Remote Sensing Processing: From Multicore to GPU

Emmanuel Christophe, *Member, IEEE*, Julien Michel, and Jordi Inglada, *Member, IEEE*

***Abstract*—As the amount of data and the complexity of the processing rise, the demand for processing power in remote sensing applications is increasing. The processing speed is a critical aspect to enable a productive interaction between the human operator and the machine in order to achieve ever more complex tasks satisfactorily. Graphic processing units (GPU) are good candidates to speed up some tasks. With the recent developments, programing these devices became very simple. However, one source of complexity is on the frontier of this hardware: how to handle an image that does not have a convenient size as a power of 2, how to handle an image that is too big to fit the GPU memory? This paper presents a framework that has proven to be efficient with standard implementations of image processing algorithms and it is demonstrated that it also enables a rapid development of GPU adaptations. Several cases from the simplest to the more complex are detailed and illustrate speedups of up to 400 times.**

***Index Terms*—CUDA, GPU, implementation, OpenCL.**

I. Introduction

THE amount of data acquired by imaging satellites has been growing steadily in recent years. There is a rapidly increasing number of applications that benefit from the decline in prices and the easier access to such data. With this proliferation of data, relying on humans to do most of the high level interpretation tasks is no longer possible. Some (but not all) advanced tasks need to be processed automatically. However, these tasks are more complex, thus raising the computational power required.

As highlighted in an insightful report from Berkeley [1], the increase in computational power for the coming years goes through a parallel approach. High performance computing (HPC) is a natural solution to provide the computational power required. There are several approaches to HPC: clusters, grids or clouds are some examples. However, we chose here to focus on desktop HPC with the use of graphics processing units (GPU). The idea is to bring the processing power as close as possible to the final user to enable better human–algorithm interaction.

It is now possible to use GPUs to do general purpose processing. Benefiting from investment from the movie and gaming industries [2], the processing power of GPUs has increased dramatically. They have evolved in a different direction than the general purpose central processing units (CPU). They harbor hundreds of processing units that are able to work at the same time. CPUs and GPUs rely on different trade-offs regarding the amount of cache memory versus the number of processing units.

To benefit from these hundreds of processing units, the inherent parallelism of the algorithms needs to be exposed. Often in the literature, the focus is on the implementation of the core algorithm. However, one critical difficulty arises from boundary conditions (when the number of pixels in the image is not a convenient multiple) and also from the data size that often cannot be kept in the hardware memory in one go (thus requiring several passes to reach the final result).

When designing a library to benefit from the capabilities of GPUs, one has to think of both the final user of the program and the developer who is going to write new programs. For the former, it is important to keep him isolated from these implementation details: the program should work for any image size on any hardware. The latter will benefit from a framework to simplify the nitty-gritty mechanisms so that he can focus on performances.

The aim of this paper is to present a framework enabling an easier implementation of the GPU kernel for some parts of a global remote sensing image processing pipeline. The framework is available as open source software in the Orfeo Toolbox library [3].

The Orfeo Toolbox (OTB) is an open source library developed by CNES (the French Space Agency). It contains numerous algorithms for preprocessing as well as for information extraction from satellite images [4].

One of the main objectives of the Orfeo Toolbox (OTB) is to provide a strong and robust software architecture to facilitate the *scalability* of newly implemented algorithms and to relieve (at least partially) the researcher from such concerns. The processing model for OTB has its roots in the Insight Toolkit [5] and has been proven to be effective for remote sensing images as well as for medical images. The general architecture of OTB is described in Section II-A and in Section II-B we describe how it can be used to exploit the GPU processing capabilities.

In Section III, several examples of implementation with increasing complexity are described and show improvement ranging from zero to 400 times faster than the comparable CPU implementation.

Section IV discusses some of the perspectives that arise from such a speedup in the way we design and work with algorithms.

Finally, Section V concludes, presenting some directions for further improvements.

II. Problem Statement

This section presents the main issues related to the processing of remote sensing images: scalability. There are two dimensions

Manuscript received April 30, 2010; revised July 22, 2010 and September 03, 2010; accepted December 13, 2010.

E. Christophe was with the Centre for Remote Imaging, Sensing and Processing (CRISP), National University of Singapore, Singapore 119260. He is now with Google Inc., Mountain View, CA 94043 USA (e-mail: emmanuel.christophe@gmail.com).

J. Michel was with Communications et Systèmes (CS), 31506 Toulouse, France. He is now with CNES, 310401 Toulouse, France (e-mail: julien.michel@cnes.fr).

J. Inglada is with the Centre d'Etudes Spatiales de la BIOsphère (CESBIO), CNES, 31401 Toulouse, France (e-mail: jordi.inglada@cesbio.cnes.fr).

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

Digital Object Identifier 10.1109/JSTARS.2010.2102340

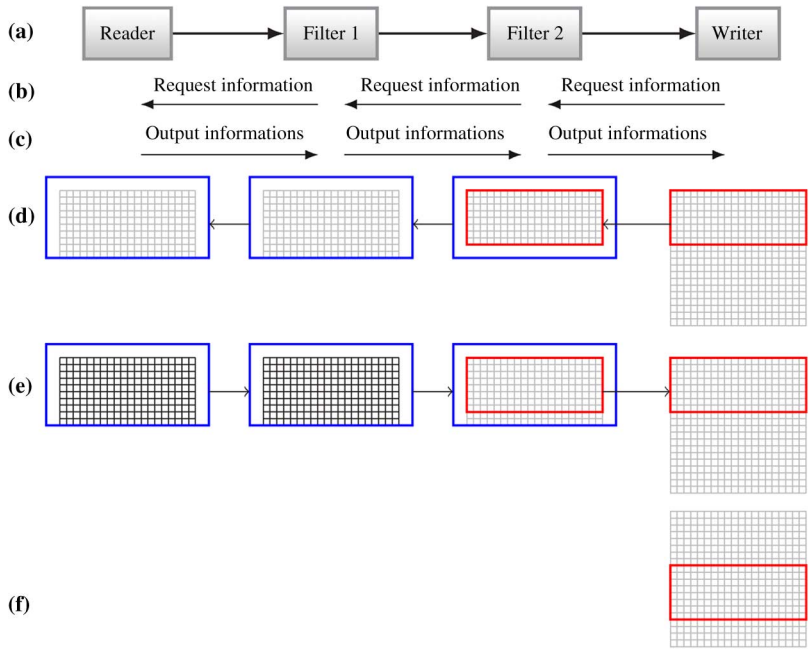

Fig. 1. Streaming model for OTB: illustration on a simple pipeline with two processing filters between a reader and a writer. One filter (filter 1) does pixel by pixel processing, while the other (filter 2) does a neighborhood processing.

to this problem: the size of the data and the time required for processing. A significant bottleneck for the use of new algorithms is the jump from toy image examples to real satellite images. The first issue is related to the size of the image which cannot be held into memory any more (Section II-A) and the second is related to the computation time and how to benefit from multiple computation units as commonly found in CPUs and GPUs (Section II-B). Even if those two problems are related, they play a part at different levels.

