalities mandatory to enable the use of the library
ERROR OpenLib (int *pArgc, char ***pArgv); ERROR CloseLib ();

// Functionalities for IMAGE management
IMAGE *ReadImageFromFile(char *pFileName, int pWidth, int pHeight, int
pDepth); ERROR WriteImageToFile(char *pFileName, IMAGE *pImPtr);

//Point Operations
ERROR UnaryPixOpC(IMAGE *pImgPtr, UNARY_OP_ID pOpId);

//Reduction Operations
ERROR ReductionOp(IMAGE pImgPtr, REDUCTION_OP_ID pOpId, ARITH *pRetVal);

//Image Arithmetic Operations (with an IMAGE parameter)
ERROR BinaryPixOpI (IMAGE *pImgPtr, IMAGE *pArgImgPtr, BINARY_OP_I_ID pOpId);

//Convolution operations
ERROR ConvolutionOp(IMAGE *pImgPtr, KERNEL *pKerPtr, CONV_OP_ID pOpId);

//Geometric operations (ROI)
ERROR GeoRoiOp(IMAGE *pImgPtr, IMG COORD pBegin, IMG AXES pNewDim,
PIXEL *pBackground, GEO_ROI_OP_ID pOpId, IMAGE **pResPtr);
```

The APIs reflect the algorithmic pattern of different classes of operators of the library; users have to specify the actual operation inside the class through the parameters `pOpId`, e.g. `REDUCTION_OP_ID` could be `MAX`; this parameter has been implemented with enumerative data type. From the user's point of view, this allows an easier management of the library operations, since he/she is no longer involved with a large number of functions, but he/she has to consider and handle a small set of clear and well-defined classes. Furthermore, the parallel environment and data partition/communication never appear in the interfaces, neither as parameters. The sequential API of the library shields a software layer devoted to manage the parallel computation with respect to the different architectures, acting as a controller. For more details about PIMA(GE)^2 Lib operations and API, please refer to [19].

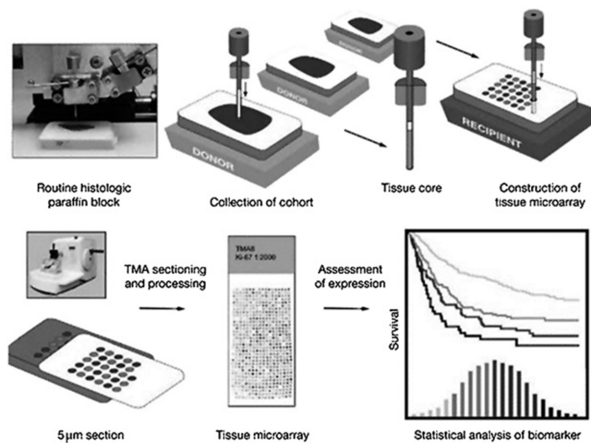

Fig. 1. Construction and use of tissue microarrays for biomarker identification, source [22].

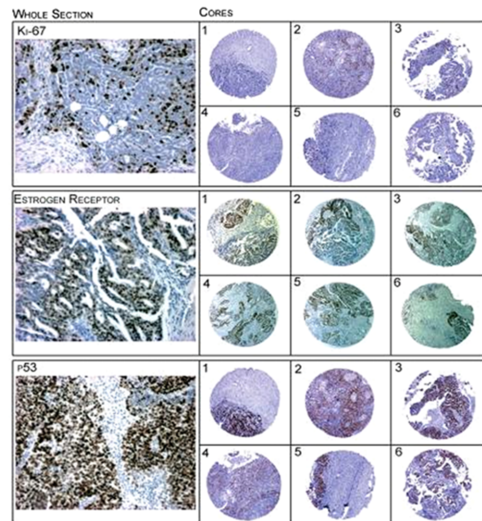

Fig. 2. TMAs used to compare whole sections of tissues with cores, source [24].

3.3. Case study: TMA image analysis

To test the performance achievable using PIMA(GE)^2 Lib, we consider a real application: the analysis of the images obtained considering the Tissue MicroArray (TMA) technology. TMA was developed to address issues derived by the use of high throughput technologies in biology; in fact, high throughput technologies are able to produce a large amount of genomic, transcriptomic and proteomic data, that need to be screened and validated by specific techniques.

TMA experiments represent a good validation for data generated following other biotechnological methods, in particular gene expression microarray data. Actually, the major limitations in the molecular clinical analysis of tissues include the cumbersome nature of the procedures, the limited availability of diagnostic reagents and the limited patient sample. TMA technique can evaluate the presence of a single biological entity (i.e. the expressed gene) in a parallel way on hundreds of different tissues included into the same paraffin block [20,21]. More in details, TMAs consist of paraffin blocks in which up to 1000 separate tissue cores are assembled in array fashion to allow simultaneous histological analysis. To obtain TMA paraffin blocks, a hollow needle is used to remove tissue cores as small as 0.6 mm in diameter from regions of interest in paraffin-embedded tissues such as clinical biopsies or cancer samples [23]. These tissue cores are then inserted in a recipient paraffin block in a precisely spaced, array pattern, as depicted in Fig. 1. Sections from this block are cut using a microtome, mounted on a microscope slide and then analyzed by any method of standard histological analysis. Each microarray block can be cut into 100–500 sections, which can be subjected to independent tests. In Fig. 2, examples of TMA images and their use are provided. Common tests on TMAs include immunohistochemistry and fluorescent in situ hybridization. Thus, each TMA block produces many treated spot views to be electronically acquired and scientists have to handle and elaborate a large number of high-resolution images.

