2014

Real-time indexing for large image databases: color and edge directivity descriptor on GPU

Bampis, Loukas

Springer

J Supercomput (2015) 71:909–937
DOI 10.1007/s11227-014-1343-2

Real-time indexing for large image databases: color and edge directivity descriptor on GPU

L. Bampis · C. Iakovidou · S. A. Chatzichristofis · Y. S. Boutalis · A. Amanatiadis

Published online: 2 December 2014

Abstract In this paper, we focus on implementing the extraction of a well-known low-level image descriptor using the multicore power provided by general-purpose graphic processing units (GPGPUs). The color and edge directivity descriptor, which incorporates both color and texture information achieving a successful trade-off between effectiveness and efficiency, is employed and reassessed for parallel execution. We are motivated by the fact that image/frame indexing should be achieved real time, which in our case means that a system should be capable of indexing a frame or an image as it becomes part of a database (ideally, calculating the descriptor as the images are captured). Two strategies are explored to accelerate the method and bypass resource limitations and architectural constrains. An approach that exclusively uses the GPU together with a hybrid implementation that distributes the computations to both available GPU and CPU resources are proposed. The first approach is strongly based on the compute unified device architecture and excels compared to all other solutions when the GPU resources are abundant. The second implementation suggests a hybrid scheme where the extraction process is split in two sequential stages, allowing the input data (images or video frames) to be pipelined through the central and the graphic processing units. Experimental results were conducted on four different combinations of GPU–CPU technologies in order to highlight the strengths and the weaknesses of all implementations. Real-time indexing is obtained over all computational setups for both GPU-only and Hybrid techniques. An impressive 22 times acceleration is recorded for the GPU-only method. The proposed Hybrid implementation outperforms the GPU-only implementation and becomes the preferred solution when a low-cost setup (i.e., more advanced CPU combined with a relatively weak GPU) is employed.

L. Bampis · C. Iakovidou · S. A. Chatzichristofis · Y. S. Boutalis · A. Amanatiadis (✉)
Department of Electrical and Computer Engineering, Democritus University of Thrace,
12 Vas. Sofias Str, Xanthi 67100, Greece
e-mail: aamanat@ee.duth.gr

Keywords GPU · Hybrid implementation · Image retrieval · Feature extraction · Database indexing

1 Introduction

An impressive number of content-based image retrieval (CBIR) methods have been lately presented in the literature [1]. CBIR emerged due to the rich information that images hold in a polysemy way, making it hard to be justified through words. The problem addressed by the proposed methods is to develop a system that is capable of describing an image with a descriptor sufficiently distinctive to identify the particular characteristics but also capable to associate the similar ones from a collection of images. CBIR consists of two stages; the indexing of the images in a database (descriptor extraction) and the searching phase.

There are mainly, two basic approaches to the CBIR problem arising according to the type of visual features that they employ. The features are either global features (GF), like color features, texture features, and shape features being computed on the entire image, or local features (LF) which have a spatial extent (local neighborhood) and are typically salient patches of the image, rich in visual information. Local-feature approaches provide a slightly better retrieval effectiveness than global features [2]. They represent images with multiple points of interest in a multi-dimensional feature space in contrast to single-point global feature representations. This high dimensionality of the LFs adds to the robustness, but makes the methods computationally expensive leading to a dramatic increase in terms of execution time as the databases grow.

CBIR methods based on LFs aim to describe the visual content of the images in a more semantic way, retrieving images with similar content apart from the visual likeness. Current research in the CBIR field has been strongly focused on the bag of visual words (BOVW) framework, which is directly inspired by the bag of words method first introduced in the text retrieval field. This approach manages to increase the effectiveness over near duplicate images compared to GF and also improves the efficiency compared to LF methods. The BOVW framework is considered as a promising framework for content-based image retrieval [3].

Methods that make use of local features and visual words are, to some extent, rotation, scale and viewpoint invariant. Unfortunately, they require computationally challenging operations like multiple scans per image, extensive training and time-consuming visual codebook generation. Their efficiency remains unfit for large databases, especially to those that are vigorously enriched with new images, like on-line databases, and whose codebook (and by extension the whole descriptor) must be reassessed when their size changes significantly.

On the other hand, in applied research CBIR often relies on global features, at least as a foundation for further research [4]. Methods that use global descriptors are lighter, database-size invariant and perform better than local features when the objective is to retrieve images with similar visual properties to the query image [5]. An advantage of the GF over the BOVW model is associated with the fact that BOVW methods, disregard the information about the spatial layout of the visual content, and they present limited descriptive ability on whole-image categorization tasks [6]. Moreover,

in two-stage multimodal retrieval systems, where both visual and textual information are available and deployed, authors in [7] found that global features perform better than the BOVW paradigm. Furthermore, the quantitative characteristics of the global features, make them widely usable and effective [8].

As image databases are growing, it becomes apparent that the efficiency, apart from the effectiveness of a method, is a matter of great importance. Flickr has an image upload rate of about 4.5 million/day, when Facebook has already an impressive 300 million image uploads by its users, and according to 2011 YouTube statistics over 4 billion hours of video is watched each month. Accelerating the descriptor extraction procedure will allow immediate indexing of the visual content of the image. In such a case, low-level features can be incorporated in the header of the captured file for future use. Additionally, speeding up the image's descriptor extraction would allow large image repositories, such as Flickr, iCloud and Dropbox, to index the uploaded images as they become part of their databases. It is worth noting, that in case of videos, real-time feature extraction would allow the real-time generation of essential low-level information that can be used for video summarization and automatic annotation.

In the meanwhile, graphics processing units (GPUs), which are powerful parallel computing devices used in embedded systems, personal computers, game consoles and mobile devices, have become attractive to general-purpose system development with the introduction of the compute unified device architecture (CUDA) by Nvidia. As far as image processing is concerned, general-purpose computing on graphics processing units (GP-GPU), which essentially means using a GPU to perform computations which are traditionally handled by the CPU, flourished as recording, enhancing and sharing multimedia data, became an everyday activity for computer and portable/mobile device users. Thus, multiple general-purpose GPUs implementations of existing image processing, recognition, indexing, and categorization algorithms or parts of those algorithms have been introduced both for computers [9–15] and mobile devices [16–18]. All methods take advantage of the available highly parallel computing architecture to perform the heavy computational tasks and the manipulation of the massive amount of data usually employed in image processing.

In this paper, we focus on real-time image indexing for retrieval tasks. Apart from the ongoing growth of on-line multimedia files, executing computer vision algorithms on devices with limited memory [19] and computation complexity capabilities has revealed the need to make descriptors faster to compute. We define as *real-time indexing* the ability of a system to index (i.e., to extract a descriptor vector) during capturing a VGA frame stream of 25 fps. We employ the color and edge directivity descriptor (CEDD) [20] which incorporates both color and texture information. CEDD is a compact descriptor and as recently the authors in [21] and [22] point out, widely accepted due to its successful trade-off between effectiveness and efficiency.

The CEDD descriptor has been widely used in recent literature. Some illustrative examples are the following: Leuken et al. [23] used CEDD to extract features as part of a technique that visually diversifies image search results. In [24], the authors propose the use of the CEDD descriptor for image visual similarity estimation while in [25], CEDD is employed as image feature for integrating content-based similarity search in content centric networks. Recently, authors in [26] employ the SURF detector to define salient image patches of blob-like textures and use the CEDD, to produce

local-feature vectors. Its lightweight nature and effective performance, is key for the inclusion of the method in a tool for summarization of arthroscopic videos presented in [27]. CEDD consist of 144 coefficients requiring <54 bytes. Moreover, CEDD is computationally lightweight relative to other feature extraction mechanisms, but has comparable accuracy. Thus, apart from being used as a stand-alone indexing and retrieval solution it is also integrated in a great variety of other methods [28–32] that in some stages require a low-level feature representation of the visual content.

We propose the implementation of CEDD with two different strategies. In both cases the approaches follow the parallel nature of the CEDD method. Initially, taking into account the principles, the guidelines and the constrains of the CUDA architecture, we reassess the model and design a parallel equivalent which is tested on four different computational setups with varying CPU and GPU technologies. In the sequel, by measuring the execution time of the different parts of the algorithm, the overhead introduced by multiple memory accesses and by estimating the necessary available GPU resources, we calculate the most computationally demanding stage. The detected bottleneck of the process, which is more evident when the GPU resources are limited, is addressed by an alternative proposed hybrid implementation employed to distribute the computations on both CPU and GPU in a pipelined scheme as to exploit all the available computational power of a system.

The rest of the paper is organized as follows: Sect. 2 provides a short overview of the CEDD descriptor and Sect. 3 briefly outlines the CUDA architecture principles and constrains in order for the paper to be self-contained. Section 4 presents the design strategies followed for both proposed implementations. Additionally, Sect. 4 proposes a new Parallel version of the Participation Identifier for the implementation of TSK Fuzzy systems. The experimental results are given and commented in Sect. 5. Conclusions are drawn in Sect. 6 where open issues for future work are stated as well.

2 CEDD: color and edge directivity descriptor

The CEDD is a global descriptor first introduced in [33] for the indexing of large image collections and the execution of retrieval tasks. As its name implies, CEDD consists of a color extraction component and a texture extraction unit. CEDD was designed with particular attention to size and storage requirements without compromising its discrimination ability. In order to highlight the effectiveness of CEDD over several other descriptors from the literature, experiments on WANG[34] and UCID[35] databases are illustrated in Table 1. To measure the performance of the descriptor, we used the mean average precision (MAP) method.