A. *Image Size Scalability: Streaming*

The sheer size of satellite images—several gigabytes—makes processing by usual methods inapplicable on standard computers. It is not desirable or possible to load the entire image into memory before doing any processing. In this situation, it is necessary to load only part of the image and process it before saving the result to the disk and proceeding to the next part. This corresponds to the concept of on-the-flow processing.

Remote sensing processing can be seen *as a chain of events or steps that lead to a final output* [6]. Each of these steps is generally independent from the following ones and generally focuses on a particular domain. For example, the image can be radiometrically corrected to compensate for the atmospheric effects, indices (such as NDVI) computed, before an object extraction based on these indexes takes place. The typical processing chain will process the whole image for each step, returning the final result after everything is done.

For some processing chains, iterations between the different steps are required to find the correct set of parameters. Due to the variability of satellite images and the variety of the tasks that need to be performed, fully automated tasks are rare. Humans are still an important part of the loop.

In these conditions, it is valuable to be able to provide some feedback quickly for only parts of the image and reprocess this part for a different set of parameters. Better yet if only the modified steps are reprocessed and not the whole chain; this is the concept of on-demand processing.

These concepts are linked in the sense that both rely on the ability to process only one part of the data. In the case of simple algorithms, this is quite easy: the input is just split into different non-overlapping pieces that are processed one by one. But most algorithms do consider the neighborhood of each pixel. As a consequence, in most cases, the data will have to be split into partially overlapping pieces.

The objective is to obtain the same result as the original algorithm as if the processing was done in one go. Depending on the algorithm, this is unfortunately not always possible.

In the Orfeo Toolbox, the processing elements are organized in the library as filters. Filters perform operations such as reading and writing the data, but also processing, e.g., linear filtering, thresholding or classification. Writing a new application (or a new processing chain) consists of plugging a few filters together to create a processing pipeline. As highlighted above, in most cases, the whole image cannot be held in memory at once and a different processing model is required.

Fig. 1 illustrates the process on a simple example. In this case, four filters are connected together:

- a reader that loads the image, or part of the image in memory from the file on disk;
- a filter which carries out a local processing that does not require access to neighboring pixels (a simple threshold for example), the processing can happen on CPU or GPU;
- a filter that requires the value of neighboring pixels to compute the value of a given pixel (a convolution filter is a typical example), the processing can happen on CPU or GPU;
- a writer to output the resulting image in memory into a file on disk, note that the file could be written in several steps.

We will illustrate in this example how it is possible to compute part of the image in the whole pipeline, incurring only minimal computation overhead.

Once all the filters are connected together and the pipeline is created as in Fig. 1(a), the processing is started by a call on the last filter of the pipeline, which is the writer in our example. This filter requests its input to provide the information regarding the size of the image it will produce [Fig. 1(b)]. The reader gets the information by reading the metadata of the file and the information is propagated through the pipeline [Fig. 1(c)]. Eventually, filters can modify this information, depending on the processing they apply on their input.

Once the writer has the information regarding the size of the data it has to produce, it can compute the splitting strategy: depending on the maximum memory specification, it will select only a small area of the image to request to its input filter. In Fig. 1(d), this area is represented by the red rectangle. The writer requests this area to its input: filter 2. This filter needs the value of the neighboring pixels of each pixel to be able to process its output. For the pixels in the middle of the region, this is not a problem, however, the region needs to be expanded to accommodate the need of the pixels at the border. This extension is represented by the blue rectangle in Fig. 1(d).

Here, there are two different cases: either the value is part of the image and can be obtained, or it is outside of the image area. In the first case (the bottom line in our example), the region is simply extended and the value will be generated by the input filter. If the value is outside of the image (top, left and right of the red region in our example), a strategy is necessary to create this value. This is handled at the filter level and several strategies (boundary conditions) are available: constant value, mirror, zero flux Neumann. In Section III-B, we will see how this strategy can ease the constraints on the GPU implementation.

Once the request reaches the reader, which is the first filter of our pipeline, it can generate the requested area from the file and pass it to the next filter [Fig. 1(e)]. Once the region reaches the writer, it is written on the disk and the process continues with the next tile [Fig. 1(f)].

This process relies on the capability of each filter to determine the size of the input needed to produce the output required by the downstream filter. Some specific algorithms cannot be rigorously implemented to work only on extracts of the image: Markov fields for example where the iterations introduce potential long range dependencies between pixels. In these situations, an approximation can usually be obtained. Note that the processing of the different tiles is independent and does not use common memory. It could ideally be coupled with distributing processing techniques (cluster or grids) where each node would process a tile, but this point is outside the scope of this paper.

The process is illustrated above in great detail, but it is worth mentioning that the user of the library does not need to understand this mechanism. Indeed, he may not even need to be aware of it. The developer of new filters, does not need to fully understand it, having a knowledge of the relevant customization points is sufficient. This design will appear to be critical in the context of the use of GPU where the memory is more limited.

B. Processing Unit Scalability: Multithreading

1) Short Review of CPU Versus GPU Architecture: Several architectures are available to enable parallel processing of data. The most common are SIMD and MIMD (using Flynn's taxonomy [7]). SIMD (Single Instruction, Multiple Data streams) where the same set of instruction is applied on several data streams is particularly suited to digital image processing. SIMD is available in CPU and enable to apply a single operation on multiple data at once. MIMD (Multiple Instruction, Multiple Data streams) corresponds to using several cores in a single die. These cores are able to execute independent instructions on different data.

Recent CPU combines several parallelization techniques to increase performances while giving the impression that they work sequentially: branch prediction, out-of-order execution, superscalar. All these techniques increase the complexity of the CPU, limiting the number of CPUs that can be included on a single chip.

On the other hand, GPUs keep each processing unit simple but pack thousands of them on the chip. One of the critical difference is the lower amount of cache memory available. As a consequence, the GPU will work well when the level of data parallelism is high and enable masking the latency of each thread.

If we omit some of the details above and compare how CPU and GPU will process the pixels of the image: we can consider that the CPU will process each pixel sequentially but very fast while the GPU will process them slower but a whole lot of them at a time.

2) The Rise of Multicore CPUs: As the frequency of processors is reaching limits due to heating issues, advances in processing capabilities of recent CPUs are geared towards the increase in the number of cores [1], [8]. Recent CPUs are able to handle 8 to 12 threads simultaneously.

However, designing an application to benefit from these multicore architectures is not straightforward. The problem is to be able to split the data into different entities that can be processed simultaneously.