As a case study we consider an edge detection applied to a single tissue example developed using the operations provided with PIMA(GE)^2 Lib. The TMA image on the right side of Fig. 3, whose size is about 5 MB, is the input of the algorithm. It represents a haematoxylin–eosin reaction: colourations differentiate basic and acid entities in a colon tissue slice. Basic elements (like cytoplasm) are highlighted in static grey while acid elements (like nuclei) are in light grey. The TMA on the left represents the result of the edge detection. Performance figures of such elaborations are reported and discussed in the specific sections, respectively MPI in Sections 4.3 and 5.3.

As for the edge detection applied, we considered the study made in [25] and already used to exemplify the performance of parallel image processing library [14]. In the proposed approach the detection is solved applying anisotropic Gaussian filters that depend at various levels on each possible line direction. The cited filtering problem can be implemented with a full bidimensional convolution for each filter along line directions; for each direction, the algorithm considers two further "*for*" loops and operations to properly provide the solution. Note that specific numerical details of the algorithm itself are out of the scope of this paper, which concerns the development of user-friendly parallel tools for image processing; for algorithmic details consider [25]. A pseudo code of the implementation using the operations provided in PIMA(GE)^2 is presented in Listing 2; thanks to the sequential interface of the library, the implementation appears equal in the three versions of the library.

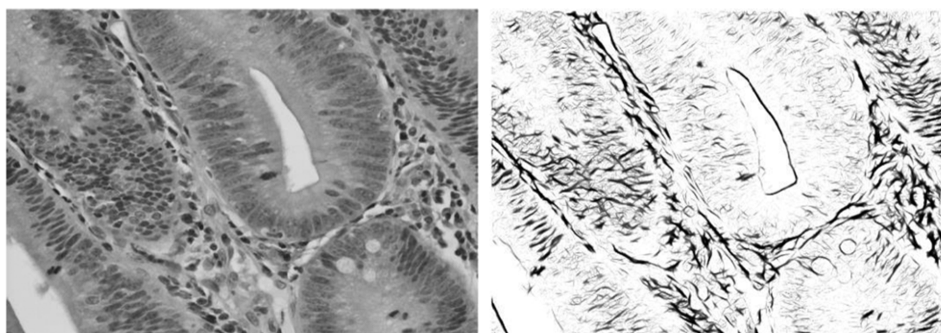

Fig. 3. Edge detection in a sample of colon tissue after a haematoxylin–eosin reaction.

Listing 2 Pseudo codes of the edge detection algorithm used to testify library efficiency.

```
for all directions
  for all smoothing scale
    for all differentiation scale
      ConvolutionOp(InputImg, Filter1, 2DGauss);
      ConvolutionOp(InputImg, Filter2, 2DGauss);
      BinaryPixOpI(Filter1, PartialRes1, pOpId);
      BinaryPixOpI(PartialRes1, PartialRes2, pOpId);
      BinaryPixOpI(OutputImg, PartialRes2, pOpId);
```

4. The MPI-based PIMA(GE)^2 Lib library

The first parallel implementation of the library is developed to run on traditional HPC architectures offering multiple CPUs. This kind of architectures can be exploited using well-known SPMD (Single Program Multiple Data) programming model on distributed memory resources; the standard de facto in this context is MPI; in particular, we used C and MPICH2. In this case, to obtain good performance, the actual criticality is represented by data communications among processes. To this aim, we considered three main topics: the efficient parallelization of single operation, the effective concatenation of the operations and the use of tools to speed up data acquisition. The latter is not discussed in this work, the first two points are detailed in the following, a discussion about performance concludes the Section.

4.1. Parallelization of single operation

Generally speaking, the aim is to subdivide the whole work into many partial subtasks taking carefully into account the load balancing of the global computation. For the parallelization of image processing operations, we started from the two remarks already mentioned: (1) most of image processing operations are easily parallelizable in a data parallel fashion, thus a key issue is the identification of a proper distribution of the global data; (2) image processing operation classes represent a restricted number of computational schemes to guide the parallelization.

We develop sequential implementation of the image processing operations and, using the definitions given in [14], we derive their parallel implementation; furthermore, we investigate issues concerning data distribution. For each class of operation, the first step is to understand the relative data parallelization schemes, i.e. the maximum amount of work that a process can perform without the need for communication, or in other words, data accesses refer to data local to the process. It is intuitively understandable that this depends on the number of pixels of the input image(s) accessed to produce the output, i.e. a pixel (of the output image) or a value depending on the operation considered. For example, the Reduction Operations produce a scalar or vector value, e.g. maximum pixel of the image, only one pixel is necessary to obtain the result value and no further data structures or pixels are required. The same happens when considering Point Operations and Image Arithmetic Operations, where of course the output is an image. This means that for these classes, images can be subdivided among parallel processes and elaborated without the need of further steps of communication or computation since data required for the elaboration are local to the process.

A different behaviour (algorithmic pattern) is adopted by the Convolution Operations class, which combines an input image with a kernel characterizing the specific convolution operation required in the computation. The value of a pixel in the output image depends on the neighbourhood of the corresponding pixel in the input image and the values of the kernel;

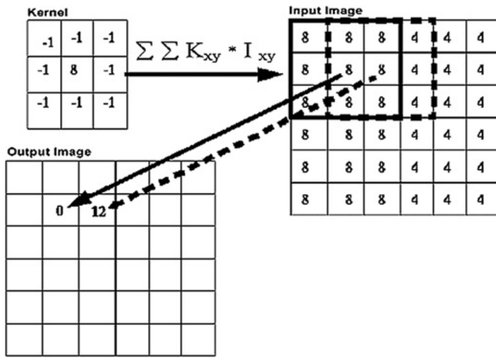

Fig. 4. An example of Convolution Operation considering a 3 × 3 kernel.

Fig. 5. Parallel schemes adopted for point and Reduction Operations exemplified on four processes.

