Color and Edge Directivity Descriptor on GPGPU

C. Iakovidou, L. Bampis, S. A. Chatzichristofis, Y. S. Boutalis and A. Amanatiadis,
Department of Electrical and Computer Engineering, DUTH, Greece
(ciakovid, loukbabi, schatzic, ybout, aamanat)@ee.duth.gr

Abstract— **Image indexing refers to describing the visual multimedia content of a medium, using high level textual information or/and low level descriptors. In most cases, images and videos are associated with noisy and incomplete user-supplied textual annotations, possibly due to omission or the excessive cost associated with the metadata creation. In such cases, Content Based Image Retrieval (CBIR) approaches are adopted and low level image features are employed for indexing and retrieval. We employ the Colour and Edge Directivity Descriptor (CEDD), which incorporates both colour and texture information in a compact representation and reassess it for parallel execution, utilizing the multicore power provided by General Purpose Graphic Processing Units (GPGPUs). Experiments conducted on four different combinations of GPU-CPU technologies revealed an impressive gained acceleration when using a GPU, which was up to 22 times faster compared to the respective CPU implementation, while real-time indexing was achieved for all tested GPU models.**

I. Introduction

The area of Content Based Image Retrieval (CBIR) has seen a steady train of improvements in performance over the last decade [1]. The main focus of any CBIR method is to capture the rich information that images hold and vectorize it building its descriptor, so as to allow fast indexing and meaningful retrieval for the user.

The wide spread of affordable image and video capturing devices led to a rapid growth of multimedia databases in areas such as private life, journalism, medicine and tourist attraction, to name a few. Thus, in real life scenarios, description methods are not evaluated solely by their achieved performance but by their efficiency, as well.

Accelerating the descriptor-extraction procedure to the point where low level features can be described and incorporated in a file's header as it is captured and becomes part of a collection, is a matter of great importance. For instance, large image repositories such as *Flickr*, *facebook* and *Dropbox*, where millions of images are uploaded daily, will be able to index the images as they become part of their databases. The contribution of an implementation that achieves fast extraction of a descriptor is more evident when dealing with videos. According to 2014 YouTube's statistics[1], 100 hours of video are uploaded every minute. Automatic video annotation and summarization could be achieved through a descriptor extraction implementation that would allow real-time indexing of the frames.

Acknowledging the restrictions on computational resources that apply due to the massive amount of data involved in CBIR, recent methods are focused on producing compact vector representations. However, despite all the algorithmic efforts towards this direction, it is clear that useful acceleration without performance degradation can only be achieved through parallel processing.

Graphics Processing Units (GPUs) were first introduced to handle graphics primitives. However, the rapid evolution of the NVIDIA Compute Unified Device Architecture (CUDA) API [2], gave access to researchers from many varying fields, to the powerful parallel architecture of GPUs. Since GPUs are primarily employed for graphics, their configuration is ideal for parallelizing image processing algorithms.

Thus, multiple General-Purpose GPU (GP-GPU) implementations of existing image processing, indexing and categorization methods, have been proposed in recent literature. All report important acceleration and improved efficiency achieved by passing computations to the GPU, carefully following the architectural principles dictated by the GPU model.

In [3] the authors propose a compact histogram representation which applies replication and padding for optimizing the voting process in shared memory. They manage to minimize position conflicts by forcing adjacent threads to vote for different sub-histograms and propose the use of padding for reducing the amount of bank conflicts. A solution for building mosaic images of printed documents from frames selected from VGA resolution video captured from a mobile device, is presented in [4]. They utilize the device's GPU to perform the most demanding computations, concluding that the deep understanding of the data and the possible parallelizations becomes the crucial step for successful designs. A GPU implementation of Local Binary Pattern feature extraction is attempted in [5]. Results show that parallelizing a method that by design consists of processes that can be handled independently, ensures great achieved efficiency even when employing older graphics cards. However, in [6] authors attempt the same implementation on a mobile device's GPU and conclude that its parallelization power is insufficient. Taking a more general outlook, authors in [7] explored the design and implementation issues of image processing algorithms on GPUs. Selecting four major domains 3D shape reconstruction, feature extraction, image compression, and computational photography, they try to employ metrics that will allow the prediction of the effectiveness of a given image processing problem for the parallel implementation, and observed that speed-up varies extensively depending on the characteristics of each algorithm.

All the authors are with the Democritus University of Thrace, DUTH.
[1]https://www.youtube.com/yt/press/statistics.html

Studying the related literature, it is clear that GPUs provide the much needed parallel computing power for feature extraction and image description methods. Both for personal computer applications or if used to make smart-phones smarter [8] by allowing applications to compute complex computations efficiently, GPGPUs consist an attractive solution. To produce desired results, in terms of efficiency and acceleration, the method selected to be parallelized needs to consist of independent procedures, while the implementation strategy must be carefully selected to fit the specifics of the GPU and the data involved.