In Section II-A, the main issue was limiting the size of the image to load into memory; here it is to draw on the availability of several processors. One common point between the two problems is that they rely on the possibility to process an extract of the data at a time. A major difference is that multithreading in the context of multicores will have access to shared memory between the cores. In the case of streaming, there is no shared memory between the tiles.

In OTB, the multithreading is done on a per filter basis with a portable implementation using one of the multithreading library available in the system (sproc, pthreads or Win32 Threads). In the previous example of Fig. 1, let us consider for example that filter 1 is multithreaded. In that case, when processing the requested blue region in step (e) of Fig. 1, the blue region is going to be subdivided into N regions each to be processed by one thread.

This ability to process a given region with multiple threads (around 10 in the case of the CPU) needs to be extended to the few thousands threads required to efficiently utilize a GPU [9].

3) Switching to GPUs: GPUs first appeared in the 1980s as hardware dedicated to graphics processing. Over the next decades, they progressively increased their capabilities to 3D processing. These chips are highly optimized for graphics-related operations, with floating point computation and massively

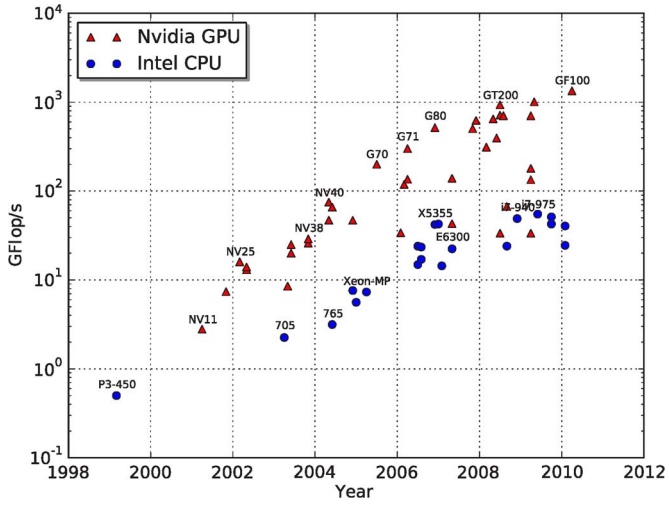

Fig. 2. Evolution of Intel CPUs and Nvidia GPUs over the last decade (source: Intel export compliance metric, Nvidia specifications and Wikipedia).

parallel computation. The parallel computation is a critical difference to the traditional CPUs architecture. The reason is that the pixels to be displayed can be computed independently. When switching one algorithm from CPUs to GPUs, the major challenge is to take advantage of this parallel organization.

Switching the programming model from sequential to parallel is not an easy task. Most of the time, the program needs to be rewritten to expose large amount of fine-grained parallelism, enabling the use of thousands of concurrent threads. The part that requires parallelization needs to be clearly exposed and defined by proper profiling. Potential gains attainable by optimizing part of the program are limited by Amdahl's law [10]:

$$\text{speedup} = \frac{1}{(1 - r_p) + \frac{r_p}{s}} \tag{1}$$

where r_p is the proportion of the program being parallelized and s is the speedup obtained on this particular part. For example, when improving a part of a program that represents 50% of the total execution time $(p_r = 0.5)$, even with an infinite speedup $(s = \infty)$, the overall speedup is only 2.

Nevertheless, the performance evolution between CPUs and GPUs justifies the effort. There is no perfect measure of comparison between CPUs and GPUs. Indeed, they are highly optimized for a specific class of problems. However, one indication of the evolution is the number of floating point operations per second (Flops/s). Fig. 2 presents the evolution of the Nvidia GPUs compared with the Intel CPUs over the last decade. One important provision is that these numbers are usually provided for double precision computation for CPUs and single precision computation for GPUs. GPUs are known to be significantly slower when double precision computation is required. However, recent GPUs address this issue. From these data, we can see that the GPUs are doubling their processing power every 12 months while CPUs do it in 18 months (according to Moore's law).

Until recently, benefiting from the GPU computing power required the developer to map the problem in terms of graphic primitives (using texture operations). This approach led some people to develop frameworks to alleviate this complexity. An example is presented in [11] to help process hyperspectral images. But mapping the problem in terms of graphic primitives remains awkward.

With the release of the first version of CUDA in 2007 and OpenCL in 2009, the programming model for GPUs was greatly simplified. CUDA is the language introduced by Nvidia on its G80 GPU series. The language is specific to one vendor and its hardware. However, it benefits from several years of usage, and the availability of numerous libraries are increasing its popularity.

OpenCL was developed by a consortium and released in 2009. It aims at supporting more hardware and to provide *a standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms* [12].

Both Cuda and OpenCL use the concept of kernel. A *kernel* is a series of operations that will typically be applied to one pixel. Each kernel will be handled by one of the numerous GPU processors. The flow of pixels forms a *stream* that will be processed by the kernel. This abstraction relieves the developer from the management tasks.

With these two languages that enable C-like programming, the learning curve to benefit from the GPU is significantly flattened [13] and several papers demonstrate good implementations for a wide range of problems such as for graph [14], sorting [9], and general purpose algorithms [15].

Recently, a paper from Intel researchers questions the 100x gains presented in several papers comparing CPU and GPU [16]. In their paper, Intel engineers use their extensive knowledge of the CPU architecture to draw the most out of it. Depending on the specific problem at hand, they conclude with a speedup for the GPU implementation over the CPU implementation ranging from 0 to 15. We have to note that the GPU cost is about half of the CPU in this particular study.

A comparison between OpenCL and Cuda is done in detail in [17]. Table I summarizes some of the different trade-offs between CUDA and OpenCL. The kernel code (the part of the program which implements the computations on the GPU) is almost identical in both cases which means that investing efforts into one technology will also benefit the other in case a change is required in the future. The setup of the kernel (the part of the code which prepares the data and transfers it to the processing units) is more complicated in the OpenCL case as there is some overhead due to the support for more heterogeneous hardware and for the compilation of the kernel during execution. However, this increased complexity enables more portability as OpenCL targets CPUs and GPUs alike. Concerning the availability of existing code, CUDA benefits from a head start of a few years compared to OpenCL and from many libraries heavily optimized for most common tasks (FFT, linear algebra, etc.).

In terms of performance, this difference in the development stage has an impact. An earlier study [17] concludes that the OpenCL implementation is 13% to 63% slower. Fig. 10 compares the performances of equivalent implementations using CUDA and OpenCL on the same hardware (OpenCL is about 45% slower).

TABLE I
COMPARISON BETWEEN CUDA AND OPENCL

	CUDA	OpenCL
Kernel code	Simple	Simple
Kernel setup	Simple	More complicated
Portability	Low	High
Library availability	High	Low

GPU memory is more limited than CPU memory. As several types of memory are available, special attention to which memory is used is required for further optimizations.