the number of neighbours required for the computation depends on the dimension of the kernel, as reported in Fig. 4. This means that, in the parallelization of such class of operations, data distribution has to consider an overlap region between two consecutive blocks of partial data, and the proper management of their update through the data communication. The behaviour of the Geometric Operations varies with the specific elaboration; in a ROI operation, data are local to the processes and no communications are needed, domain transformations work differently. These operations consider the use of auxiliary MATRIXs and may require data communications. These peculiarities have to be taken into account in the parallelization of each operation class.

Starting from these remarks, we developed a number of parallel schemes that is equal to the number of algorithmic patterns. For Point Operations and Image Arithmetic Operations, data are distributed and elaborated independently using the sequential code. The output image is obtained through data collection without further elaborations. This concludes their parallel implementation. Reduction Operations, after data gathering, require a further computation to properly combine local results in order to obtain the output, e.g. to calculate the maximum pixel of an image it is necessary to select maximum values among the partial results—maximum pixel of partial data. For these classes in Fig. 5 we schematize the corresponding parallel behaviour. Convolution Operations require "larger" partial data to include the overlapping regions. The elaboration of the partial image requires the KERNEL and communication phases could be necessary to update the overlapping regions of partial data. The output image is obtained through data collection, without further elaboration. This concludes the parallel implementation of such class. In Listing 3, we report the sequential (pseudo) code and the MPI parallel one of Image Arithmetic Operations and Convolution Operations.

Listing 3 Pseudo codes of Image Arithmetic and (simplified) Convolution Operations, sequential on the left side, the corresponding MPI implementation on the right.

Sequential Image Arithmetic	**MPI Parallel Image Arithmetic**
	MPI Image Distribution(InputImage1, PartialIn1)
	MPI Image Distribution(InputImage2, PartialIn2)
for i = all rows in InputImage	for all rows i in PartialIn1
for j = all columns in InputImage	for all columns j in PartialIin1
OutImage[i][j] = FuncOf(In1[i][j], In2[i][j])	PartOutImg[i][j] = FuncOf(PartiaIn1[i][j],PartialIn2[i][j]);
	MPI Image Collection(PartialOutImage, OutImage)
Sequential Convolution (simplified)	**Parallel Convolution (simplified)**
	MPI Image Distribution(InputImage, PartialImage+Overlap)
for i = all rows in InputImage	for all rows i in PartialImage+Overlap
for all columns j in InputImage	for all columns j in PartialImage+Overlap
sum=0	sum=0;
for all rows x in KERNEL	for all rows x in KERNEL
for all columns y in KERNEL	for all columns y in KERNEL
sum + = I[...][...] * K[x][y]	sum + = PartialImage+Overlap[...][...] * K[x][y]
OutImage[i][j] = sum / DimKERNEL	PartialOutImage[i][j] = sum / DimKERNEL
	MPI Image Collection(PartialOutImage, OutImage)

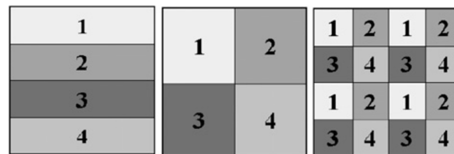

Fig. 6. Data partitioning patterns.

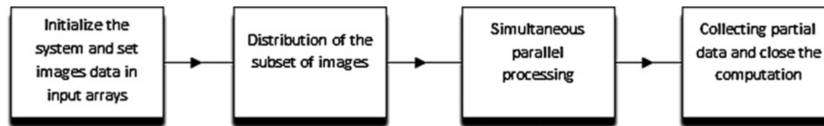

Fig. 7. General processing flow.

The second step involves data partitioning; in general, the data distribution pattern has a high impact on the performance of an application and depends on the nature and regularity of the application itself. We assume that the same pattern of distribution for each image operation class, however in this way, we are not applying an optimal parallelization policy to all image operation classes. To define the distribution pattern suitable for image processing operations, we tested three different configurations, adopting the (more intuitive) terminology of the 2D case: row block partition, row and column block partition, cyclic row and column block partition as applied in ScaLAPACK. In Fig. 6, we graphically present the three mentioned patterns distributed among four processes. Due to the regular nature of the operations considered, the row block partition results the most suitable in terms of overall performance. In fact, it limits the number of communication necessary to parallelize the code, and realizes the most efficient memory access. Technical details and performance comparison are discussed in [19].

4.2. Optimization of a pipeline of operations

Till now, we have defined an efficient parallelization of the single image operation class. However, an application is typically composed of several library operation calls aimed to elaborate one or more images; for this reason, it is necessary to define further rules in order to define an efficient sequence of image operations.

An effective sequence of operations should avoid data collection at the end of a single operation, if such data/image is still involved in the computation, i.e. the image has to be distributed again in the next calls. It is intuitively clear that this behaviour results in redundant communications among parallel processes and useless data (re)distribution, thus in a loss of efficiency. In PIMA(GE)^2 Lib, to prevent this kind of redundancy, we apply a simple strategy, thus avoiding more sophisticated approach [14]. The strategy is based on several assumptions: the same data distribution pattern is used in each operation class; the MPI environment to execute the computation has a fixed number of processes N, i.e. during the execution, the number of processes N cannot change. These hypotheses are not restrictive, since they describe the scenario usually adopted for a MPI computation. We further assume that each process is responsible for the computation of a piece of the global data for the part of the computation that involves such data; partial data distributed among processes represent the most updated copy of the images, unless differently specified. This has been codified with the definition of a flag *status* for each image. In each parallelization schema, a set of controls on this flag, with consequent actions, ensures the coherence of the computation.