In this paper, we focus on real-time image indexing for CBIR tasks. A *real-time image processing system* requires at least 25 frames per second [9]. In the same direction, we define as *real-time indexing* the ability of a system to extract a descriptor vector during capturing a VGA frame stream of 25fps. We employ the Color and Edge Directivity Descriptor (CEDD) [10] which incorporates both color and texture information. CEDD is a compact descriptor that achieves successful trade-off between effectiveness and efficiency and has been widely used in recent literature [11], [12], [13], [14]. Moreover, it divides the indexing problem into many identical and independent sub-problems making it ideal for parallelization. Taking into account the principles and the constrains of the CUDA architecture, we reassess and design a parallel equivalent which is tested on four different computational setups with varying CPU and GPU technologies.

II. CEDD - Color and Edge Directivity Descriptor

This section provides a brief presentation of the structural elements of the Color and Edge Directivity Descriptor. For a more detailed description, kindly refer to [10], [15].

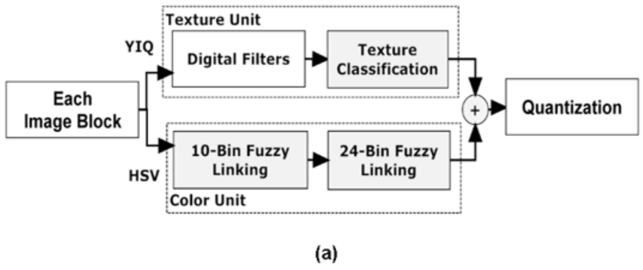
CEDD is a global feature image descriptor[2], designed for CBIR tasks, that achieves good performance with relatively low size and storage requirements. CEDD divides an image of any size into 1600 rectangular image areas. Those Image-Blocks are then handled independently to extract their colour and texture information. Each Image-Block is represented by a quantized vector that captures its colour and texture attributes. When all 1600 Image-Block vectors have been calculated, they are combined (fused) to form a single Image vector. The final CEED descriptor is produced by normalizing and quantizing into 8 predefined levels the aforementioned Image vector. The following subsections provide a closer look at the *Colour Extraction* and the *Texture Extraction* Units.

A. *Colour Extraction Unit*

As depicted in Figure 1(a), each Image-Block enters the Colour Unit after its RGB values are converted into the HSV colour space.

Then, a two-staged fuzzy system is employed to produce a fuzzy linking histogram. Linking is defined as the combination of more than one histograms to a single one

[2]Recently, a local features' version of the CEDD descriptor was proposed in [16].



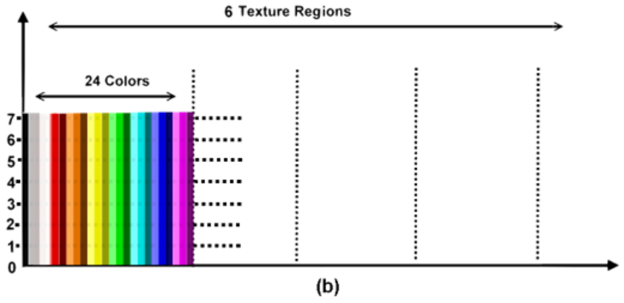


Fig. 1. (a) CEDD Implementation Flowchart, (b) Descriptor's Structure[10].

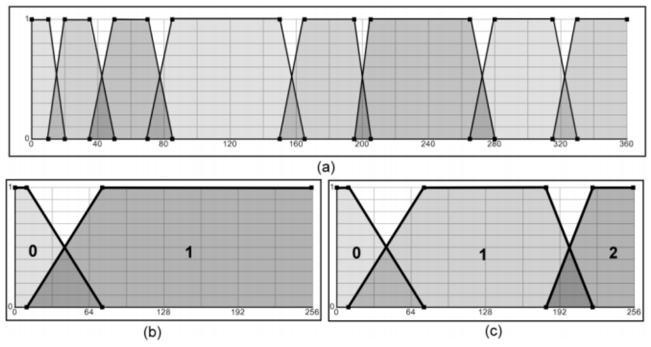


Fig. 2. Membership Functions of H (a) , S (b) and V (c) for the first stage of the fuzzy system [10].

[10]. The first stage of the fuzzy system has the three mean HSV channels of an Image-Block as inputs, and forms a 10-bins histogram output. The three inputs of the fuzzy system are described as follows: Hue (H) is divided into 8 fuzzy areas, Saturation (S) is divided into 2 fuzzy regions while the channel Value (V) is divided into 3 areas (kindly refer to Figure 2). The output of the fuzzy system is enabled by a set of 20 rules and returns a crisp value ranging from 0 to 1 (TSK like fuzzy system) to produce the 10-bins first-stage histogram. The first three bins represent Black, Grey and White, respectively, while the rest seven bins represent a preset colour each.