In the case studies described in the next section, the amount of data to be processed can be above the memory size. Another issue appears related to the size of the images. In most cases, each thread of the GPU will process one pixel. Due to the hardware architecture, threads need to be gathered into thread blocks, forming, for example, a group of 16x16 threads. This is fine if the image size is a multiple of 16. However, when processing regular images, this is unlikely, and this problem needs to be accounted for.

It can be handled at the GPU level, but it usually involves branching conditions. If possible, it is better to handle it before transferring the data to the GPU, making sure that the data size is suitable (a multiple of 16 in the example above). The pipeline model (Fig. 1) does just that: when the processing requires some neighborhood information (a simple case is a convolution), the pipeline is able to adapt the requested region to ensure that the necessary data are available. To comply with the GPU requirements, we just adapt this request to make sure that the region size is a multiple of 16.

In the following examples, the size of the image is handled as described in Section II-A. The only modification, which fits perfectly in the pipeline structure, is how the filter is computing the region required to produce the output.

III. CASE STUDIES OF GPU MIGRATION

In this section, we introduce several examples of processing algorithms in order to illustrate different trade-offs in terms of memory size and computation complexity, which will allow the reader to get insight on the benefit of GPU-based approaches and when it would be most profitably applied. We have selected three algorithm categories that cover most of the steps of a classical remote sensing image processing chain.

Relative performances are compared between the original program on CPU using either a multithreaded implementation or otherwise, and the same program running on GPU which provides similar results. Outputs are compared to make sure that no differences other than those due to the single precision computation appears.

It is always a delicate task to compare programs using their execution time as it depends heavily on the quality of the implementation. Unlike what was done in the Intel's study [16], we do not push the implementation optimization to the maximum, but instead choose to focus on good quality implementation attainable with reasonable effort and hardware knowledge by the typical remote sensing scientist. There is probably room for improvement on the CPU side (using SIMD) as well as on the GPU side (coalescing access).

The hardware is an Intel i7-920 with 6 GB of RAM; the GPU is a Nvidia GTX-260 used purely for processing, the display being handled by another card. In terms of software, the C/C++ compiler gcc/g++ 4.4.3 is used with version 3.0 of the CUDA toolkit for both CUDA and OpenCL simulations. Compilation option used is -O3 which turns on all the optimization available.

A. Pixel-Based Processing, a First Naive Example

The first category of algorithms refers to those which perform operations on single pixels without the need of context. This category can include any arithmetic or logical operation on pixels such as simple additions and thresholdings or more complex computations such as numerical solutions of equations where the pixel value is a parameter. This category also includes pixel-based classification such as maximum likelihood, neural networks, or SVM classifiers. Finally, another interesting subset of algorithms for remote sensing image processing which belong to this category are the coordinate transformations used in image orthorectification (through sensor models), map projection transforms, and any analytical model-based in preparation for image resampling.

As the data transfer from the CPU to the GPU is relatively slow, the key factor in order to benefit from the GPU's massively parallel architecture will be the complexity of the pixel-wise operation. In order to illustrate this, we have selected two classical algorithms.

1) Algorithm Description: The first algorithm is the computation of the normalized difference vegetation index (NDVI) [18] which is a radiometric index which combines the red (R) and near infrared (NIR) reflectances in order to estimate the amount of vegetation:

$$NDVI = \frac{NIR - R}{NIR + R}. \tag{2}$$

The second algorithm is the spectral angle mapper (SAM), which computes, for each pixel of the image p with n_b bands, the spectral angle with respect to a reference pixel r. The spectral angle is defined as

$$\mathrm{SA} = \arccos\left(\sum_{b=1}^{n_b} r(b).p(b) / \sqrt{\sum_{b=1}^{n_b} r(b)^2 \sum_{b=1}^{n_b} p(b)^2}\right) \tag{3}$$

where b is the spectral band, r is the reference pixel, and p is the current pixel. The interest of evaluating the SAM is that its computation is more costly due to the square root and the arccos function.

2) Implementation Details: The straightforward way to implement these pixel processing algorithms is to get each thread of the GPU to process one pixel. In this case, the kernel is very simple as shown in Fig. 3.

Unfortunately, in this case, the GPU processing appears to be slower than the CPU by about 20% (Fig. 4). A quick profiling of this case shows that the kernel spends about 92% of its time for memory transfer and only 8% for the processing. This shows clearly that the NDVI computation is too simple to get any benefit by itself from the GPU architecture.

On the other hand, the computation cost for the spectral angle (3) is higher than the NDVI. In this case, we start to see some

```
__global__ void ndviKernel(float* pix, float* ndvi,
           int numBands, int indexRed, int indexNIR,
           int imageWidth)
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  float nir = pix[numBands*(x +  y*imageWidth)
                  + indexNIR];
  float red = pix[numBands*(x +  y*imageWidth)
                  + indexRed];
  ndvi[x +  y*imageWidth] = (nir - red)/(nir + red);
}
```

Fig. 3. Naive kernel example for NDVI: unfortunately, the amount of processing for each pixel is too low to get any gain from GPU.

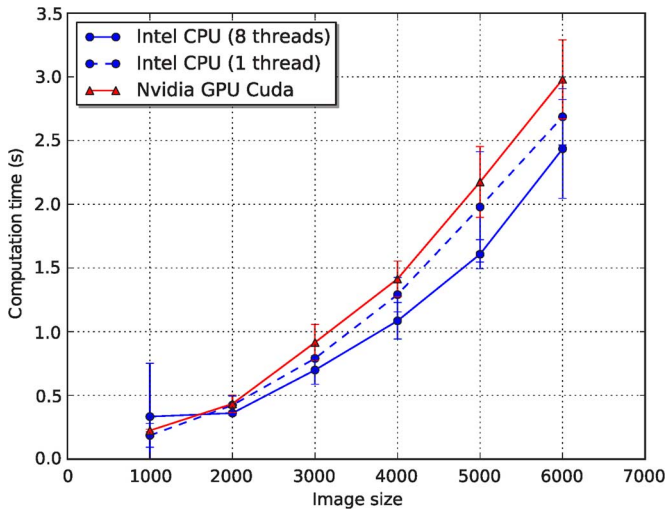

Fig. 4. Computation time for the NDVI for increasing image sizes: NDVI is not a good candidate to benefit from a GPU implementation on its own.

gain from using the GPU, but it is not yet convincing (Fig. 5): the GPU is faster than the CPU using one thread, but comparable to the CPU fully using its 8 threads.

3) Results: Fig. 4 shows the computation time for the NDVI for image size from 1000×1000 to 6000×6000. Due to the significance of the IO operations compared to the computation, the execution time displays a large variance; average timing and the standard deviation for these timings on at least 20 runs are represented. As one can observe, the NDVI computation is too simple to be a good candidate for GPU optimization on its own. Actually, the GPU version is slower than the CPU versions. However, the multithreaded CPU implementation brings some benefit with respect to the single-threaded one but the improvement is also limited and much lower that the factor 8 expected.