Behind the sequential API of the library, the MPI processes are spawned with the OpenLib function and data are distributed among them when a user instantiates an image; in particular data are subdivided in contiguous row regions with the same granularity, during the I/O phases or during memory management operations. This corresponds to set a specific value in the flag *status*. Explicit requests for data collection, e.g. I/O operations and/or data structure management as replacement of an image with a new one, correspond to a modification of the flag. Each process acquires and elaborates its own chunk of the image, following the adequate parallelization scheme derived by the specific algorithmic patterns. The controls on the flag verify that during the execution the most up-to-date data are the partial data under elaboration among parallel processes. If it is not the case, a proper management or collection/redistribution phase has been implemented thus to ensure correct results. Data are definitely collected when the execution is completed, or because of an explicit operation, such as delete operation. An intuitive view of the general management of parallel computations is given in Fig. 7. To improve the general performance of the library, we adopted a parallel I/O in PIMA(GE)^2 Lib that permits to speed up the I/O operations as well as the distribution phase; in this paper we do not report this experimentation, more details are presented in [26]. The software layer aimed to implement the parallelization strategy is transparent to the users through the sequential API of the library. The functions of this layer start up the algorithmic patterns and check the coherence of parallel execution with proper control instructions when necessary. As demonstrated in the next Subsection, the parallelization strategy leads to good speedup values.

Table 1
Speedup values achieved with the TMA edge detection.

Linear	2	4	6	8	12	16	24	32	36
IMATI-HPC	1.96	3.62	5.33	7.34	11.20	14.47	21.70	26.33	32.40

4.3. MPI scalability for TMA image analysis

To evaluate speedup values achievable using PIMA(GE)^2 Lib, we implemented the edge detection on a TMA introduced in Section 3.3. We report the speedup values obtained on the resource in our domain, i.e. three blades, each one is composed by two Intel Xeon E5645 processors (six cores each) 32 GB of RAM and an nVidia GTX 580 GPU equipped with a 1.5 GB of GPU memory. The interconnection is provided by a Gigabit switched Ethernet. In Table 1, we refer to such resources as IMATI-HPC and we report speedup at varying the MPI parallel processes, together with the optimal (linear) one.

We can see that the library, in its MPI implementation, obtains quite good speedup performance, especially when launched on few cores, growing the number of cores the scalability of the code decreases. This is reasonably due to the Gigabit used for the connection among blades; in fact, scalability has a reduction when the communications among processes involve two or three blades, while when all MPI processes are spawned on a single blade, speedup values are almost linear. This also agrees with the fact that although MPI is born for distributed architectures, it is able to effectively exploit also a shared memory environment, as the twelve cores on the single blade, thus optimizes its behaviour accordingly. The scalability obtained testifies the effectiveness of the choices made.

5. Moving towards GPU computing, the CUDA-based PIMA(GE)^2 Lib library

The second parallel implementation of the library is developed to run on GPU device, which offers a large number of lightweight threads dedicated to floating-points calculations. Let us spend a few words about such devices, just to make clearer the policies explained in the following Section.

GPUs are highly parallel programmable microprocessors born to support graphics elaboration. This kind of devices are used in combination with CPU, where the GPU is exploited as a coprocessor to speed up numerically intensive parts of code by the means of a massive fine grained parallelism. Thus typically, the parts of the code that exhibit a rich amount of data parallelism are performed on the GPU (*device*) in a SIMD mode; the CPU (*host*) performs the serial parts. To enable such hybrid computation, data have to be transferred on the GPU memory using the PCI-Express bus. This data transfer represents the actual bottleneck of the computation. The efforts necessary to perform general-purpose computation on GPUs were considered actually high until more friendly tools appeared, e.g. nVidia released the Compute Unified Device Architecture (CUDA) [11], that still requires the careful design implementation and optimization efforts to achieve high performance figures [7]. CUDA specifies extensions to the C programming language to manage hybrid computations and to write program (*kernel*) directly targeting GPUs.

To develop the CUDA-version of PIMA(GE)^2 Lib, we followed almost the same design steps of the MPI implementation, i.e. the parallelization of the classes of operations focusing on the mapping between data and threads and the efficient execution of the concatenation of operations to avoid useless data transfers between the host and the device. They are described in the following, with performance figures.

5.1. Parallelization of single operation

Actually, the experience gained with the previous version of the library allowed a faster design and implementation of the CUDA parallelization of the operation classes. Using CUDA, a kernel is executed on threads organized in a grid of blocks; each block shares a local memory (shared) and synchronizes the accesses to memory. A simple parallelization of the image processing classes of operations could consider a one-to-one mapping between pixel (composing an image) and threads; referring to the sequential codes exemplified in Listing 4, this means to avoid the "*for*" loops on the rows and the columns of the images. However, the effectiveness of the mapping depends on the granularity of the operations; for example in the case of simple operations as the Point ones, in the proposed mapping each thread performs a single operation. Although a significant improvement in speedup values could be obtained, it is clear that this approach does not maximize the utilization of the GPU's arithmetic units, neither makes good use of memory access leading to a poor ratio between memory access and operations computed. When more complex operations are required, as Convolution Operations, a bidimensional block organization of the thread makes sense, since the ratio grows. Therefore, the development of the parallelization strategy for the image processing operations is based on the algorithmic pattern used to derive their parallel schemes. Each class of operations organizes the grid of thread according to the configuration that better fits the specific algorithmic pattern. For Point Operations and Image Arithmetic Operations, we structure the blocks of thread as one-dimensional, each thread is in charge of the elaboration of one row of the image. This kind of mapping is actually employed in CUDA implementation and ensures coalesced access to the global memory [9,27].

Listing 4 Pseudo codes of Image Arithmetic and (simplified) Convolution Operations, sequential on the left side, the corresponding CUDA parallelization on the right.

Sequential Image Arithmetic Operations	**CUDA Parallel Image Arithmetic Operations**
	CUDA Image Copy To Device (InputIn1, In1Device)
	CUDA Image Copy To Device (InputIn2, In2Device)
for i = all rows in InputImage	i = threadIdx.x+blockIdx.x* blockDim.x;
for j = all columns in InputImage	for all columns j in PartialIin1
OutImage[i][j] = FuncOf(In1[i][j], In2[i][j])	OutDevice [i][j] = FuncOf(In1Device [i][j], In1Device [i][j]);
	Image Copy To Host (OutDevice, OutImage)
Sequential Convolution (simplified)	**CUDA Parallel Convolution (simplified)**
	CUDA Image Copy To Device (InputImage, ImDevice)
	CUDA Kernel Copy To Device (K, KDevice)
for i = all rows in InputImage	i = threadIdx.x+blockIdx.x* blockDim.x;
for all columns j in InputImage	j = threadIdx.k+blockIdx.y* blockDim.y;
sum=0	sum=0;
for all rows x in KERNEL	for all rows x in KERNEL
for all columns y in KERNEL	for all columns y in KERNEL
sum + = I[...][...] * K[x][y]	sum + = ImDevice [...][...] * KDevice [x][y];
OutImage[i][j] = sum / DimKERNEL	OutDevice[i][j] = sum / DimKERNEL
	CUDA Image Copy To Host (OutDevice, OutImage)

Furthermore, since images are processed row by row, this point will be actually useful in the merging of the MPI and CUDA parallelization. We develop the related kernel adapting the sequential implementation to exploit the thread blocks organization and to perform the proper transfer between CPU memory and the global memory of the GPU. As for the parallelization of the Reduction Operations class, we prefer to use an extern library, CUDPP—CUDA Data Parallel Primitives Library [28], which offers primitives such as parallel prefix-sum (*scan*), parallel sort and parallel reductions. The parallelization of the Convolution Operations class has to take into account the proper management of the neighbourhood of the pixel in the input image and the KERNEL necessary for computation. We structure threads in bidimensional blocks; every thread combines a pixel and the neighbourhood involving the specific KERNEL. Thus, thread blocks are responsible for a patch of pixels in the output image, in the so-called tartan distributed thread block distribution [29]. Since the KERNEL has to be accessed by all threads involved in the computation, a future direction of this work may investigate the use of GPU constant memory and/or shared memory. In Listing 4, we report a sequential (pseudo) code and the corresponding CUDA parallelization of Point and Convolution Operations.

5.2. Optimization of a pipeline of operations

As in the MPI implementation case, we have to define an efficient execution of the sequence of single operations provided in PIMA(GE)^2 Lib; in the CUDA implementation, the data transfer between CPU and GPU represents the actual bottleneck of the computation. The effective concatenation of operations exploiting GPU devices should avoid the data transfer between CPU and GPU at the end of a single operation, if such data/image has to be elaborated in the next calls, i.e. transferred again. This results in an additional PCI-Express bus latency via CUDA streams, where kernels and their arguments are serialized. Experimental results presented in Table 2 testify this issue and the considerable improvements achievable avoiding useless/redundant data transfer.

To overcome the problem, we adopt a strategy similar to the MPI implementation one; several assumptions are made and codified with the definition of the flag *status*. Data are transferred on the GPU global memory when a user instantiates an image, each thread processes its own chunk of the image according to the CUDA parallelization schemes of each image operation class. Again, the library assumes that the most updated data are present on the GPU global memory during the entire execution, unless explicit requests for data collection, e.g. I/O operations or data structure management. Also in this case, the proper management of these phases has to be carefully implemented with a set of instructions based on the flag *status*. The CPU works as a controller submitting kernel executions, synchronizing operations and executing the proper control/correction instructions. Data are definitely transferred to the CPU when the execution is completed, or again, because of an explicit operation, e.g. delete operations.

The policy to concatenate the parallel operations is managed through a dedicated software layer implementing such controls; the functionalities of this layer are hidden behind the sequential API of PIMA(GE)^2 Lib, thus completely transparent

Table 2
Execution times expressed in seconds; rows report the operation classes varying on data dimensions, columns report respectively sequential run, the kernel run plus data transfer between CPU–GPU, pure kernel run without data transfer.

		Sequential code	CUDA with data transfer	CUDA without data transfer
Test unary	1 MB	0.0011	0.004	0.00001
	5 MB	0.0037	0.008	0.00001
	10 MB	0.006	0.012	0.00001
Test binary	1 MB	0.0010	0.0073	0.00006
	5 MB	0.0046	0.022	0.00008
	10 MB	0.0075	0.035	0.00008
Test reduction	1 MB	0.0009	0.0054	0.001
	5 MB	0.0043	0.017	0.0032
	10 MB	0.0073	0.027	0.0045
Test geometric	1 MB	0.0016	0.0063	0.0042
	5 MB	0.012	0.029	0.018
	10 MB	0.021	0.048	0.030
Test convolution	1 MB	0.42	0.011	0.0076
	5 MB	1.20	0.034	0.026
	10 MB	2.93	0.053	0.030

to the users. As demonstrated in the next Subsection, the parallelization strategy leads to good performance values. In the exploitation of GPUs, the transparent management of the hybrid computation is actually valuable because of the expertise and programming skills required to the users to achieve a good level of performance [7].

5.3. CUDA scalability for TMA image analysis

Tests consider a nVidia GTX 580 GPU equipped with a 1.5 GB of GPU memory; we report the performance achieved by the classes of image processing operations considering one operator for each class. Experimental results are collected on images of about 1, 5 and 10 MB; execution times (expressed in seconds) are presented in Table 2, distinguishing sequential run, kernel run plus data transfer between CPU–GPU and pure kernel run without data transfer.