The second-stage fuzzy linking system (TSK) is responsible for adding the brightness value to the seven colours (Black, Grey and White are not computed). Again the S and V mean values of an Image-Block become fuzzy inputs, as illustrated in Figure 3. The output is a 3 bin histogram of crisp values, indicating if the colour will be characterized as light, normal or dark hued.

The two outputs (first and second stage histograms) are

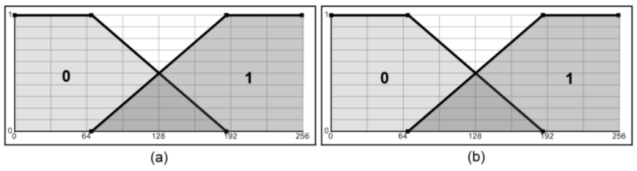


Fig. 3. Membership Functions for the S (a) and V (b) channels used in the second stage of the fuzzy system [10].

combined and the final 24-bin colour histogram is produced. Each bin represents a colour 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.

B. Texture Extraction Unit

In parallel with the Colour Unit, Image-Blocks enter the Texture Unit, after being converted to the YIQ colour space. For the extraction of the texture information the method employs the five digital filters proposed by the MPEG-7 Edge Histogram Descriptor-EHD [17] which represent five broadly grouped edge types: vertical, horizontal, 45 diagonal, 135 diagonal, and isotropic (Figure 4(a)), along with an additional Non-Edge filter. In order to employ the filters, each Image-Block must be subdivided into four Sub-Blocks. The value representing each Sub-Block is the mean value of the luminosity (Y) of the pixels consisting the Sub-Block.

The digital filters are applied and the obtained responses became inputs to the fuzzy mapping scheme, illustrated in Figure 4(b). In its essence, this mapping system is responsible for indicating which kinds of edges are present for every Image-Block. Please note that more than one edge types can be simultaneously present.

The normalized maximum responses (edge magnitudes) from the applied filters per Image-Block, are placed in the heuristic pentagon diagram (Figure 4(b)). Each value is placed along the line that pertains to the filter it emerged from. If that value is greater than the corresponding line's threshold, the Image-Block is classified in the respective type of edge. If none of the five thresholds are met the Image-Block is categorized as Non-Edge.

The Texture Unit produces a 6-bin vector output for each Image-Block. Every bin represents one of the five employed textures, while the first bin represents the Non-Edge case. When an edge type was found present in an Image-Block the corresponding bin is marked with "1". Otherwise it is marked as "0", producing the binary Image-Block texture vector.

C. Producing the CEDD descriptor

When the 24-bins colour histogram and the 6-bins texture vector have been calculated for an Image-Block, the two are combined and a 144-bins vector for every Image-Block is generated as follows: the bins are divided to six regions (that represent a different texture) of 24-bins each. According to the Image-Block's texture vector, and for those of its bins that were marked as "1", the respective region in the 144-bins vector is filled with the 24-bins colour histogram that was calculated for the Image-Block. 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 image's CEDD descriptor (Figure 1(b)) has been formed and will represent the visual content of the image in a compact and distinct fashion.

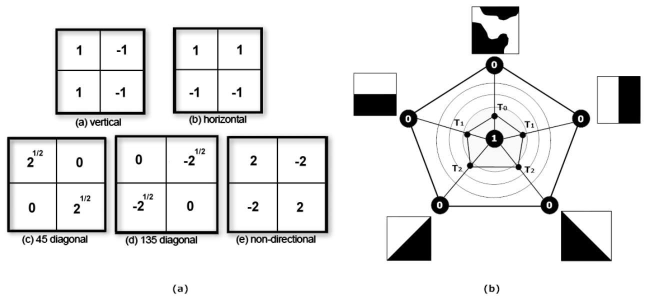


Fig. 4. (a) Filter Coefficients for Edge Detection, (b) Edge Type Diagram (Heuristic Pentagon Diagram) [10].

III. CUDA Architecture

In CUDA terms, the GPU is called device and the CPU that calls the CUDA functions is the host. The GPU architecture contains a large number of computing cores which can handle a large number of operations, simultaneously. The GPU is divided in a number of multicore processors named multiprocessors (MPs). Each multiprocessor is a set of processors with a single instruction multiple data (SIMD) architecture. Due to the SIMD nature of CUDA, at one time the threads must perform identical operations. [7], [18]. Otherwise, the computations will be partially serialized because different instructions must be executed in different clock cycles, leaving groups of threads idle [19].

