J Supercomput (2013) 65:978–996
DOI 10.1007/s11227-013-0875-1

A GPU implementation of a structural-similarity-based aerial-image classification

Rok Češnovar · Vladimir Risojević · Zdenka Babić · Tomaž Dobravec · Patricio Bulić

Published online: 24 January 2013

Abstract There is an increasing need for fast and efficient algorithms for the automatic analysis of remote-sensing images. In this paper we address the implementation of the semantic classification of aerial images with general-purpose graphics-processing units (GPGPUs). We propose the calculation of a local Gabor-based structural texture descriptor and a structural texture similarity metric combined with a nearest-neighbor classifier and image-to-class similarity on CUDA supported graphics-processing units. We first present the algorithm and then describe the GPU implementation and optimization with the CUDA programming model. We then evaluate the results of the algorithm on a dataset of aerial images and present the execution times for the sequential and parallel implementations of the whole algorithm as well as measurements only for the selected steps of the algorithm. We show that the algorithms for the image classification can be effectively implemented on the GPUs. In our case, the presented algorithm is around 39 times faster on the Tesla C1060 unit than on the Core i5 650 CPU, while keeping the same success rate of classification.

Keywords Aerial-image classification · Structural texture similarity · Local images descriptors · GPU · CUDA · Image processing

R. Češnovar (✉) · T. Dobravec · P. Bulić
Faculty of Computer and Information Science, University of Ljubljana, Ljubljana, Slovenia
e-mail: rok.cesnovar@fri.uni-lj.si

V. Risojević · Z. Babić
Faculty of Electrical Engineering, University of Banja Luka, Banja Luka, Bosnia-Herzegovina

V. Risojević
e-mail: vlado@etfbl.net

1 Introduction

One of the most important problems in aerial-image analysis is semantic classification. The ultimate goal of the semantic classification of aerial images is to assign a class from a predefined set, e.g., urban, industry, forest, etc., to each image pixel. Since aerial images are frequently multi-spectral and of high resolution, in order to reduce the computational complexity, this problem is usually approached by dividing the aerial image into tiles, and assigning a class from a predefined set to each tile. Such an obtained classification of image tiles can then be used in content-based image retrieval or for constructing a thematic map, for example.

In recent years an image-similarity measure, which takes into account the properties of the human visual system, has been proposed [16, 17, 22–26]. It consists of three terms. The first two are measures of the similarities between the means and the standard deviations of the pixel values in respective images, and the third one is a *structural term*, which is based on cross-correlations between the images being compared. It is believed that the third term captures the structural information in the images, which is very important for any image-similarity assessment by human subjects. Therefore, this similarity measure is named the *structural similarity measure* (SSIM). This SSIM has shown good results in image-quality assessments.

The main drawback of the proposed algorithms for aerial-image classification is their computational complexity. But such an approach with the usage of structural-similarity metrics has a large amount of inherent parallelism, so it can be effectively implemented on parallel computers such as modern GPUs. Modern GPUs are low-cost and powerful parallel platforms used to accelerate many applications, i.e. audio processing [1], option pricing [5], sparse linear systems [6], medical imaging and simulation [9, 19, 21], bioinformatics [4], scientific simulations [2, 20], and image classification [18].

We focused on an optimized GPU implementation of the algorithm for aerial-image classification proposed in [16], because we found that this classifier advances the state-of-the-art. The optimization is based on the properties of the GPU we used. We present the results of this implementation and the results of the optimization steps. The method was implemented on graphics-processing units with the CUDA architecture and programming model [3, 7, 14, 15].

This paper is organized as follows. In Sect. 2 the image representation and similarity measure are introduced, and a nearest-neighbor classifier is presented. In Sect. 3 we propose the method for parallelizing and optimizing the algorithm explained in Sect. 2. The experimental results are presented in Sect. 4. And, finally, in Sect. 5 we outline the conclusions.

2 Aerial-image classification

In this section we give an overview of the method for aerial-image classification proposed in [16]. A broad class of metrics, the structural similarity metrics (SSIM), that attempt to incorporate structural information into image comparisons was proposed in [22]. They are based on a set of local image statistics. These metrics are computed

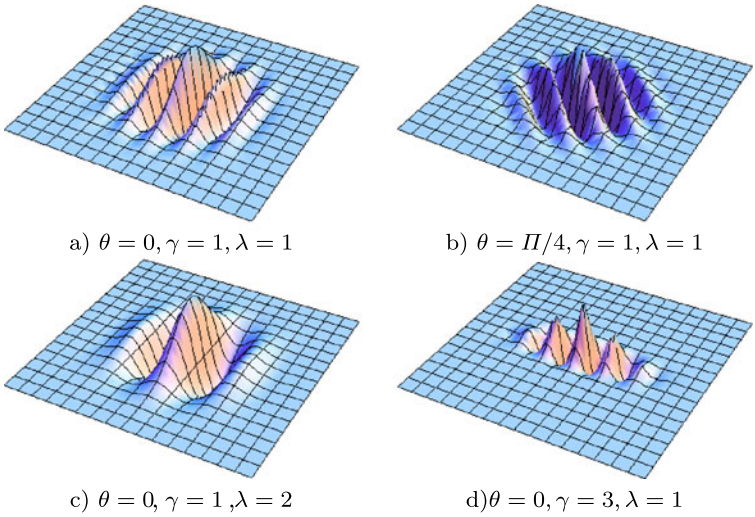

Fig. 1 Real components of the Gabor function with different parameter sets

after the channel decomposition that separates the images into sub-bands that are selective for spatial frequency as well as orientation. As the frequency and orientation representations of the Gabor filters are similar to those of the human visual system, the authors in [16] chose to decompose the images using Gabor filters [8].

2.1 Gabor filters

The impulse response of a Gabor filter in a spatial domain corresponds to the value of the Gabor function, which is defined as a product of the 2D Gaussian-shaped function (the envelope) and a complex sinusoid (the carrier), as follows:

$$g(x, y) = e^{-\frac{x'^2+\gamma^2 y'^2}{2\sigma^2}} \cdot e^{i(2\pi \frac{x'}{\lambda}+\phi)}. \tag{1}$$

Here, vector (x', y') represents a 2D rotation of the original vector (x, y) in the clockwise direction θ, i.e.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix},$$

thus θ defines the orientation of the normal to the parallel stripes of a Gabor function (see Fig. 1a and 1b).

In the Gabor function λ represents the wavelength of the sinusoidal factor (see Fig. 1c), ϕ is the phase offset, σ is the standard deviation of the Gaussian envelope and γ is the spatial aspect ratio and specifies the ellipticity of the support of the Gabor function (see Fig. 1d).

