K12is – 231

Computing efficiently spectral-spatial classification of hyperspectral images on commodity GPUs

Pablo Quesada-Barriuso, Francisco Argüello and Dora B. Heras

Abstract The high computational cost of the techniques for segmentation and classification of hyperspectral images makes them good candidates for parallel processing, in particular, for computing on Graphics Processing Units (GPUs). In this paper an efficient projection on the GPU for the spectral–spatial classification of hyperspectral images using the Compute Unified Device Architecture (CUDA) for NVIDIA devices is presented. A watershed transform is applied after reducing the hyperspectral image to one band through the calculation of a morphological gradient, while the spectral classification is carried out by Support Vector Machines (SVMs). The results are combined with an adaptive majority vote. The different computational stages are concatenated in a pipeline that minimizes the data transfer between the main memory of the host computer and the global memory of the graphics device to maximize the computational throughput. The memory hierarchy and the thousands of threads available in this architecture are efficiently exploited. It is possible to study different data partitioning strategies and thread block arrangements in order to promote concurrent execution of a large number of threads. The objective is to efficiently exploit commodity hardware with the aim of achieving real-time execution for on-board processing.

Key words: Hyperspectral images, Watershed, Classification, CUDA

231.1 Introduction

Recent advances in sensor technology have led to hyperspectral images being now widely available [1, 2]. The special characteristics of hyperspectral images, which provide a detailed spectrum for each pixel, allow distinguishing among physical materials and objects even at pixel level, presenting new challenges to spectral analysis, target detection, image segmentation or classification. Nevertheless, the large number of spectral channels of the hyperspectral images makes most of the commonly used methods designed for the processing of grey level or color images not appropriate. To take full advantage of the rich information provided by the spectral dimension new algorithms are required.

The supervised classification of hyperspectral images has been a very common topic in the last decades. Pixel-wise classifiers, for instance, consider only the spectral information of the pixel [1, 3–5]. In particular, pixel-wise classification by Support Vector Machine (SVM) classifiers has been introduced and shown good results when a small number of training samples are available [3]. However, this pixel-wise classification does not consider information about spatial structures. Therefore, the classification can also take advantage of the spatial relationships among pixels, allowing more elaborate spectral–spatial models for a more accurate segmentation and classification of the image [6–8]. The spatial information can be included considering different approaches. The first approach consists in including information from the closest neighborhood of a pixel through morphological filtering [9], morphological leveling [6] or Markov random fields [10]. The second approach consists in carrying out a segmentation of the image by methods that are usually based in graphs [11]. Among these some unsupervised methods have been widely used: partitional clustering [7], hierarchical segmentation [12], MSF [13] and watershed [8]. The watershed transform is a widely used method for non-supervised image segmentation, specially suitable for low contrast images [14]. It is usually applied to the morphological gradient of a two dimensional image for extracting homogeneous regions with respect to grey level values.

Recently, Tarabalka et al. [8] have presented a spectral–spatial classification scheme for hyperspectral images that uses the watershed transform. It is based on an SVM spectral classification, followed by a Majority Vote (MV) process among the classified pixels within the same watershed region. Among the proposals presented by the authors to reduce the image to one band, such as multidimensional or vectorial gradients. One of the most efficient approaches, in terms of classification quality, is obtained through a Robust Color Morphological Gradient (RCMG) calculation. The good classification results of this proposal in urban and open areas had led us to adopt it in this work.

The computational cost of the techniques for segmentation and classification of hyperspectral images is high, which makes them good candidates for parallel and, in particular, for General-Purpose Computing on Graphics Processing Units (GPGPU). The focus of this study is to provide a solution for a GPU platform, adapting the hyperspectral processing to a low cost parallel computing architecture. With this

approach the on-board processing of information is possible without the need for bulky high performance computing infrastructures.

In most cases neither sequential nor existing parallel algorithms can be directly implemented in the GPU and it is necessary to modify the flow of the computations in order to fully exploit the architecture. The use of Graphics Processing Units to process hyperspectral images has been gaining popularity in recent years. For instance, algorithms for spectral unmixing [15, 16], target detection [17, 18], classification [1] and segmentation [19, 20] have led to more complete tools [21, 22]. A spectral–spatial GPU classification tool was presented in [22]. In this tool the spatial information is introduced by MV within a fixed window where each pixel is assigned to the most predominant class, so the spatial structure of the image is not fully considered. As a result, this MV implementation may generate different classes within the same watershed region, unlike in [8].

The interest is on exploring GPU architectures for hyperspectral processing by developing techniques that can be efficiently projected on GPU consumer platforms with the objective of achieving real-time execution that makes on-board processing possible. In this paper a spectral–spatial classification scheme for hyperspectral images based on [8] is presented, specially adapted for GPU processing using CUDA. The process consists in the calculation of a morphological gradient operator, that reduces the dimensionality of the hyperspectral image, followed by the calculation of a watershed transform based on Cellular Automata (CA) over the resulting 2D image, and a spectral classification based on SVM. A MV process combines the spectral and spatial results. The thousands of threads available in the GPU are efficiently exploited. The different stages are concatenated in a pipeline processing that minimizes the data transfers between the host and the device and maximizes the computational throughput. Furthermore, data are reused within the GPU, taking advantage of the shared memory and cache hierarchy of the architecture. In addition, different hyperspectral data partitioning strategies and thread block arrangements are studied in order to effectively exploit the memory and computing capabilities of the GPU architecture.

The reminder of this paper is organized as follows: in Section 231.2 some GPU and CUDA fundamentals are introduced. Section 231.3.1 introduces the morphological gradient, Section 231.3.2, the watershed transform, and Section 231.3.3 the majority vote approach for spectral–spatial classification. The implementations of the algorithms and the results obtained are discussed in Sections. 231.4 and 231.5, respectively. Finally, Section 231.6 presents the final remarks.

231.2 GPU Architecture

The most recent GPUs provide massively parallel processing capabilities based on a data parallel architecture. The NVIDIA GPU architecture is organized into a set of Streaming Multiprocessors (SMs), each one with many cores called streaming processors [35], as shown in Fig. 231.1 (a). These cores can manage hundreds of threads
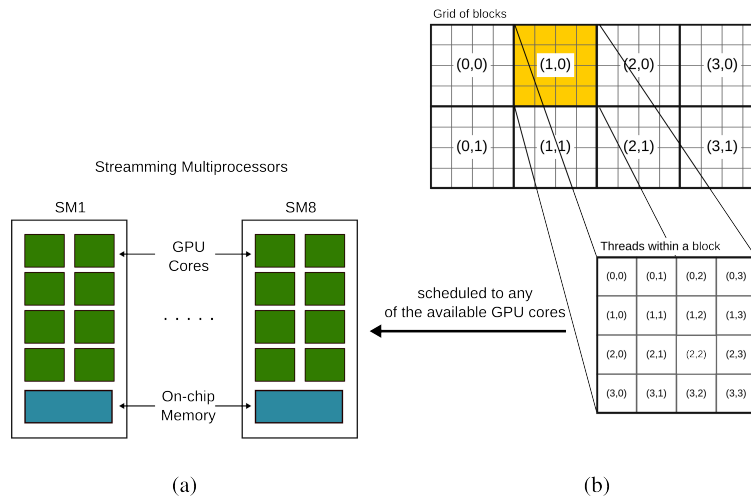
Fig. 231.1 NVIDIA CUDA architecture. Streaming multiprocessors (a) and organizations of Grid, blocks and threads (b)

in a Single Instruction Multiple Data (SIMD) programming model. The GPU cores execute the same instruction simultaneously on different data unlike the multicore processors that are Multiple Instruction Multiple Data (MIMD) (different cores execute different threads operating on different data).

CUDA for NVIDIA devices, is an Application Programming Interface (API) for writing programs that are executed in the GPU. A CUDA program, which is called a kernel, is executed by thousands of threads grouped into blocks, as illustrated in Fig. 231.1 (b). The Compute Unified Device Architecture (CUDA) architecture has a global memory of Dynamic Random Access Memory (DRAM) that is available for all the blocks. There is also an on-chip *shared memory* space only available per block. This feature enables an extremely rapid read/write access to the data in this memory but with the lifetime of the block. Furthermore, it is not possible to read or write data to the shared memory allocated to another block. Finally, each thread has its own *local memory* and *registers*. Examples include the NVIDIA G80 and GT200 graphics cards series.

The Fermi and Kepler architectures [36] have also a *cache hierarchy* consisting of a configurable L1 and a unified L2 caches. The 64 KB of on-chip memory can be configured as 48 KB of shared memory and 16 KB of L1 cache or vice versa. There are 64 KB of this memory available for each SM. The L2 is a unified cache up to 1536 KB shared by all the SMs. The accesses to the DRAM are cached in this memory hierarchy. The NVIDIA Tesla GF100 and the GeForce 500 series are examples of the Fermi architecture. The Tesla K-series family of products includes the Kepler K10, K20 and K20X GPU accelerators with different chipsets. In particular the Tesla K20X based on the GK110 chipset incorporates 2688 CUDA cores and 6 GB of memory. These chipsets can be found in commodity GPUs like the GTX680

graphics card, used in this work, which has a GK104 chipset (1536 CUDA cores and 2 GB of memory).

The challenge of GPU programming is to increase the computational throughput. To achieve this, important aspects that must be considered are [37]: minimizing CPU–GPU data transfers, aligning accesses to consecutive memory locations, maximizing data reuse, balancing the workload among threads, and minimizing their divergence.

231.3 Spectral–Spatial Classification of Hyperspectral Images

Hyperspectral images are basically digital pictures where each pixel is represented by a set of *n* values. Each value corresponds to a spectral component across the visible and infrared light bands [18]. The number of captured bands depends on the properties of the hyperspectral sensor. For example, the well known Reflective Optics System Imaging Spectrometer (ROSIS) is able to record 103 spectral bands [24], while the Airbone Visible-Infrared Imaging Spectrometer (AVIRIS) is able to record 224 spectral bands [25].

Most classification methods for hyperspectral images process each pixel independently using pixel-wise classifiers, but do not take into account the spatial information of the neighborhood [26]. Nevertheless, it has been proved that the classification results significantly improve when spatial information is incorporated [6–8].

An efficient approach to integrate spectral and spatial information in a classification system is defined by Tarabalka et al. [8]. The process consists of the stages
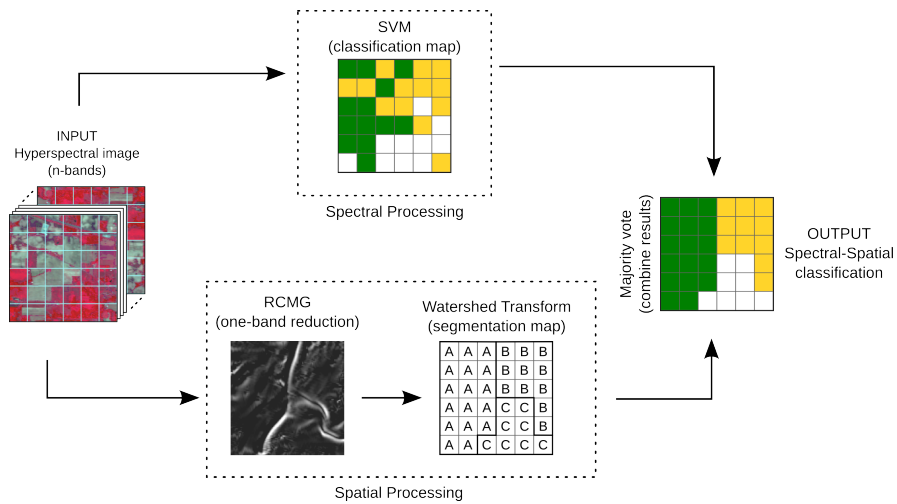

Fig. 231.2 Spectral-spatial classification scheme, which consists of a spectral stage (*top*), a spatial stage (*bottom*), and a final stage to combine the results
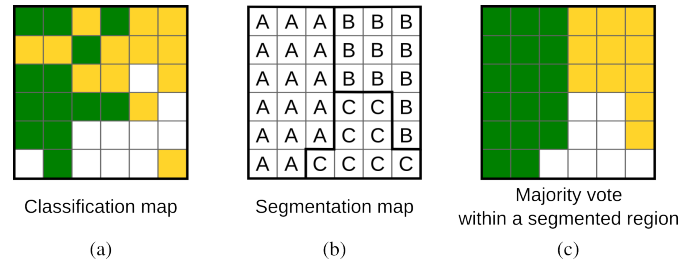
Classification map   Segmentation map   Majority vote
                                                       within a segmented region

(a)   (b)   (c)

Fig. 231.3 Example of majority vote application for spectral–spatial classification

shown in Fig. 231.2. On one hand, the spectral processing is applied over the hyperspectral input image using a SVM that produces a classification map (shown at the top of the figure). Each pixel of this map belongs to one class predicted by the SVM (three classes in this example). On the other hand, the spatial processing, applied to the one–band image generated after a RCMG calculation, creates a segmentation map using a watershed transform (shown in the bottom of the figure). In this map, all of the pixels are labelled according to the region they belongs to.

Finally, the spectral and spatial results are combined using a majority vote process. Each pixel in a watershed region is assigned to the most predominant class among the classes within the same region. The output of this scheme, as shown in Fig. 231.2, is a more accurate hyperspectral classification of the image compared to a standalone spectral classification. The procedure for combining the results is illustrated in detail in Fig. 231.3 for the case of three spectral classes, represented as three colors in Fig. 231.3 (a). The segmentation map, with regions A, B, C, and the results of the MV are displayed in Figs. 231.3 (b) and 231.3 (c).

In the following sections we explain in detail the different steps of the spatial processing. The Robust Color Morphological Gradient, Section 231.3.1, the watershed transform based on CA, Section 231.3.2, and how to combine the results with the spectral ones, Section 231.3.3.

231.3.1 Robust Color Morphological Gradient

The basic morphological gradient operator for grey scale images is defined as Eq. 231.1:

$$\nabla(f) = \delta_g(f) - \varepsilon_g(f) \ , \tag{231.1}$$

where δ_g and ε_g are the dilation and erosion morphological operators, and g the structuring element which defines the neighborhood of a pixel in the image f. In an alternative form, Eq. 231.1 can be expressed as follows in Eq. 231.2:

$$\nabla(f) = \max_{x \in g} \{f(x)\} - \min_{y \in g} \{f(y)\}$$
$$= \max(|f(x) - f(y)|) \quad \forall x, y \in g \ , \tag{231.2}$$

giving the greatest intensity difference between any two pixels within the structuring element. In this way Eq. 231.2 can easily be extended to color images [27], which have a pixel vector of three components, i.e. the red, green and blue channels of color.

Let $\mathbf{x}$ be a pixel vector of a color image and $\chi = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]$ be a set of n pixel vectors in the neighborhood of $\mathbf{x}$, and the set χ contains $\mathbf{x}$. The Color Morphological Gradient (CMG), $\nabla(\mathbf{f})$, using the Euclidean distance, is defined as Eq. 231.3:

$$\nabla(\mathbf{f}) = \max_{i,j \in \chi} \{||\mathbf{x}_i - \mathbf{x}_j||_2\} \ , \tag{231.3}$$

whose response is the maximum of the distances between all pairs of vectors in the set χ. As the CMG is very sensitive to noise and may produce edges that are not representative of the gradient, a RCMG is proposed in [27], based on pairwise pixel rejection of Eq. 231.3. The RCMG, $\nabla(\mathbf{f})_{Robust}$, is defined as Eq. 231.4:

$$\nabla(\mathbf{f})_{Robust} = \max_{i,j \in \chi - R_s} \{||\mathbf{x}_i - \mathbf{x}_j||_2\} \ , \tag{231.4}$$

where R_s is the set of s pairs of pixel vectors removed. The pairs removed are those that are furthest apart. The RCMG is therefore a vectorial gradient operator based on the Euclidean distances of pixel vectors.

A pixel vector also refers to a pixel of the hyperspectral image with all the n-bands as components of a n-dimensional vector. Thus, using the RCMG, a hyperspectral image may be reduced to a single band and be used as input for the watershed transform.

Regarding GPU concerns, the calculation of Eq. 231.4 is split into partial operations and then the partial results are combined to find the maximum distance. There are two possible work distribution strategies among thread blocks which will be explained in Section 231.4.1.1 and Section 231.4.1.2.

231.3.2 Watershed Transform Based on Cellular Automata

Regarding segmentation, a watershed transform based on CA is applied, because of the simplicity of the computing model of the CA that can model complex problems easily, and because the computations for pixels are highly independent, and thus very adequate for streaming parallel processing architectures like the GPU.

The watershed algorithm is a widely used method for non-supervised image segmentation, specially suitable for low-contrast images [14]. If a grey scale image is represented as a topographic relief, where the height of each pixel is directly related to its grey level, the dividing lines of the basins of attraction of rain falling
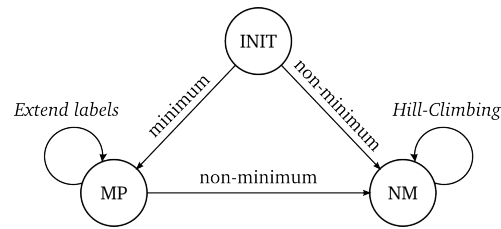
Fig. 231.4 Three-state automaton implementing the Hill-Climbing algorithm [31]

over the regions are called watershed lines [14]. Various definitions, algorithms and implementations can be found in the literature [28]. In this paper the Hill-Climbing algorithm based on the topographical distance by Meyer is adopted [29]. This algorithm starts by detecting and labelling all minima in the image with unique labels. Then, the labels are propagated upwards, *climbing up the hill*, following the path defined by the lower slope of each pixel. At the end, all pixels have a label that identifies the region to which they belong. No implicit lines are generated with this algorithm, so the watershed lines are the limits between these regions.

CA are computing models composed of a set of cells arranged in a regular grid, with each cell connected to its adjacent neighbors. The CA evolve in discrete time steps, according to a collection of states and a set of transition rules. One of the main characteristics of CA is that updates are made for each cell considering only local information, so the concept of parallelism and, in particular, streaming processing, is implicit in the automata. The updates of the cells can be carried out synchronously or asynchronously [30]. In the latter case, the grid can be partitioned into different regions which can be independently updated an unbounded number of times.

Galilée et al. proposed a three-state cellular automaton implementing the Hill-Climbing algorithm [31] that is shown in Fig. 231.4 (MP stands for Minimum or Plateau state and NM for non-minimum state). The main advantage of this Watershed Transform based on Cellular Automata (CA–Watershed) is that minima detection, labeling, and climbing the steepest paths are simultaneously and locally performed.

Each cell of the automaton computes a pixel of the image. First, the pixels are sequentially labelled and the state of each pixel is initialized to one of two possible states. Considering that a plateau is a region of constant grey value within the image, these states are MP and NM. If a pixel is within a plateau, it switches to the MP state. Otherwise, the state of the pixel switches to NM. Figure 231.5 (a) shows an example of a 1D image represented as a terrain (lines) and the corresponding grey values of each pixel (squares). The numbers within each square in this figure are initial label values.

Once the pixels have been initialized, the following steps update the automaton. This is an iterative task that processes the MP and NM states as follows: The pixels of a plateau, i.e. MP state, extend the label with the minimum value along the pixels belonging to that plateau, in case of a plateau that is minimum as indicated in Fig. 231.5 (b), and change their state if the plateau is non-minimum. If the state
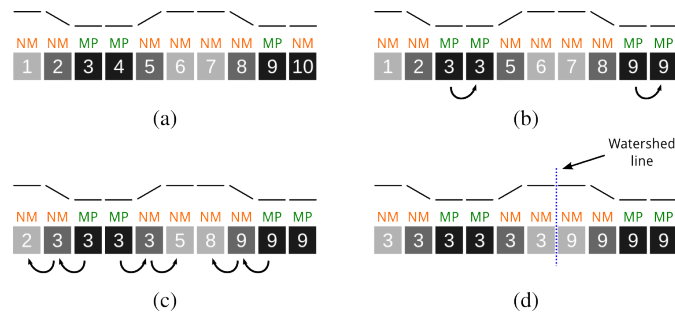
Fig. 231.5 1D image represented as a terrain (*top lines*) and the corresponding grey values of each pixel (*bottom squares*). (a) Init state, (b) MP update, (c) NM update, (d) final segmentation

of the pixel is NM, the label is propagated through the lower slope as shown in Fig. 231.5 (c), where labels 3 and 9 are being propagated upwards, climbing up the hills. This iterative task ends when no more changes occur as in Fig. 231.5 (d). The result is a segmentation map where each region is represented by the label corresponding to the seed pixel that generated the region. The watershed lines can be later defined as the borders among regions.

The CA–Watershed can be synchronously or asynchronously implemented. The asynchronous implementation is non-deterministic and may lead to different segmentation results. A formal proof of correctness and convergence towards a watershed segmentation using a mathematical model of data propagation in a graph is presented in [31].

The asynchronous algorithm is particularly suitable for the CUDA computing model as it was shown by P. Quesada-Barriuso et al. [32]. Different regions of the image can be simultaneously and independently updated during certain number of steps, thus reducing the number of points of synchronization, so the exploitation of parallelism is maximized.

231.3.3 Majority Vote

The MV is a process for determining which out of an arbitrary number of candidates has received most votes, considering a vote as a particular property or attribute. One possible implementation of MV takes as input an array with the votes for each candidate, and returns the element with most votes after one pass over the whole vector [33]. In the hyperspectral classification context, the MV within a fixed window, i.e. fixed neighborhood, is a standard spatial regularization procedure when it is applied after a pixel-wise classification [8]. However, using the regions created by the segmentation process as in this work, i.e. using an adaptive neighborhood, the spatial structures that may be present in the image are taken into account in a more realistic way. So, using an adaptive neighborhood, the MV process integrates the spectral

and the spatial information that are available per pixel within each watershed region, summing up the votes that identify the spectral class for each pixel [34].

From an implementation perspective, in order to combine the results, it is necessary to identify with the same label all the pixels belonging to a region. This may become a challenge when the algorithm is executed on a GPU, because each watershed region can be computed by independent blocks of threads. So, it could be necessary to connect the labels identifying the watershed regions among different blocks.

231.4 Spectral–Spatial Processing in GPU

In this section the GPU projection of the classification process described in Section 231.3 is detailed. The hyperspectral image must be divided into regions that are distributed among the thread blocks. The regions will be one, two or three dimensional depending on the executed stage, enabling all the threads to perform useful work, and therefore exploiting the thousands of threads available in the GPU.

For the RCMG and the CA–Watershed stages, each data region must be extended with a border of size one because the processing of each pixel requires data of its neighbors. As an example, Fig. 231.6 (a) shows an image divided into 4×4 pixel regions assigned to blocks of 4×4 threads, and Fig. 231.6 (b) the extended region for one of the blocks. Threads on the edge of the block must perform extra work loading the data corresponding to the border. In practice, rectangular regions are considered. Using a rectangular block, with the longest dimension being the one along which data is stored in global memory, the data of the border is packed in the minimum number of cache lines. This way the overhead associated to global memory accesses is minimized [38].

Thanks to the pipeline processing, the number of computations and the required bandwidth are reduced in the majority vote stage, because all the pixels in the same
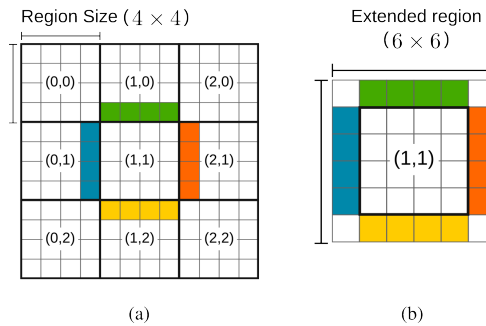

Fig. 231.6 An image divided into 4×4 pixel regions (a) and the extended region for one of the blocks (b)

watershed region are already identified by the same watershed label. So, there is no need to create new data structures and copy them to the GPU memory.

Regarding the spectral processing, different implementations in GPU of SVM are available in the literature [39–43]. Among those that provide the source code performing training and classification, and producing a final classification map, the selected library is the GPUSVM by Catanzaro et al. [39]. This implementation considers the standard two-class soft-margin SVM classification problem. With the use of the CUDA Basic Linear Algebra Subroutines (CUBLAS)[1] to perform the classification, the library takes maximum profit from the latest CUDA releases.

231.4.1 Robust Color Morphological Gradient

The workflow of the RCMG algorithm, summarized in Fig. 231.7 is divided into three steps. First, for all the pixel vectors, threads within a block cooperate to obtain the distances of the set χ, for calculating Eq. 231.3, and computing the CMG. Second, the pair of pixels R_s that are furthest apart, required for Eq. 231.4, are found and the RCMG is calculated with the remaining distances in the third step. So, finally a one-band gradient is obtained. In this work $R_s = 1$, i.e. only one pair of pixels is removed.

The hyperspectral image can be partitioned in the spatial or the spectral domains. From a processing point of view, two different algorithms have been implemented. One based on *spatial partitioning* within a block, as shown in Fig. 231.8 (a) and described in Section 231.4.1.1. Another based on *spectral partitioning* within a block, described in Section 231.4.1.2 and shown in Fig. 231.8 (b). In both cases, the input image is stored in global memory so that consecutive threads access consecutive global memory locations. The intermediate results that are necessary in order to calculate the distances are stored in shared memory.

231.4.1.1 Spatial Partitioning RCMG

In this implementation, each thread processes one spectral component, as shown in Fig. 231.8 (a), and a group of threads cooperate in a reduction operation, where the largest dimension of a thread block indexes the different spectral components. For each region of the image, all the spectral components of each pixel vector are consecutively stored in global memory. The kernel is configured to work in blocks

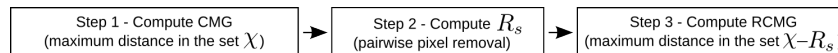| Step 1 - Compute CMG (maximum distance in the set χ) | Step 2 - Compute R_s (pairwise pixel removal) | Step 3 - Compute RCMG (maximum distance in the set $\chi - R_s$) |

Fig. 231.7 RCMG algorithm work-flow

---

[1] See CUBLAS at https://developer.nvidia.com/cublas

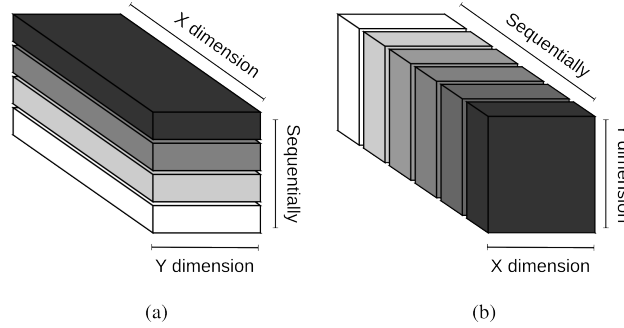(a)   (b)

Fig. 231.8 Kernel configuration for spatial (a) and spectral (b) partitioning

of $x \times y$ threads, corresponding to the X and Y dimensions of Fig. 231.8 (a). For each block, all the threads load different components of each pixel vector simultaneously and compute a partial result $(\mathbf{x}_i)^2 - (\mathbf{x}_j)^2$ of Eq. 231.3. Then, the threads in the X dimension cooperate in a reduction operation [45] for computing the CMG (step 1). Half of the threads work in the reduction, and the number of active threads is halved at each iteration as the reduction proceeds.

One thread in the X dimension finds the pair of pixels that generated the maximum distance (step 2) and computes the RCMG (step 3) with the remaining distances. Finally, the RCMG is written in global memory.

231.4.1.2 Spectral Partitioning RCMG

In the spectral partitioning RCMG, each thread processes all the spectral components of a pixel, as shown in Fig. 231.8 (b). For each region, data are stored in row-major order for each band. The kernel is configured to work in blocks of 32×4 threads corresponding to the X and Y dimensions in Fig. 231.8 (b). Threads within a block process a region of each spectral band in a loop through all the bands (sequential processing). At each iteration, data corresponding to a new band are loaded in shared memory, and the partial results $(\mathbf{x}_i)^2 - (\mathbf{x}_j)^2$ of Eq. 231.3 are computed and stored. At the end of the loop, all the distances for each pixel are available in shared memory.

To compute the CMG (step 1 in Fig. 231.7), each thread finds the maximum of the distances of its set χ and the corresponding pair of pixels which generated that maximum. Having identified the two pixel vectors that are furthest apart (step 2), each thread computes Eq. (231.4) with the remaining distances (step 3) and writes the result back to global memory. This implementation is expected to use less shared memory that the previous one owing to the sequential scanning in the spectral domain. So, more concurrent blocks per SM are also expected.

231.4.2 Watershed Based on CA

The input data to the CA–Watershed algorithm is the 2D gradient image obtained from the RCMG algorithm. The CA–Watershed can be asynchronously implemented as it was mentioned in Section 231.3.2, which is up to four times faster than the CUDA synchronous implementation [32]. In this section the asynchronous algorithm is described, which has the advantage of reusing information within a block, efficiently exploiting the shared and cache memories of the GPU.

The algorithm has two kernels implementing the initialization and updating stages of the CA–Watershed. These kernels are configured to work in blocks of 32×4 threads operating over 32×4 pixel regions of the image. Data structures have been compressed in order to reduce the storage requirements to 8 bytes per pixel as in [32]. With the first kernel, the automaton is initialized. Once all the data have been initialized, they are packed into 8 bytes per pixel before transferring them to global memory.

The updating stage is a hybrid iterative process that includes intra-block updates and inter-block updates. Each region is synchronously updated, i.e. all cells within a region are updated at each time step, while the regions themselves are asynchronously updated (an update of all the blocks is performed at each inter-block step).

During the intra-block updating the values used from outside the block (a border of size one) are kept constant and equal to their values at the beginning of the stage. In the inter-block updating process, data are read at the block borders, which allows the data propagation across the entire grid.

On each call to the CUDA kernel, an inter-block update takes place, where each step is a set of intra-block updates. For each block, once data are loaded in shared memory from an input buffer, the pixels are modified in registers according to their state, and stored back to shared memory in an iterative intra-block process within each region.

The intra-block updating ends when no new modifications are made with the available data within the region. Then the data in shared memory are packed and stored in global memory in an output buffer. This operation is repeated several times in an iterative inter-block process. The algorithm ends when all regions have been flooded and each pixel is labelled with a value indicating the region it belongs to.

The CA–Watershed implementation not only exploits efficiently the resources of the GPU, as the shared memory, but also generates a segmentation map where the pixels are connected. Figure 231.9 (a) shows an example of an image segmented into three regions, represented as "A", "B", "C". The grey lines in each region indicate that after the segmentation process every pixel of each region has the same label, that of the pixel from which the region was created. So, the pixels are connected as shown in Fig. 231.9 (b) without the need of performing any component labelling process [46]. Thus, the output of this algorithm can be used directly in the final stage of the spectral–spatial classification scheme.
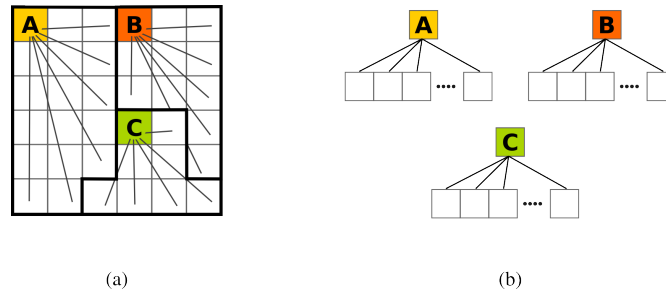
<center>(a)   (b)</center>

Fig. 231.9 An example of an image segmented into three regions, represented as "A", "B", "C" (a), and the connected components created from the labels (b)

231.4.3 Majority Vote

The MV, when applied to this hyperspectral classification, processes the pixels within each segmented region. In this implementation, a region can be assigned to different thread blocks, therefore all the pixels belonging to the same region must be connected, as shown in Fig. 231.9 (b).

By using the segmentation map, such as the one in Fig. 231.9 (a), the pixels of each watershed region are already connected, so the MV can be projected in the GPU following the steps: *voting, winner and updating*. The voting step counts the number of SVM classes for each region. The winner step finds the class with the maximum number of votes per region, and finally, the updating step assigns the winner class to the pixels within the region. Each step is performed by a separate kernel that is configured to work in one dimensional blocks of threads. In the first and third kernel each thread operates on one pixel of the image, while in the second kernel each pixel operates on the information collected for one region of the segmentation map. So, for the second kernel less blocks need to be executed.

One majority vote per watershed region is performed. As the number of these regions is unknown a priori, the first approach would be to allocate in global memory data structures of a large enough size to compute as many regions as pixels in the image. With the aim of saving memory resources, the number of regions generated by the CA–Watershed algorithm are calculated prior to the voting step. Once the number of regions is known, a two dimensional data structure is defined in global memory being the number of watershed regions and the number of spectral classes the dimensions of the structure.

For the voting kernel, each thread adds one vote to the corresponding class, using the label as an index to reference its region. As two or more threads can vote in the same region to the same class with no predictable order, the voting is done by atomic operations. In the winner step, each thread finds the class with the maximum number of votes (winning class) and saves its class identifier in global memory. The last kernel updates the pixels of the classification map with the winning class, producing a new spectral–spatial classification map.

231.5 Results

The algorithms have been evaluated on a PC based in the Nehalem microarchitecture with an Intel quad-core i7-860 microprocessor (8MB Cache, 2.80 GHz) and 8 GB of Double Data Rate type three (DDR3) Synchronous DRAM. The code has been compiled using gcc version 4.4.3 with OpenMP 3.0 support under Linux. For the CUDA implementation we run the algorithms on a NVIDIA Kepler architecture with the GK104 chipset (1536 CUDA cores and 2 GB of Graphics Double Data Rate type five (GDDR5) Synchronous DRAM). The GPU is a GeForce GTX680 with 64 KB of on-chip memory that can be distributed among L1 cache and shared memory and 8 SMs which can execute up to 16 concurrent blocks giving a total maximum of 2048 threads per SM. The CUDA code has been compiled using nvcc and the 4.2 toolkit, also under Linux.

The results are expressed in terms of execution times and speedups. For the SVM spectral classification the speedups are calculated with respect to the LIBSVM [47] that is a sequential library. For the remaining steps of the spectral–spatial classification scheme of Fig. 231.2, the reference codes in CPU are optimized OpenMP parallel implementations considering 4 threads because four cores are available in the Intel Core i7. The tests were run on two hyperspectral airborne images that were obtained from the Basque University (UPV/EHU)[2]: A 103-band ROSIS image from the *University of Pavia* (Pavia) with a spatial dimension of 610×340 pixels, and a 204-band AVIRIS image of 512×217 pixels taken over the *Salinas Valley, California* (Salinas). Although both images are of approximately the same global size, Pavia is larger in the spatial domain while Salinas is larger in the spectral domain.

The final results were compared to the available ground truth of each image. these results are validated using the Overall Accuracy (OA), which is the percentage of correctly classified pixels in the whole image, the Class Accuracy (CS), which is the percentage of correctly classified pixels for a given class, and the Average Accuracy (AA), which is the mean of the CS for all the classes [13]. Tables 231.1 and 231.2 shows the OA, AA, and CS percentages for the SVM and the spectral–spatial classification scheme in CPU and GPU for the images of Pavia and Salinas. RO stands for the ratio between the number of training samples and the number of testing samples for each class. These results are similar to those published in [6–8] when combining spectral and spatial information. Overall, the image of Pavia gives the best results in terms of spectral–spatial classification with an OA improvement of 4.85 points over the SVM. Similar results are obtained in CPU and GPU. The image of Salinas has a very high OA score with the SVM classification and thus, less room for improvement with the spectral–spatial scheme. The improvement for this image is 0.92.

Figures 231.10 and 231.11 show from left to right the SVM classification map, the RCMG results, the segmentation map represented as watershed lines, and the majority vote for the University of Pavia and the Salinas Valley, respectively. The

---

[2] Hyperspectral Remote Sensing Scenes available at `http://www.ehu.es/ccwintco/index.php/Hyperspectral_Remote_Sensing_Scenes`

Table 231.1 Classification accuracy as percentages for the SVM and the whole spectral–spatial scheme in CPU and GPU for the hyperspectral image of Pavia in terms of OA, AA and CS

		SVM CPU	SVM GPU	Spect–Spat. CPU	Spect–Spat. GPU
OA		89.77	89.78	94.55	**94.63**
AA		91.49	91.50	**95.00**	94.97
CS	RO				
Asphalt	0.083	84.80	84.83	**94.74**	94.59
Meadows	0.029	90.37	90.39	94.89	**95.12**
Gravel	0.187	78.75	78.85	86.37	**85.66**
Trees	0.171	**96.57**	**96.57**	93.93	93.90
Metal sheets	0.197	99.55	99.55	**99.63**	**99.63**
Bare Soil	0.106	88.51	88.53	92.86	**93.30**
Bitumen	0.282	95.04	95.04	96.69	**96.84**
Bricks	0.140	89.90	89.90	**95.98**	95.79
Shadows	0.244	**99.89**	**99.89**	**99.89**	**99.89**

number of watershed regions are 22,678 for the first image and 8,423 for latter one. An over-segmented result is observed in Pavia, while the number of regions in Salinas is smaller due to the larger plateaus present in that image

Table 231.2 Classification accuracy as percentages for the SVM and spectral–spatial scheme in CPU and GPU for the hyperspectral image of Salinas in terms of OA, AA and CS

|		SVM CPU	SVM GPU	Spect–Spat. CPU	Spect–Spat. GPU
OA		93.55	93.45	**94.43**	94.37
AA		96.82	96.76	**96.90**	96.89
CS	RO				
Brocoli_green_weeds_1	0.100	99.75	99.75	99.75	**99.80**
Brocoli_green_weeds_2	0.100	99.79	99.76	**100.00**	**100.00**
Fallow	0.100	99.85	99.85	**100.00**	**100.00**
Fallow_rough_plow	0.100	99.71	99.71	**99.78**	**99.78**
Fallow_smooth	0.100	98.77	98.77	**99.14**	**99.14**
Stubble	0.100	99.65	99.65	**99.85**	**99.85**
Celery	0.100	99.62	**99.92**	99.80	99.80
Grapes_untrained	0.100	89.64	89.57	**93.42**	93.12
Soil_vinyard_develop	0.100	99.95	99.81	**99.98**	**99.98**
Corn_senesced_green_weeds	0.100	98.05	97.53	**98.78**	98.72
Lettuce_romaine_4wk	0.100	**98.97**	**98.97**	**98.97**	**98.97**
Lettuce_romaine_5wk	0.100	**99.79**	**99.79**	99.58	**99.79**
Lettuce_romaine_6wk	0.100	**99.67**	**99.67**	95.96	95.74
Lettuce_romaine_7wk	0.100	**95.51**	95.42	94.77	94.77
Vinyard_untrained	0.100	71.05	70.82	**71.68**	71.66
Vinyard_vertical_trellis	0.100	**99.11**	**99.11**	98.95	99.17
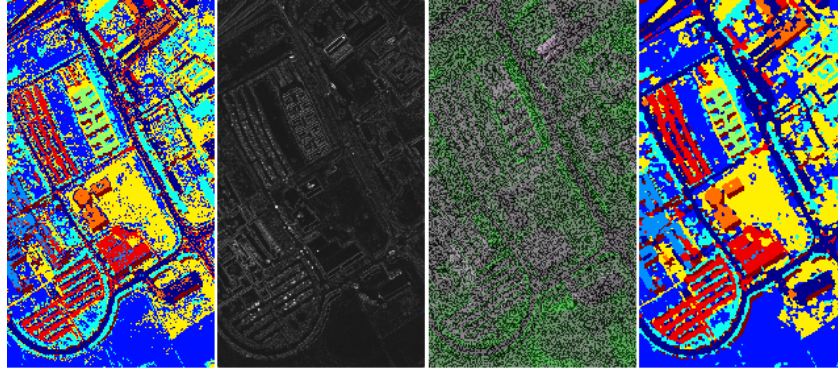
Fig. 231.10 From left to right, the GPUSVM classification map, the RCMG, the CA–Watershed lines imposed over a false color composition to assist in visualizing the segmentation map, and the final classification by majority vote, of the hyperspectral image of Pavia

The performance results are summarized in Table 231.3 for Pavia and Table 231.4 for Salinas. The time to transfer the hyperspectral image from CPU to the GPU global memory at the beginning is included in the spectral stage. The data transfer time for copying the final results back to the CPU is 0.6 milliseconds for Pavia and 0.3 milliseconds for Salinas, resulting in less than $0,003\%$ of the total time. The total times indicate that, even with the high speedups obtained, the times required in GPU are around 17 seconds for Pavia and 59 seconds for Salinas. These values are far from real-time, mainly due to the cost of the spectral classification, that accounts for the 81.2% of this GPU time. So the real-time objective can only be achieved if a less costly spectral technique is applied. Overall, the best results for the whole classification scheme are obtained for Pavia with a speedup of 5.9x. In the next sections we will explain in detail the results for each stage.

231.5.1 SVM Spectral Classification

The standard two-class SVM spectral classification has two phases: training and classification. The training phase builds a model which is used to predict if new

Table 231.3 Performance results for the University of Pavia hyperspectral image (execution times in seconds)

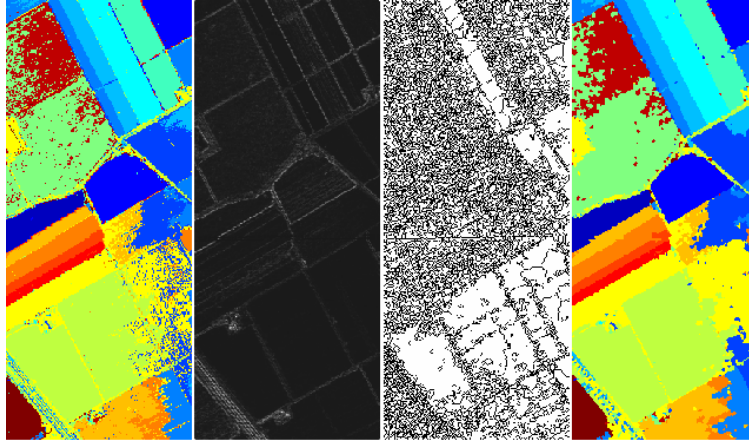|	SVM Training	SVM Classification	Spect. Part. RCMG	Async. CA Watershed	Majority Vote	Total
CPU	0.5760s	101.4484s	0.1517s	0.0186s	0.0022s	102.1969
GPU	3.2466s	14.0497s	0.0085s	0.0010s	0.0003s	17.3067
Speedup	**0.2x**	**7.2x**	**17.8x**	**18.6x**	**7.3x**	**5.9x**

Fig. 231.11 From left to right, the GPUSVM classification map, the RCMG, the CA–Watershed lines and the final classification by majority vote, of the hyperspectral image of Salinas.

samples belong to one category or another in the second phase of classification, which is the most time consuming one as it can be observed in Tables 231.3 and 231.4. The percentage of time corresponding to classification is the same for both images, $81,2\%$ of the time in GPU required for the whole classification process.

When more than two classes are present the classification must be multiclass and different strategies can be applied in order to solve it. Hsu [44] found that the One-Against-One (OAO) method is more suitable for practical use than the One-Against-All (OAA), mainly because the total training time is shorter. In this work GPUSVM with the OAO method is used to classify the hyperspectral images. The kernel function for the SVM is a Gaussian Radial Basis Function (RBF) [26]. The number of classes considered for the classification was taken from the ground truth, with nine classes for the first image and sixteen for the second one.

First, the SVM was trained with the same values of C, γ, and number of training samples for the Pavia image as in [8]: $C = 128, \gamma = 0.125$ and 3192 samples. In the case of the Salinas image, the values $C = 256$ and $\gamma = 0.125$ were determined by 5–fold cross validation, and the number of training samples for each class was selected as 10% of the total samples for the class.

Table 231.4 Performance results for the Salinas Valley hyperspectral image (execution times in seconds)

	SVM Training	SVM Classification	Spect. Part. RCMG	Async. CA Watershed	Majority Vote	Total
CPU	1.5559s	112.2305s	0.1959s	0.0963s	0.0023s	114.0809s
GPU	11.0552s	47.7323s	0.0092s	0.0035s	0.0001s	58.8006s
Speedup	**0.1x**	**2.3x**	**21.3x**	**27.5x**	**23.0x**	**1.9x**

As shown in Tables 231.3 and 231.4, the speedup results are worse for the GPU in the training phase. The SVM requires a small number of training samples in this phase [3] and, therefore, the GPU performance is low because the number of samples is not enough to exploit the big number of threads that can be simultaneously available in the GPU, up to 2048 threads per SM in the GTX680. This is not a problem, as the training phase only needs to be performed once for each type of hyperspectral image and it is responsible for only 18.8% of the total time.

The second phase in the spectral classification, which consumes 81.2% of the time for the Pavia and Salinas images, obtained speedups of 7.2x and 2.3x respectively. The tests are carried out as in [39], excluding the file I/O time for both, the LIBSVM and GPUSVM, but including CPU–GPU data transfer in the GPU implementation times.

231.5.2 Robust Color Morphological Gradient

The RCMG is the vectorial gradient, described in Section 231.3.1, that is applied to the hyperspectral image in order to reduce it to one-band. The approaches described in Section. 231.4.1, called spatial partitioning RCMG and spectral partitioning RCMG have been developed. Different block configurations were tested and finally the spectral partitioning RCMG implementation was configured with blocks of 32×4 threads. For the spatial partitioning RCMG, 128×4 threads per block for the Pavia image and 256×2 threads per block for the Salinas image were considered. Each block in the spatial partitioning RCMG processes a region of 4×4 pixel vectors for the first image and a region of 2×2 pixel vectors for the second one. Table 231.5 shows a summary of performance for the images. The best results are for the spectral partitioning RCMG with speedups of 17.8x and 21.3x. The shared memory requirements for the spectral partitioning RCMG are 5.7 KB per block, while the spatial partitioning RCMG requires up to 20.6 KB, depending on the block size. Thus, more blocks per SM are concurrently executed in the first approach which leads to a better speedup as shown in Table 231.5.

Table 231.5 Performance results for the spatial and spectral partitioning RCMG with the Pavia and the Salinas hyperspectral images.

	Spatial partitioning		Spectral partitioning	
	Pavia	Salinas	Pavia	Salinas
CPU (OpenMP)	0.1517s	0.1959s	0.1517s	0.1959s
GPU (CUDA)	0.0537s	0.0638s	0.0085s	0.0092s
Speedup	**2.8x**	**3.1x**	**17.8x**	**21.3x**

231.5.3 Asynchronous CA–Watershed

As explained in Section 231.4.2, the asynchronous CA–Watershed takes as input the 2D gradient image obtained from the RCMG calculation. This implementation presents the advantage of reusing information within each thread block, efficiently exploiting the shared and cache memories of the GPU. In addition, it achieves better results when the image has large plateaus because in this situation the labels must be propagated through large regions. In the asynchronous implementation the labels are propagated faster within a block, unlike the synchronous implementation which performs more steps to propagate them within a plateau. Thus, less inter-block synchronizations are needed [32].

The kernels were configured to work with blocks of 32×4 threads and the shared memory was maximized to 48 KB, because only 21.4 KB are required for the 16 blocks that are simultaneously active per SM.

This proposal achieves speedups of 18.6x and 27.5x, that can be observed in Tables 231.3 and 231.4. The speedup is better for the Salinas image, as a consequence of presenting larger plateaus.

231.5.4 Majority Vote

The Majority Vote was projected on the GPU taking advantage of the pipeline processing explained in Section 231.4, and reducing the requirements of global memory, which also means less data transfer. The times shown in Tables 231.3 and 231.4 include, as it was described in Section 231.4.3, the step for counting the watershed regions, as well as the global memory allocation time.

The MV obtained speedups of 7.3x and 23.0x for the images of Pavia and Salinas. The difference in the speedups are related mainly to the number of regions because the number of blocks executed in the *winner* step is directly related to the number of watershed regions in the image. The segmentation map of Pavia has $22,678$ regions which is approximately three times more than Salinas, which has $8,423$ regions, that is approximately the speedup factor observed in the performance tables.

The kernels were configured to work in one dimensional blocks. Different block sizes have been tested and it was found that the best performance is achieved for blocks of 128 threads. With this configuration each SM is fully exploited with 16 blocks simultaneously active.

231.6 Conclusions

In this work a GPU projection scheme for a spectral–spatial classification of hyperspectral images was presented. The scheme efficiently exploits the memory hierar-

chy and the thousands of threads available in the GPU architecture. The different stages of the scheme have been concatenated with a pipeline processing that minimizes the data transfers between the CPU and the GPU and maximizes the computational throughput. Different hyperspectral data partitioning strategies and thread block arrangements were studied in order to have a larger number of blocks being concurrently executed. The spectral classification stage was carried out with SVM using the GPUSVM, a third party library. The spatial processing stages consists in the calculation of a RCMG, that reduces the dimensionality of the hyperspectral image to a two dimensional image, followed by the asynchronous calculation of a watershed transform based on cellular automata. The spectral and the spatial results are combined by a MV technique commonly used in classification of hyperspectral images.

The projection of the classification process in the GPU requires working with data blocks of different dimensionality depending on the stage of the process: 3D for RCMG, 2D for watershed and 1D for MV. For the RCMG, two different approximations of data distribution among blocks were studied: spectral and spatial partitioning. The spectral partitioning takes better advantage of the memory hierarchy of the GPU maximizing the number of active blocks per SM. For the watershed transform an asynchronous strategy based on a cellular automaton was proposed. This asynchronous approach has the advantage that it can efficiently exploit the shared memory of the GPU being up to four times faster than a synchronous implementation. Finally, the MV was designed to save global memory space and to directly operate on the output of the other two stages using pipeline processing. This way, there is no need to move new data structures to the GPU.

The speedup values for the whole classification process were 5.9x for Pavia and 1.9x for Salinas showing the efficiency of the GPU projections while maintaining the same classification quality as when it is computed on CPU. The best performance values for the RCMG, 17.8x and 21.3x, were obtained for the spectral partitioning approach, with the images of Pavia and Salinas, respectively. The asynchronous CA–Watershed reached speedups of 18.6x and 27.5x, and the MV speedups of 7.3x and 23.0x, respectively. These results show that the GPU is an adequate computing platform for on-board processing of hyperspectral information.

As the most costly part of the spectral–spatial classification process, and therefore the critical part in terms of real-time execution, was the classification stage by SVM, other spectral classification algorithms more adequate for their efficient projection on GPU should be investigated.

References

[1] Plaza, A., Benediktsson, J. A., Boardman, J. W., Brazile, J., Bruzzone, L., Camps-Valls, G., Chanussot, J., Fauvel, M., Gamba, P., Gualtieri, A., Marconcini, M., Tilton, J. C. and Trianni, G.: Recent advances in techniques for hyperspectral image processing, Remote Sensing of Environment, **113**, S110–S122 (2009)

[2] van der Meer, F.D., van der Werff, H.M., van Ruitenbeek, F.J., Hecker, C.A., Bakker, W.H., Noomen, M.F., van der Meijde, M., Carranza, E.J.M., de Smeth, J.B. and Woldai, T.: Multi- and hyperspectral geologic remote sensing: A review, International Journal of Applied Earth Observation and Geoinformation, **14**(11), pp. 112–128 (2012)

[3] Gualtieri, J. A. and Cromp, R. F.: Support vector machines for hyperspectral remote sensing classification, Proceedings of SPIE, **3584**, pp. 221–232 (1998)

[4] Jia, X., Richards, J. A. and Ricken, D. E.: Remote sensing digital image analysis: an introduction. Berlin, Springer Verlag (1999)

[5] Varshney, P. K., Arora, M. K. (Eds.).: Advanced Image Processing Techniques for Remotely Sensed Hyperspectral Data, Springer Verlag (2004)

[6] Fauvel, M., Chanussot, J., Benediktsson, J. A. and Sveinsson, J. R.: Spectral and spatial classification of hyperspectral data using SVMs and morphological profiles, IEEE Trans. Geosci. Remote Sens. **46**(10), pp. 3804–3814 (2008)

[7] Tarabalka, Y., Benediktsson, J.A. and Chanussot, J.: Spectral-spatial classification of hyperspectral imagery based on partitional clustering techniques, Geosci. and Remote Sensing, IEEE Trans. on **47**(8) pp. 2973–2987 (2009)

[8] Tarabalka, Y., Chanussot, J. and Benediktsson, J.A.: Segmentation and classification of hyperspectral images using watershed transformation. Pattern Recogn. **43**(7) pp. 2367–2379 (2010)

[9] Fauvel, M.: Spectral and spatial methods for the classification of urban remote sensing data, Ph.D. dissertation, Grenoble Inst. Technol., Grenoble, France (2007)

[10] Farag, A.A., Mohamed, R.M. and El-Baz, A.: A unified framework for MAP estimation in remote sensing image segmentation, Geoscience and Remote Sensing, IEEE Transactions on , **43**(7), pp. 1617-1634 (2005)

[11] Couprie, C., Grady, L. Najman, L. and Talbot, H.: Power Watershed: A Unifying Graph-Based Optimization Framework, Pattern Analysis and Machine Intelligence, IEEE Transactions on , **33**(7), pp. 1384–1399 (2011)

[12] Tarabalka, Y., Benediktsson, J.A. and Chanussot, J.: Classification of hyperspectral data using support vector machines and adaptive neighborhoods, In Proc. 6th EARSeL SIG IS Workshop, (2009)

[13] Bernard, K., Tarabalka, Y., Angulo, J., Chanussot, J. and Benediktsson, J. A.: Spectral-Spatial Classification of Hyperspectral Data Based on a Stochastic Minimum Spanning Forest Approach, Image Processing, IEEE Transactions on, **21**(4), pp. 2008–2021 (2012)

[14] Vincent, L. and Soille, P.: Watersheds in digital spaces: An eficient algorithm based on immersion simulations. IEEE Trans. Pattern Anal. Mach. Intell. **13** pp. 583–598 (1991)

[15] Plaza, A., Plaza, J. and Vegas, H.: Improving the Performance of Hyperspectral Image and Signal Processing Algorithms Using Parallel, Distributed and Specialized Hardware-Based Systems, Journal of Signal Processing Systems, **61**(3), pp. 293–315 (2010)

[16] González, C., Sánchez, S., Paz, A., Resano, J., Mozos, D. and Plaza, A.: Use of FPGA or GPU-based architectures for remotely sensed hyperspectral image processing, Integration, the VLSI Journal, **46**(2), pp. 89–103 (2013)

[17] Tarabalka, Y., Haavardsholm, T. V., Kåsen, I. and Skauli, T.: Real-time anomaly detection in hyperspectral images using multivariate normal mixture models and GPU processing, Journal of Real-Time Image Processing, **4**(3), pp. 287–300 (2009)

[18] Heras, D.B., Argüello, F., Gómez, J.L., Becerra, J.A. and Duro, R.J.: Towards Real-time Hyperspectral Image Processing, a GP-GPU Implementation of Target Identification. Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2011 IEEE 6th Int. Conference on. **1** pp. 316–321 (2011)

[19] Priego, B., Souto, D., Bellas, F. and Duro, R. J., Unsupervised Segmentation of Hyperspectral Images through Evolved Cellular Automata, Advances in Knowledge-Based and Intelligent Information and Engineering Systems, **243**, pp. 2160–2169, (2012)

[20] Quesada-Barriuso, P., Argüello, F. and Heras, D.B, Efficient segmentation of hyperspectral images on commodity GPUs, Advances in Knowledge-Based and Intelligent Information and Engineering Systems, **243** pp. 2130–2139, (2012)

[21] Christophe, E., Michel, J. and Inglada, J.: Remote Sensing Processing: From Multicore to GPU, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, **4**(3), pp. 643–652 (2011)

[22] Bernabé, S., Plaza, A., Reddy Marpu, P. and Benediktsson, J.A.: A new parallel tool for classification of remotely sensed imagery, Computers & Geosciences, **46**, pp. 208–218 (2012)

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

[24] Gege, P., Beran, D., Mooshuber, W., Schulz, J. and Van Der Piepen, H.: System analysis and performance of the new version of the imaging spectrometer rosis, in Proceedings of the First EARSeL Workshop on Imaging Spectroscopy. University of Zurich Remote Sensing Laboratories, pp. 29–35 (1998)

[25] Green, R. O., Eastwood, M. L., Sarture, C. M., Chrien, T. G., Aronsson, M., Chippendale, B. J., Faust, J. A., Pavri, B. E., Chovit, C. J., Solis, M. S., Olah, M. R. and Williams, O.: Imaging spectroscopy and the Airborne Visi-

ble/Infrared Imaging Spectrometer (AVIRIS), Remote Sens. Environ., **65**(3), pp. 227–248 (1998)

[26] Camps-Valls, G. and Bruzzone, L.: Kernel-based methods for hyperspectral image classification. IEEE Trans. on Geosci. and Remote Sensing, **43**(6) pp. 1351–1362 (2005)

[27] Evans, A. and Liu, X.: A morphological gradient approach to color edge detection. Image Processing, IEEE Trans. on **15**(6) pp. 1454–1463 (2006)

[28] Roerdink, J. B. T. M. and Meijster, A.: The watershed transform: definitions, algorithms and parallelization strategies, Fundamenta Informaticae, **41**(1–2), pp. 187–228 (2000)

[29] Meyer, F.: Topographic distance and watershed lines. Mathematical Morphology and its Applications to Signal Processing **38**(1) pp. 113–125 (1994)

[30] Nehaniv, C.L.: Evolution in asynchronous cellular automata. In: Proc. of the eighth Int. Conf. on Artificial life, MIT Press pp. 65–73 (2003)

[31] Galilée, B., Mamalet, F., Renaudin, M. and Coulon, P.Y.: Parallel asynchronous watershed algorithm-architecture. IEEE Trans. Paral. Distrib. Syst. **18**(1) pp. 44–56 (2007)

[32] Quesada-Barriuso, P., Heras, D.B. and Argüello, F., Efficient GPU asynchronous implementation of a watershed algorithm based on cellular automata, in Proc. IEEE Int. Symp. on Parallel and Distributed Processing with Applications, pp. 79-86, (2012)

[33] Boyer, R. S. and Moore, J. S.: MJRTY - A Fast Majority Vote Algorithm, in Boyer, R. S. (Eds), Automated Reasoning **1** Automated Reasoning Series, Springer Netherlands, pp. 105–117 (1991)

[34] Santos, A., Araújo, A. and Menotti, D.: Combining Multiple Approaches for Accuracy Improvement in Remote Sensed Hyperspectral Images Classification, Workshop of Thesis and Dissertations - XXV Conference on Graphics, Patterns and Images, pp. 54–59 (2012)

[35] NVIDIA Corporation: NVIDIA CUDA C Programming Guide 4.2, Santa Clara (2011)

[36] NVIDIA Corporation: NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110 Whitepaper (2012)

[37] NVIDIA Corporation: CUDA C Best Practices Guide (2012)

[38] Balasalle, J., López, M. A. and Rutherford, M. J.: Optimizing Memory Access Patterns for Cellular Automata on GPUs, in GPU Computing Gems, Jade Edition, W. W. Hwu, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 67–75 (2012)

[39] Catanzaro, B., Sundaram, N. and Keutzer, K.: Fast support vector machine training and classification on graphics processors, in Proc. of the 25th international conference on Machine learning - ICML'08, pp. 104–111 (2008)

[40] Carpenter, A.: CUSVM: A CUDA implementation of support vector classification and regression, patternsonscreen (2009)

[41] Herrero-López, S., Williams, J. R. and Sanchez, A.: Parallel multiclass classification using SVMs on GPUs, Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, ACM, pp. 2–11 (2010).

[42] Athanasopoulos, A., Dimou, A., Mezaris, V. and Kompatsiaris, I.: GPU acceleration for support vector machines, in Procs. 12th Inter. Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011), Delft, Netherlands (2011)

[43] Li, Q., Salman, R., Test, E., Strack, R. and Kecman, V.: GPUSVM: a comprehensive CUDA based support vector machine package, Central European Journal of Computer Science, **1**(4), pp. 387–405 (2011)

[44] Hsu, C.-W. and Lin, C.-J., A comparison of methods for multiclass support vector machines, IEEE Trans. Neural Netw., **13**(2), pp. 415–425 (2002)

[45] Harris, M.: Optimizing parallel reduction in CUDA. NVIDIA Developer Technology, (2007)

[46] Hawick, K.A., Leist, A. and Playne, D.P.: Parallel graph component labelling with GPUs and CUDA, Parallel Computing **36**(12), pp. 655–678 (2010)

[47] Chang, C. C. and Lin, C. J., LIBSVM: a library for support vector machines. ACM Transactions on Intelligent Systems and Technology (TIST), **2**(3), p. 27 (2011)