A CUDA kernel function is a set of instructions that the device's threads will execute in parallel. The threads are organized in a two-level hierarchy, Thread Block and Grid. Every Grid is a set of Thread Blocks and every Thread Block is a set of threads. The maximum number of Thread Blocks and threads per Block that can be executed in parallel varies, and is defined by the GPU model.

A GPU also has an efficient memory architecture divided in global and local memories. Registers are local memory spaces assigned per processor. The threads belonging to the same Thread Block can share data through the Shared Memory. Threads from different Thread Blocks coordinate only through the Global Memory. a large and long-latency memory which has read/write operations. Accessing the Global Memory space is much slower, typically two orders of magnitude slower than floating point multiplication and addition [20], [19]. Global Memory read operations from threads whose `id` follows the memory alignment guidelines can be coalesced, leading to faster execution [2].

Generally, when a method demands the usage of the same data multiple times, the efficient strategy is to copy the data to Registers and access it from there, as long as this is possible. An informative summarization of the properties of the different types of memories and the suitability of CUDA can be found in [21].

IV. CUDA Implementation

Indexing images, whether dealing with an image collection or frames from a video stream, is a computationally demanding and time consuming process. GPUs are low cost, powerful processors, available on every personal computer that when utilized to handle computations can significantly improve the efficiency of a method. However, in order to benefit from the parallel computational power that they can offer, one needs to carefully select the appropriate method and design around the architectural principles of GPUs. In this section we propose and design a parallel equivalent of the original CEDD implementation. CEDD was selected because it has a parallel structure by design since it divides the indexing problem into many smaller independent and identical processes.

Figure 5 depicts the implementation's flowchart. From left to right the implementation consists of three successive stages: The Average RGB values are calculated in parallel for every Image Sub-Block, the colour and texture vectors per Image-Block are extracted and combined to form the Image-Blocks' descriptors, the descriptors are added to produce a single vector and normalized and quantized to form the final CEDD descriptor of the image.

1) Preparing the data to be forwarded to the GPU: Due to the absence of a direct communication channel between the hard drive (where the images are stored) and the GPU, the CPU is activated to read the images. Designing for the GPU implementation leads us early on to investigate about the preferred input data manipulation strategy. We explored two different approaches. First we forwarded the data in the format they were originally stored and read by the CPU. Then, we performed additional rearrangement of the data so that values belonging to an Image Sub-Block would be stored in neighbouring memory slots per RGB channel. Ideally, the accesses to the Global Memory that the threads perform should be coalesced. Thus, data that will be accessed by the same Thread Block should be sequential.

Reading the images takes up on average 62% of the total execution time. When further engaging the CPU to rearrange the data, and even though the GPU part of the implementation achieved almost a 10% speed-up, the total CPU execution time (reading and rearranging the data) significantly impacted the overall execution time and therefore data rearrangement was abandoned.

2) Calculating the Average RGB Values: CEDD divides the image into 1600 Image-Blocks. The smallest structural unit, however, is defined by the texture extraction procedure which demands each Image-Block to be further divided into four equally sized Sub-Blocks. Thus, we must calculate the average R, G and B values of $4 \times 1600 = 6400$ blocks of pixels. For this to be done 6400×3 Thread Blocks are activated.

To achieve maximum parallelization each Thread Block should contain a number of threads equal to the number of pixels in the Sub-Block. The size of a Sub-Block is directly related to the actual 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 procedure depends mainly on the employed technique, but can benefit from input data organized in a suitable manner, a scenario that we explored but found unfit for our implementation. A popular technique that semi-parallelizes the summation problem follows the Reduction process [22], [23] architecture. It is a tree-based approach where every involved thread sums two values. The outcome becomes the input data of the next step and the process continues until only two values are left for one thread to sum.

In our design we assign a device Thread Block for every RGB channel of an Image Sub-Block. This one-to-one assignment leads to a maximum image size of about 6.5 Mega-Pixels. This upper limit is defined by our weakest GPU of 512 threads per Thread Block. Since the Reduction process assigns one thread for the summation of two values, the maximum Image Sub-Block size can be up to 1024 pixels. The maximum image size can be further increased if more than one Thread Blocks are assigned per Image Sub-Block. If so, the output of every Thread Block responsible for the same Sub-Block must be summed, as well. In any case, the optimum exploitation of the resources during 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.

In our method, the number of values to be summed is defined by the size of the Image Sub-Block. Thus, when the size of the input image is small, the utilization is not optimum because many available threads per Block remain idle. This is evident through our experimental results presented in the next section. By the end of the summation process the method can be divided into 1600 independent problems.

3) Color Extraction Unit - TSK Fuzzy System Implementation: Every four Sub-Blocks that form an Image Block enter the Texture and the Color Units. We will begin by describing the Color Extraction Unit (Figure 1(a)). The previously calculated average RGB values per Sub-Block are used to find the Image Block's average red, green and blue values. Those values are then converted into the HSV (Hue, Saturation, Value) colour 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 sys-

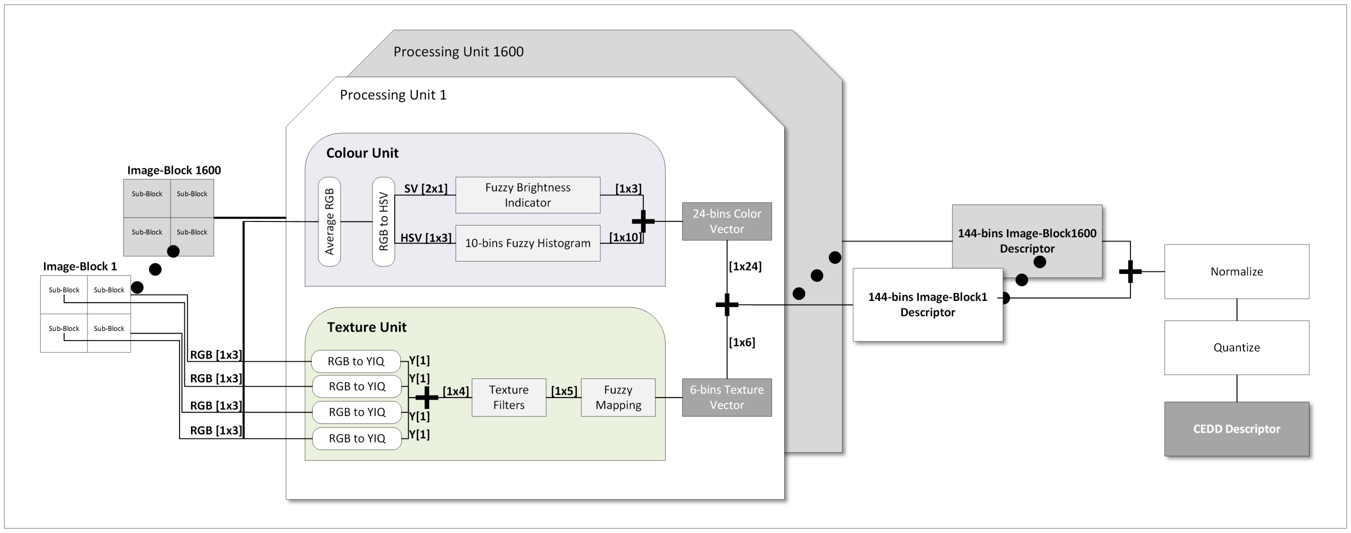


Fig. 5. GPU Implementation Flowchart.

tems share the same architectural principles in the GPU implementation. We will proceed by describing our Parallel version of a Participation Identifier (PPI).

The PPI technique that we employ ensures that the requested output will be produced in one time-step. The task in hand is to indicate whether a value belongs to a specific sub-region of a range of values. 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. 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 involved. Due to the SIMD architecture, at one time, threads must perform identical operations. PPI successfully adapts to this principle. In every time-step the output is produced and a new set of identical operations can take place in the next time-step.

We applied the PPI to the Fuzzy systems implementation. One thread per membership function region is activated. Given 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, 1600×48 threads are activated. Likewise, for the 24-bin Fuzzy Histogram (Fuzzy Brightness Indicator), 4 threads are employed simultaneously per Image-Block (i.e 1600×4 for the whole image).

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 Colour Vector. The following pseudo code describes the combination process.

```
**Data**: x $\in [0,7] \cup N$, y $\in [0,2] \cup N$, z=0
**for** *24 threads with threadIDx.x/y/z* **do**
    **if** *threadIDx.x==0* **then**
        24Bin_Color_Vector[threadIDx.x $\times$ 3 + threadIDx.y]=
        10Bin_Color_Vector[threadIDx.y]
    **else**
        24Bin_Color_Vector[threadIDx.x $\times$ 3 + threadIDx.y]=
        10Bin_Color_Vector[threadIDx.x +2] $\times$
        3Bin_Brightness_Vector[threadIDx.y]
    **end**
**end**
```

Algorithm 1: The Color Combination Process.

According to CEDD, the first three bins (Black, Grey, White) from the 10-bin Histogram Unit are forwarded unchanged to become the first three components of the 24-bins Colour Vector. As for the remaining bins, every colour bin is multiplied with all three brightness bins to produce a three-shaded colour representation. In CUDA, we enable 24 threads to process as follows: 3 threads are responsible for transferring the first three 10-bin values, and another set of three threads per colour bin (i.e. 3 threads for every one of the remaining seven colours) is enabled to execute the multiplication of the brightness values with the colour. All computations are implemented in parallel and the 24-bins Colour Vector is formed at once.