Fig. 5 shows the same kind of simulation for the spectral angle computation. Here, the parallel implementations are much more efficient than the single threaded one, but the spectral angle is still too simple for the GPU implementation to provide a speedup with respect to the CPU multithreaded one.

B. Neighborhood-Based Processing

The second category of algorithms we are interested in is the ones which use a pixel and its close neighbors (on a regular grid) in order to compute the output value for a single pixel. This category includes many image processing tasks such as linear

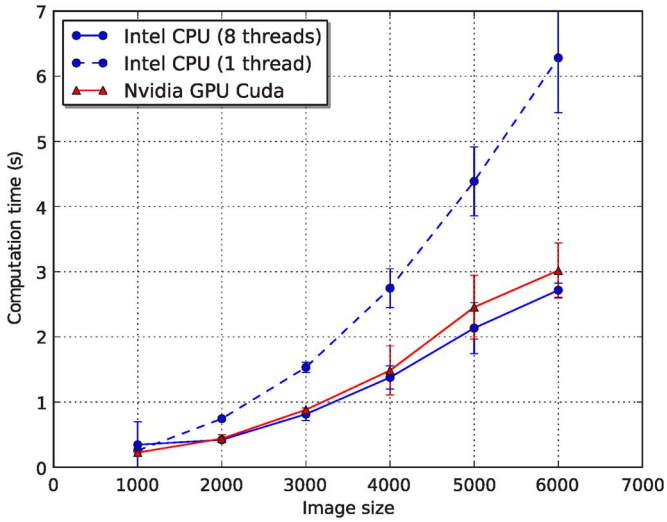

Fig. 5. Computation time for the spectral angle for image size from 1000×1000 to 6000×6000: there is some improvement over the CPU implementation on a single thread, but it is still not sufficient to justify a GPU implementation.

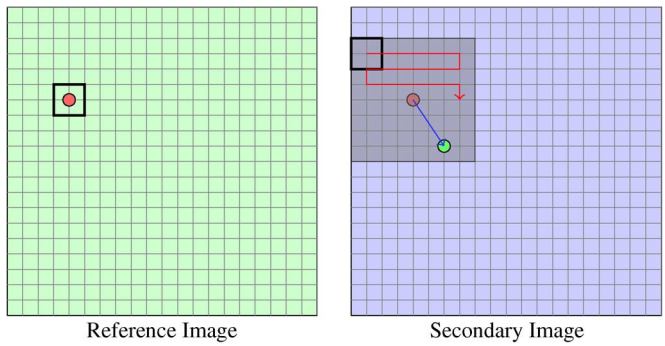

Fig. 6. Illustration of the fine correlation process.

filtering by convolution, but also nonlinear filtering (median, mathematical morphology), local statistics, etc.

Texture estimations using the Grey-Level Co-occurrence Matrices (GLCM) are also in this category, but they are more computationally intensive since two shifted neighborhoods are used, so they are very interesting candidates for GPU implementation.

1) Algorithm Description: We choose here another particular case where the local computation is made using several pixel neighbors: fine image correlation. This technique is used for disparity map estimation between stereo image pairs [19]. Let I be the reference image and J be the secondary image, which are supposed to be roughly superimposable; one is interested in finding the local shift $(\Delta x, \Delta y)$ between small image patches which maximizes the correlation coefficient:

$$\rho_{I,J}(\Delta x, \Delta y) = \frac{\sum_{x,y} I(x,y) J(x+\Delta x, y+\Delta y)}{\sqrt{\sum_{x,y} I(x,y) \sum_{x,y} J(x+\Delta x, y+\Delta y)}}. \quad (4)$$

This processing is applied for every shift in a given exploration area and for every pixel in the image (see Fig. 6).

2) Implementation Details: This particular problem of the estimation of a disparity map by fine correlation poses several issues for the implementation on GPUs and the adaptation is not as straightforward as the previous examples. On the other hand,

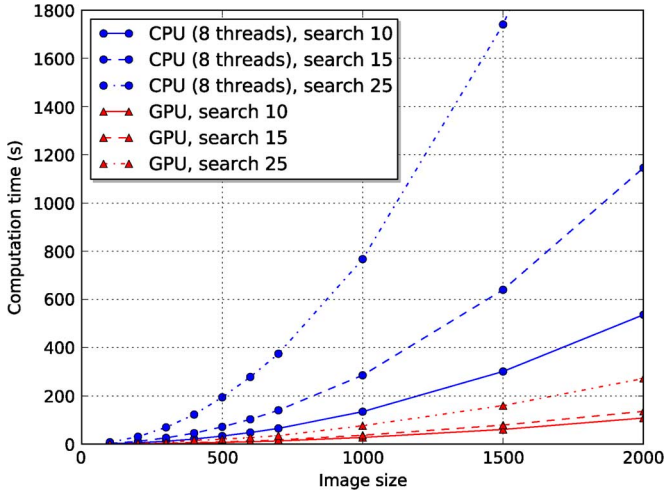

Fig. 7. Computation time of the fine correlation for different hardware for different search radius size and for a image size from 100×100 to 700×700 pixels: the GPU version is 8 to 12 times faster that the multithreaded CPU version fully using 8 threads.

the amount of computation per pixel is much greater than in the previous cases, so the potential gain is significant.

An adaptation of a similar algorithm to perform the auto-correlation of an image is given in [20], but the search window size was specific to their problem and involved different trade-offs.

The first question is which part of the processing should be implemented on the GPU. After profiling the CPU version of the algorithm, it appeared that 95% of the time was spent in computing the correlation for each shift. Producing the correlation map for each pixel seems to be the ideal part to be implemented.

The second question concerns the degree of flexibility required in the implementation. Of course, we do not want any limit concerning the size of the image to be processed. But given the streaming process described in Section II-A, it is not an issue. However, we also do not want a restriction on the size of the patch used to compute the local correlation or on the size of the exploration window.

The approach chosen for this case is that each thread will compute the correlation value corresponding to one displacement. Each block of threads will compute the correlation map for one pixel. Of course as there is no restriction in the size of the search window, the number of possible displacements for one pixel (which is computed by one thread block) can be greater than the maximum number of threads in one block (which is currently limited to 512). To go around this limitation, when the window search size is too big, the whole correlation map is computed through multiple kernel launches with different parameters.

There is an issue with the matching window which can extend outside of the image. One way to solve the problem is to make costly checking for each access directly in the GPU kernel. Here we avoid the issue altogether by using the strategy presented in Section II-A which ensures that all accesses will be valid.

The final result (Fig. 7) shows a speedup of 8 to 12 times compared to the CPU implementation for the whole fine registration process. The correlation computation is no longer the limiting factor and further improvement would require improving the interpolation process to find subpixel displacements.