Gabor filters are widely used in image processing (edge and pattern detection), computer vision, neuroscience and psychophysics. A Gabor filter bank usually consists of Gabor filters with various scales and orientations [8]. The filters in a Gabor

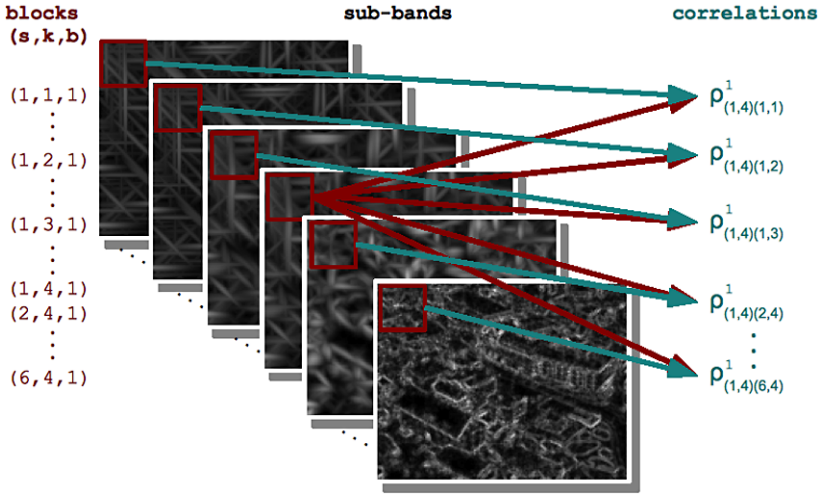

Fig. 2 An example of calculating the correlations for a block $(s, k, b) = (1, 4, 1)$ assuming $S = 4$, $K = 6$

filter bank can be considered as edge detectors with a tunable orientation and scale so that the information on the texture can be derived from the statistics of the outputs of those filters. In the feature-extraction phase of the method used in this paper, the input image is convolved with a Gabor filter bank at S scales and K orientations resulting in SK sub-bands. Each sub-band is partitioned into a grid of $\sqrt{B} \times \sqrt{B}$ blocks, where B represents the total number of image blocks.

For each $(s, k, b) \in \{1, \ldots, S\} \times \{1, \ldots, K\} \times \{1, \ldots, B\}$ let $G_{(s,k)}$ denote the Gabor filter at scale s and orientation k, and let $Y_{(s,k)}$ denote the filter response (i.e., the convolution of the input image and the Gabor filter $G_{(s,k)}$) and $Y^b_{(s,k)}$ its bth block (blocks are enumerated using the row-major order).

For each block $Y^b_{(s,k)}$ the following statistics are computed:

1. the means (i.e., the expected value)

$$\mu^b_{(s,k)} = E\big(Y^b_{(s,k)}\big),$$

2. the standard deviations

$$\sigma^b_{(s,k)} = E\big(\big(Y^b_{(s,k)} - \mu^b_{(s,k)}\big)^2\big),$$

3. the Pearson cross-correlation (Fig. 2) with other sub-bands (s_1, k_1) where (a) $s_1 = s, k_1 = 1, \ldots, K$, $k_1 \neq k$ (the same scale, different orientations) and (b) $k_1 = k$, $s_1 = 1, \ldots, S$, $s_1 \neq s$ (the same orientation, different scales),

$$\begin{aligned}\rho^b_{(s,k)(s_1,k_1)} &= corr\big(Y^b_{(s,k)}, Y^b_{(s_1,k_1)}\big) \\ &= \frac{E((Y^b_{(s,k)} - \mu^b_{(s,k)})(Y^b_{(s_1,k_1)} - \mu^b_{(s_1,k_1)}))}{\sigma^b_{(s,k)}\sigma^b_{(s_1,k_1)}}.\end{aligned}$$

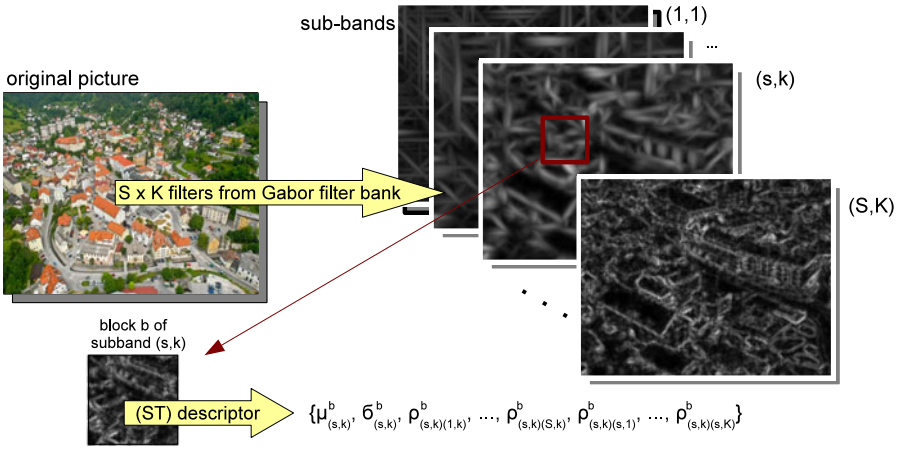

Fig. 3 Each block b of a sub-band (s, k) is associated with an (ST) descriptor set with $S + K$ coefficients

In the above equations, $E(Y)$ denotes the expected value or mean of all values of the random variable Y. Each block b is now described with $(S + K)SK$ coefficients stored in a structural texture (ST) descriptor set (Fig. 3)

$$ST = \left\{\mu_p^b;\, p \in \mathcal{S} \times \mathcal{K}\right\} \cup \left\{\sigma_p^b;\, p \in \mathcal{S} \times \mathcal{K}\right\} \cup \left\{\rho_{p_1 p_2}^b;\, p_1, p_2 \in \mathcal{S} \times \mathcal{K}, p_1 \sim p_2\right\},$$

where $\mathcal{S} = \{1, \ldots, S\}$, $\mathcal{K} = \{1, \ldots, K\}$ and the relation $\sim$ requests equality in exactly one pair component (i.e., $(s_1, k_1) \sim (s_2, k_2) \iff s_1 = s_2 \oplus k_1 = k_2$; here $\oplus$ works like an *exclusive or* operator, meaning that autocorrelations $\rho_{(s,k)(s,k)}^b$ are not included in ST).