The measurements show one or two orders-of-magnitude improvements when comparing sequential and kernel performance; when also data transfers are measured, only a little improvement is obtained with elementary operations and in the worst case, a performance degradation. More compute intensive operations always achieve interesting performance, see the last row of the Table. Actually, data movement is the limiting factor for the performance of the CUDA image processing operations and the defined policy is necessary, as testified by the last column of the Table.

We test performance achievable with the CUDA implementation of PIMA(GE)^2 Lib on the edge detection introduced in Section 3.3, where the defined policy to manage the pipeline of operation is applied. For the analysis of an image of 10 MB, the sequential execution takes about 4420 s, i.e. about 1 h and 10 min, while the GPU implementation takes about 2980 s, i.e. less than 50 min, thus gained about the 30% of the efficiency in terms of computational time.

6. Merging MPI and CUDA parallel computing paradigms

The last implementation of PIMA(GE)^2 Lib is aimed to exploit features of clusters of GPUs; the idea is to add a further level of parallelism to the previous implementations of PIMA(GE)^2 Lib by integrating both codes. In this case, MPI is used to distribute the computation and CUDA as the main execution engine. This approach has been already adopted in the scientific community to speed up compute intensive computations [30,31], in particular in the development of parallel library for cluster of GPUs [32,33], with successful examples also in the image processing field [3,9]. The integration of the MPI and CUDA has worked quite smoothly thanks to the experience with the previous library versions, we can affirm that it is a few intrusive thanks to the library design in terms of code organization and modularity. Indeed, the original sequential part of the MPI implementation is substituted with the CUDA parallel kernel combining the related parallelization strategies. Experimentations about the integration provide interesting results; in the following, we describe the design steps with performance obtained on a set of operations.

6.1. Parallelization of single operation

Starting from parallel schemes defined in the MPI and CUDA library versions, the development of the MPI–CUDA implementation has been obtained quite smoothly. The MPI code has been carefully combined with CUDA parallel kernels mostly replacing the procedures implementing the sequential operations with corresponding CUDA kernels and properly combining MPI communications and data transfer to/from GPU. With this approach, CUDA kernels elaborate the partial data in charge of the MPI processes, i.e. a piece of the original image, and GPUs are the main execution engines.

Listing 5 Pseudo codes of Image Arithmetic and (simplified) Convolution Operations, sequential on the left side and the corresponding MPI–CUDA parallelization on the right.

Sequential Image Arithmetic Operations	**MPI–CUDA Parallel Image Arithmetic Operations**
	MPI Image Distribution(InputImage1, PartialIn1)
	MPI Image Distribution(InputImage2, PartialIn2)
	CUDA Image Copy To Device (PartialIn1, In1Device)
	CUDA Image Copy To Device (PartialIn2, In2Device)
for i = all rows in InputImage	i = threadIdx.x+blockIdx.x* blockDim.x;
for j = all columns in InputImage	for all columns j in PartialIin1
OutImage[i][j] = FuncOf(In1[i][j], In2[i][j])	OutDevice [i][j] = FunOf(In1Device [i][j], In1Device [i][j]);
	CUDA Image Copy To Host (OutDevice, PartialOutImage)
	MPI Image Collection(PartialOutImage, OutImage)
Sequential Convolution (simplified)	**MPI–CUDA Parallel Convolution (simplified)**
	MPI Image Distribution(InputImage, PartialImage+Overlap)
	CUDA Image Copy To Device (InputImage+Overlap, ImDevice)
	CUDA Kernel Copy To Device (K, KDevice)
for i = all rows in InputImage	i = threadIdx.x+blockIdx.x* blockDim.x;
for all columns j in InputImage	j = threadIdx.k+blockIdx.y* blockDim.y;
sum=0	sum=0;
for all rows x in KERNEL	for all rows x in KERNEL
for all columns y in KERNEL	for all columns y in KERNEL
sum + = I[...][...] * K[x][y]	sum + = ImDevice [...][...] * KDevice [x][y];
OutImage[i][j] = sum / DimKERNEL	OutDevice[i][j] = sum / DimKERNEL
	CUDA Image Copy To Host (OutDevice, PartialOutImage)
	MPI Image Collection(PartialOutImage, OutImage)

Depending on the image operations classes, the integration may require further efforts; for example in the simple case of Point Operations and Image Arithmetic Operations, the three steps consisting of MPI data distribution, CUDA elaboration of partial images and MPI partial data gathering, conclude the computation. This is exactly the same situation depicted in Fig. 5, but in this case, the processing part is performed on the GPU devices. More attention has to be paid in the development of the Convolution and Geometric Operations classes, because of possible data communications. MPI communications have to be tightly coupled with a proper data transfer to/from GPU devices, e.g. when the exchange of the overlap regions occurs in the convolution case, we have to integrate the transfer of data from the device before the communications among MPI processes and the transfer in the opposite direction when the exchange is completed. In Listing 5, we exemplify the merge of MPI and CUDA parallelization, again, for Point and Convolution Operations.

6.2. Optimization of a pipeline of operations

On a cluster of GPUs, the problem of redundant/useless data movement is represented by both communications among MPI processes and transfers to/from GPUs memory. Such problems have been managed separately through the definition of specific policies previously described; in this case, the issue is addressed through the proper integration of the existing policies. Starting from several assumptions, the integration is obtained with a coherent management of the flags *status* of both policies and related sets of controls/instructions.

As the MPI-based policy requires, the MPI parallel environment considers a number of processes that cannot be varied during the computation; each process is responsible for the elaboration of a piece of the global data, i.e. the partial data. The library assumes that the most updated data during the execution are represented by the partial data under elaboration among parallel processes, i.e. data present on the GPUs. Such conditions are at the bases of both policies; they are assumed true unless explicit request for data collection. When a user instantiates an image, through image acquisition/creation, partial data are distributed among MPI processes and transferred on the connected GPUs; the MPI flag and the CUDA flag are modified accordingly. Threads elaborate such data according to the parallelization schemes of the operation classes, thus managing communication phases (if any) according to the policy applied in the MPI implementation of the library, with a proper data transfer to/from the GPU. The controls on both flags verify that the most up-to-date data are the partial data under elaboration among parallel processes by the connected GPUs. If it is not the case, a proper management or collection/redistribution phase has been implemented thus to ensure correct results. Data are definitely collected when the execution is completed, or because of an explicit operation.

Table 3
Execution times expressed in seconds; rows report the operation classes varying on data dimensions, columns report respectively sequential run, the MPI run and the MPI–CUDA run.

		Sequential	MPI	MPI–CUDA
Test unary	1 MB	0.0011	0.004	0.0007
	5 MB	0.0037	0.008	0.0007
	10 MB	0.006	0.012	0.006
Test binary	1 MB	0.0010	0.0046	0.00005
	5 MB	0.0046	0.022	0.00006
	10 MB	0.0075	0.035	0.00004
Test reduction	1 MB	0.0009	0.00043	0.0001
	5 MB	0.0043	0.0017	0.0007
	10 MB	0.0073	0.0025	0.0004

As already said, MPI is used to distribute the computation of partial subtasks and CUDA as the main execution engine. Thanks to the SPMD model implemented with MPI, each single node/processor of the cluster acts as the controller of its own CUDA elaboration, while the software layer develops to manage the MPI computation, still verifies the MPI computation correctness.

6.3. Performance tests

We used the resource in our domain, i.e. a GPU cluster composed of three node interconnected with a Gigabit Ethernet. In Table 3, we report experimental results collected on a set of operations run on images of about 1, 5 and 10 MB, distinguishing the execution times (expressed in seconds) of the sequential run and of the MPI–CUDA run. Since I/O and distribution operations are dominant in the selected tests, execution times are evaluated excluding these phases.

On simple operations as the ones reported in the Table, the integration of the two levels of parallelism enables an improvement of about one order; however, even if promising, a deeper analysis have to be performed, in particular on more complex operations. Solid works presented in the scientific community provide actually interesting results [9], and this leads us to continue the consolidation of the policy and performance analysis.

7. Conclusions

In recent years, the evolution and growth of the techniques and platforms commonly used for HPC have been truly astonishing and it is now possible to access highly heterogeneous systems, equipped with powerful specialized hardware. If their availability and accessibility could be considered no longer a critical point, the actual barrier posed is represented by the effective exploitation of such heterogeneous resources. In particular, the low price of GPUs and their high performance make them a desirable architecture to speed up computations, however to achieve good performance, the level of expertise and programming skills required to the users are not trivial.

Starting from these considerations, in this work we presented the PIMA(GE)^2 Lib, the Parallel IMAGE processing GEnoa Library, developed to provide the most common low level image processing operations. The library is able to exploit traditional compute clusters through MPI, GPU device through CUDA and a new study is oriented to GPU-clusters through the combination of both. Optimization policies have been defined to tackle the specific issues arising from each specific architecture and to enable the proper execution of pipeline of PIMA(GE)^2 Lib operations. The parallelism of the library and the specific policies implemented at each level, are transparent to the users thanks to a sequential interface suitably defined. We presented the incremental approach applied in the development of the three implementations; the third implementation is derived through the integration of the first two. Actually, this process worked quite smoothly and few invasively thanks to the library organization in terms of code design, i.e. organization and modularity, and the experience matured during the library development. To evaluate performance figures, we consider a complex application developed with PIMA(GE)^2 Lib operations, an edge detection applied to biological data, i.e. images obtained through the TMA technologies. Results are quite satisfying, achieving good speedup values in the MPI parallelization, about a 30% improvement in the execution time on a single GPU and the first experimentations on cluster of GPUs show promising results with respect to the operations already tested. As a future direction, we have to consolidate the work on cluster of GPUs and we would further investigate the GPU's exploitations in terms of optimized memory, as the constant memory and the shared, as well as the use of multiples GPUs on the same node.

References

[1] G. Ritter, J. Wilson, Handbook of Computer Vision Algorithms in Image Algebra, second ed., CRC Press Inc., 2001.
[2] A. Murli, L. D'Amore, L. Carracciuolo, M. Ceccarelli, L. Antonelli, High performance edge preserving regularization in 3D spect imaging, Parallel Comput. 34 (1) (2008) 115–132.
[3] T.D.R. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, M. Ujaldon, Biomedical image analysis on a cooperative cluster of GPUs and multicores, in: Proceedings of the 22nd Annual International Conference on Supercomputing, ACM, 2008, pp. 15–25.

[4] J.J. Dongarra, A.J. van der Steen, High performance computing systems: status and outlook, Acta Numer. 21 (2012) 379–474.
[5] C. Brown, H.-W. Loidl, K. Hammond, Paraforming: forming parallel haskell programs using novel refactoring techniques, in: Proceedings of the 12th International Conference on Trends in Functional Programming, TFP'11, Springer-Verlag, 2012, pp. 82–97.
[6] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, K.A. Yelick, The landscape of parallel computing research: a view from Berkeley, in: Technical Report No. UCB/EECS-2006-183, 2006.
[7] D. D'Agostino, A. Clematis, E. Danovaro, Heterogeneous parallel computing platforms and tools for compute-intensive algorithms: a case study, in: E. Jeannot, J. Žilinskas (Eds.), High-Performance Computing on Complex Environments, 2014, pp. 193–213. http://dx.doi.org/10.1002/9781118711897.ch11.
[8] C. Gonzalez, S. Sanchez, A. Paz, J. Resano, D. Mozos, A. Plaza, Use of FPGA or GPU-based architectures for remotely sensed hyperspectral image processing, Integr. VLSI J. 46 (2) (2013) 89–103.
[9] V.W. Ben, J. Maassen, F.J. Seinstra, Towards user transparent parallel multimedia computing on GPU-clusters, Comput. Archit. (2012) 28–39.
[10] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J.J. Dongarra, MPI: The Complete Reference, MIT Press, Cambridge MA, USA, 1995.
[11] CUDA. Home page maintained by Nvidia. http://developer.nvidia.com/object/cuda.html.
[12] R.F. Boisvert, Mathematical Software: Past, Present and Future. Computational Science, Mathematics, and Software, Purdue University Press, 2002.
[13] J. Lebak, J. Kepner, H. Hoffmann, E. Rudtledge, Parallel VSIPL++: an open standard library for high-performance parallel signal processing, Proc. IEEE 93 (2) (2005) 313–330.
[14] F. Seinstra, D. Koelma, J.M. Geusebroek, A software architecture for user transparent parallel image processing, Parallel Comput. 28 (7–8) (2002) 967–993.
[15] S. Balay, W. Gropp, L.C. McInnes, B.F. Smith, PETSc 2.0 users' manual, in: Technical Report ANL-95/11 Revision 2.0.17, Argonne National Laboratory, 1996.
[16] J. Dongarra, L.S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, ScaLAPACK User's Guide, Society for Industrial and Applied Mathematics, Philadelphia, 1997.
[17] A. Clematis, D. D'Agostino, A. Galizia, An object interface for interoperability of image processing parallel library in a distributed environment, in: Proceedings of ICIAP 2005, in: LNCS, vol. 3617, Springer, 2005, pp. 584–591.
[18] P. Oliveira, H. du Buf, SPMD image processing on beowulf clusters: directives and libraries, in: IEEE Proceedings of 7th IPDPS, 2003, pp. 8–15.
[19] A. Galizia, Design of Parallel Image Processing Libraries for Distributed and Grid Environments (Ph.D. thesis), 2008, Available at http://www.disi.unige.it/dottorato/THESES/2008-05-GaliziaA.pdf.
[20] J. Kononen, L. Bubendorf, A. Kallioniemi, M. Barlund, P. Schraml, S. Leighton, J. Torhorst, M.J. Mihatsch, G. Sauter, O.P. Kallioniemi, Tissue microarrays for high-throughput molecular profiling of tumor Specimens, Nat. Med. 4 (1998) 844–847.
[21] S. Mousses, L. Bubendorf, U. Wagner, G. Hostetter, J. Kononen, R. Cornelison, N. Goldberger, A.G. Elkahloun, N. Willi, P. Koivisto, W. Ferhle, M. Raffeld, G. Sauter, O.P. Kallioniemi, Clinical validation of candidate genes associated with prostate cancer progression in the CWR22 model system using tissue microarrays, Cancer Res. 62 (2002) 1256–1260.
[22] J.M. Giltnane, D.L. Rimm, Technology Insight: identification of biomarkers with tissue microarray technology, Nat. Clin. Pract. Oncol. 1 (2004) 104–111.
[23] A. Nocito, J. Kononen, O.P. Kallioniemi, G. Sauter, Tissue microarrays (TMAs) for high-throughput molecular pathology research, Int. J. Cancer 94 (2001) 1–5.
[24] D.G. Rosen, X. Huang, M.T. Deavers, A. Malpica, E.G. Silva, J. Liu, Validation of tissue microarray technology in ovarian carcinoma, Mod. pathol. 17 (7) (2004) 790–797.
[25] J.M. Geusebroek, A.W.M. Smeulders, H. Geerts, A minimum cost approach for segmenting networks of lines, Int. J. Comput. Vis. 43 (2) (2001) 99–111.
[26] A. Clematis, D. D'Agostino, A. Galizia, Parallel I/O aspects in PIMA(GE)^2 Lib, in: Parallel Computing: Architectures, Algorithms and Applications, Proceedings of the ParCo 2007, in: Advances in Parallel Computing, vol. 15, IOS Press, 2007, pp. 441–448.
[27] D. D'Agostino, S. Decherchi, A. Galizia, J. Colmenares, A. Quarati, W. Rocchia, A. Clematis, CUDA accelerated blobby molecular surface generation, in: Parallel Processing and Applied Mathematics, in: LCNS, vol. 7203, 2012, pp. 347–356.
[28] S. Sengupta, M. Harris, Y. Zhang, J. Owens, Scan primitives for GPU computing, in: Proceedings of Graphics Hardware 2007, 2008, pp. 97–106.
[29] J. Colmenares, J. Ortiz, S. Decherchi, A. Fijany, W. Rocchia, Solving the linearized Poisson- Boltzmann equation on GPUs using CUDA, in: 2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP, IEEE, 2013, pp. 420–426.
[30] T.M. Benson, D.P. Campbell, D.A. Cook, Gigapixel spotlight synthetic aperture radar back projection using clusters of GPUs and CUDA, in: IEEE Proceedings of Radar Conference, RADAR 2012, 2012, pp. 0853–0858.
[31] N.P. Karunadasa, D.N. Ranasinghe, Accelerating high performance applications with CUDA and MPI, in: IEEE Proceedings of the International Conference on Industrial and Information Systems, ICIIS, 2009, pp. 331–336.
[32] M. Fatica, Accelerating linpack with CUDA on heterogeneous clusters, in: ACM Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009, pp. 46–51.
[33] R. Babich, M.A. Clark, B. Joó, Parallelizing the QUDA library for multi-GPU calculations in lattice quantum chromo dynamics, in: IEEE Proceeding of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, 2010, pp. 1–11.