3) Results: Fig. 7 shows the computation time for the fine correlation for different image size and different radius for the search windows. A search window of radius 25 means that 2601 (51×51) different possible displacements will be explored for each pixel. In all cases, the size of the patch was fixed to 11×11 pixels. The CPU time corresponds to the multithreaded version, making full use of the processor. Here the difference between the GPU version and the CPU version is significant. For example, a processing that takes 4 min 30 s on the GPU takes more than 51 min on the CPU. It is worth noting that the correlation computation part, the only one implemented on the GPU here, used to represent 95% of the total computation time. The execution time for this part has been reduced by a factor of 20 and now represents only 50% of the total computation time. To obtain significant further improvements it is required to work on another part of the computation such as the correlation map interpolation [cf. Amdahl's law in (1)].

C. Irregular or Non-Local Processing

This third class of algorithms consists of cases where the pixels we are interested in are in irregular positions or represent only a small percentage of the pixels of the image and the computation to be performed is very demanding. In these cases, the amount of data to be transferred to the processing unit is small and the computing cost is large.

In remote sensing image analysis, some examples of these algorithms are irregular interpolation by thin plate splines, histogram kernel estimation, Voronoi/Delaunay triangulations, and vector object processing (operations on polygons or poly-lines yielded by segmentation and detection algorithms).

1) Algorithm Description: One example of such processing is point density estimation. This program estimates the point density for every point in an image for a given set of points. Several applications use this density estimation: we can mention point feature extraction such as SIFT [21], permanent scatterers in SAR images [22], among others. One example of possible output for this process is illustrated in Fig. 8.

The estimation is done using a Gaussian kernel as in (5) where $d_{p,h}$ denotes the distance between the pixel p where the density is computed and the point h of the set of points:

$$\rho_p = \frac{1}{2\pi\sigma^2}\sum_h e^{\frac{d^2_{p,h}}{2\sigma^2}}. \tag{5}$$

This equation means that the density is spread around point h following a Gaussian model.

The decision to optimize this problem was taken after noticing that the CPU performances of the original version of the program were not satisfying, taking hours to provide a density map of permanent scatterers.

Another example is the learning step of SVM classifiers [23]. As mentioned in Section III-A, SVM classification belongs to pixel-based processing, since class prediction for a given pixel is only a matter of a few scalar products with support vectors. Prior to SVM classification, SVM learning is the task of identifying these support vectors into a training set and involves several iterations over a set of training examples which can be located

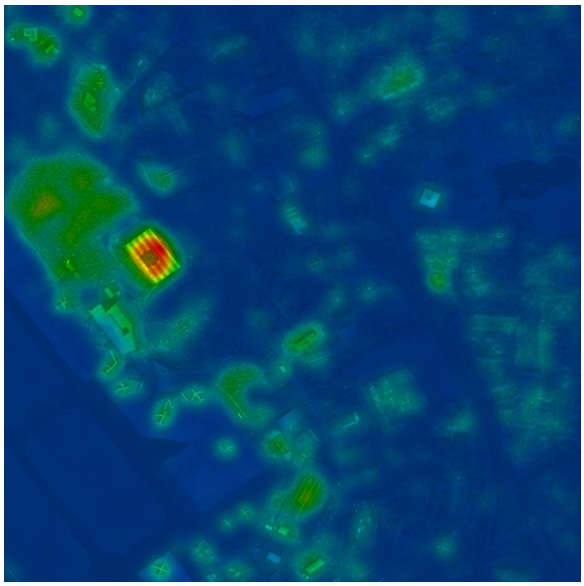

Fig. 8. Example of possible output of a point density estimation: in this example, the points correspond to SIFT detections. About 10,000 points are detected on this image.

anywhere in an image. This makes the SVM learning algorithm fall into the class of irregular algorithms.

2) Implementation Details: For the density estimation example, there are two different approaches to compute the final image. One is to go through the set of points and, for each of them, compute their impact on the whole image using an accumulator. The other approach is to go through all pixels and, for each of them, sum up the impact of all points in the point set. Depending on the relative size of the image and the point set, one or the other can be privileged in the case of CPU implementation. The first one corresponds to a *scatter* approach while the second one is a *gather* approach.

The *gather* approach is one where the computation of one output pixel is done at once by a single computation unit, gathering information from several positions from the input. The *scatter* approach is the reverse, when input are accessed once by a single computation unit and their impact is reported to the relevant output pixel which works as an accumulator.

GPUs are more suited to the *gather* approach and this is the one selected here: for each pixel, we go through the point set and compute the impact of each point to this pixel. The other advantage is that this approach is perfectly suitable for the pipeline model described in the previous sections.

This simple approach, where one thread processes one pixel, iterating over the list of points, already provides an impressive gain compared to the CPU version. However, using additional features available from GPUs, additional performance gains are possible.

The first improvement is to use the constant memory of the GPU. Accesses to the constant memory are much faster when they are synchronized between the threads. Constant memory is the ideal candidate to store the point coordinates as they are frequently accessed by each thread.

Another source of improvement is to factorize some computations that are common to different threads, thus reducing the total amount of computations to be performed. When computing

```
__constant__ float pt_c[CHUNK_SIZE*2];

__global__ void pointDensityKernel(float* pix,
  int numPoint, int originX, int originY,
  float spacingX, float spacingY,
  int imageWidth, int radiussq)
{
   int x1 = blockIdx.x*blockDim.x*PIX_PER_THREAD
            + threadIdx.x;
   int x2 = x1 + blockDim.x;
   int x3 = x1 + 2*blockDim.x;
   int x4 = x1 + 3*blockDim.x;
   int y = blockIdx.y*blockDim.y + threadIdx.y;

   float accum1 = 0.0f;
   float accum2 = 0.0f;
   float accum3 = 0.0f;
   float accum4 = 0.0f;
   for (int k = 0; k < numPoint; k++)
   {
     float ptX = pt_c[2 * k];
     float ptY = pt_c[2 * k + 1];
     float pixX1 = x1 * spacingX + originX;
     float pixX2 = x2 * spacingX + originX;
     float pixX3 = x3 * spacingX + originX;
     float pixX4 = x4 * spacingX + originX;
     float pixY = y * spacingY + originY;
     float disty_sq = (ptY - pixY) * (ptY - pixY);
     float distsq;
     distsq = (ptX - pixX1) * (ptX - pixX1)
              + disty_sq;
     accum1 += __expf(-distsq/radiussq/2);

     distsq = (ptX - pixX2) * (ptX - pixX2)
              + disty_sq;
     accum2 += __expf(-distsq/radiussq/2);

     distsq = (ptX - pixX3) * (ptX - pixX3)
              + disty_sq;
     accum3 += __expf(-distsq/radiussq/2);

     distsq = (ptX - pixX4) * (ptX - pixX4)
              + disty_sq;
     accum4 += __expf(-distsq/radiussq/2);
   }
   pix[x1+y*imageWidth] += accum1/NORMALIZATION;
   pix[x2+y*imageWidth] += accum2/NORMALIZATION;
   pix[x3+y*imageWidth] += accum3/NORMALIZATION;
   pix[x4+y*imageWidth] += accum4/NORMALIZATION;
}
```