2.2 Similarity metrics

Wang et al. [22] proposed comparing two images by looking at their luminance, contrast and structure. For the image blocks b_1 and b_2 and for any $p \in \mathcal{S} \times \mathcal{K}$ they defined the luminance comparison $l_p(b_1, b_2)$ as the similarity of the mean values:

$$l_p(b_1, b_2) = \frac{2\mu_p^{b_1}\mu_p^{b_2}}{(\mu_p^{b_1})^2 + (\mu_p^{b_2})^2}. \tag{2}$$

Similarly, they proposed the contrast-comparison function $c_p(b_1, b_2)$ as the similarity of the standard deviations:

$$c_p(b_1, b_2) = \frac{2\sigma_p^{b_1}\sigma_p^{b_2}}{(\sigma_p^{b_1})^2 + (\sigma_p^{b_2})^2}. \tag{3}$$

The comparison of the image structures is performed by averaging the similarities of all the cross-correlations [26]:

$$r_p(b_1, b_2) = \frac{1}{S + K - 2} \sum_{p_1 \sim p} \left(1 - 0.5 \left| \rho_{pp_1}^{b_1} - \rho_{pp_1}^{b_2} \right| \right). \tag{4}$$

To compute the similarity between two images, the algorithm from [16] uses the similarity metrics already proposed in [22], i.e., the following similarity metrics between two signals x and y:

$$Q(x, y) = l^{\alpha}(x, y) c^{\beta}(x, y) r^{\gamma}(x, y) \tag{5}$$

where $\alpha > 0$, $\beta > 0$ and $\gamma > 0$ are the parameters used to adjust the relative importance of the three components. The authors in [16] compute the similarity between two blocks, b_1 and b_2, by averaging their similarities in all sub-bands:

$$Q(b_1, b_2) = \frac{1}{SK} \sum_{p \in \mathcal{S} \times \mathcal{K}} l_p^{\frac{1}{3}}(b_1, b_2) c_p^{\frac{1}{3}}(b_1, b_2) r_p^{\frac{1}{3}}(b_1, b_2). \tag{6}$$

2.3 Image-classification based on ST-descriptors

The problem of image classification consists of assigning a test image I to a class from the predefined set $\{C_1, C_2, \ldots, C_m\}$. The image I is classified to the class for which the similarity measure is maximal (nearest neighbor). Each image is partitioned into B blocks, i.e., $\mathrm{I} = \{b_1, b_2, \ldots, b_B\}$. The block-to-class similarity is defined as

$$Q(b, C) = \max_{c \in C} Q(b, c). \tag{7}$$

The similarity of the image $\mathrm{I} = \{b_1, b_2, \ldots, b_B\}$ to the class C is based on the block-to-class similarity and is defined as

$$Q(\mathrm{I}, C) = \prod_{i=1}^{B} Q(b_i, C). \tag{8}$$

The classification of the image $\mathrm{I} = \{b_1, b_2, \ldots, b_B\}$ is performed using the algorithm from [16] in the following way:

1. Calculate the ST descriptor sets for all the blocks $b_1, b_2, \ldots, b_B$.
2. For each block b_i, $i = 1, \ldots, B$, and for each class C_j, $j = 1, \ldots, m$, compute the block-to-class similarity $Q(b_i, C_j)$.
3. Compute the image-to-class similarity $Q(\mathrm{I}, C_j)$ between the image I and each class C_j, $j = 1, \ldots, m$.
4. Assign the image I to the class C for which the maximum similarity has been obtained.

3 GPU Implementation

In this section we present the optimized GPU implementation of the algorithm explained in Sect. 2 and previously proposed in [16]. The presented implementation follows the recommendations in [11].

3.1 Calculation of the ST descriptors

As explained in Sect. 2, the first step in image classification is to calculate the ST descriptors for an image I. This calculation is performed by filtering the image with the Gabor filters from the filter bank. The filtering is done in frequency space to reduce the required number of arithmetic operations. The ST descriptors are calculated for all blocks $b_1, b_2, \ldots, b_B$ in an input image I.

The input data needed to compute the ST descriptors are:

1. the set of images that need to be classified, $\mathcal{I}$
2. the Fourier transforms of the Gabor filters, $\mathcal{FFT}(G(s,k))$, which are calculated only once and stored in the filter bank.

This computation of the ST descriptors is implemented in five steps:

1. Calculate the discrete FFT for all the images to be classified, $\mathcal{FFT}(I)$, for all $I \in \mathcal{I}$,
2. Calculate the element-wise products $\mathcal{FFT}(I) \times \mathcal{FFT}(G(s,k))$ for all $I \in \mathcal{I}$ and for all sub-bands $(s,k) \in \{1,\ldots,S\} \times \{1,\ldots,K\}$
3. Calculate the inverse FFT of each product, i.e. $Y(s,k) = \mathcal{IFFT}[\mathcal{FFT}(I) \times \mathcal{FFT}(G(s,k))]$
4. For each $(s,k,b) \in \{1,\ldots,S\} \times \{1,\ldots,K\} \times \{1,\ldots,B\}$, calculate the means $\mu^b_{(s,k)}$ and the standard deviations $\sigma^b_{(s,k)}$ for a block $Y^b_{(s,k)}$.
5. Calculate the cross-correlations $\rho^b_{(s,k)(s_1,k_1)}$ with the other sub-bands. Note that $\rho^b_{(s_1,k_1)(s,k)} = \rho^b_{(s,k)(s_1,k_1)}$, thus we need to calculate only half of the cross-correlations.

For each step we implement a CUDA kernel. Each CUDA kernel and associated thread/block organization are explained in the following subsections.

3.1.1 Computation of discrete FFTs

To compute the discrete FFT for each image $I \in \mathcal{FFT}(I)$ we use the CUFFT Library [12]. In order to speed-up the computation of the discrete FFT, we applied batch execution. The batch execution is used to find the discrete FFT of multiple images in parallel in a single call. This is much more efficient than simply calling the FFT over and over in a loop since some of the intermediate twiddle factors can be reused. The batch input parameter in this step is the number of images N_I in $\mathcal{FFT}(I)$.

3.1.2 Computation of the filter responses in the frequency domain

Each image I is now filtered with a Gabor filter bank, resulting in $S \times K$ sub-bands. Each image has a size of $M \times M$, where M depends on the dataset and is 256 in our

case. Each thread now computes one product between the pixel in the input image I and the pixel at the same position in the Gabor filter. The input for this step is the N_I transforms of the input images and the $S \times K$ transforms of the Gabor filters. The thread organization in this step is as follows:

1. the number of threads in a block is 16×16,
2. the number of blocks in a grid is

$$\left[M^2/(16 \times 16)\right] \times \left[(S \times K) \times N_I\right].$$

In such a way for each sub-band we use $[M^2/(16 \times 16)]$ blocks of threads to compute the filter response of an input image.

3.1.3 Computation of inverse transforms

Now we have to compute the discrete inverse FFT for each filter response in the frequency domain. The input for this step is composed of $(S \times K) \times N_I$ filter responses in the frequency domains, i.e., $(S \times K)$ sub-bands for each input image. Again, we apply the batch execution available in the CUFFT library, where the batch parameter is equal to the number of filter responses, i.e., $(S \times K) \times N_I$.

3.1.4 Computation of mean values and standard deviations

In this kernel we compute the mean values and the standard deviations for each of the blocks $b_1, b_2, \ldots, b_B$ in an input image I. After filtering we have $(S \times K)$ sub-bands for each input image, and thus the input for this step is composed of $(S \times K) \times N_I$ filter responses. Each filter response is partitioned into a grid of $\sqrt{B} \times \sqrt{B}$ blocks, where B represents the total number of blocks in one filter response. In the current implementation of the proposed algorithm, the total number of blocks B is 16.

One thread is used to calculate the $\mu^b_{(s,k)}$ and $\sigma^b_{(s,k)}$ for each block $G^b_{(s,k)}$. Thus we need to create $(S \times K) \times N_I \times B$ threads that are organized as follows:

1. The number of threads in a block is $\sqrt{B} \times \sqrt{B} \times 16$, i.e., each block of threads works on 16 images in parallel. In such a way we keep each streaming multiprocessor busy.
2. The number of blocks in a grid is

$$\left[(S \times K)\right] \times [N_I/16]$$

The reader may notice that the number of blocks in the grid increases with the number of images N_I. In the case of a very large number of images to be classified (more than 2100) we should fix the number of blocks in the grid and implement the threads in such a way that each thread calculates the mean and the standard deviations for more than one block. However, for the image dataset used in this paper this is not the case.

Algorithm 1 The pseudo-code of the kernel for the cross-correlation computation

s = blockIdx.x/K; k = blockIdx.x%K;
b = threadIdx.x*$\sqrt{B}$ + threadIdx.y;
I = blockIdx.y;
$s_1 = k_1$ = threadIDx.z;

if s != s_1 **then**
    $\rho^b_{(s,k)(s_1,k)}(I) = \frac{E((Y^b_{(s,k)} - \mu^b_{(s,k)})(Y^b_{(s_1,k)} - \mu^b_{(s_1,k)}))}{\sigma^b_{(s,k)}\sigma^b_{(s_1,k)}}$
end if
if k != k_1 **then**
    $\rho^b_{(s,k)(s,k_1)}(I) = \frac{E((Y^b_{(s,k)} - \mu^b_{(s,k)})(Y^b_{(s,k_1)} - \mu^b_{(s,k_1)}))}{\sigma^b_{(s,k)}\sigma^b_{(s,k_1)}}$
end if

3.1.5 Computation of cross-correlations with other sub-bands

In this kernel we compute cross-correlation with the other sub-bands. The input for this kernel is composed of $(S \times K) \times N_I$ filter responses. Each filter response is partitioned into the grid of $\sqrt{B} \times \sqrt{B}$ blocks, where B represents the total number of blocks in one filter response. In the current implementation of the proposed algorithm, the total number of blocks B is 16.

One thread is used to calculate $\rho^b_{(s,k)(s_1,k_1)}$ between two blocks $Y^b_{(s,k)}$ and $Y^b_{(s_1,k)}$, i.e., a sub-band with the same orientation, and between two blocks $Y^b_{(s,k)}$ and $Y^b_{(s,k_1)}$, i.e., a sub-band with the same scale. In this kernel we launch $(S \times K) \times N_I \times B \times \max(S, K)$ threads that are organized as follows:

1. the number of threads in a block is $\sqrt{B} \times \sqrt{B} \times \max(S, K)$, i.e. each block of threads works on $\max(S, K)$ orientations or scales in parallel. In such a way we keep each streaming multiprocessor busy.
2. the number of blocks in a grid is

$$\left[(S \times K)\right] \times [N_I]$$

Algorithm 1 contains the pseudo-code for the computation of the cross-correlation coefficients. $G[s][k][b]$ represents the bth block of the filter response, while $ST[s][k][b]$ represents the descriptor of the bth block, both at the scale s and the orientation k. The first coefficient of the descriptor is the mean, followed by the standard deviation and the cross-correlation coefficients, as described in Sect. 2.

3.2 Classification

We propose two different ways to parallelize the classification method, depending on the number of blocks (B) in an image I. We will refer to them as the *full-image parallel* and the *block-wise parallel* classification. The pseudo-code of the sequential classification is shown in Algorithm 2 and should help the reader to understand the proposed classification algorithm.

Algorithm 2 The pseudo code of the sequential classification

```
1:  for c = 1 to m do                                   ▷ m is the number of classes
2:      for i = 1 to B do
3:          for j = 1 to m_c do          ▷ m_c is the number of labeled images in class c
4:              for k = 1 to B do
5:                  q(i, k_j^c) =block similarity(I(i), labeled(c, j, k))
6:                                  ▷ k_j^c is the kth block of the jth labeled image from class c
7:              end for
8:          end for
9:          Q(i, c) = max ( q(i, :) )
10:     end for
11:     Q(I, c) = product ( Q(:, c) )
12: end for
13: C = index of max(Q)
14: return C
```

When to use either of them depends on the compute capability(CC) of the GPU [13]. If $(CC = 1.X \wedge B \leq 16) \vee (CC = 2.X \wedge B \leq 25)$ the use of full-image parallel computation is recommended, due to a significant improvement in the speed (see Sect. 4). Otherwise, the block-wise parallel classification should be used, due to restrictions on the number of threads in a block (for details see Sect. 3.4).

For the purpose of parallelization, we divided the classification of the test image into three steps:

1. the computation of the similarity metrics $Q(b_1, b_2)$ between two blocks b_1 and b_2 using Eq. (6),
2. the computation of the block-to-class similarities $Q(b, C)$ using Eq. (7) and
3. the computation of the image-to-class similarities $Q(\mathrm{I}, C)$ using Eq. (8).

Each of the above steps is implemented with a different kernel and thread organization. Before the parallelized classification, we need to transfer all the precomputed ST-descriptors for the labeled images in the device's global memory. If we only parallelize the classification step, we also need to transfer the ST-descriptors for the images that are not yet classified; otherwise they are already present in the global memory of the device.

The input data to the above steps are the ST descriptors of the images that should be classified (test images) and the ST descriptors of the labeled images. Let us suppose that we have N_I test images and N_L labeled images.

3.3 Full-image parallel classification

3.3.1 Computation of similarity metrics

In the computation of similarity metrics a single thread computes one similarity metric $Q(b_i^I, b_j^L)$ between the block i from the test image I and the block j from the labeled image L. The threads are organized as follows:

1. the number of threads in a block is $(\sqrt{B} \times \sqrt{B}) \times (\sqrt{B} \times \sqrt{B})$, i.e., one thread block computes the similarity metrics between all the blocks from a test image and all the blocks from a labeled image.
2. the number of blocks in a grid is $(N_I \times N_L)$.

The threads in a block work as follows: all the threads with the same index on the x-axes (index tid_x) compute the similarity metrics $Q(b^I_{tid_x}, b^L_j)$, $j = 1, \ldots, B$, while all the threads with the same index on the y-axes (index tid_y) compute the similarity metrics $Q(b^I_i, b^L_{tid_y})$, $i = 1, \ldots, B$.

The result of this step (kernel) is an array of $N_I \times N_L \times B^2$ similarity metrics.

3.3.2 Computation of block-to-class similarities

After computing the similarity metrics, we run the second kernel where a single thread computes one block-to-class similarity $Q(b^I_i, C_j)$ between the block i from the test image I and the class j. The threads are organized as follows:

1. the number of threads in a block is $(\sqrt{B} \times \sqrt{B}) \times 16$, i.e., one thread block computes the block-to-class similarity metrics between all the blocks from 16 test images and one class;
2. the number of blocks in a grid is $(N_I/16) \times m$, where m is the number of classes.

The result of this step (kernel) is an array of $N_I \times m \times B^2$ block-to-class similarity metrics.

3.3.3 Computation of image-to-class similarities

With this last kernel each thread in a block computes the image-to-class similarity $Q(I_i, C_j)$ between the test image i and the class j. The threads are organized as follows:

1. the number of threads in a block is m (where m is the number of classes), i.e. one thread block computes the image-to-class similarity metrics between one image and all the classes;
2. the number of blocks in a grid is N_I.

The number of threads in a block is relatively small; therefore, the streaming multiprocessors are not optimally loaded. Nevertheless, the computation on the GPU is still faster than the computation on the host CPU (that would also involve the data transfer to the host memory).

3.3.4 Shared memory

In the first step all the threads in a block calculate the similarity metrics $Q(b^I_i, b^L_j)$ between two images that are described with two ST descriptors. To improve the performance, we would like to keep both descriptors in the shared memory. Each descriptor has a size of $B \times S \times K \times (S + K)$. In practice, B is usually 16, K is 6 and S is 4. This leads to a descriptor with a size of 3840 floats (15360 bytes). As in GPUs with $CC = 1.X$, the shared memory has a size of 16 kB, we can hold only one

descriptor at a time in the shared memory if $B > 8$, $K = 6$ and $S = 4$. In GPUs with $CC = 2.X$ we can have both descriptors in the shared memory when $B < 25$, $K = 6$ and $S = 4$.

The impact of using the shared memory is discussed and presented in Sect. 4.

3.4 Block-wise parallel classification

In the full-image parallel classification (Sect. 3.3), the first step required $B \times B$ threads in a block. NVIDIA's specifications state that the maximum number of threads per block is 512 ($CC = 1.X$) or 1024 ($CC = 2.X$). This means that we can only divide an image into up to 16 blocks for $CC = 1.X$ or up to 25 blocks for $CC = 2.X$. In the cases when the number of blocks is larger than 16, we propose the block-wise parallel classification, as follows.

3.4.1 Computation of similarity metrics

In the computation of similarity metrics a single thread computes one similarity metrics $Q(b_i^I, b_j^L)$ between the block i from the test image I and the block j from the labeled image L. The threads are organized as follows:

1. The number of threads in a block is now reduced and fixed, i.e. the number of the threads in a block is $(\sqrt{B} \times \sqrt{B}) \times (\lfloor 256/B \rfloor)$, i.e., one thread block computes the similarity metrics between $(\lfloor 256/B \rfloor)$ blocks from a test image and all the blocks from a labeled image.
2. The number of blocks in a grid is $[(N_I/(\lfloor 256/B \rfloor)] \times [N_L \times B])$.

3.4.2 Computation of block-to-class similarities

After computing the similarity metrics, we run the second kernel where a single thread computes one block-to-class similarity $Q(b_i^I, C_j)$ between the block i from the test image I and the class j. Again, we have to reduce the number of threads in a block. The threads are now organized as follows:

1. The number of threads in a block is $(\sqrt{B} \times \sqrt{B}) \times (\lfloor 256/B \rfloor)$, i.e., one thread block computes the block-to-class similarity metrics between all the blocks from $(\lfloor 256/B \rfloor)$ test images and one class.
2. The number of blocks in a grid is $(N_I/(\lfloor 256/B \rfloor)) \times m$, where m is the number of classes.

3.4.3 Computation of image-to-class similarities

Here, the number of threads in a block and the thread organization is the same as in the full-image parallel classification, because the number of classes m is relatively small.

4 Experimental results

For the evaluation of the classifier we used the UC Merced Land Use Dataset, which is publicly available at http://vision.ucmerced.edu/datasets/landuse.html. This dataset has recently been used in similar experiments [24].

The UC Merced Land Use Dataset consists of aerial images of 21 land-use classes. All the images are 256 × 256 pixels and in RGB colorspace. They are manually classified into the following 21 classes: agricultural, air-plane, baseball diamond, beach, buildings, chaparral, dense residential, forest, freeway, golf course, harbor, intersection, medium density residential, mobile home park, overpass, parking lot, river, runway, sparse residential, storage tanks, and tennis courts. Each class contains 100 images, which makes this dataset the largest publicly available dataset for remote sensed image classification.

We filter each image using a Gabor filter bank at four scales and six orientations, and compute ST-descriptors for sub-band blocks on 1 × 1 (global sub-band coefficients statistics), 2 × 2, and 4 × 4 grids. We use 80 % of the images from each class as labeled images, and the rest as test images. We repeated the experiment five times with different random splits of the dataset, and averaged the results. The results of the aerial-image classification accuracy can be found in [16].

4.1 Comparison of CPU and GPU implementations

The experiments for the sequential implementations were performed using the HP Compaq 8100 Elite CMT PC (Intel(R) Core(TM) i5 650 CPU, that operates at 3.2 GHz, 4 GB DDR3 RAM that operates at 1.33 GHz, 128 kB L1, 512 kB L2, 4 MB L3). The experiments for the GPU implementation were performed using the NVIDIA Tesla C1060 Computing Processor with 240 processor cores and 4 GB GDDR3 with 102 GB/s peak bandwidth per GPU [10]. The NVIDIA Tesla C1060 is installed in the same HP Compaq 8100 Elite CMT PC.

The measurements were performed for the computation of the ST descriptors and the classification individually, as well as both steps together.

First, we present the execution times for the computation of the ST descriptors and the classification, separately. Then we present the execution times for the whole algorithm, i.e., the ST descriptors' computation plus classification.

4.1.1 Computation of the ST descriptors

The computation of the ST descriptors is required for each test image and for each labeled image. The former computation is performed only once and the ST descriptors of the labeled images are stored in the memory. Figure 4 presents the execution time for the ST descriptors' computation. In every run of the algorithm proposed in this paper, the computation of the ST descriptors for all the test images is required. With the CPU implementation, the computation of the ST descriptors for a single test image takes 130 ms and 51 ms with the GPU implementation, so the speed-up factor is around 2.54. The computation of the ST descriptors for 500 test images on the CPU takes 65.71 seconds and 25.57 on the GPU, thus the speed-up factor is around 2.56.

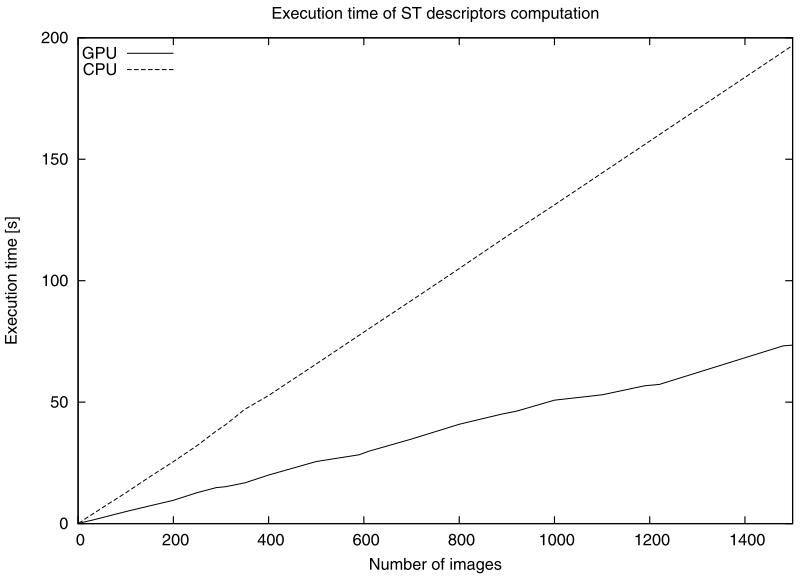

Fig. 4 Execution times for the ST descriptors' computation

As each image has 256×256 pixels, up to around 300 images can reside in the global memory of the device. If there are more test images, the computation is split into two or more equal steps. This is the reason for a steeper rise of the execution time when the number of images is a factor of 300. We can see this in Fig. 4.

4.1.2 Classification

In Fig. 5 we can see the execution times for the classification step on the GPU for the whole UC Merced Land Use Dataset containing 2100 images. We can see that the use of the shared memory reduces the execution time. When classifying one test image, the shared memory reduces the execution time by around 17 %. When classifying more images, the impact of the shared memory becomes less significant, due to longer execution times for the other parts of the classification.

When performed on the whole dataset, the CPU implementation takes 4.36 s to classify one image, while the GPU full-image parallel classification takes 63 ms. When classifying 500 images, the classification takes 36 minutes on the CPU, while the GPU implementation takes 31 seconds.

Figure 6 shows the speed-up factors for the GPU-based classification of the whole dataset. The maximum speed-up factor is approximately 69 for the full-image parallel classification when using the shared memory and approximately 67 without using the shared memory. This maximum speed-up is reached when we classify four test images. As stated in Sect. 3.4, when $B \geq 25$, we have to use the block-wise parallel classification. The speed-up factor for the block-wise parallel classification is around 62 and is presented in Fig. 6.

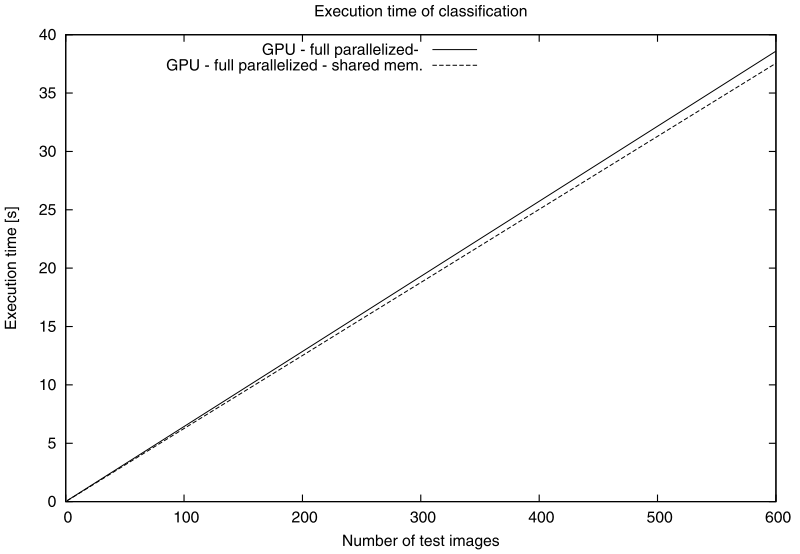

Fig. 5 Execution times of the classification procedure for the whole UC Merced Land Use Dataset

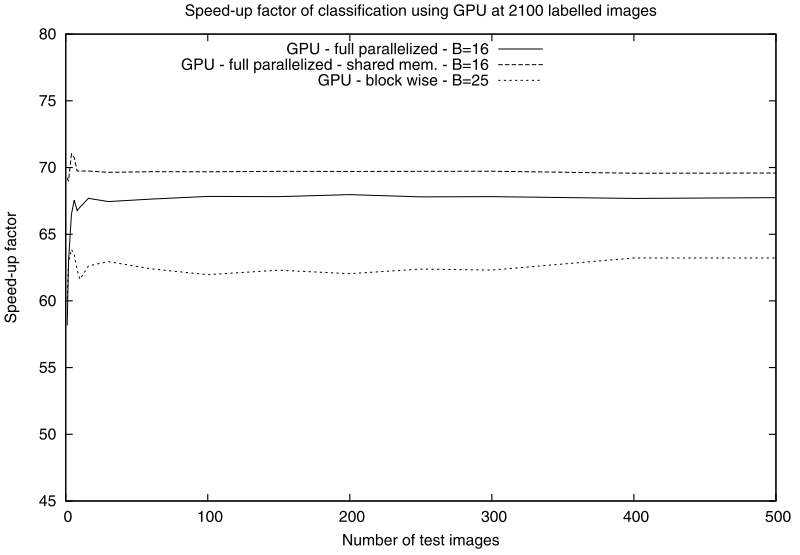

Fig. 6 Speed-up factor for the GPU-based classification for the whole UC Merced Land Use Dataset

In the previous figures we presented the execution times and speed-up factors when the number of labeled images is constant. Figure 7 presents the speed-up factor when we classify only one test image against various numbers of labeled images. The maximum speed-up factor for the classification of one test image on the GPU using the shared memory is around 70. Even with very small sets of labeled images