The remainder of this section briefly presents the CEDD descriptor and its structural elements. For a more detailed description, kindly refer to [20,33]. CEDD begins by dividing images of any size into 1,600 rectangular image areas, referred to as Image-Blocks. The algorithm's objective is to categorize the Image-Blocks according to their combined color and texture information and compactly represent them with a single Image-Block vector. When all vectors of the 1,600 Image-Blocks are produced they are further combined to form a single vector that serves as the descriptor of the image for indexing and retrieval tasks. As depicted in Fig. 1, the Texture and the Color Units

Table 1 MAP values on WANG and UCID databases

Descriptor	WANG	UCID
CEDD	0.5891	0.6748
FCTH [20]	0.5736	0.6723
BTDH [36]	0.3503	0.5353
C.CEDD [20]	0.5296	0.6584
C.FCTH [20]	0.5222	0.6487
JCD [37]	0.5880	0.6876
SpCD [38]	0.4578	0.5840
MPEG-7 EHD [39]	0.3097	0.5326
MPEG-7 SCD [39]	0.2557	0.4998
MPEG-7 CLD [39]	0.4626	0.5361
Color histograms	0.3018	0.4443
Tamura directionality	0.2586	0.4411
Auto-correlograms [40]	0.3634	0.5507
Top-surf (10,000 visual words) [41]	0.2526	0.4248
Top-surf (200,000 visual words) [41]	0.1612	0.3952

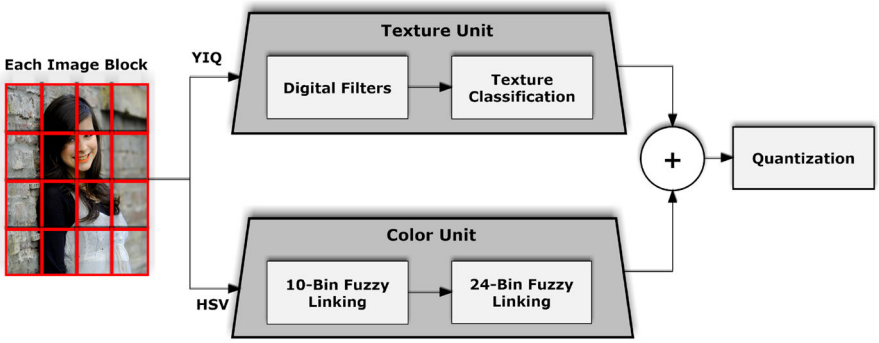

Fig. 1 CEDD parallel color and texture extraction units

are computed in parallel and their outputs are combined and quantized to produce the final descriptor.

A commonly used way to describe the color information of an image is by linking the color space channels. Linking is defined as the combination of more than one histograms to a single one [20]. In our case, after the division of the image into 1,600 Image-Blocks, the average color of each block is converted from the RGB to the HSV color space. Then, a two-staged fuzzy system is employed to produce a fuzzy linking histogram. The first stage of the fuzzy system has the three mean HSV channels of an Image-Block as inputs, and forms a 10-bin histogram as an output. Each one of the 10-bins represents a preset color. The three inputs of the fuzzy system are described as follows: Hue (H) is divided into eight fuzzy areas, Saturation (S) is divided into two fuzzy regions while the channel Value (V) is divided into three areas. The output

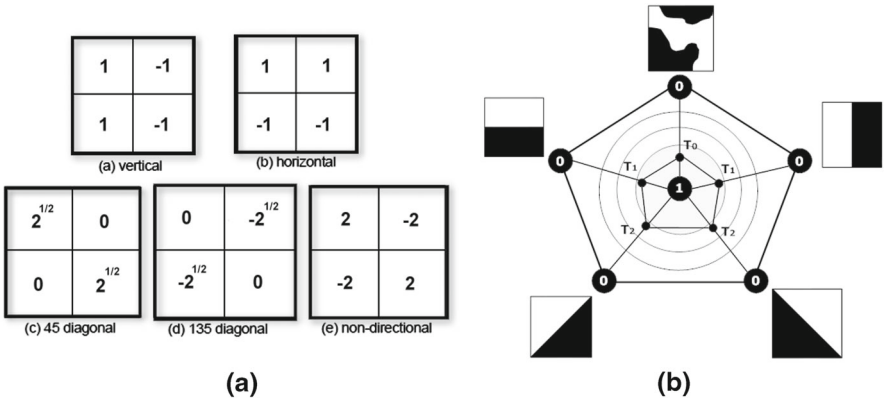

Fig. 2 **a** Filter coefficients for edge detection, **b** Edge type diagram (heuristic pentagon diagram) [20]

of the fuzzy system is enabled by a set of 20 rules and returns a crisp value ranging from 0 to 1 (TSK type fuzzy system).

Depending on the actual mean value of H, S and V values, the fuzzy rules produce a 10-bin histogram output (Image-Block Stage 1 Vector). After the first stage is completed 1,600 10-bin histograms have been produced, for every Image-Block accordingly.

The next stage consists of a second fuzzy linking system (TSK) which is responsible for adding the brightness value of a hue. Again the S and V mean values of an Image-Block, previously calculated, become fuzzy inputs.

The output is a 3-bin histogram of crisp values indicating if the color will be characterized as light, normal or dark hued. These two outputs (Stage 1 vector and Stage 2 vector) are combined so that each Image-Block is characterized by the color extraction procedure. On the completion of the process, a 24-bin color histogram is produced. Each bin represents a preset color as follows: (0) Black, (1) Grey, (2) White, (3) Dark Red, (4) Red, (5) Light Red, (6) Dark Orange, (7) Orange, (8) Light Orange, (9) Dark Yellow, (10) Yellow, (11) Light Yellow, (12) Dark Green, (13) Green, (14) Light Green, (15) Dark Cyan, (16) Cyan, (17) Light Cyan, (18) Dark Blue, (19) Blue, (20) Light Blue, (21) Dark Magenta, (22) Magenta, (23) Light Magenta.

The Texture information extraction unit employs the five digital filters proposed by the MPEG-7 Edge Histogram Descriptor-EHD [39]. These filters are illustrated in Fig. 2a. In this case, each Image-Block is converted to the YIQ color space and separated into four Image Sub-Blocks. The value of each Image Sub-Block is the mean value of the luminosity (Y) of the pixels that participate in it. The result obtained from the application of the digital filters to the Image Sub-blocks for each Image Block, serves as input to a fuzzy mapping scheme, illustrated in Fig. 2b. In its essence this mapping system is responsible for indicating which kind of texture (from five available plus one "Non-Texture") is present for every Image-Block. More than one textures can simultaneously be present. The normalized maximum responses (edge magnitudes) from the applied filters per Image Block are placed in the heuristic pentagon diagram, as shown in Fig. 2b. Each value is placed along the line that pertains to the filter it

emerged from. If that value is greater than the threshold associated with the line it belongs to, the Image-Block is classified in the respective type of texture. This means that every Image Block can participate in more than one type of textures, as long as the corresponding edge magnitude scores higher than the threshold. Overall, the output of the texture unit is a 6-bin vector for each Image Block. The texture regions are described as follows: Non Edge, Non Directional Edge, Horizontal Edge, Vertical Edge, 45-Degree Diagonal and 135-Degree Diagonal.

Finally 1,600 6-bin texture vectors, one for every Image-Block, are produced. Every bin represents one of the five available textures while the first bin represents the non-textured case. When a texture was found to be present in that Image-Block, the corresponding bin is marked with 1. Otherwise it is marked as 0, producing the binary Image-block texture vector.

When the color vectors and the texture vectors have been calculated for every Image-Block, the final Texture-Color vector for every Image-Block is produced by combining the 1×6 texture vector and 1×24 color vector. Then all Image-Block descriptors are added to form the image descriptor. This vector is normalized and quantized into 8 predefined levels. On completion the CEDD descriptor has been formed and will represent the visual content of the image for indexing and retrieval tasks.

3 CUDA architecture

Compute unified device architecture (CUDA) is a general-purpose parallel computing architecture introduced by Nvidia. This section provides a quick overview of the CUDA architecture and its optimization issues in order for the paper to be self-contained.

Generally in CUDA terms, the GPU is called device and the CPU that calls the CUDA functions is the host. The GPU is divided in a number of multicore processors named multiprocessors (MPs), as shown in Fig. 3. Each multiprocessor is a set of processors with a single instruction multiple data (SIMD) architecture, meaning that at each clock cycle, a multiprocessor executes the same instruction on a group of threads, called a warp. The maximum size of a warp that can be handled is 32. Due to the SIMD nature of CUDA, at one time the threads must perform identical operations [10]. By ignoring this constraint and by enabling control or memory divergence [42] the application is partially serialized because different instructions will be executed in different clock cycles and groups of threads will unnecessarily stay idle [43].

CUDA-enabled GPUs have a number of cores that can collectively run computing threads. In every CUDA application there are two types of threads, the host thread of the CPU (host) and the threads of the device. The host is responsible for copying data between the CPU and the GPU and initiates the parallel functions called kernels. The number of device threads that can be executed in parallel on such devices is currently in the order of hundreds but still provides many applications with the opportunity to excel, benefiting from the parallelism that the GPUs introduce, compared to the CPUs.