Fig. 9. CUDA kernel for the point density computation: each thread computes the density value for four points of the same line simultaneously. The OpenCL kernel is very similar.

the distance between one point and one pixel, the computation is done for the x component and the y component. The y component will be the same for all the points of a line. By using one thread to process not only one point, but several points on the same line, this part of the computation can be done only once. A trade-off has to be made to keep the number of threads high enough to use all the computation units of the GPU and not to use too many registers (as that would reduce the number of threads that can run concurrently). In the present example, each thread processes four consecutive points on the same line.

In this case, as shown in Fig. 10, the improvement over the GPU is impressive: 130 times faster than the multithreaded CPU version.

Regarding the SVM learning problem, the CPU implementation relies on LibSVM [24], a widely known library to perform SVM classification and regression, while the GPU version was handled by cuSVM [25], a CUDA implementation of

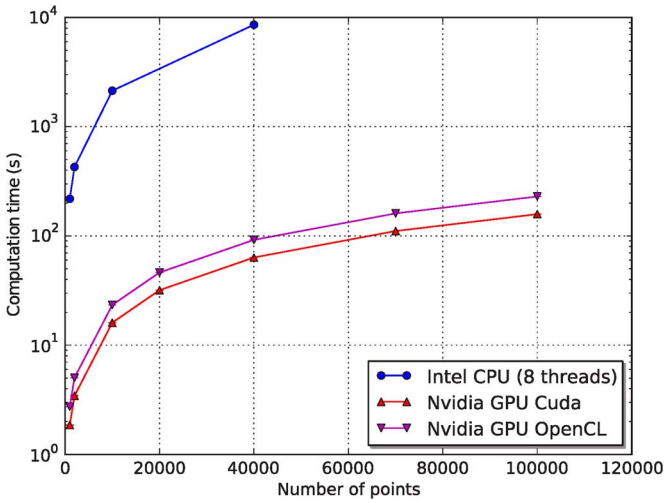

Fig. 10. Computation time of the point density for different hardware for different number of points for an image of 36 MPixels: the GPU version is about 130 times faster that the multithreaded CPU version (Note that the timing is in logarithm scale).

SVM restricted to regression and two-class classification with a Gaussian kernel. In this case, all that is needed is to fit the LibSVM and cuSVM calls into the Orfeo Toolbox framework. In each case, the learning and classification steps are implemented in separate filters and the classification filters are given streaming capabilities. Parameters from both libraries are tuned to match results as closely as possible.

3) Results: Fig. 10 presents the results obtained by the implementation of the point density computation. Here, the gain is so significant that a logarithm scale is used to show the CPU and the GPU time on the same plot. The speedup from the GPU version is about 130 times. For this particular case, two implementations for CUDA and OpenCL are realized. It appears that for this particular case the OpenCL implementation is about 45% slower than the CUDA implementation. However, the gain compared to the CPU is still impressive, and with respect to the CUDA implementation, there is the advantage of supporting a wider range of hardware including CPUs and GPUs. Computation speed improvements of such an order of magnitude open a whole range of possibilities where interactive processing becomes convenient.

Regarding the SVM learning and classification problem, both CPU and GPU filters have been used to perform change detection on a pair of registered SPOT5 images in the context of a severe flooding event. This dataset contains 2320×4320 pixels with three spectral bands for each date. A training mask of 62,676 pixels denoting change and no-change areas was used to train the SVM. The learning step took 572 seconds with the CPU implementation, while only 1.35 seconds were necessary for the GPU implementation to converge, which is tremendously faster (more than 400 times), although we should mention that the CPU version of the learning step is mono-threaded in this case. As expected, being a simpler algorithm, the classification step shows smaller time improvements, with 489 seconds for the multithreaded CPU version and 146 seconds for the GPU one—only about 3 times faster.

IV. Discussion and Perspectives

As we have seen, some classes of algorithms can benefit tremendously from a GPU implementation. Typically, these algorithms can be identified as algorithms that do mostly local processing (limited distance impact) and intensive computation for each pixel. We have seen for example that the SAM is at the lower bound in terms of computation to make it valuable to implement on GPU for an image with four bands (Fig. 5). In that particular case, the limit appears to be a few hundred operations (mainly due to the trigonometric function). Below this limit, the time spent transferring the data from the CPU memory to the GPU overcomes any benefit in the computation speed.

Above this limit, the benefits can be very important. In some cases, they can be so important that they could change the way the human interacts with the process. When the process takes hours, or even minutes, it is not conceivable to have the user sit in front of the computer, waiting for the results. In this case, the data will be processed from end to end and the user will exploit the final result. If there is a need to adjust some parameters, the data will be reprocessed.

One drawback of this classical scheme is that it tends to limit the interactivity between the human and the algorithm. When we reach a situation where we can process a screen size area in about a second, it becomes possible to do the processing in real time. In this situation, any modification of any parameter will trigger immediate feedback: the user is able to interact much more with the algorithm. The human can become a real part of the processing chain.

This can lead to an improvement of the classic processing chains in use, but it can also lead to the development of new paradigms. One obvious example is the application of active learning to remote sensing problems [26]. Benefiting from the major speed-up brought by the GPU implementation of SVM learning (see Section III-C3), the training samples selection step could change drastically: near-real-time feedback on the pertinence of selected samples and on primitive-wise confidence of the classifier becomes achievable. Other common remote sensing image processing tasks could benefit from immediate quality feedback, such as image co-registration for instance.

V. Conclusion

As demonstrated in this paper, adapting the most expensive part of a processing pipeline to benefit from the processing power of GPUs is quite simple. With a minimum investment (hardware cost is around US$200, the software used here is free and open source), performance gains can attain 10 to 400 times on the critical portion of the processing.

One of the main shortcomings, which is the relatively slow computation in double precision (important for some scientific computations), has been addressed by the new Fermi architecture released by Nvidia in April 2010. We will definitely witness an increasing number of GPU implementations for remote sensing processing algorithms in the near future.

Still, benefiting from this massive speed-up requires one to carefully select those algorithms which fit well in the GPU computing architecture, identify the critical sections to optimize, and look closely at how things are implemented.

All this complexity should remain hidden to the end-user, which is exactly what the high level of abstraction provided by the Orfeo Toolbox framework allows. Further improvements can be made in that direction by proposing a mechanism to switch seamlessly from CPU to GPU versions of algorithms depending on available hardware. Another interesting perspective would be to run GPU-enabled filters on GPU blades, which gather several GPU devices on a single hardware.