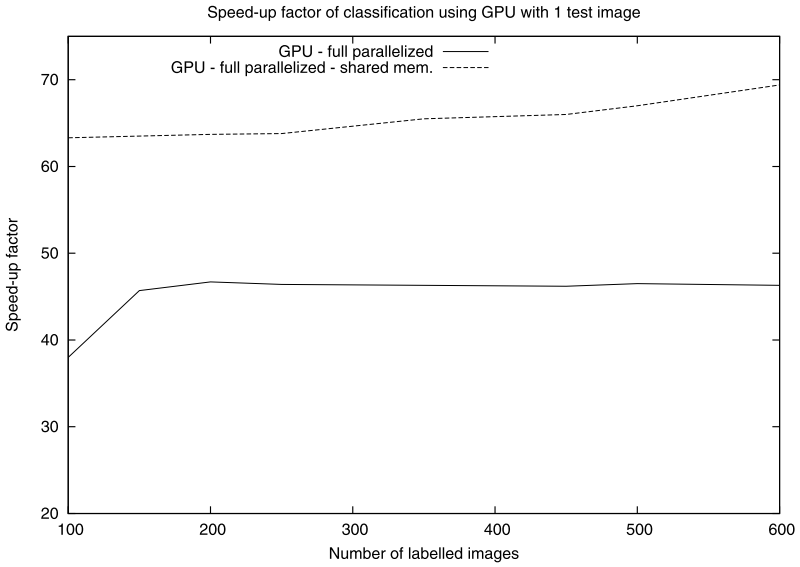

Fig. 7 Speed-up factor for the GPU-based classification of one test image

($N_L < 10$) the speed-up factor is around 30. Without the shared memory the maximum speed-up factor is 46, when $N_L = 606$.

4.2 Computation of the ST descriptors and classification

The last measurements were performed for the whole algorithm (the computation of the ST descriptors plus the classification). The implementation of the algorithm includes the transfer of the input images to the device, the transfer of the precomputed ST descriptors of the labeled images to the device and the transfer of the image-to-class similarities back to the host machine.

Figure 8 presents the execution times of the whole aerial-image classification algorithm. We can see that, even though there is an additional overhead in the parallel implementation, the GPU implementation is still faster. This is also true even for the classification of a single image.

The larger part of the execution time is consumed by the classification; with 500 test images the ratio between the classification time and the ST descriptor computation is around 1.22:1 on the GPU and 33:1 on the CPU.

For the whole UC Merced Land Use Dataset the CPU takes 4.49 s to classify an image, while the GPU performed in 114 ms, resulting in a speed-up factor of 39. The speed-up factor for various numbers of test images is presented in Fig. 9. Again, the speed-up factor reaches its maximum at around four test images.

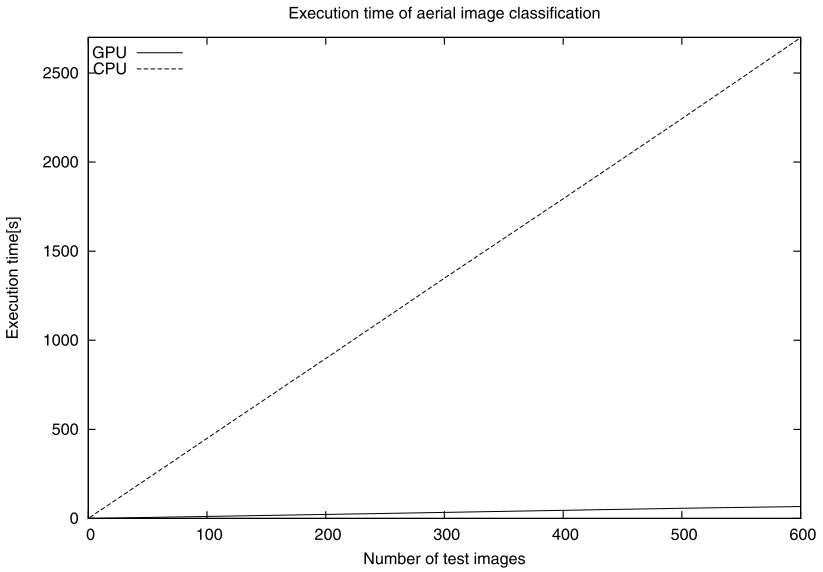

Fig. 8 Execution time for the algorithm with the full-image parallel classification

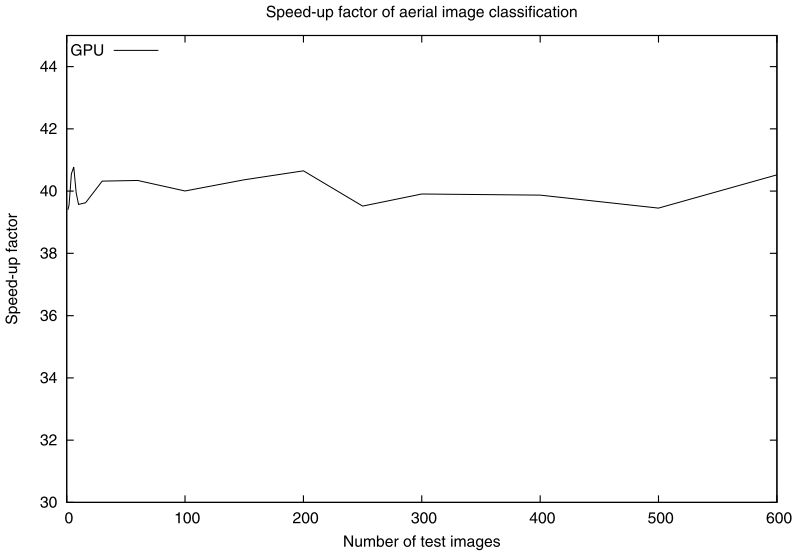

Fig. 9 Speed-up factor for the GPU implementation of the algorithm with the full-image parallel classification

5 Conclusions

In this paper we proposed the GPU implementation of a nearest-neighbor classifier using local Gabor-based structural texture descriptors and structural texture similarity

for the semantic classification of aerial images. We tested the performance of the GPU-implemented classifier on a real dataset of aerial images.

The main drawback of such a classifier is its computational complexity, which could be overcome with a GPU implementation due to a large amount of parallelism inherent to the proposed classifier. We showed that it benefits from the use of the parallel implementation, even for very small datasets and the very small number of images that are classified.

The results show that for the given dataset with images of the size 256×256 pixels, the parallel computation of the ST descriptors alone is around 2.54 times faster than the sequential implementation. Meanwhile, the parallel implementation of the classification alone is even more than 69 times faster than the sequential classification. If we parallelize both steps in the classification we can see that for the whole dataset the speed-up is around 39. Our experimental results show that the classification step contributes the most to the execution time of the algorithm on the CPU. With the use of massively parallel processing units we have successfully decreased this ratio.