4) Texture Information Extraction Unit: As in the Colour Unit, the inputs of the Texture Extraction Unit are the four Image Sub-Blocks that comprise an Image Block. We will continue describing the implementation of the Texture Extraction Unit for one Image Block (4 Image Sub-Blocks). All 1600 Image Blocks that compose the input image are processed in parallel by identical kernels.

The RGB average values of each Sub-Block are converted into the YIQ colour space. All four extracted Y (Luminance) values become a $[1 \times 4]$ vector used for the application of the texture filters. A single thread is enabled per filter. Thus, a total of 5 threads are executed simultaneously to calculate the response of the five different texture filters. The output enters the Fuzzy Mapping system.

This system is responsible for identifying which kind of texture is located in an Image Block. First, we locate the highest scoring filter. If a predefined threshold (hereby refereed to as T) is not met by the highest scoring filter, the whole Image Block is categorized as Non-Edge and no further computations take place. Otherwise, the corresponding response scores of the five filters are normalized and used to form the final Texture Vector.

A total of six threads are activated to carry out the computations. One is the Non-Edge detector, and the rest are the five filter-threads. The maximum found response value is stored in the shared device memory, in order for it to be available for all threads to access. 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 filter-threads mark their corresponding elements as "0". In any other case, 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 their threshold. If so, they affect the final Texture Vector by marking as "1" their respective element.

5) The CEDD Descriptor: The 24-bins Colour Vector and the 6-bins Texture Vector are combined to form a 144-bins vector that carries the colour and texture information of an Image-Block. The combination procedure of the two input vectors follows the method described earlier, when the 10-bin Fuzzy Histogram and the Fuzzy Brightness vectors were combined. All 144-bins vectors of every Image-Block are extracted in parallel. 144 threads for every one of the 1600 thread blocks are activated. Every thread is undertaking the task of multiplying one of the colour's bins with one of the texture's bin and store the result to the new vector. The produced vectors enter the Normalization Unit. They are summed into a single vector and normalized. For the restriction of the descriptor's length, a 3bits/bin quantization is used, resulting 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 8 threads, each one responsible for checking if the bin value belongs to the corresponding quantization region. Similarly to the TSK systems implementation, the quantized vector is extracted in a single time step.

When all bins have been quantized the new vector is the final CEDD descriptor of the image. Please note that the CEDD descriptor is produced without any quality degradation compared to the original CPU implementation.

V. Experimental Results

In order to highlight the efficiency of the proposed models, we tested the implementation on four different computer systems that consist of different combinations of CPU and GPU technologies (kindly refer to Table I) and calculated the achieved execution time and speed-up. Speed-up refers to how much a parallel algorithm is faster than the corresponding sequential algorithm and is calculated as: $S_p = T_1/T_p$ where p is number of processors, T_1 the is execution time of the sequential algorithm and T_p is the execution time of the parallel algorithm with p processors.

Seven different image sizes were employed, ranging from VGA 640 $\times$ 480 pixels up to 2048 $\times$ 2048 pixels. Please note that the execution time depends solely on the width and the height of the image (frame) to be indexed. As our implementation is based on parallelizing the procedure of forming one descriptor and not to create multiple descriptors in parallel, the achieved speed-up is going to remain the same regardless of the number of images in the database.

In order to obtain robust results, the indexing challenge comprised 1000 images of the WANG image database [24], which we resized to produce the different resolutions, and was repeated 10 times per setup. The average execution time of those 10 iterations was used to calculate the Frames/sec value. In Table II we included the standard deviation (stdv) of the iterations per image dimension and per setup. The small stdv values reveal the reliability of our implementation. Table III summarizes the number of descriptors extracted per image dimension, per setup and per implementation, evaluated by the obtained speed-up percentage.

As depicted in Table III, real-time indexing (i.e. at least 25fps for VGA frame sizes) is achieved on all setups. More specifically, for the first setup the experiments show that our implementation manages to significantly speed-up the indexing process (up to 7.5 times). Generally, the GPU performs better when the device occupancy is high. The achieved indexing time of the GPU implementation is very closely related to the available resources that the technology offers. The utilization strategy of those resources that we followed during the different stages of our method, occasionally breaks the expected proportionally behaviour of the achieved accelerations as image sizes increase.

To take a closer look into how the device occupancy impacts the speed-up we will focus on the first setup. The available threads to be activated in parallel are 768, according to the GPU used in this setup. The possible threads per Thread Block due to the Reduction method that is employed for the summation of a Sub-Block's values, demands the number of threads to be:

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

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

TABLE I

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	1404 / 256
Max Threads per Block / per MP	512 / 768	512 / 768	1024 / 1536	1024 / 1536
Warp Size / GPU Gflops	32 / 38	32 /43	32 / 240	32 /486.4

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

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

The three last frame sizes meet these criteria and allow the GPU implementation to perform better (in terms of speed-up) than frame sizes that are smaller but are not in-line with the specifics of the GPU model.

TABLE II

THE STANDARD DEVIATION CALCULATED FOR THE 10 EXPERIMENTAL RUNS, PER IMAGE DIMENSION, PER SETUP.

	Setup1	Setup2	Setup3	Setup4
Img Dim	**Stdv**	**Stdv**	**Stdv**	**Stdv**
640x480	0.0097	0.0076	0.0087	0.0066
800x600	0.0102	0.0083	0.0112	0.0068
1024x768	0.0087	0.0075	0.0085	0.0071
1024x1024	0.0095	0.0131	0.0162	0.0081
1600x1200	0.0097	0.0072	0.0079	0.0068
2048x1536	0.0082	0.0076	0.0086	0.0071
2048x2048	0.0092	0.0088	0.0089	0.0078

Setup2 is armed with a GPU similar in resources to setup1 but with a much more advanced CPU. Thus, even though the number of indexed images per second is slightly greater than setup1, the total speed-up percentage is less. In setup3 the GPU has a Computational Capability 2.$\times$ with a total of 1536 $\times$ 2 available Max threads, 1024 Max threads/block and 16 maximum resident blocks, and as expected performs better. In this setup real-time indexing of at least 25 frames per second is achieved for even larger frame sizes of 1024 $\times$ 768 pixels. As before, the frame sizes that meet the criteria set by formulas 1-3 utilize the available resources and therefore the highest speed-up percentage is reported for the 2048 $\times$ 2048 pixels frame size (7.6 times acceleration).

Setup4 has an impressive total of 8 available MPs which enables a higher parallel computational power through the richer resources. This setup allows us to fully show off the power and the potential of our GPU 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, accelerating the image indexing up to 22.2 times.

Finally, we would like to highlight that the experimental results confirm the great acceleration that is achieved when parallelizing the indexing method. Even employing the weakest GPU (setup1), the obtained frame rate is always greater compared to the strongest CPU model (setup4).

VI. CONCLUSION

Efficiently indexing images and videos has become a matter of great importance with on-line databases growing rapidly. In this paper we employed the CEDD descriptor, a lightweight, effective and widely used method which by design applies for parallelization.

Our implementation strategy was focused on locating the parallelization bottlenecks and designing the parallel equivalent taking under considerations the possible available computational resources from the user's end. Real-time indexing (25 frames/sec for VGA resolution) was achieved by the proposed GPU implementation in all tested combinations of CPU-GPU technologies. The potential of our implementation shone through when involving a powerful GPU, resulting into a 22 times acceleration when compared to the respective CPU implementation.

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.

REFERENCES

[1] R. Datta, D. Joshi, J. Li, and J. Z. Wang, "Image retrieval: Ideas, influences, and trends of the new age," *ACM Comput. Surv.*, vol. 40, no. 2, 2008.

[2] NVIDIA Corporation, "Cuda online programming guide," in *http://docs.nvidia.com/cuda/cuda-c-programming-guide/*, 2013.

[3] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "An optimized approach to histogram computation on gpu," *Mach. Vis. Appl.*, vol. 24, no. 5, pp. 899–908, 2013.

[4] M. B. López, J. Hannuksela, O. Silvén, and M. Vehviläinen, "Graphics hardware accelerated panorama builder for mobile phones," in *IS&T/SPIE Electronic Imaging*, 2009, pp. 72 560D–72 560D.

[5] G. Zolynski, T. Braun, and K. Berns, "Local binary pattern based texture analysis in real time using a graphics processing unit," *VDIBERICHT*, vol. 2012, p. 321, 2008.

[6] M. Lopez, H. Nykänen, J. Hannuksela, O. Silven, and M. Vehviläinen, "Accelerating image recognition on mobile devices using gpgpu," in *Proceedings of SPIE*, vol. 7872, 2011, p. 78720R.

TABLE III

Frames per Second and %Speed-up, Indexed per Image Dimension, per Implementation and per Setup.

	Setup1			Setup2			Setup3			Setup4		
Img Dim	**GPU**	**CPU**	**Speed-up**	**GPU**	**CPU**	**Speed-up**	**GPU**	**CPU**	**Speed-up**	**GPU**	**CPU**	**Speed-up**
640x480	25.63	8.86	289.4%	28.54	13.39	213.1%	52.50	11.13	247.1%	288.10	28.58	1008.1%
800x600	19.34	3.67	527.0%	23.28	9.79	237.6%	49.01	9.16	535.2%	244.53	18.97	1289.4%
1024x768	11.83	2.70	438.4%	12.51	5.35	234.1%	29.51	4.24	696.2%	185.75	10.22	1818.2%
1024x1024	9.38	1.46	640.8%	8.63	3.47	248.9%	16.72	2.39	698.7%	102.87	5.60	1838.5%
1600x1200	6.23	1.08	576.2%	5.32	2.22	239.2%	10.97	1.81	606.6%	69.88	4.50	1552.4%
2048x1536	4.53	0.60	**755.7%**	4.21	1.29	**327.1%**	6.72	1.04	644.4%	48.42	2.44	1984.1%
2048x2048	2.66	0.44	603.1%	3.63	1.19	305.2%	6.05	0.79	**763.2%**	40.84	1.84	**2221.4%**

[7] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C. W. Kim, "Design and performance evaluation of image processing algorithms on gpus," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 91–104, 2011.

[8] A. Amanatiadis and S. Chatzichristofis, "How smart are smartphones?: Bridging the marketing and information technology gap," *IEEE Consumer Electronics Magazine*, vol. 3, no. 4, pp. 51–54, 2014.

[9] R. Ureña, C. Morillas, and F. J. Pelayo, "Real-time bio-inspired contrast enhancement on gpu," *Neurocomputing*, vol. 121, pp. 40–52, 2013.

[10] S. Chatzichristofis, K. Zagoris, Y. Boutalis, and N. Papamarkos, "Accurate image retrieval based on compact composite descriptors and relevance feedback information," *International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI)*, vol. 24, no. 2, pp. 207–244, 2010.

[11] R. H. van Leuken, L. G. Pueyo, X. Olivares, and R. van Zwol, "Visual diversification of image search results," in *WWW*. ACM, 2009, pp. 341–350.

[12] X. Jin, A. C. Gallagher, L. Cao, J. Luo, and J. Han, "The wisdom of social multimedia: using flickr for prediction and forecast," in *ACM Multimedia*, 2010, pp. 1235–1244.

[13] P. Daras, T. Semertzidis, L. Makris, and M. G. Strintzis, "Similarity content search in content centric networks," in *ACM Multimedia*, 2010, pp. 775–778.

[14] M. Lux, O. Marques, K. Schoffmann, L. Boszormenyi, and G. Lajtai, "A novel tool for summarization of arthroscopic videos," *Multimedia Tools Appl.*, vol. 46, no. 2-3, pp. 521–544, 2010.

[15] S. Chatzichristofis and Y. Boutalis, "CEDD: Color and edge directivity descriptor: A compact descriptor for image indexing and retrieval," *LNCS, Computer Vision Systems*, pp. 312–322, 2008.

[16] C. Iakovidou, N. Anagnostopoulos, A. C. Kapoutsis, Y. S. Boutalis, and S. A. Chatzichristofis, "Searching images with mpeg-7 (& mpeg-7-like) powered localized descriptors: The simple answer to effective content based image retrieval," in *CBMI*, 2014, pp. 1–6.

[17] B. Manjunath, J. Ohm, V. Vasudevan, and A. Yamada, "Color and texture descriptors," *IEEE Transactions on circuits and systems for video technology*, vol. 11, no. 6, pp. 703–715, 2001.

[18] A. Herout, R. Josth, R. Juránek, J. Havel, M. Hradis, and P. Zemcík, "Real-time object detection on cuda," *J. Real-Time Image Processing*, vol. 6, no. 3, pp. 159–170, 2011.

[19] W. J. van der Laan, A. C. Jalba, and J. B. Roerdink, "Accelerating wavelet lifting on graphics hardware using cuda," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 132–146, 2011.

[20] M. Hussein, A. Varshney, and L. Davis, "On implementing graph cuts on cuda," in *First Workshop on General Purpose Processing on Graphics Processing Units*, vol. 2007, 2007.

[21] W.-M. Hwu, C. Rodrigues, S. Ryoo, and J. Stratton, "Compute unified device architecture application suitability," *Computing in Science & Engineering*, vol. 11, no. 3, pp. 16–26, 2009.

[22] NVIDIA Corporation, "Cuda samples," in *http://docs.nvidia.com/cuda/cuda-samples/*, 2013.

[23] H. Nguyen, *Gpu gems 3*, 1st ed. Addison-Wesley Professional, 2007.

[24] J. Wang, J. Li, and G. Wiederhold, "Simplicity: Semantics-sensitive integrated matching for picture libraries," *IEEE Transactions on pattern analysis and machine intelligence*, pp. 947–963, 2001.