CUDA kernel function is a set of instructions that the device's threads will execute. When launching a CUDA kernel function, a developer specifies how many copies (tasks) of it to run, or in other words, how many threads will be activated and execute

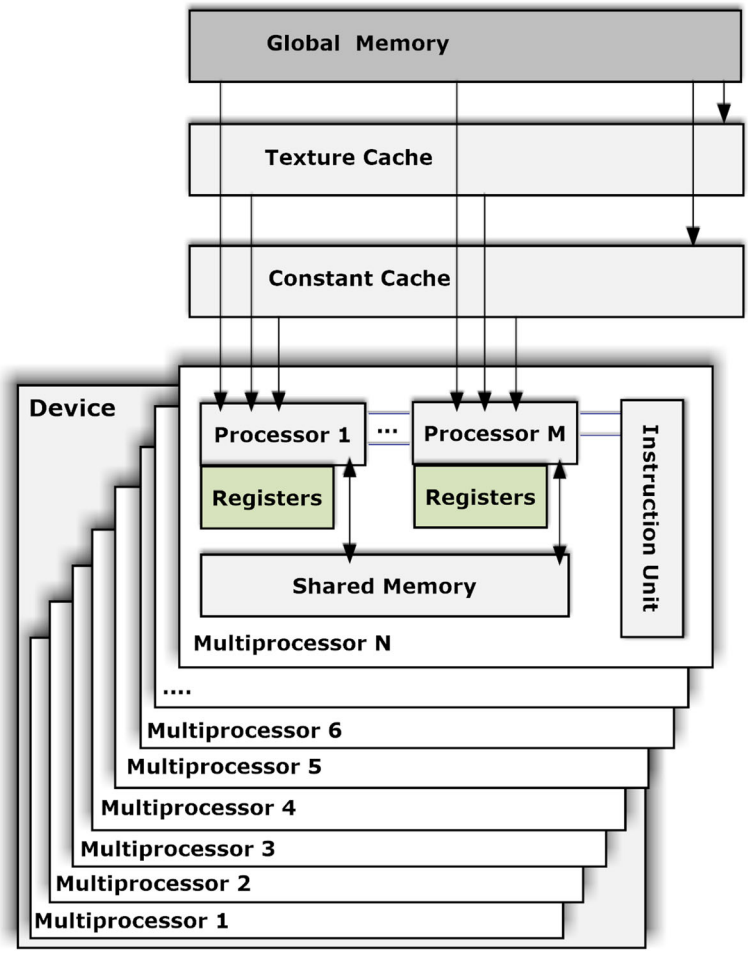

Fig. 3 Nvidia's device architecture. The device consists of multiprocessors (MPs), and each of them encloses a number of processors (Cores). Each core has its own registers and can access both global and shared memory

the kernel. The threads executing a kernel are organized in a two-level hierarchy, Thread Block and Grid as depicted in Fig. 4. Every Grid is a set of Thread Blocks and every Thread Block is a set of threads. Based on this hierarchy every thread and Thread Block has its own `id` that can be of one, two or three dimensions (`threadIdx.x` ,`threadIdx.y` and `threadIdx.z` for threads and `blockIdx.x`,`blockIdx.y` and `blockIdx.z` for Thread Blocks) depending on the applied GPU technology. The maximum number of Thread Blocks that can be executed in parallel varies and is defined by the GPU model. In order to assure high device occupancy (i.e., engaging as many resources as possible), a robust and therefore preferable approach is to use less Thread Blocks of more threads wherever possible.

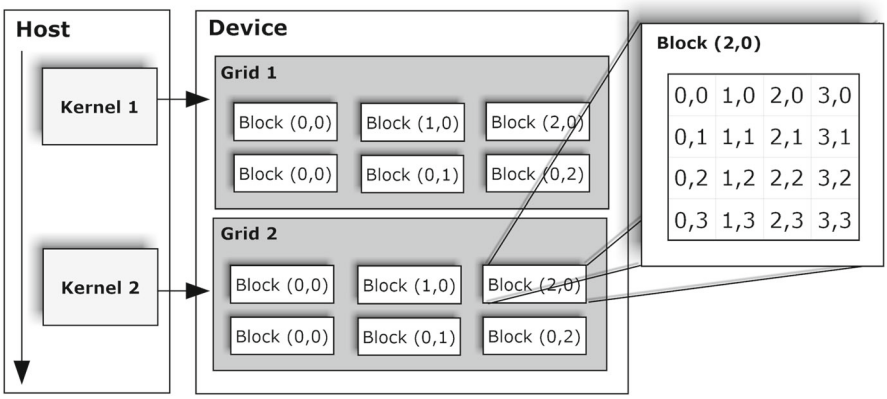

Fig. 4 Structure of thread block and threads per kernel

The device memory space consists of various types of memories, as shown in Fig. 3. Registers are local memory spaces assigned per processor. The threads belonging to the same Thread Block can share data through the Shared Memory without sending it over the system memory bus. Threads from different Thread Blocks coordinate only through Global Memory, a large and long-latency memory which has read/write operations. Besides this, the Constant Memory and the Texture Memory allocated for a grid are read-only and basically cached global memories. The Texture Memories are preferred for handling 2D/3D arrays [44].

It is important to highlight that Constant Memories and Texture Memories are as fast to access as Registers on cache. The Shared Memory is as fast as accessing Registers too, as long as there is no bank conflict. It is divided into equal-sized memory modules called banks in order to succeed bandwidth-wise. However, if two memory requests fall in the same bank, then the access is serialized, reducing the bandwidth. Accessing the Global Memory space is much slower, typically two orders of magnitude slower than floating point multiplication and addition [45]. Global Memory read operations from threads whose `id` follows the memory alignment guidelines can be coalesced leading to faster execution.

In conclusion, concerning memory manipulation, if it is necessary to use multiple times the same data from the Global Memory, the Texture Memory, the Constant Memory or the Shared Memory, the efficient strategy is to copy the data to Registers and access it from there, as long as this can be achieved. As a final note, two different threads, in the same warp, can write simultaneously to the same address in the Global Memory. This introduces a parallelization limitation since the writing order can not be specified which can lead to unexpected results.

4 CUDA implementation

CEDD, as thoroughly explained previously, is a perfect candidate for parallel processing because it divides the problem into many identical and independent sub-problems.

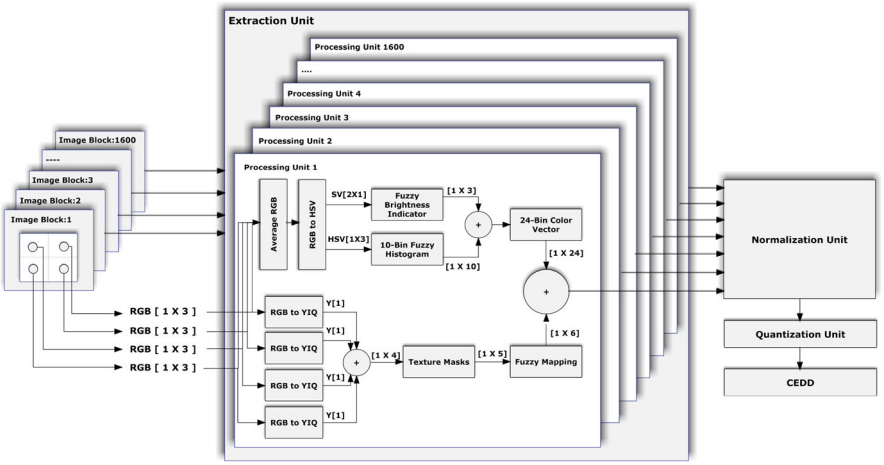

Fig. 5 GPU-only implementation flowchart

Moreover, employing the GPU to handle calculations allows us to take advantage of its powerful computational resources for our application, which otherwise remains unused. GPUs are available on every personal computer, their architecture is utilized for image processing (since their main objective is to handle graphics) and thus, comprise a low cost and widely applicable solution for image processing applications. When employed to accelerate algorithms that have a parallel structure by design, such as CEDD, successful outcome is guaranteed.

The image indexing problem has yet another important aspect to take under consideration; the massive input data and the needed iterations as to successfully index large image or video collections. This parameter prompted us to face the challenge with two different approaches. The first one is a GPU-only implementation oriented towards the optimal manipulation of CUDA's features. The second one is a hybrid implementation committed to deploying every available resource (CPU–GPU) in a pipelined-based scheme. Both CUDA implementations together with the original c# source codes are available on-line at http://www.tinyurl.com/CEDD-CUDA.

This section describes the first adopted approach where the whole descriptor is generated by using exclusively GPU resources. The implementation, which is depicted in Fig. 5, consists of three successive stages: the Average RGB values calculation of every Image Sub-Block, the main Extraction Unit with color and texture extraction components for every Image Block and the third stage which forms the CEDD descriptor of the image.

4.1 Reading the image/frame to be indexed

Before the GPU can begin to process the images/frames, the CPU starts by reading the image from the hard drive. This step is handled by the CPU due to the absence of a direct communication channel between the GPU and the storage device. At this point

a critical decision must be made; we must decide whether we will ask the CPU to simply read the image in the current saved format or burden the CPU with the task of storing values belonging to an Image Sub-Block in neighboring memory slots per RGB channel. Ideally, the accesses to the Global Memory that the threads perform should be coalesced. This essentially means that the stored data accessed by the same Thread Block should be sequential. For the first attempted implementation, we performed no such data manipulation. As shown in the "Appendix", the reading of the images takes up on average 62 % of the total execution time. When further engaging the CPU to rearrange the data in a suitable format, the GPU part of the implementation achieved almost a 10 % speed-up, but the new CPU execution time (reading the image and rearranging the data) significantly slowed down the overall implementation and therefore data rearrangement was abandoned. Figure 6 illustrates the bmp storage format as it was copied into the Global Memory of the GPU, as well as the thread accessing pattern that occur during the first GPU kernel, i.e., the calculation of the average RGB values explained right after.

4.2 Calculating the average RGB values of image sub-blocks

As mentioned earlier in Sect. 2, the method requires the image to be divided in 1,600 Image-Blocks in order to extract their color information and further divides each block into four equal Image Sub-Blocks in order to extract the texture information. In total, 6,400 average color values need to be calculated for each of the three channels of the RGB color space and become inputs both in the texture and the color units. For this to be done 6,400*3 Thread Blocks are activated.

Each Thread Block contains as many threads as needed for the pixels to be summed based on the original image size. As described before, the number of threads in each block is limited. For instance in the weakest GPU that we tested, this limit is 512 threads per block. In this case, if an Image Sub-Block is formed by more than 512 pixels, their processing becomes partly serialized, delaying the total execution time.

The calculation of the average RGB values of an Image Sub-Block is essentially a summation problem. Summation is by default a not fully parallelizable procedure. The efficiency of the summation procedure depends mainly on the employed technique, but can benefit from input data organized in a suitable manner, a scenario that we explored but in our case was found unfit.

The widely employed technique that manages to semi-parallelize the summation problem follows the Reduction process [46] architecture. This method is a tree-based approach where every involved thread sums two values of the input data. The outcome becomes the input data of the next step and the process continues until only two values are left for one thread to finally sum.

In our implementation, for every RGB channel of an Image Sub-Block, a device Thread Block is assigned. This one-to-one assignment that we adopt leads to a maximum image size of about 6.5 Mega-Pixels defined by the weakest GPU of 512 threads per block. Since the Reduction process assigns one thread for the summation of two values, the maximum Image Sub-Block size can be up to 1,024 pixels for each one of the 6,400 Image Sub-Blocks that the image consists of. The maximum image size can

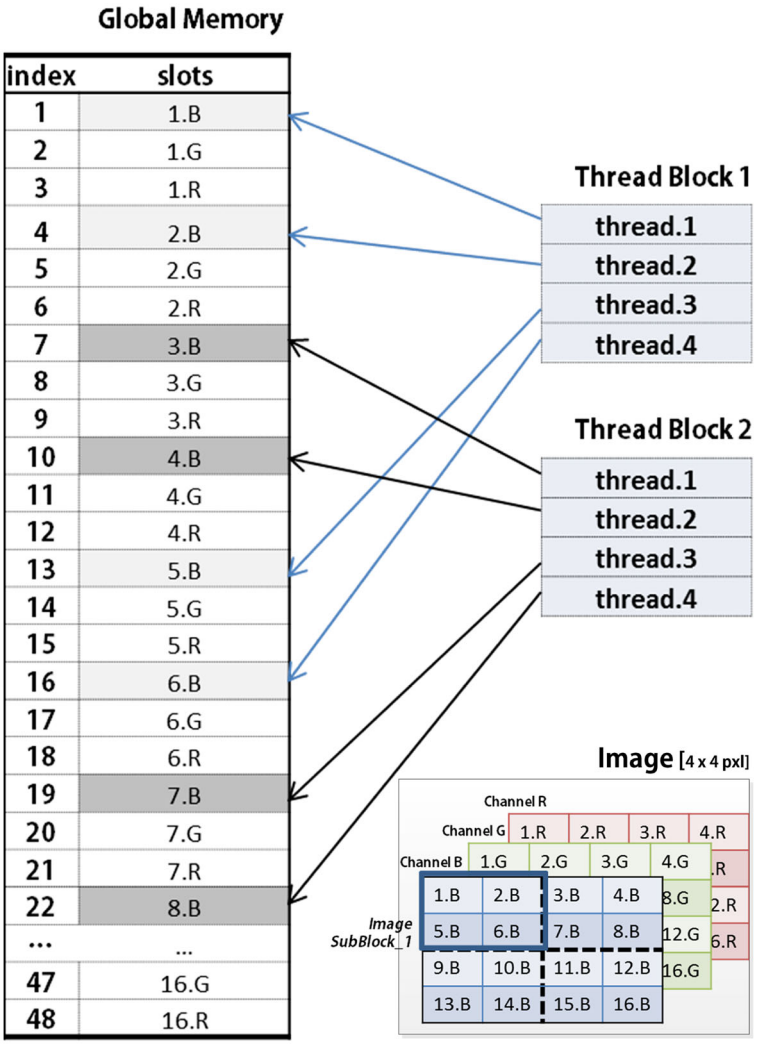

Fig. 6 The BMP image is copied to the GPU with its standard storage format

be further increased if more than one Thread Blocks are assigned per Image Sub-Block. If that is the case, the output of every Thread Block responsible for the same Image Sub-Block must be added. The optimum exploitation of the resources in a summation process is achieved when all the available parallel threads per clock cycle are engaged in the process. This can be ensured when all provided Thread Blocks are employed and utilized to their maximum threading capacity, defined by the GPU model.

In our method, the values to be summed depend on the size of the Image Sub-Block. Thus, when the size of the input image is small the utilization is not optimum and that is also evident in our experimental results provided in the next section. By the end of the summation process, the method can be divided into 1,600 independent problems.

We will continue describing the implemented architecture for one of them, since they all share the same instructions applied on different input data (Image Blocks).

4.3 Color information extraction unit: TSK fuzzy system implementation using a parallel participation identifier

Every Image Block (four Image Sub-Blocks) enters simultaneously the Texture and the Color Units. We will begin by describing the Color Extraction Unit (Fig. 1). The average pixel values of the four Image Sub-Blocks are used to find the average red, green and blue value of the entire Image Block they belong to. These RGB values are converted into the HSV (Hue, Saturation, Value) color space. The S and V components become inputs to the Fuzzy Brightness Indicator while in parallel all three components (HSV) enter the 10-bin Fuzzy Histogram. Both fuzzy systems share the same architectural principles in the GPU implementation using a proposed parallel version of a participation identifier (PPI).

The PPI technique is employed to indicate whether or not a value belongs to a specific sub-region of a range of values. PPI ensures that the requested output will be produced in one time-step since it evaluates the participation of the input on all sub-regions in parallel. We assume that a set $S_k, k \in [0, n]$ may be classified into $V_i, i \in [0, l], l, n \in N$ regions. We also assume that each S_k belongs to V_i. Using the straightforward approach, for each S_k, one up to l number of participation checks are required so as to form the outcome. In case of PPI, each S_k is compared in parallel with all V_i so as to determine the participation of S_k in one or more of them. Even though, following the straightforward approach could allow for the output to be produced in one single time-step (if the participation was located in the first tested sub-region) using the minimum amount of resources, PPI ensures that regardless of the sub-region that S_k will be found to participate, always one time step will be needed. For this to be achieved we engage the maximum needed resources, i.e., number_of_thread = number_of_sub-regions.

This approach can be adopted by a variety of clustering applications, especially when many sub-regions/clusters are employed. As mentioned earlier, due to the SIMD CUDA architecture, at one time, threads must perform identical operations. Thus, PPI adapts to this principle providing the outcome at once, allowing in the next time-step the procedure to be continued with a new set of identical operations. The PPI method described above applies to the TSK-Fuzzy systems implementation. One thread per membership function region is activated. Given the fact that the 10-bin Fuzzy Histogram is calculated using three inputs (H, S and V), containing 8, 2 and 3 membership functions respectively, $8 \times 2 \times 3 = 48$ combinations per Image-Block are tested in parallel by a respective number of threads. Overall, in a single time step, $1{,}600 \times 48$ threads are activated. Similarly, in case of the 24-bin Fuzzy Histogram (Fuzzy Brightness Indicator), four threads are employed simultaneously per Image-Block.

When both outputs from the Fuzzy Brightness Indicator and the 10-bin Fuzzy Histogram are produced, they are combined to form the $[1 \times 24]$ long Color Vector. The following pseudo code describes the combination process.

```
Data: x ∈ [0, 7] ∪ N, y ∈ [0, 2] ∪ N, z=0
for 24 threads with threadIDx.x/y/z do
    if threadIDx.x==0 then
        24Bin_Color_Vector[threadIDx.x × 3 + threadIDx.y]=
        10Bin_Color_Vector[threadIDx.y]
    else
        24Bin_Color_Vector[threadIDx.x × 3 + threadIDx.y]=
        10Bin_Color_Vector[threadIDx.x +2] × 3Bin_Brightness_Vector[threadIDx.y]
    end
end
```

Algorithm 1: The Color Combination Process.

According to CEDD, the first three bins from the 10-bin Histogram Unit are forwarded unchanged to become the first three components of the 24-bin Color Vector. As for the remaining bins, every color bin is multiplied with all three brightness bins to produce a three-shaded representation of the particular color. In CUDA, we enable 24 threads to process as follows: three threads are responsible for transferring the first three 10-bin values, and another set of three threads per color bin (i.e., three threads for every one of the remaining seven colors) is enabled to execute the multiplication of the brightness values with the color. All computations are implemented in parallel and the 24-bin Color Vector is formed at once.

4.4 Texture information extraction unit

The inputs of the Texture Extraction Unit are the four Image Sub-Blocks that comprise an Image Block. The Image Sub-Blocks are the required building blocks in this part of the method, due to the texture masks' design properties that were followed in CEDD. All 1,600 Image Blocks that compose the input image are processed in parallel by identical kernels.

The RGB components of each Image Sub-Block are transformed into the YIQ color space. Only the Y (Luminance) component will be employed for the rest of the procedure. All four extracted Y values become a $[1 \times 4]$ vector used for the application of the Texture Masks. A single thread is enabled per mask. Thus, a total of five threads are executed simultaneously to calculate the five different Texture Masks. The output enters the Fuzzy Mapping system.

In short, the system will calculate which kind of texture is located in an Image Block. The method that is applied starts by locating the Texture Mask with the highest response. If a predefined threshold (hereby refereed to as T) is not met by the highest scoring mask, the whole Image Block is categorized as Non-Edge and no further computations take place. Otherwise, the corresponding values of the five masks are normalized and take part in the formation of the final Texture Vector if they surpass their individually set threshold (T_0, T_1 and T_2).

Six threads are activated to carry out the aforementioned Fuzzy Mapping system. One is the Non-Edge detector, and the rest are the five mask-threads. The maximum found value is stored in the shared device memory available for all threads to access. Their contribution to the final Texture Vector depends partially on that maximum value. Thus, all threads access the stored value to compare it with T simultaneously.

When T is not met, the Non-Edge detector thread marks the first element of the Texture Vector with 1 while the five mask-threads mark their corresponding elements as 0. Otherwise, the Non-Edge detector is zeroed and the remaining threads use the maximum value to normalize and compare their value to confirm if the response is higher than the threshold. If so, they affect the final Texture Vector by marking as 1 their corresponding element.

4.5 Composing the CEDD descriptor

The 24-bin Color Vector extracted by the Color Unit and the 6-bin Texture Vector extracted by the Texture Unit are combined to form a 144-bin vector that carries the color and texture information of an Image Block. The combination procedure of the two input vectors is governed by the same architectural principles as described earlier when the 10-bin Fuzzy Histogram and Fuzzy Brightness vectors were combined.

All 1,600 Image Blocks produce their 144-bin vector in parallel. To do this we activate 144 threads for every one of the 1,600 thread blocks. Every one of those threads is responsible for multiplying one of the color's bins with one of the texture's bin and store the result to the new vector. The vectors enter the Normalization Unit. They are summed into a single vector and normalized. For the restriction of the CEDD length, a 3 bits/bin quantization is used, constraining its total length to $144 \times 3 = 432$ bits.

The quantization procedure includes the bin-by-bin comparison of their normalized value to a group of preset ranged quantization levels. Eight quantization levels, each one of them occupying a specific sub-region of values are employed for every bin area, which leads to a total of 32 levels for the whole vector. Employing the PPI method, every bin of the image descriptor is handled by eight threads, each one responsible for checking if the bin value belongs to the corresponding quantization region. Similarly to the TSK-Fuzzy systems implementation, the quantized vector is extracted in a single time step.

Finally, the output of the Quantization Unit is the CEDD descriptor of the image. When the CEDD representation of an image is completed the CPU reads the next image/frame from the hard drive and forwards it to the GPU. The procedure continues until all images from a collection are indexed according to their CEDD descriptor.

We must highlight the fact that the descriptor is produced without any quality degradation compared to the original CPU implementation. The produced descriptors of images from all methods (CPU, GPU as well as the hybrid implementation subsequently described) match exactly.

5 The hybrid implementation

The image indexing problem starts with an image collection or a frame collection extracted from video streams. Image and video collections on-line increase their size daily with new footage uploaded by users globally. Systems that try to index images have to face a massive amount of already accumulated input data. Utilizing every possible computational resource available proves to be imperative.

Most computer systems consists of two processing units the CPU and the GPU. In this sub-section we propose a Hybrid implementation of the CEDD indexing method that exploits both resources. After studying the CEDD algorithm together with GPU parallelism and efficiency limitations, we broke down the CEDD method into two successive parts. The computation of the average RGB values per Image Sub-Block and the Color/ Texture Extraction, Vector Normalization and Quantization processes.

The first obvious benefit seized by this split, comes from the disengagement of the image size from the needed transferred to the GPU data. The CPU calculates the average RGB values for all the 1,600 $\times$ 4 Image Sub-Blocks, thus the transferred data to the GPU are 3 $\times$ 6,400 values, independently of the initial image size. This also radically decreases the required accesses (which were also not coalesced) from threads to the Global Memory, a memory space significantly slower that the other available memory spaces on a GPU. Furthermore, the splitting point was defined by the following fact; after the summation of the Image Sub-Block pixel values, the number of dependent with each other data that enter all other units of the method are of a constant and much lower order of magnitude.

The most significant factor though, is the realization that the summation procedure cannot be efficiently parallelized on any resource. Techniques in the literature addressing this problem manage to speed up the procedure compared to a serial implementation, by partially parallelizing the computations. However, the execution time of the summation either implemented on a CPU or "parallelized" on a GPU, depends on the employed technology in either cases. As in many image processing problems, our method introduces yet another delaying factor. The need to sum multiple times small non-sequential parts of an array organized in image blocks.

Thus, if we have to choose a specific task to be assigned to another than the GPU processing unit, then the summation, which is a not efficiently parallelized procedure, would be our best option. The Appendix presents the step by step timing results of the different parts of the implementation. The summation process (SumSubBlocks + SumBlocks) takes up an average of 72 % of the total GPU execution time.

Even though the summation executed on the CPU is always slower than if implemented on a GPU, by splitting the method and assigning different parts to two separate resources enables us to pipeline the procedure. Figure 7a illustrates the Hybrid implementation. The first image in a collection or the first frame of a video stream enters the CPU-Stage. Its average RGB values per Image Sub-Block are calculated and a [3 $\times$ 6400] vector of those values is sent to the GPU-Stage. Simultaneously, the second image enters the CPU-Stage. A CPU-thread is employed to execute the CPU-Stage procedures and another CPU-thread coordinates the GPU-Stage. A synchronization barrier is set between the two stages, to ensure that the input data on both parts are valid. As depicted in Fig. 7b, after the CEDD descriptor of the first image is composed, a new descriptor is produced at every pipeline time-step. The time-step is defined by the most time-consuming stage of the pipeline, i.e., the CPU-Stage.

The experimental results have confirmed that for systems armed with powerful CPUs compared to their GPUs abilities, the pipelined approach succeeds a reduction of the overall execution time compared to the GPU-only implementation, which is more evident in small image sizes.

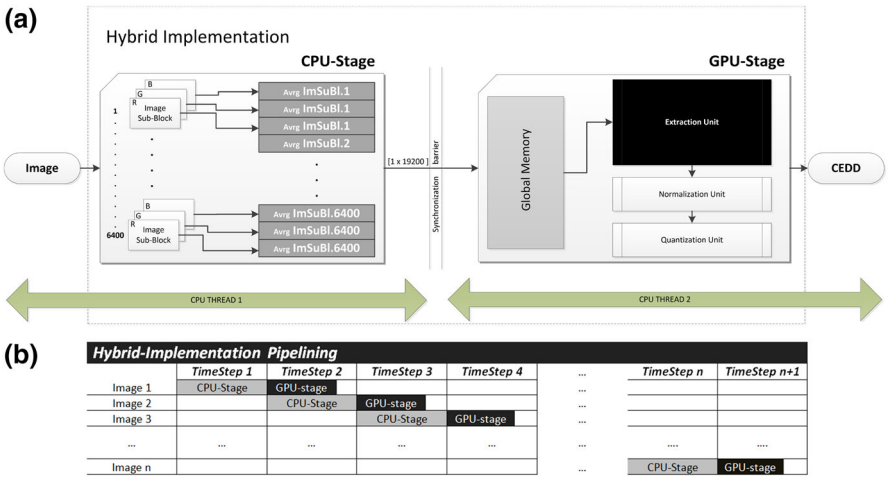

Fig. 7 **a** The hybrid implementation flowchart. **b** The hybrid implementation pipelining

Table 2 The features of the four hardware setups

	Setup1	Setup2	Setup3	Setup4
CPU model	Intel pentium dual-core	Intel core i5	Intel core i5	Intel core i7
Clock rate (GHz)/Mem. Bits (bit)/Gflops	2.20/32/16	3.20/64/49	2.60/64/42	4.10/64/53
GPU model	GeForce G 103M	GeForce 8400 GS	GeForce GT 620M	Quadro 4000
CUDA capability	1.1	1.1	2.1	2
Multiprocessors/cores per MP	1/8	1/8	2/48	8/32
Max resident blocks per MP	8	8	8	8
GPU clock rate (GHz)	1.60	1.62	1.25	0.95
Memory clock rate (Mhz)/bus width (bit)	500/64	400/64	900/64	1,404/256
Max threads per block/per MP	512/768	512/768	1,024/1,536	1,024/1,536
Warp size/GPU Gflops	32/38	32/43	32/240	32/486.4

6 Experimental results

The two implementations of the CEDD indexing method (GPU-only and Hybrid) along with the original CPU-only (c# official implementation) method were tested on four different computer systems summarized in Table 2. The different combinations of CPU and GPU technologies will highlight the efficiency of the proposed models. In particular, the first setup that consists of relatively weak technologies for both CPU and

Table 3 Frames per second indexed per image dimension, per implementation and per setup

Image dim.	Setup1			Setup2		
	GPU-only	Hybrid	CPU-only	GPU-only	Hybrid	CPU-only
640 × 480	**25.63**	***28.68***	8.86	**28.54**	***32.05***	13.39
800 × 600	19.34	*20.43*	3.67	23.28	***25.57***	9.79
1,024 × 768	11.83	*13.88*	2.70	12.51	*15.70*	5.35
1,280 × 1,024	*9.38*	9.14	1.46	8.63	*9.96*	3.47
1,600 × 1,200	*6.23*	6.12	1.08	5.32	*6.63*	2.22
2,048 × 1,536	*4.53*	3.80	0.60	4.21	*4.31*	1.29
2,048 × 2,048	2.66	*2.72*	0.44	3.63	*3.88*	1.19
Average	11.37	*12.11*	2.69	12.30	*14.01*	5.24
Image dim.	Setup3			Setup4		
	GPU-only	Hybrid	CPU-only	GPU-only	Hybrid	CPU-only
640 × 480	***52.20***	**26.67**	11.13	***288.10***	**39.92**	**28.58**
800 × 600	***49.01***	23.66	9.16	***244.53***	**28.07**	18.97
1,024 × 768	***29.51***	15.13	4.24	***185.75***	18.52	10.22
1,280 × 1,024	*16.72*	9.74	2.39	***102.87***	10.26	5.60
1,600 × 1,200	*10.97*	6.23	1.81	***69.88***	6.97	4.50
2,048 × 1,536	*6.72*	4.65	1.04	***48.42***	4.39	2.44
2,048 × 2,048	*6.05*	3.54	0.79	***40.84***	3.47	1.84
Average	*24.45*	12.80	4.37	*140.06*	15.94	10.31

Bold values indicate real-time executions while italicized ones highlight the best framerate achieved per setup

GPU serves as a baseline experiment that evaluates the worst case scenario for all three methods. The second setup is armed with a more powerful CPU compared to Setup1, while the GPU model is of the same capability, with equal number of multiprocessors (MP), cores and thread per MP. This setup is employed to give an edge to the CPU-only and the Hybrid implementations in order to examine the benefits that derive —if any—from a parallelized scheme on a weak GPU. The third setup tries to even out the technologies for both CPU and GPU, to test all implementations on a contemporary computer system. The fourth and final setup, offers a powerful CPU and a significantly more advanced GPU compared to the other setups to fully explore the potential of the parallelized indexing method.

The experiments on each setup were carried out for seven different image sizes, ranging from VGA 640 × 480 pixels up to 2,048 × 2,048 pixels. Please note that the execution time depends solely on the width and the height of the image (frame) to be indexed as described earlier in Sect. 4.

Table 3 number of descriptors extracted per image dimension, per setup and per implementation. In order to obtain robust results the indexing challenge comprised 1,000 images and was repeated 10 times per setup. The average execution time of those 10 iterations was used to calculate the Frames/s value. Table 4 illustrates the Speed-

Table 4 Achieved speed-up% per image size, per implementation and per setup

Image dim. (%)	Setup1		Setup2		Setup3		Setup4	
	GPU-only (%)	Hybrid (%)	GPU-only (%)	Hybrid(%)	GPU-only (%)	Hybrid (%)	GPU-only (%)	Hybrid (%)
640 × 480	289.40	323.70	213.10	239.40	247.10	126.20	1,008.10	139.70
800 × 600	527.00	556.80	237.60	261.10	535.20	258.40	1,289.40	148.00
1,024 × 768	438.40	514.40	234.10	293.70	696.20	356.90	1818.20	181.30
1,280 × 1,024	640.80	624.60	248.90	287.10	698.70	406.90	1,838.50	183.30
1,600 × 1,200	576.20	566.00	239.20	298.20	606.60	344.40	1,552.40	154.80
2,048 × 1,536	**755.70**	634.20	327.10	**334.70**	644.40	446.00	1984.10	179.70
2,048 × 2,048	603.10	616.10	305.20	325.90	**763.20**	446.60	**2,221.40**	188.70

Bold values indicate the implementation that achieves the best speed-up for each setup

up%. Speed-up refers to how much a parallel algorithm is faster than the corresponding sequential algorithm and is calculated as: $S_p = T_1/T_p$. p is number of processors, T_1 the execution time of the sequential algorithm and T_p is the execution time of the parallel algorithm with p processors.

Real-time indexing (i.e., at least 25 fps for VGA frame sizes) is achieved on all setups for both proposed implementations (GPU-only and Hybrid). More specifically, for the first setup the experiments prove that both implementations that employ the GPU manage to significantly speed-up the indexing process (up to 7.5 times). Furthermore, an important observation can be obtained through this setup; pipelining the procedure and exploiting all available resources through the proposed hybrid implementation allows for better execution time for small frame sizes. Generally, the GPU performs better when the device occupancy is high. A clear example of the aforementioned statement is observed in our first setup. The overall available threads to be activated in parallel are 768. The necessary threads per Thread Block due to the Reduction method that is employed for the summation of a Sub-Block's values are calculated as follows:

$$\begin{aligned} &\text{Num of Threads} = 2^n, \quad \text{where} \quad n \in N \\ &\text{Num of Threads} \leq \text{Max Threads per Block} \end{aligned} \tag{1}$$

Moreover, the maximum number of activated in parallel Thread Blocks is 8. Thus, in order to achieve an efficient GPU implementation on the first setup the following formulas must be met:

$$\frac{\text{Max Threads per MP} \times \text{MP}}{\text{Num of Threads}} \in N^* \tag{2}$$

$$\begin{aligned} A &= \text{Max Resident Blocks per MP} \times \text{MP} \\ A_t &= \text{Num of Threads} \times A \\ A_t &\leq \text{Max Threads per MP} \times \text{MP} \end{aligned} \tag{3}$$

The three last frame sizes meet these criteria and allow the GPU-only implementation to perform slightly better.

The advantages of the pipelined hybrid approach are more evident in Setup2. The powerful CPU combined with a weak GPU allows for the hybrid implementation not only to achieve real-time performance for even larger frame sizes than before (800×600 pixels), but also outperforms the GPU-only implementation on all occasions. The effect that the higher thread occupancy has on the execution time is still noticeable for the three last frame sizes. The GPU-only implementation manages to narrow the performance gab up to a point where it directly competes with the efficiency of the hybrid model.

The rest of the setups focus on highlighting the advantages of parallel programming on powerful GPUs. In Setup3 the GPU has a Computational Capability $2.\times$ and presents therefore higher performance with a total of $1{,}536 \times 2$ available Max threads, 1,024 Max threads/block and 16 maximum resident blocks while Setup4 has an impressive total of eight available MPs which enables a higher parallel computational power through the richer resources. Starting our comments on the third setup, the GPU-only implementation presents a constant higher performance against the other

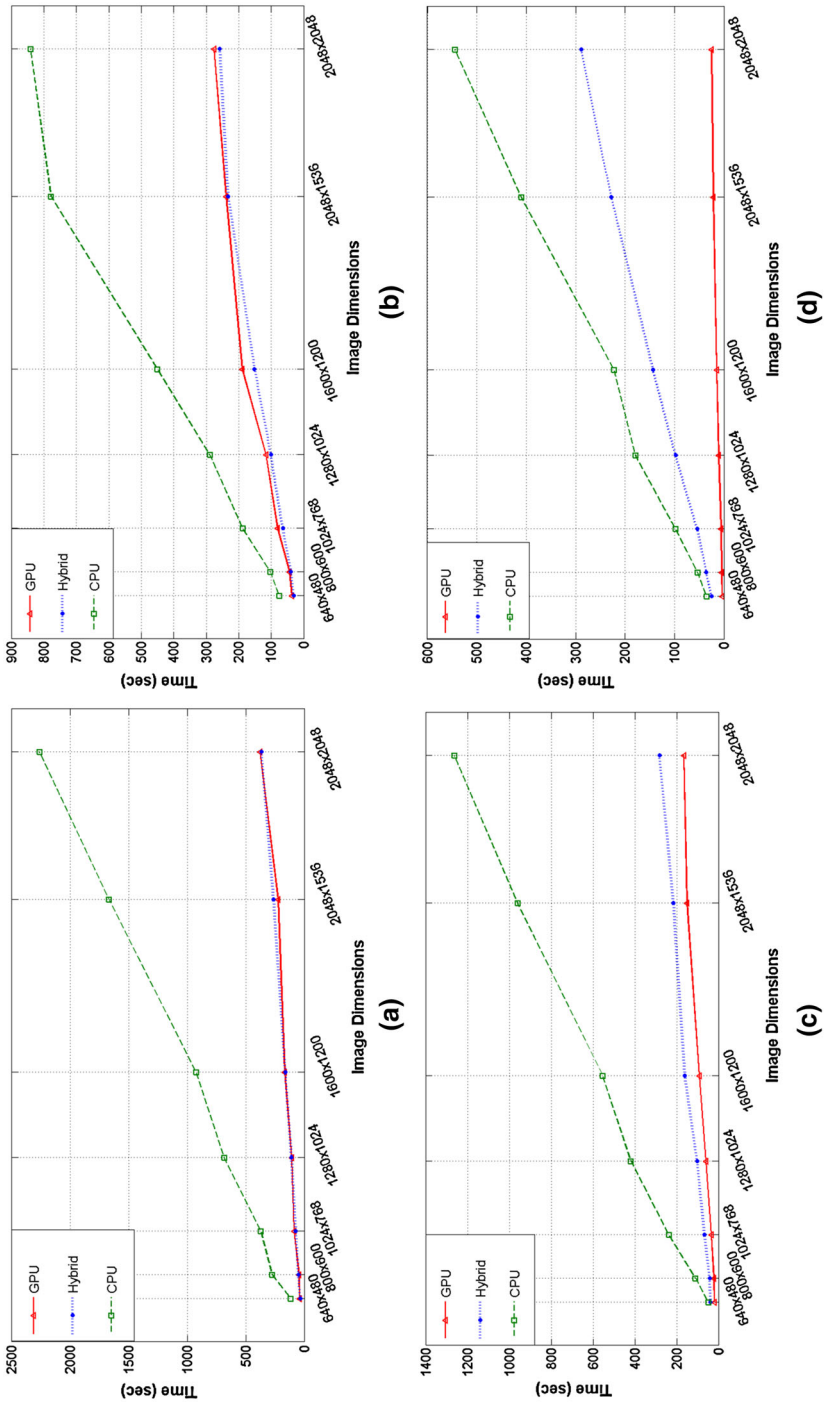

Fig. 8 Average indexing time for 1,000 images for (**a**) Setup1 , **b** Setup2, **c** Setup3 and **d** Setup4

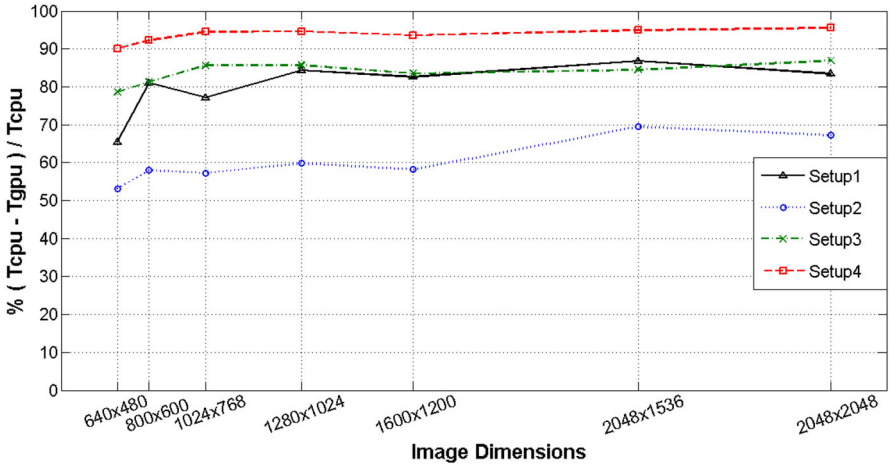

Fig. 9 Execution time improvement over all setups

models. Real-time indexing of at least 25 frames per second is achieved for even larger frame sizes of 1,024 × 768 pixels. As before, the frame sizes that meet the criteria set by formulas 1–3 take advantage of the available resources and therefore the highest speed-up percentage occurs for the 2,048 × 2,048 pixels frame size (accelerating the procedure 7.6 times).

Finally, the last setup which is armed with a very powerful GPU, was employed to show off the power of the GPU-only implementation. The available resources allow for real-time indexing of all the tested frame sizes. The achieved speed-up percentage is of a much greater order of magnitude compared to all other setups and implementations, as expected (accelerating the image indexing 22.2 times). The calculated average frames-per-second value over all tested frame sizes states that in the case of databases that consist of varying images sizes of the most commonly used formats, the GPU-only implementation is able to losslessly index about 140 images per second.

Figure 8 depicts the indexing time obtained for 1,000 images per image size per setup. As expected, the indexing time that the CPU implementation requires, increases proportionally along with the image sizes for all setups. The Hybrid implementation presents an analogical linear behavior, because the execution time, due to the pipelined scheme employed, is essentially the time-step of the summation process that is handled by the CPU.

The indexing time of the GPU-only implementation is very closely dependent on the available resources that the technology offers. Moreover, a number of other factors that affect the overall device occupancy can change the execution time of the GPU-only method and occasionally break the expected proportionally behavior of the achieved accelerations as image sizes increase. Figure 9 presents the execution time improvement over all setups when indexing is handled by the GPU. The improvement of the indexing execution time remains consistent over all image sizes per setup. The more abundant the GPU resources are, the more stable the improvement. When the GPU resources are limited (Setup1, Setup2) the outcome is subject to the device occupancy, and thus the fluctuations in the experimental results.

The experimental results confirm the great acceleration that is achieved when parallelizing the indexing method. The frame rate (i.e., execution time per image size) obtained by the strongest CPU model (Setup4) are always lower than the corresponding frame rate achieved by employing even the weakest GPU (Setup1). As anticipated, the proposed Hybrid implementation makes excellent use of the available computational resources by passing the summation part of the algorithm, which is by default a non-fully parallelizable procedure, to the CPU in a pipelined scheme. Thus, in cases where the GPU offers slim resources (Setup1, Setup2) the Hybrid implementation surpasses the limitations by simultaneously occupying the CPU and in many cases outperforms both CPU-only and GPU-only approaches.

7 Conclusion

When planning our implementation strategy two aspects were taken under consideration; locating the parallelization bottlenecks and making the implementation cost-effecting, i.e., comprehending the computational resource limitations that may exist. Two approaches were proposed in this paper. The first one parallelizes the whole method exclusively using the GPU, while the second one splits the procedure employing both CPU and GPU simultaneously in a pipelined scheme. It is important to note that the extracted descriptors from the CPU, the GPU and the Hybrid implementation match with no deflection in accuracy. Experiments were carried out on four different computational setups assembled to test the implementations on different combinations of available resources.

Real-time indexing was achieved by both proposed GPU involving implementations in all occasions. Real-time or higher frame rates were recorded for even higher resolutions, depending on the computational resources. The GPU-only proposed technique gained up to 22 times acceleration when compared to the CPU-only implementation, while the Hybrid implementation achieved a peak of 6.34 speed-up. Overall, the Hybrid approach excels when the available CPU technology is advanced compared to the GPU technology employed. Thus, we propose that a hybrid architecture should be considered when a method presents parts that are by default not fully parallelizable but can be independently assigned to the CPU in order to improve the total execution time. Moreover, in cases where the input data consists of a collection of items (e.g., image databases or video flow) a pipelined strategy, as the one proposed in this paper, where computations are distributed to all available resources can also lead to a low-cost and efficient solution.

Acknowledgments This research has been co-financed by the European Union (European Social Fund-ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF), Research Funding Program: Heracleitus II. Investing in knowledge society through the European Social Fund.

Appendix

The following Tables 5, 6, 7, and 8 are illustrating the step-by-step timing (in seconds for 1,000 images) of the GPU-only implementation.

Table 5 Step-by-step times obtained by Setup1

	640 × 480	800 × 600	1,024 × 768	1,280 ×1,024	1,600 × 1,200	2,048 × 1,536	2,048 × 2,048
CPU							
Read image	19.52	23.85	54.29	64.67	78.93	127.15	194.94
Read image (% exec. time)	50	46	64	61	49	58	52
GPU							
SumSubBlocks	12.51	18.17	21.36	34.97	74.61	86.76	173.04
SumBlocks	0.35	0.52	0.60	0.41	0.48	0.48	0.77
SumSubBlocks + SumBlocks (% exec. time)	66	67	73	84	92	93	96
ToYIQ	0.24	0.31	0.30	0.23	0.24	0.23	0.45
ToHSV + Fuzzy10	1.30	1.33	1.33	1.32	1.30	1.33	1.50
ToHSV + FuzzyBrightness	0.33	0.36	0.38	0.31	0.32	0.31	0.47
ApplyMasks + CreateTextureVector	0.31	1.33	0.40	0.34	0.32	0.32	0.36
Create24BinColorVector	0.76	0.90	0.88	0.74	0.72	0.72	0.77
CreateCEDD144Bin	1.60	2.25	2.23	1.61	1.55	1.58	1.60
Sum1600TO2	1.83	2.42	2.42	1.81	1.78	1.78	1.81
Sum2TO1	0.13	0.14	0.17	0.15	0.14	0.14	0.16
ApplyQuantization	0.14	0.15	0.15	0.12	0.14	0.13	0.15
GPU − 1000 frames	19.49	27.86	30.22	42.00	81.61	93.77	181.07
GPU + CPU − total (1,000 frames)	39.01	51.71	84.51	106.66	160.54	220.92	376.00

Table 6 Step-by-step times obtained by Setup2

	640 × 480	800 × 600	1,024 × 768	1,280 × 1,024	1,600 × 1,200	2,048 × 1,536	2,048 × 2,048
CPU							
Read image	7.78	7.10	40.38	61.41	90.93	128.72	139.43
Read image (% exec. time)	22	17	51	53	48	54	51
GPU							
SumSubBlocks	13.45	20.82	24.52	39.96	82.33	92.23	120.60
SumBlocks	3.15	3.94	4.32	4.39	4.69	6.51	4.64
SumSubBlocks + SumBlocks (% exec. time)	61	69	73	82	90	91	92
ToYIQ	0.48	0.53	0.48	0.33	0.50	0.53	0.51
ToHSV + Fuzzy10	1.21	1.24	1.24	1.20	1.19	1.16	1.23
ToHSV + FuzzyBrightness	0.68	0.61	0.63	0.67	0.46	0.46	0.51
ApplyMasks + CreateTextureVector	0.58	0.52	0.55	0.51	0.46	0.46	0.49
Create24BinColorVector	1.18	1.21	1.33	1.17	1.19	1.07	1.18
CreateCEDD144Bin	3.09	3.14	3.22	3.04	2.97	2.95	3.07
Sum1600TO2	2.84	3.15	2.61	2.57	2.61	2.54	2.84
Sum2TO1	0.46	0.55	0.49	0.42	0.61	0.69	0.71
ApplyQuantization	0.13	0.17	0.16	0.16	0.14	0.15	0.17
GPU − 1000 frames	27.27	35.86	39.55	54.41	97.14	108.74	135.94
GPU + CPU − total (1,000 frames)	35.05	42.96	79.93	115.82	188.07	237.46	275.37

Table 7 Step-by-step times obtained by Setup3

	640 × 480	800 × 600	1,024 × 768	1,280 × 1,024	1,600 × 1,200	2,048 × 1,536	2,048 × 2,048
CPU							
Read image	13.89	15.28	28.03	53.46	82.34	140.05	149.90
Read image (% exec. time)	72	75	83	89	90	94	91
GPU							
SumSubBlocks	3.22	3.31	3.81	4.26	6.30	6.79	13.50
SumBlocks	0.12	0.10	0.10	0.08	0.09	0.09	0.08
SumSubBlocks+SumBlocks (% exec. time)	63	66	67	68	73	78	87
ToYIQ	0.07	0.07	0.09	0.04	0.40	0.11	0.05
ToHSV + Fuzzy10	0.37	0.24	0.26	0.22	0.39	0.25	0.25
ToHSV + FuzzyBrightness	0.07	0.13	0.09	0.05	0.18	0.08	0.14
ApplyMasks + CreateTextureVector	0.05	0.07	0.12	0.14	0.14	0.09	0.12
Create24BinColorVector	0.11	0.11	0.19	0.11	0.06	0.10	0.15
CreateCEDD144Bin	0.19	0.17	0.26	0.23	0.27	0.27	0.22
Sum1600TO2	0.94	0.79	0.84	1.04	0.79	0.89	0.89
Sum2TO1	0.06	0.10	0.07	0.10	0.05	0.09	0.04
ApplyQuantization	0.07	0.04	0.02	0.07	0.12	0.09	0.11
GPU − 1,000 frames	5.27	5.13	5.85	6.34	8.79	8.85	15.54
GPU + CPU − total (1,000 frames)	19.16	20.41	33.88	59.80	91.13	148.90	165.44

Table 8 Step-by-step times obtained by Setup4

	640 × 480	800 × 600	1,024 × 768	1,280 × 1,024	1,600 × 1,200	2,048 × 1,536	2,048 × 2,048
CPU							
Read image	1.60	1.78	3.32	7.39	10.69	17.06	18.17
Read image (% exec. time)	46	44	62	76	75	83	74
GPU							
SumSubBlocks	1.03	1.62	1.30	1.58	2.85	2.93	5.58
SumBlocks	0.05	0.00	0.08	0.05	0.05	0.08	0.05
SumSubBlocks+SumBlocks (% exec. time)	57	70	66	70	80	84	89
ToYIQ	0.03	0.03	0.06	0.05	0.03	0.05	0.02
ToHSV + Fuzzy10	0.13	0.17	0.08	0.09	0.11	0.11	0.17
ToHSV + FuzzyBrightness	0.05	0.02	0.03	0.03	0.02	0.02	0.05
ApplyMasks + CreateTextureVector	0.00	0.05	0.05	0.03	0.00	0.03	0.05
Create24BinColorVector	0.08	0.02	0.05	0.03	0.11	0.03	0.02
CreateCEDD144Bin	0.08	0.05	0.06	0.11	0.11	0.08	0.05
Sum1600TO2	0.38	0.31	0.32	0.27	0.22	0.22	0.31
Sum2TO1	0.03	0.03	0.03	0.05	0.08	0.02	0.00
ApplyQuantization	0.03	0.02	0.02	0.05	0.05	0.03	0.05
GPU − 1,000 frames	1.87	2.31	2.07	2.33	3.62	3.59	6.32
GPU + CPU − total (1,000 frames)	3.47	4.09	5.38	9.72	14.31	20.65	24.49

References

1. Datta R, Joshi D, Li J, Wang JZ (2008) Image retrieval: ideas, influences, and trends of the new age. ACM Comput Surv 40(2):5:1–5:60
2. Wetzel A (1997) Computational aspects of pathology image classification and retrieval. J Supercomput 11(3):279–293
3. Ren R, Collomosse J, Jose J (2011) A bovw based query generative model. In: Proceedings of the 17th international conference on advances in multimedia modeling. Volume Part I, ser. MMM'11, 2011, pp 118–128
4. Lux M, Chatzichristofis S (2008) Lire: lucene image retrieval: an extensible java cbir library. In: Proceeding of the 16th ACM international conference on multimedia. ACM, 2008, pp 1085–1088
5. Chatzichristofis S, Iakovidou C, Boutalis Y, Marques O (2013) Co.vi.wo.: color visual words based on non-predefined size codebooks. IEEE Trans Cybernet 43(1):192–205
6. Lazebnik S, Schmid C, Ponce J (2006) Beyond bags of features: spatial pyramid matching for recognizing natural scene categories. CVPR 2:2169–2178
7. Zagoris K, Chatzichristofis SA, Arampatzis A (2011) Bag-of-visual-words vs global image descriptors on two-stage multimodal retrieval. In: Proceedings of the 34th international ACM SIGIR conference on research and development in information Retrieval, pp 1251–1252
8. Amanatiadis A, Kaburlasos V, Gasteratos A, Papadakis S (2011) Evaluation of shape descriptors for shape-based image retrieval. IET Image Process 5(5):493–499
9. Sevilla J, Bernabe S, Plaza A (2014) Unmixing-based content retrieval system for remotely sensed hyperspectral imagery on GPUs. J Supercomput, pp 1–12
10. Park IK, Singhal N, Lee MH, Cho S, Kim CW (2011) Design and performance evaluation of image processing algorithms on gpus. IEEE Trans Parallel Distrib Syst 22(1):91–104
11. Antikainen J, Havel J, Josth R, Herout A, Zemcík P, Hauta-Kasari M, Zemcík P (2011) Nonnegative tensor factorization accelerated using GPGPU. IEEE Trans Parallel Distrib Syst 22(7):1135–1141
12. Zhu L, Jin H, Zheng R, Feng X (2013) Effective naive bayes nearest neighbor based image classification on GPU. J Supercomput, pp 1–29
13. Risojević V, Babić Z, Dobravec T, Bulić P et al (2013) A GPU implementation of a structural-similarity-based aerial-image classification. J Supercomput 65(2):978–996
14. van de Sande KEA, Gevers T, Snoek CGM (2011) Empowering visual categorization with the GPU. IEEE Trans Multimed 13(1):60–70
15. Alvarado R, Tapia JJ, Rolón C (2013) Medical image segmentation with deformable models on graphics processing units. J Supercomput, pp 1–26
16. Song B, Tang W, Nguyen T-D, Hassan MM, Huh EN (2013) An optimized hybrid remote display protocol using GPU-assisted m-JPEG encoding and novel high-motion detection algorithm. J Supercomput 66(3):1729–1748
17. López MB, Nykänen H, Hannuksela J, Silvén O, Vehviläinen M (2011) Accelerating image recognition on mobile devices using GPGPU. In:Proceedings of SPIE 7872:78720R
18. Amanatiadis A, Bampis L, Gasteratos A (2014) Accelerating image super-resolution regression by a hybrid implementation in mobile devices. In: Proceedings IEEE international conference on consumer electronics, pp 335–336
19. Nalpantidis L, Amanatiadis A, Sirakoulis G, Gasteratos A (2011) Efficient hierarchical matching algorithm for processing uncalibrated stereo vision images and its hardware architecture. IET Image Process. 5(5):481–492
20. Chatzichristofis S, Zagoris K, Boutalis Y, Papamarkos N (2010) Accurate image retrieval based on compact composite descriptors and relevance feedback information. Int J Pattern Recogn Artif Intell 24(2):207–244
21. Jiang Y, Xu X, Terlecky P, Abdelzaher T, Bar-Noy A, Govindan R (2013) Mediascope: selective on-demand media retrieval from mobile devices. In: Proceedings of the 12th international conference on information processing in sensor networks, ser. IPSN '13. New York, NY, USA: ACM, 2013, pp 289–300
22. Zha Z-J, Tian Q, Cai J, Wang Z (2013) Interactive social group recommendation for flickr photos. Neurocomputing 105:30–37
23. van Leuken RH, Pueyo LG, Olivares X, van Zwol R (2009) Visual diversification of image search results. In: WWW. ACM, 2009, pp 341–350

24. Jin X, Gallagher AC, Cao L, Luo J, Han J (2010) The wisdom of social multimedia: using flickr for prediction and forecast. In: ACM Multimedia, 2010, pp 1235–1244
25. Daras P, Semertzidis T, Makris L, Strintzis MG (2010) Similarity content search in content centric networks. In: ACM multimedia, 2010, pp 775–778
26. Iakovidou C, Anagnostopoulos N, Kapoutsis AC, Boutalis YS, Chatzichristofis SA (2014) Searching images with MPEG-7 (& mpeg-7-like) powered localized descriptors: the SIMPLE answer to effective content based image retrieval. In 2014 12th International workshop on content-based multimedia indexing (CBMI), Klagenfurt, Austria, June 18–20(2014), 2014, pp 1–6. [Online]. doi:10.1109/CBMI.2014.6849821
27. Lux M, Marques O, Schoffmann K, Boszormenyi L, Lajtai G (2010) A novel tool for summarization of arthroscopic videos. Multimed Tools Appl 46(2–3):521–544
28. Rafailidis D, Manolopoulou S, Daras P (2013) A unified framework for multimodal retrieval. Pattern Recogn 46(12):3358–3370
29. Piras L, Giacinto G (2012) Synthetic pattern generation for imbalanced learning in image retrieval. Pattern Recogn Lett 33(16):2198–2205
30. Vallet D, Cantador I, Jose JM (2013) Exploiting semantics on external resources to gather visual examples for video retrieval. Int J Multimed Inf Retriev 2(2):117–130
31. Daras P, Manolopoulou S, Axenopoulos A (2012) Search and retrieval of rich media objects supporting multiple multimodal queries. IEEE Trans Multimed 14(3–2):734–746
32. Yu J, Jin X, Han J, Luo J (2011) Collection-based sparse label propagation and its application on social group suggestion from photos. ACM TIST 2(2):12
33. Chatzichristofis S, Boutalis Y (2008) CEDD: color and edge directivity descriptor: a compact descriptor for image indexing and retrieval. LNCS, Computer Vision Systems
34. Wang J, Li J, Wiederhold G (2001) Simplicity: semantics-sensitive integrated matching for picture libraries. IEEE Transactions on pattern analysis and machine intelligence, pp 947–963
35. Schaefer G, Stich M (2004) UCID-an uncompressed colour image database. Storage and retrieval methods and applications for multimedia 2004, vol 5307, pp 472–480
36. Chatzichristofis S, Boutalis Y (2010) Content based radiology image retrieval using a fuzzy rule based scalable composite descriptor. Multimed Tools Appl 46:493–519
37. Chatzichristofis S, Arampatzis A, Boutalis Y (2010) Investigating the behavior of compact composite descriptors in early fusion, late fusion and distributed image retrieval. Radioengineering 19(4):725
38. Chatzichristofis SA, Boutalis YS, Lux M (2010) SpCD–spatial color distribution descriptor. A fuzzy rule based compact composite descriptor appropriate for hand drawn color sketches retrieval. In: ICAART, 2010, pp 58–63
39. Manjunath B, Ohm J, Vasudevan V, Yamada A (2001) Color and texture descriptors. IEEE Trans Circuits Syst video Technol 11(6):703–715
40. Huang J, Kumar S, Mitra M, Zhu W (2001) Image indexing using color correlograms. US Patent 6,246,790, 12, pp 1–16
41. Thomee B, Bakker EM, Lew MS (21010) Top-surf: a visual words toolkit. In ACM multimedia, 2010, pp 1473–1476
42. Sartori J, Kumar R (2013) Branch and data herding: reducing control and memory divergence for error-tolerant gpu applications. IEEE Trans Multimed 15(2):279–290
43. van der Laan WJ, Jalba AC, Roerdink JB (2011) Accelerating wavelet lifting on graphics hardware using CUDA. IEEE Trans Parallel Distrib Syst 22(1):132–146
44. Li R, Saad Y (2013) Gpu-accelerated preconditioned iterative linear solvers. J Supercomput 63(2):443–466
45. Thibault JC, Senocak I (2012) Accelerating incompressible flow computations with a pthreads-CUDA implementation on small-footprint multi-GPU platforms. J Supercomput 59(2):693–719
46. Cano A, Luna JM, Ventura S (2013) High performance evaluation of evolutionary-mined association rules on GPUS. J Supercomput 66(3):1438–1461