Furthermore, our experimental results show that we can successfully harness the power of the massively parallel processing units and overcome the computational complexity of the whole algorithm, and thus the algorithm can be run in a reasonable amount of time.

Acknowledgements This research was supported by Slovenian Research Agency (ARRS) under grant P2-0359 (National research program Pervasive computing) and by Slovenian Research Agency (ARRS) and Ministry of Civil Affairs, Bosnia and Herzegovina, under grant BI-BA/10-11-026 (Bilateral Collaboration Project) and by the Ministry of Science and Technology of the Republic of Srpska under contract 06/0-020/961-220/11 (Automatic land cover/land use classification).

References

1. Belloch JA, Gonzalez A, Martínez-Zaldívar FJ, Vidal AM (2011) Real-time massive convolution for audio applications on GPU. J Supercomput 58(3):449–457. doi:10.1007/s11227-011-0610-8. http://www.springerlink.com/index/10.1007/s11227-011-0610-8
2. Cecilia JM, Abellán JL, Fernández J, Acacio ME, García JM, Ujaldón M (2012) Stencil computations on heterogeneous platforms for the Jacobi method: GPUs versus cell BE. J Supercomput 62(2):787–803. doi:10.1007/s11227-012-0749-y. http://www.springerlink.com/index/10.1007/s11227-012-0749-y
3. Che S, Boyer M, Meng J, Tarjan D, Sheaffer J, Skadron K (2008) A performance study of general-purpose applications on graphics processors using CUDA. J Parallel Distrib Comput 68(10):1370–1380. doi:10.1016/j.jpdc.2008.05.014
4. Comput JPD (2012) G-MSA—a GPU-based, fast and accurate algorithm for multiple. J Parallel Distrib Comput 73(1):32–41. doi:10.1016/j.jpdc.2012.04.004
5. Fatone L, Giacinti M, Mariani F, Recchioni MC, Zirilli F (2012) Parallel option pricing on GPU: barrier options and realized variance options. J Supercomput 62(3):1480–1501. doi:10.1007/s11227-012-0813-7. http://www.springerlink.com/index/10.1007/s11227-012-0813-7
6. Gravvanis GA, Filelis-Papadopoulos CK, Giannoutakis KM (2011) Solving finite difference linear systems on GPUs: CUDA based parallel explicit preconditioned biconjugate conjugate gradient type methods. J Supercomput 61(3):590–604. doi:10.1007/s11227-011-0619-z. http://www.springerlink.com/index/10.1007/s11227-011-0619-z
7. Halfhill T (2008) Parallel processing with CUDA. Microprocessor report pp 1–8
8. Manjunath B, Ma W (1996) Texture features for browsing and retrieval of image data. IEEE Trans Pattern Anal Mach Intell 18(8):837–842. doi:10.1109/34.531803

9. Nimmagadda VK, Akoglu A, Hariri S, Moukabary T (2011) Cardiac simulation on multi-GPU platform. J Supercomput 59(3):1360–1378. doi:10.1007/s11227-010-0540-x. http://www.springerlink.com/index/10.1007/s11227-010-0540-x
10. NVIDIA Corporation (2010) NVIDIA TESLA Computing Processor Datasheet. http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C1060_US_Jan10_lores_r1.pdf
11. NVIDIA Corporation (2011) CUDA C best practices guide, version 4.0. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf
12. NVIDIA Corporation (2011) CUDA CUFFT Library. http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUFFT_Library.pdf
13. NVIDIA Corporation (2011) NVIDIA CUDA C Programming Guide, Version 4.0. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
14. Owens J, Houston M, Luebke D, Green S, Stone J, Phillips J (2008) GPU computing. Proc IEEE 96(5):879–899. doi:10.1109/JPROC.2008.917757
15. Owens J, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn A, Purcell T (2007) A survey of general-purpose computation on graphics hardware. Comput Graph Forum 26(1):80–113. doi:10.1111/j.1467-8659.2007.01012.x
16. Risojevic V, Babic Z (2011) Aerial image classification using structural texture similarity. In: IEEE international symposium on signal processing and information technology (ISSPIT), pp 190–195. doi:10.1109/ISSPIT.2011.6151558
17. Risojevic V, Momic S, Babic Z (2011) Gabor descriptors for aerial image classification. In: Dobnikar A, Lotric U, Ster B (eds) ICANNGA (2). Lecture notes in computer science, vol 6594. Springer, Berlin, pp 51–60
18. van de Sande K, Gevers T, Snoek C (2011) Empowering visual categorization with the GPU. IEEE Trans Multimed 13(1):60–70. doi:10.1109/TMM.2010.2091400
19. Schellmann M, Gorlatch S, Meiländer D, Kösters T, Schäfers K, Wübbeling F, Burger M (2010) Parallel medical image reconstruction: from graphics processing units (GPU) to grids. J Supercomput 57(2):151–160. doi:10.1007/s11227-010-0397-z. http://www.springerlink.com/index/10.1007/s11227-010-0397-z
20. Thibault J, Senocak I (2012) Accelerating incompressible flow computations with a Pthreads-CUDA implementation on small-footprint multi-GPU platforms. J Supercomput 59:693–719. doi:10.1007/s11227-010-0468-1
21. Valero P, Sánchez JL, Cazorla D, Arias E (2011) A GPU-based implementation of the MRF algorithm in ITK package. J Supercomput 58(3):403–410. http://www.springerlink.com/index/10.1007/s11227-011-0597-1
22. Wang Z, Bovik A, Sheikh H, Simoncelli E (2004) Image quality assessment: from error visibility to structural similarity. IEEE Trans Image Process 13(4):600–612. doi:10.1109/TIP.2003.819861
23. Wang Z, Bovik AC (2009) Mean squared error: love it or leave it. IEEE Signal Process Mag 26(1):98–117
24. Yang Y, Newsam S (2010) Bag-of-visual-words and spatial extensions for land-use classification. In: Proceedings of the 18th SIGSPATIAL international conference on advances in geographic information systems, GIS'10. ACM, New York, pp 270–279. doi:10.1145/1869790.1869829. http://doi.acm.org/10.1145/1869790.1869829
25. Zhao X, Reyes M, Pappas T, Neuhoff D (2008) Structural texture similarity metrics for retrieval applications. In: Proceedings of 15th IEEE international conference on image processing ICIP 2008, San Diego, CA, USA, pp 1196–1199
26. Zujovic J, Pappas TN, Neuhoff DL (2009) Structural similarity metrics for texture analysis and retrieval. In: Proceedings of the 16th IEEE international conference on image processing, ICIP'09. IEEE Press, Piscataway, pp 2201–2204. http://portal.acm.org/citation.cfm?id=1819298.1819352