The source code corresponding to the examples presented in this paper used to generate the results is available for download from the Orfeo Toolbox web site (http://www.orfeo-toolbox.org/OTB-GPU). Most of it will be integrated in the upcoming releases of the library.

REFERENCES

[1] K. Asanovíc, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The Landscape of Parallel Computing Research: A View From Berkeley," University of California, Berkeley, Tech. Rep., Dec. 2006.

[2] D. Blythe, "Rise of the graphic processor," *Proc. IEEE*, vol. 96, no. 5, pp. 761–778, May 2008.

[3] *"The Orfeo Toolbox Software Guide"* 2010 [Online]. Available: http://www.orfeo-toolbox.org

[4] E. Christophe and J. Inglada, "Open source remote sensing: Increasing the usability of cutting-edge algorithms," *IEEE Geosci. Remote Sens. Newsletter*, pp. 9–15, Mar. 2009.

[5] ITK, the Insight Toolkit. [Online]. Available: http://www.itk.org

[6] J. R. Schott, *Remote Sensing: The Image Chain Approach*. New York: Oxford Univ. Press, 2007.

[7] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Computers*, vol. 21, no. 9, pp. 948–960, Sep. 1972.

[8] J. L. Manferdelli, N. K. Govindaraju, and C. Crall, "Challenges and opportunities in many-core computing," *Proc. IEEE*, vol. 96, no. 5, pp. 808–815, May 2008.

[9] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Proc. IEEE Int. Symp. Parallel and Distributed Processing*, Rome, Italy, May 2009, pp. 1–10.

[10] G. M. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities," in *AFIPS Conf. Proc.*, Apr. 1967, vol. 30, pp. 483–485.

[11] J. Setoain, M. Prieto, C. Tenllado, and F. Tirado, "Real-time onboard phyperspectral image processing using programmable graphic hardware," in *High Performance Computing in Remote Sensing*. Boca Raton, FL: Chapman & Hall/CRC, 2007, pp. 411–451.

[12] *The OpenCL Specification*, Khronos Group Std. 1.0, Dec. 2008.

[13] D. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: A Hands-On Approach*. New York: Elsevier Science & Technology, 2010.

[14] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *High Performance Computing—HiPC 2007*, ser. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer, 2007.

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *J. Parallel and Distributed Comput.*, vol. 68, no. 10, pp. 1370–1380, Oct. 2008.

[16] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," in *Proc. 37th Annual Int. Symp. Computer Architecture (ISCA'10)*, Jun. 2010.

[17] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of CUDA and OpenCL," in *CoRR*, 2010, abs/1005.2581.

[18] C. J. Tucker, "Red and photographic infrared linear combinations for monitoring vegetation," *Remote Sens. Environ.*, no. 8, pp. 127–150, 1979.

[19] J. Inglada and A. Giros, "On the possibility of automatic multi-sensor image registration," *IEEE Trans. Geosci. Remote Sens.*, vol. 42, no. 10, pp. 2104–2120, Oct. 2004.

[20] P. J. Lu, H. Oki, C. A. Frey, G. E. Chamitoff, L. Chiao, B. J. Au, M. Christiansen, A. B. Schofield, D. A. Weitz, P. J. Lu, D. A. Weitz, H. Oki, C. A. Frey, G. E. Chamitoff, L. Chiao, E. M. Fincke, D. M. Tani, P. A. Whitson, J. N. Williams, W. V. Meyer, R. J. Sicker, B. J. Au, M. Christiansen, and A. B. Schofield, "Orders-of-magnitude performance increases in GPU-accelerated correlation of images from the international space station," *J. Real-Time Image Process.*, vol. 5, no. 3, pp. 179–193, 2009.

[21] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Computer Vision*, vol. 60, pp. 91–110, Feb. 2004.

[22] A. Ferretti, C. Prati, and F. Rocca, "Permanent scatterers in SAR interferometry," *IEEE Trans. Geosci. Remote Sens.*, vol. 39, no. 1, pp. 8–20, Jan. 2001.

[23] V. Vapnik, *Statistical Learning Theory*. New York: Wiley, 1998.

[24] C.-C. Chang and C.-J. Lin, LIBSVM: A Library for Support Vector Machines. 2001 [Online]. Available: http://www.csie.ntu.edu.tw/cjlin/libsvm

[25] A. Carpenter, cuSVM: A CUDA Implementation of Support Vector Classification and Regression. Jan. 2009 [Online]. Available: http://patternsonascreen.net/cuSVMDesc.pdf

[26] D. Tuia, F. Ratle, F. Pacifici, M. F. Kanevski, and W. J. Emery, "Active learning methods for remote sensing image classification," *IEEE Trans. Geosci. Remote Sens.*, vol. 47, no. 7, pp. 2218–2232, Jul. 2009.

Emmanuel Christophe received the Engineering degree in telecommunications from École Nationale Supérieure des Télécommunications de Bretagne, Brest, France, and the DEA in telecommunications and image processing from University of Rennes 1, France, in 2003. In October 2006, he received the Ph.D. degree from Supaero and University of Toulouse, France, in hyperspectral image compression and image quality.

He was a visiting scholar in 2006 at Rensselaer Polytechnic Institute, Troy, NY. From 2006 to 2008, he was a research engineer at CNES, the French Space Agency, focusing on information extraction for high resolution optical images. Between 2008 and 2010, he was with CRISP, National University of Singapore, where he was tackling new challenges for remote sensing in tropical areas. He is now with Google Inc. in California.

Julien Michel received the Telecommunications Engineer degree from the École Nationale Supérieure des Télécommunications de Bretagne, Brest, France, in 2006.

From 2006 to 2010, he was with Communications et Systèmes, Toulouse, France, where he worked on studies and developments in the field of remote sensing image processing. He is now with the Centre National d'Études Spatiales (French Space Agency), Toulouse, France, where he is in charge of the development of image processing algorithms for the exploitation of Earth observation images, mainly in the field of very high resolution image analysis.

Jordi Inglada received the Telecommunications Engineer degree from both the Universitat Politècnica de Catalunya, Barcelona, Spain, and the École Nationale Supérieure des Télécommunications de Bretagne, Brest, France, in 1997, and the Ph.D. degree in signal processing and telecommunications in 2000 from Université de Rennes 1, Rennes, France.

He is currently with the Centre National d'Études Spatiales (French Space Agency), Toulouse, France, working in the field of remote sensing image processing at the CESBIO laboratory. He is in charge of the development of image processing algorithms for the operational exploitation of Earth observation images, mainly in the field of multitemporal image analysis for land use and cover change.

Dr. Inglada is an Associate Editor of the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING.