FPGA-based System for Real-time Video Texture Analysis

Dimitris Maroulis, Dimitris K. Iakovidis, Dimitris Bariamis

Real Time Systems and Image Analysis Laboratory, Department of Informatics and Telecommunications, University of Athens, Greece

{d.maroulis, d.iakovidis, d.bariamis}@di.uoa.gr

Abstract. This paper describes a novel system for real-time video texture analysis. The system utilizes hardware to extract 2nd-order statistical features from video frames. These features are based on the Gray Level Co-occurrence Matrix (GLCM) and describe the textural content of the video frames. They can be used in a variety of video analysis and pattern recognition applications, such as remote sensing, industrial and medical. The hardware is implemented on a Virtex-XCV2000E-6 FPGA programmed in VHDL. It is based on an architecture that exploits the symmetry and the sparseness of the GLCM and calculates the features using integer and fixed point arithmetic. Moreover, it integrates an efficient algorithm for fast and accurate logarithm approximation, required in feature calculations. The software handles the video frame transfers from/to the hardware and executes only complementary floating point operations. The performance of the proposed system was experimentally evaluated using standard test video clips. The system was implemented and tested and its performance reached 133 fps and 532 fps for the analysis of CIF and QCIF video frames respectively. Compared to the state of the art GLCM feature extraction systems, the proposed system provides more efficient use of the memory bandwidth and the FPGA resources, in addition to higher processing throughput, that results in real time operation. Furthermore, its fundamental units can be used in any hardware application that requires sparse matrix representation or accurate and efficient logarithm estimation.

Keywords: Field Programmable Gate Arrays, Parallel Architectures, Pattern Recognition, Video Signal Processing, Real-Time System.

1. Introduction

Texture is an innate property of the natural objects, and it is widely used for video content description. The utility of texture feature extraction from video extends to a wide range of advanced modern applications, including segmentation of objects in image sequences [1,2], object recognition [3], tracking of moving objects [4,5], video transcoding for video content adaptation [6] and adaptive intra refreshment schemes for improved error resilience in object-based video coding [7].

The Gray Level Cooccurrence Matrix (GLCM) features [8] describe the textural image content by encoding the second order statistical properties of texture. These properties are mostly related to the human perception and discrimination of textures [9]. The GLCM features have been successfully utilized in a number of applications including medical [10], remote sensing [11] and industrial visual inspection applications [12,13].

A major drawback of the GLCM feature extraction method is its high computational complexity, which is prohibiting for real-time video texture analysis in software. A system capable of performing video texture analysis in real-time would be useful for a variety of applications including temporal analysis of video frame sequences [14], and analysis of medical video streams, such as endoscopic [10] or ultrasound screening [15]. In such cases, the software implementations are usually incapable of achieving real-time performance when full resolution video streams are used rather than downscaled videos. To overcome the limitations imposed by the software, we considered the utilization of dedicated hardware based on Field Programmable Gate Arrays (FPGAs). FPGAs are low cost and high density gate arrays capable of performing many complex computations in parallel while hosted by conventional computer hardware. They have been the choice for the implementation of computationally intense feature extraction tasks, including fingerprint feature extraction [16], facial feature extraction [17], the computation of Zernike moments [18], etc.

An FPGA-based system for the computation of two GLCM features has been proposed by Heikkinen and Vuorimaa [19]. This system approximates only two simple

features, namely mean and contrast, without actually computing the GLCMs. Tahir et al. [20] has presented another architecture that calculates the GLCM of multispectral images. The calculation of the GLCM is performed by one FPGA core whereas the computation of the GLCM features is performed by a second core that is subsequently programmed onto the FPGA. However, the use of this second core results in a time overhead for reprogramming the FPGA, affecting the overall feature extraction performance.

A similar system was proposed by our research group [21]. This system calculates both the GLCMs and features in hardware, but relies on software for a significant part of the computations. It performs well when the input image is divided into overlapping blocks, but in the case of non-overlapping blocks, it is inefficient as several of the units are unused for extended periods of time. Another FPGA-based system for GLCM calculation has been proposed by our research group [22], which provides more efficient calculation of GLCMs, however it does not calculate any GLCM features in hardware. Furthermore, the transfer of GLCMs over the PCI bus incurs a significant performance overhead, which can be prohibiting for real-time video texture analysis. The system presented in [23] was capable of GLCM features calculation in hardware, but employed data redundancy in order to achieve high processing throughput. However, the redundancy led to high memory capacity requirements and redundant transfers of data over the PCI bus.

In this paper we propose a novel FPGA-based system for real-time extraction of GLCM texture features from video frames. The motivation for the development of this system was to cover the need for real time extraction of texture features from uncompressed video streams, such as the input of the Colorectal Lesion Detection (CoLD) software [24]. The proposed system is capable of calculating a total of 64 GLCM features in parallel, namely angular second moment, correlation, inverse difference moment and entropy, at four different directions in a video frame, for four video frame blocks. It is implemented on a single FPGA core that performs the calculation of both the GLCMs and the features, exploiting the symmetry and sparseness of the GLCM and using integer and fixed point arithmetic. The proposed system also incorporates an algorithm for efficient approximation of the logarithm

in the entropy feature, and an effective buffering scheme, which only occupies a small fraction on the FPGA area and reduces the external memory requirements, while retaining a high processing throughput.

The rest of this paper is organized in five sections. The methodology used for the extraction of the GLCM texture features is described in Section 2. The architecture of the proposed system presented in Section 3 is followed by a complexity analysis in Section 4. The results obtained from the experimental evaluation on standard video clips are apposed in Section 5. Finally, Section 6 summarizes the conclusions derived from this study.

2. Texture Features Extraction

GLCMs encode the gray level spatial dependence based on the estimation of the 2nd order joint-conditional probability density function, which is computed by counting all pairs of pixels of a video frame block at distance d having gray levels i and j at a given direction θ. The cooccurrence matrix can be regarded symmetric if the distribution between opposite directions is ignored, so the angular displacement is usually included in the range of the values {0°, 45°, 90°, 135°} [25]. Among the 14 statistical GLCM features, originally proposed by Haralick et al [12], we consider four (Eqs. 1 to 4), namely, angular second moment (f_1), correlation (f_2), inverse difference moment (f_3) and entropy (f_4). These four selected features are widely used in the literature because they provide high discrimination accuracy, which can be only marginally increased by adding more features in the feature vector [12,26].

$$f_1 = \sum_{i=1}^{N_g} \sum_{j=1}^{N_g} p_{ij}^2 \tag{1}$$

$$f_2 = \frac{\sum_{i=1}^{N_g} \sum_{j=1}^{N_g} i \cdot j \cdot p_{ij} - \mu_x \cdot \mu_y}{\sigma_x \cdot \sigma_y} \tag{2}$$

$$f_3 = \sum_{i=1}^{N_g}\sum_{j=1}^{N_g} \frac{1}{1+(i-j)^2} p_{ij} \tag{3}$$

$$f_4 = -\sum_{i=1}^{N_g}\sum_{j=1}^{N_g} p_{ij} \cdot \log_2 p_{ij} \tag{4}$$

where p_{ij} is the ijth entry of the normalized cooccurrence matrix, N_g is the number of gray-levels of the video frame, μ_x, μ_y, σ_x, and σ_y are the means and standard deviations of the marginal probabilities $P_x(i)$ and $P_y(j)$ obtained by summing up the rows or the columns of matrix p_{ij} respectively.

The calculation of the Eqs. 1-4 requires floating point operations that would result in high FPGA area utilization and low operating frequencies. To implement the calculation of the features efficiently in hardware, we have reformulated the equations by extracting five expressions V_1 to V_5 as follows:

$$f_1 = \frac{V_1}{r^2} \tag{5}$$

$$f_2 = \frac{\left(N_g - 1\right)\cdot\left(r \cdot N_g^2 \cdot V_2 - r^2\right)}{N_g^2 \cdot V_5 - N_g r^2} \tag{6}$$

$$f_3 = \frac{1}{r} \cdot V_3 \tag{7}$$

$$f_4 = r \cdot \log_2 r - \frac{1}{r} \cdot V_4 \tag{8}$$

where

$$V_1 = \sum_{i=1}^{N_g}\sum_{j=1}^{N_g} c_{ij}^2 \tag{9}$$

$$V_2 = \sum_{i=1}^{N_g}\sum_{j=1}^{N_g} i \cdot j \cdot c_{ij} \tag{10}$$

$$V_3 = \sum_{i=1}^{N_g}\sum_{j=1}^{N_g} c_{ij} \cdot IDMLUT[|i-j|] \tag{11}$$

$$V_4 = \sum_{i=1}^{N_g} \sum_{j=1}^{N_g} c_{ij} \cdot \log_2 c_{ij} \tag{12}$$

$$V_5 = \sum_{i=1}^{N_g} C_x^2(i) \tag{13}$$

$$c_{ij} = r \cdot p_{ij} \tag{14}$$

$$C_x(i) = r \cdot P_x(i) \tag{15}$$

$$IDMLUT[|i-j|] = \frac{1}{1+(i-j)^2},\ 0 \le |i-j| < N_g \tag{16}$$

In the above equations, the operations needed to calculate V_1 to V_5 (Eqs. 9-13) are performed using integer or fixed point arithmetic. These values are computed in hardware and subsequently used for the calculation of the four features f_1 to f_4 in software, requiring only 16 floating-point operations that incur a negligible time overhead. In contrast to this approach, the implementation of a floating-point unit capable of performing the division operation on FPGA would significantly diminish the performance of the system, by reducing the achieved frequency and throughput of the hardware architecture.

The parameter r (Eqs. 14 and 15) represents the number of pixel pairs of a block, for a specific direction and distance. $IDMLUT[|i\text{-}j|]$ is an N_g×32-bit lookup table, which contains a 32-bit fixed point representation of the function $1/(1+(i\text{-}j)^2)$, $0 \le |i\text{-}j| < N_g$. The calculation of the logarithm $\log_2 c_{ij}$ is implemented using a fast approximation method in hardware. The implementation returns a 32-bit fixed point value $\log_2 c_{ij}$ for each integer c_{ij}.

3. System Description

The proposed system is based on a Xilinx XCV2000E-6 FPGA, programmed in VHDL [27]. The FPGA features 19,200 slices, includes 160 256×16-bit Block RAMs and can support up to 600kbit of distributed RAM. It is packaged in a 560-pin ball grid array (BGA560) that provides 404 user I/O pins. It is hosted by the Celoxica RC-1000 board that includes four 2MB static RAM banks [28]. These RAM banks can be accessed by the FPGA

or the host computer independently, whereas simultaneous access is prohibited. This is ensured by the board's arbitration circuit which assigns the ownership of each bank to either the host or the FPGA using isolator circuits. The FPGA and the host processor can communicate through the board in two ways: by bulk PCI transfers from/to the memory banks, and by control and status byte transfers. Bulk transfers are commonly used for large data transfers, but can only be initiated by the host. Control and status byte transfers are mostly used for synchronization and can be initiated by either the FPGA or the host.

The architecture of the implemented hardware is illustrated in Fig. 1. The software iteratively feeds the FPGA board with four video frame blocks per iteration. Each pixel is represented by 6 bits (N_g=64) and the pixels of the four blocks are interleaved, allowing the utilization of the memory bank width and the retrieval of one pixel from each block in one clock cycle, for a total of four pixels. The FPGA reads each block's pixels, calculates the GLCM of each block and their respective feature vectors for the θ=0°, 45°, 90° and 135° directions and d=1 distance, and stores them into memory bank 1 and 2.

The FPGA architecture consists of:

- A control unit
- Three memory controllers (for memory banks 0, 1 and 2)
- A circular buffers unit
- Sixteen GLCM calculation units (GCUs)
- Four vector calculation units (VCUs)

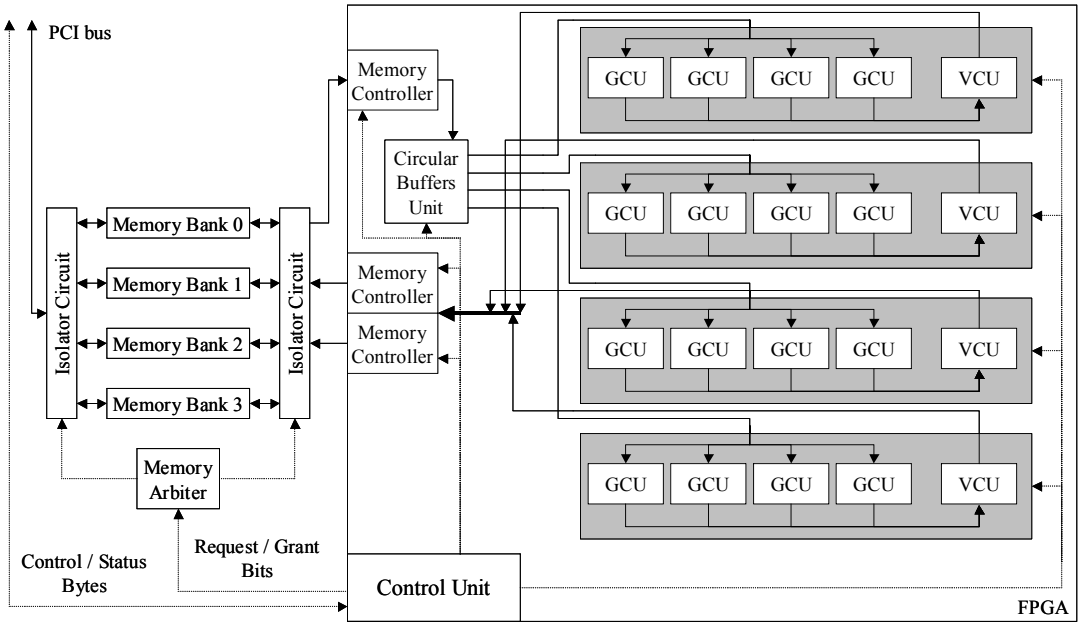


Fig. 1. The hardware architecture

Control Unit

The control unit coordinates the FPGA functions by generating synchronization signals that coordinate the memory controllers, the circular buffers unit, the GLCM calculation units (GCUs) and the vector calculation units (VCUs). It also handles the communication with the host, by exchanging control and status bytes and by requesting or giving up the ownership of the memory banks.

Memory Controllers

Three memory controllers handle the transactions between the FPGA and the asynchronous memory banks. Each controller is assigned to a specific memory bank, providing a 32-bits synchronous interface through which the data can be accessed for read/write operations.

Circular Buffers Unit

The pixels of each input block are read from memory bank 0 sequentially. During the calculation of the four GLCMs, each pixel in the block is visited five times: First, the pixel is regarded to be the center of a 3×3-pixel neighborhood, and after a number of sequential reads the same pixel becomes non-central in four other 3×3-pixel neighborhoods at 0°, 45°, 90° and 135° directions from the central pixel of each neighborhood. It is noted that the dimension of the neighborhood is 3×3 because the distance between the neighboring pixels for the calculation of the GLCM is d=1. Moreover, only four out of the eight possible neighborhoods are taken into account due to the symmetry of the GLCM. In order to avoid multiple reads of the same pixels, a circular buffers unit has been implemented, reducing the external bandwidth requirements of the proposed architecture.

Figure 2 illustrates a part of the input block. The squares in the grid represent the pixels in the block. The pixels marked with diagonal lines are stored in the circular buffer. The circular buffer outputs five pixels, namely the central pixel of the 3×3 neighborhood (black background) and its four neighboring pixels for the four directions (gray background). These five pixels are forwarded to the GCUs. As shown in Fig. 2, in every clock cycle (A, B and C snapshots) the last pixel is removed from the circular buffer, the neighborhood is slided by one pixel to the right and a new pixel is inserted into the buffer. The circular buffer contains the pixels marked with diagonal lines in every consecutive snapshot.

The circular buffers unit contains four circular buffers, one for each input block of $W\times W$ dimensions. Each buffer consists of W+3 6-bit cells, as shown in Fig. 2. Through the use of buffering, the external read bandwidth is 4 pixels per clock cycle, while the GCUs operate with a throughput of 20 pixels per clock cycle, received from the circular buffers unit. Therefore, the implementation of this unit enables a fivefold increase of throughput and a consequent fivefold reduction of input bandwidth requirements.

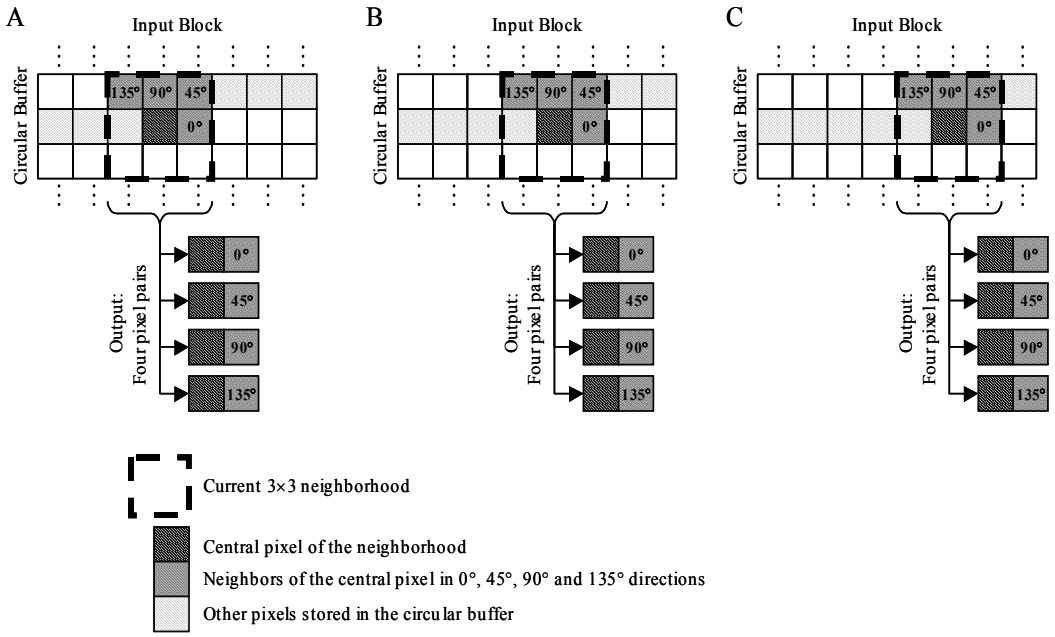


Fig. 2. Three consecutive snapshots of the operation of the circular buffer

GLCM Calculation Units

A GLCM calculation unit (GCU) is used for the calculation of the GLCM of a single block for a particular direction. It consists of an n-way set associative array [29] with a capacity of N_c cells and the auxiliary circuitry needed for the calculation of the GLCM. Set associative arrays can be used for efficient storage and retrieval of sparse matrices because each of their elements can be accessed or updated in four clock cycles while pipelining ensures a throughput of one operation per cycle. Other methods for calculating and storing the GLCM include the utilization of the available BlockRAMs or the implementation of standard sparse structures that store indices and values. The former does not exploit the sparseness of the GLCM, as all its elements need to be stored, while the latter cannot ensure a high throughput, as the cycles needed to traverse the indices is proportional to the length of the array. In contrast to the aforementioned methods, the set associative array was chosen, as it is a flexible alternative that can provide efficient utilization of FPGA resources, high throughput per clock cycle and high frequency potential.

An n-way set associative array consists of n independent tag arrays (tag_0 - tag_{n-1}) as illustrated in Fig. 3. Each tag array consists of N_c/n cells. The set associative array uniquely maps an input pair of 6-bit gray-level intensities (i, j) to an address of the N_c-cell data array, which is implemented using FPGA Block RAMs. A Block RAM has the capacity to store 256 GLCM elements, therefore N_c is set to multiples of 256. The data array cells contain the number of occurrences of the respective (i, j) pairs, using a 16-bit integer representation. Each of the (i, j) pairs is represented by a single 12-bit integer, which results from merging i and j. This integer is split into two parts: the $set(i, j)$ part, which consists of its $\log_2(N_c/n)$ least significant bits and the $tag(i, j)$ part, which consists of its $12\text{-}\log_2(N_c/n)$ most significant bits. The procedure of incrementing a data array cell that corresponds to an input pair (i, j) is implemented in four pipeline stages:

1) All valid tag array cells located in the $set(i, j)$ row are retrieved and stored to temporary registers. The validity of each tag is confirmed using one valid bit per tag.

2) The temporary registers' values are compared to $tag(i, j)$.

a) If a match is found, then the column number of the matching tag is written in the offset register.

b) If there are no matches, the $tag(i, j)$ is stored in the tags array, at the first available cell of the $set(i, j)$ row. The proper selection of the parameters n and N_c for each application ensures that the $set(i, j)$ row has enough available cells for the calculation.

3) The contents of both the offset register and $set(i, j)$ form an address a. The data array element in address a is read and stored in a temporary register.

4) The value of the temporary register is increased by one and it is written back to the address a of the data array.

After all input pairs are read and processed, the data array will contain the GLCM of a video frame block for a particular direction. The GLCM calculation unit gives triplets (i, j, c_{ij}) as output, which is forwarded to a vector calculation unit.

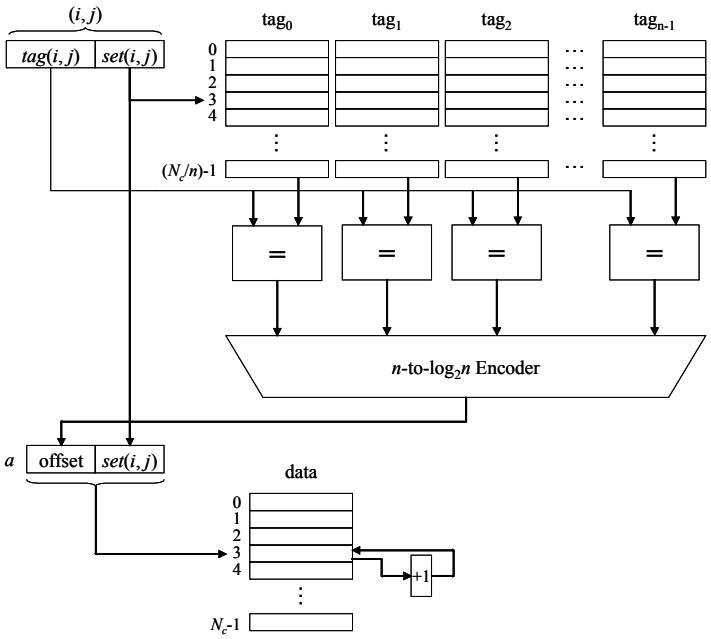


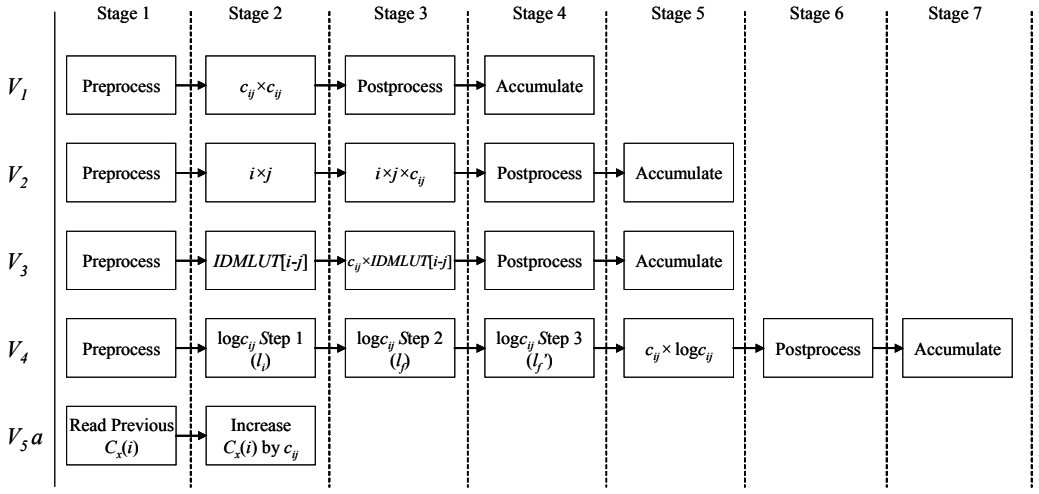
Fig. 3. GLCM calculation unit block diagram

Vector calculation units

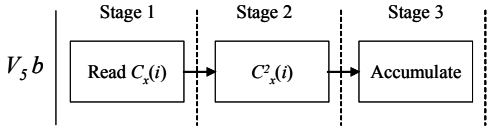
Four special units, named as vector calculation units (VCUs) have been assigned for the calculation of vectors $\overline{V} = \left[V_1, V_2, V_3, V_4, V_5\right]$ (Eqs. 9-13). Each vector calculation unit receives a GLCM generated by a GLCM calculation unit as input and outputs a vector $\overline{V}$. The vectors are stored to the board's memory banks through the corresponding memory controllers.

The calculation of V_1 to V_4 values is implemented in four independent pipelined circuits. The pipeline stages for each circuit are illustrated in Fig. 4a. All circuits implement a pre-processing stage, computation stages, a post-processing stage and an accumulation stage. Both pre-processing and post-processing stages facilitate the operations needed for the transition from/to the lower diagonal representation of the GLCM. The pre-processing stage uses a shifter to implement the multiplication of the GLCM's main diagonal by two, whereas the post-processing stage involves the use of a shifter for the multiplication of the results of the intermediate computation stages by two, for all c_{ij} except for those belonging to the main

diagonal. The computation stages involve table lookup, logic or arithmetic operations such as multiplication, addition and subtraction. The multipliers needed for the multiplication operations are implemented using FPGA slices, as there are no dedicated multipliers in the XCV2000E-6 FPGA. The timing analysis shows that the multipliers implemented in a single pipeline stage do not limit the maximum frequency of the design. If the maximum frequency was affected, alternatively, the multipliers could have been easily extended to two or more pipeline stages. If the architecture were to be implemented on a Virtex2 or on a more recent FPGA, the multiplier blocks of the FPGA would be used, reducing the FPGA area utilization of the design. The l_i, l_f and l_f' symbols (Fig. 4a) correspond to the integer, the fractional and the corrected fractional part of $\log_2 c_{ij}$ as computed by the logarithm calculation unit (described in the next subsection), which is included as a sub-unit of each of the vector calculation units. The output of each post-processing stage is accumulated in a corresponding register during the accumulation stage.



(a)



(b)

Fig. 4. Vector calculation unit pipelines

The calculation of V_5 is implemented in two pipelines. A dual ported 64×16-bit Block RAM is used for the storage of $C_x(i)$. The first pipeline (V_5 *a* in Fig. 4a) is used to calculate $C_x(i)$ from c_{ij} values. At the first stage of this pipeline, the previous value of $C_x(i)$ is retrieved and at the second it is increased by c_{ij} and it is written back to the Block RAM. The second pipeline (V_5 *b* in Fig. 4b) is activated when all of the c_{ij} values have been read. At the first stage of this pipeline, $C_x(i)$ is retrieved from the Block RAM, at the second stage it is squared and at the third stage it is added into an accumulator register.

The fixed point calculation of the $\overline{V}$ vector is performed without rounding or truncating any intermediate results by increasing the bit width of the operands in each calculation stage. Thus, no error is introduced during the conversion of floating to fixed point operations. For example, in the first stage of the V_2 computation the input values i and j are 6-bit wide and their product $i \cdot j$ is 12 bits wide. In the next stage, c_{ij} is 16 bits wide and the width of the resulting product $i \cdot j \cdot c_{ij}$ is 28 bits. The accumulators of all pipelines have a width of 64 bits to prevent overflow. The approximation error of the logarithm function used for the entropy feature is discussed in the following subsection.

Logarithm Calculation Unit

Several methods have been proposed in the literature for the implementation of the log-operation, including Look-Up-Table (LUT) [30], CORDIC-based [31,32] and power series implementations [33]. The use of a LUT for 32-bit representation of the logarithm for all 16-bit integers would require a 65536-element array with a total size of 65536 × 32-bit = 256kB. The implementation of such an array on FPGA would not be feasible since its size exceeds the available resources. CORDIC-based and power series implementations require several computation steps and need long pipelines and many computational units, in order to be implemented in hardware.

In the proposed system we have implemented an efficient method for the approximation of the base-2 logarithm of 16-bit integers. It is based on the linear approximation method originally proposed by Mitchell [34] but includes an additional error-correcting step, in which the logarithm is approximated by two linear segments. A VLSI implementation of an error-correcting circuit for the logarithm approximation using Mitchell's method has been proposed in [35]. This implementation is optimized for low power consumption and combinatorial operation, and it does not minimize the approximation error. The implementation we propose optimizes the logarithm computation for minimum approximation error. Furthermore, the proposed implementation achieves high frequency by using three pipeline stages; still it requires limited hardware resources. Other approaches to logarithm approximation based on Mitchell's method [36,37] employ more linear segments, but would require more slices and reach lower frequency when implemented on FPGA.

It involves three steps:

1) The integer part of the logarithm, $l_i = \lfloor \log_2(x_0) \rfloor$ is determined by the position of the Most Significant Bit (MSB) of the input integer x_0, where

$$2^n \le x_0 < 2^{n+1} \Rightarrow n \le \log_2(x_0) < n+1 \Rightarrow l_i = n,\ n \ge 0 \tag{17}$$

2) The fractional part, $l_f = \log_2(x_0) - l_i$ is estimated using the linear approximation of $\log_2(x)$, between the points $(2^n, n)$ and $(2^{n+1}, n+1)$ of the x–$\log_2(x)$ as follows:

$$\frac{x_0 - 2^n}{2^{n+1} - 2^n} = \frac{l_f}{n+1-n} \Leftrightarrow l_f = \frac{x_0}{2^n} - 1 \tag{18}$$

The above expression for the computation of l_f can be easily extracted from the binary representation of x_0, as it is equal to x_0 without its most significant bit, shifted right by n bits.

3) The approximation accuracy achieved can be further increased by a transformation performed on the fractional part of the logarithm. This transformation involves a segmentation of the $[2^n, 2^{n+1})$ interval into two halves. This is mathematically expressed in the following equation:

$$l'_f = \begin{cases} (1+a)\cdot l_f & if \quad l_f \le 1/2 \\ (1-a)\cdot l_f + a & if \quad l_f > 1/2 \end{cases} \tag{19}$$

where the parameter a is experimentally determined. For different values of a the error E between the actual values of the logarithm $\log_2(x)$ and its approximation ($l_i + l_f'$) was estimated by Eq. 20, as illustrated in Fig. 5.

$$E = \frac{1}{65535}\sum_{x=1}^{65535}\left|\frac{\log_2(x) - (l_i + l'_f)}{\log_2(x)}\right| \tag{20}$$

Although the minimum error was achieved for $a = 0.22$ ($E = 0.07\%$), we selected $a = 0.25$ ($E = 0.08\%$) because it can be easily implemented in hardware using shift and add operations and leads to a comparable error E. In order to further reduce the approximation error, the first 16 values of $\log_2(x)$ are stored in a 16×32-bit lookup table.

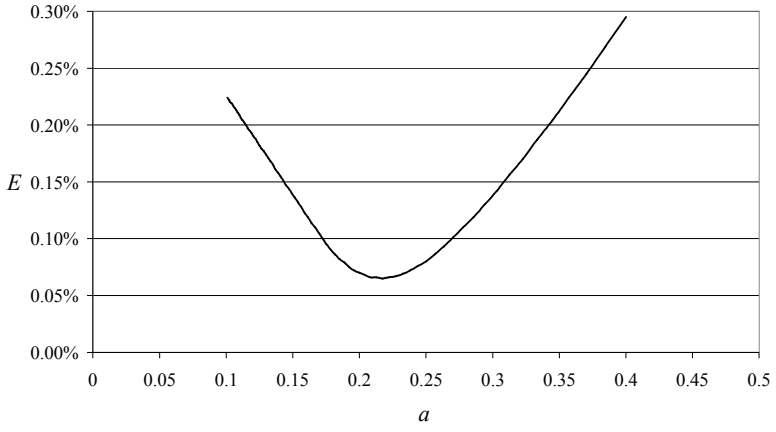


Fig. 5. Error E estimated for different values of parameter a

The three steps of the proposed method are implemented in a three-stage pipelined circuit, which allows a throughput of one result per clock cycle. The implementation of the proposed method on the XCV2000E-6 FPGA requires 123 FPGA slices and achieves a maximum frequency of 121.5MHz. In comparison, a fully pipelined CORDIC core requires more than 200 slices, as generated by Xilinx CoreGen. Furthermore, the CORDIC core requires scaling of the input in order to converge, which must be implemented in additional circuitry that occupies approximately 70 more slices.

4. Complexity Analysis

The parallel computation of sixteen vectors $\overline{V}$ from four input video frame blocks of $W \times W$ dimensions (one vector for each angle and block) requires 6 steps. The number of cycles needed for each of the steps is shown in Table I.

TABLE I
COMPLEXITY ANALYSIS

Step	Cycles	Task
1	N_c/n	Reset the GLCM calculation units
2	$W \times W$	Read all pixels from RAM bank 0 (four pixels per cycle)
	$W+3$	Wait for the circular buffer to become empty
	4	Wait for the last pixel pair output by the circular buffer to be processed by the GLCM calculation units
3	2	Reset the vector calculation units
	N_c	Read all (i, j, c_{ij}) triplets from the GLCM calculation units for 0^o direction
	67	Wait for the vector calculation units to complete the computation of V_5
	20	Write vectors $\overline{V}$ of all vector calculation units to RAM
4	N_c+89	Repeat step 3 for 45^o direction
5	N_c+89	Repeat step 3 for 90^o direction
6	N_c+89	Repeat step 3 for 135^o direction

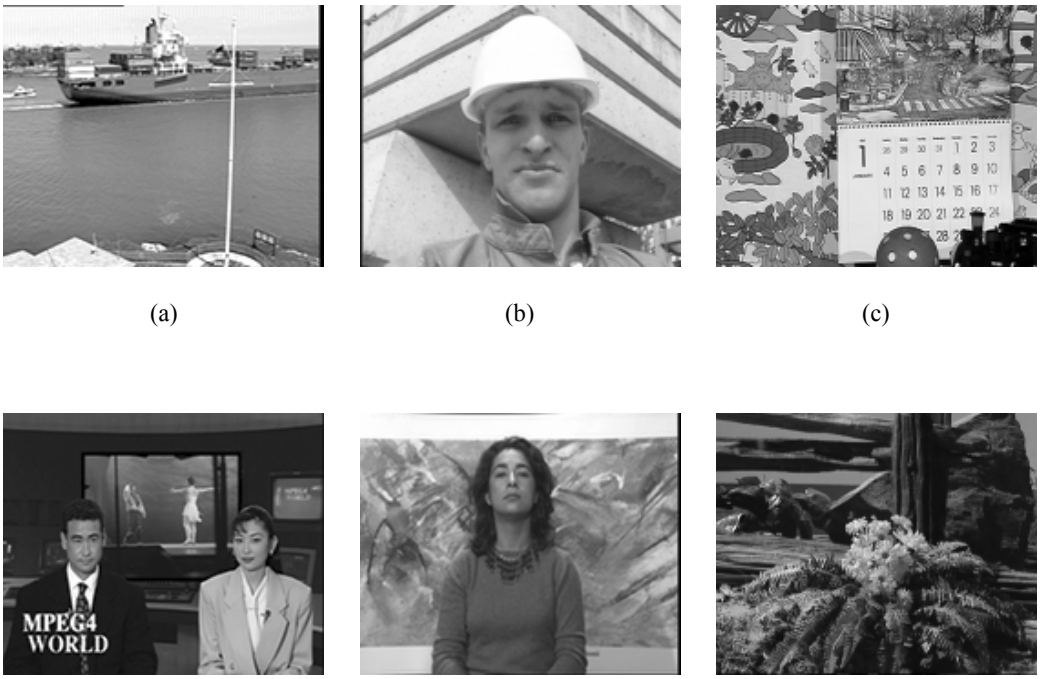
The first step involves resetting the valid bits for the tags array of the GLCM calculation units to zero. This requires one cycle for each row of the array or N_c/n cycles in total. In the second step, the input video frame blocks are read from memory bank 0 into the circular buffers, in $W \times W$ cycles. In the next $W+3$ cycles, the circular buffers do not receive

any input but they produce the last pixel pairs from the pixels that are still stored in them. The GLCM calculation units have a latency of four cycles, thus the system needs to wait four more cycles for the units to complete the GLCM calculation. In the third step, the system resets the vector calculation units in two cycles and forwards the calculated GLCM for 0^o direction to the vector calculation unit in N_c cycles. The vector calculation units produce the $\overline{V}$ vector 67 cycles after the last element of the GLCM has been read. Furthermore, 20 cycles are needed to write the four $\overline{V}$ vectors produced by the four vector calculation units to memory banks 1 and 2. The third step is then repeated for the 45^o, 90^o and 135^o directions.

The total number of cycles needed for the computation of the feature vectors for four video frame blocks in parallel is $4 \cdot N_c + N_c/n + W^2 + W + 363$.

5. Results

The performance of the proposed system was experimentally evaluated using standard test video clips, six of which are shown in Fig. 6. The videos are encoded in both CIF and QCIF formats. The host processor used was a 1GHz Athlon, which was state of the art at the time XCV2000E-6 was introduced. The results are organized in two parts; in the first part, we present the system performance measurements for several hardware simulations and in the second part, we present the system performance measurements for two hardware implementations, compared to the software implementation running on two different processors, the Athlon 1GHz and the Athlon XP 2800+.



(a) (b) (c)

(d) (e) (f)

Fig. 6. Video clips used in the experiments: (a) container, (b) foreman, (c) mobile, (d) news, (e) silent and (f) tempete

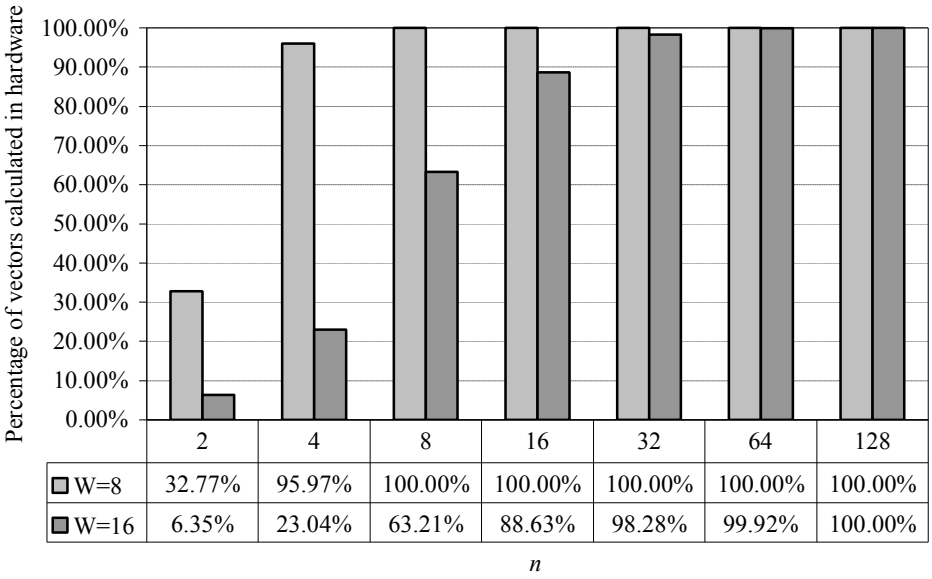
A. Simulation

Real-time texture analysis requirements impose a high processing rate, which can be achieved when all vectors $\overline{V}$ (Eqs. 9-13) of a video frame sequence are calculated in hardware. The number of vectors successfully calculated in hardware highly depends on the number of gray-level transitions appearing on each video frame, which in turn affects the number of non-zero GLCM elements. This can be adjusted by proper selection of the GLCM calculation parameters, N_c and n.

A series of hardware simulations were performed in order to experimentally determine the pairs of (N_c, n) values that could efficiently support real-time video texture analysis, without exceeding the available FPGA hardware resources. The output of the simulations is the percentage of vectors that can be successfully calculated in hardware using

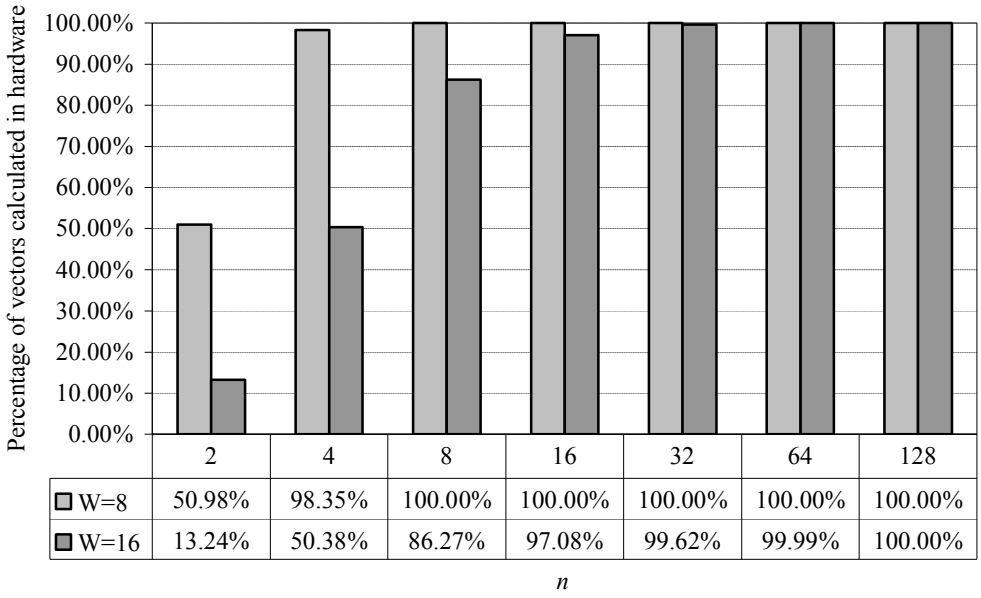
non-overlapping consecutive blocks of standard dimensions ($W \times W$, W=8 or W=16) that cover the whole CIF and QCIF video frames.

The results are illustrated in Fig. 7-10 and show that the choice of n mainly depends on the dimensions of the block used. Small blocks (W=8) can be handled entirely in hardware by set associative arrays of $n \geq 8$ with N_c=256. Larger blocks (W=16) require set-associative arrays of $n \geq 16$ and N_c=512. Under these circumstances a choice of (N_c, n)=(256, 8) for W=8 and (N_c, n)=(512, 16) for W=16 is preferable if all calculations are to be performed in hardware.



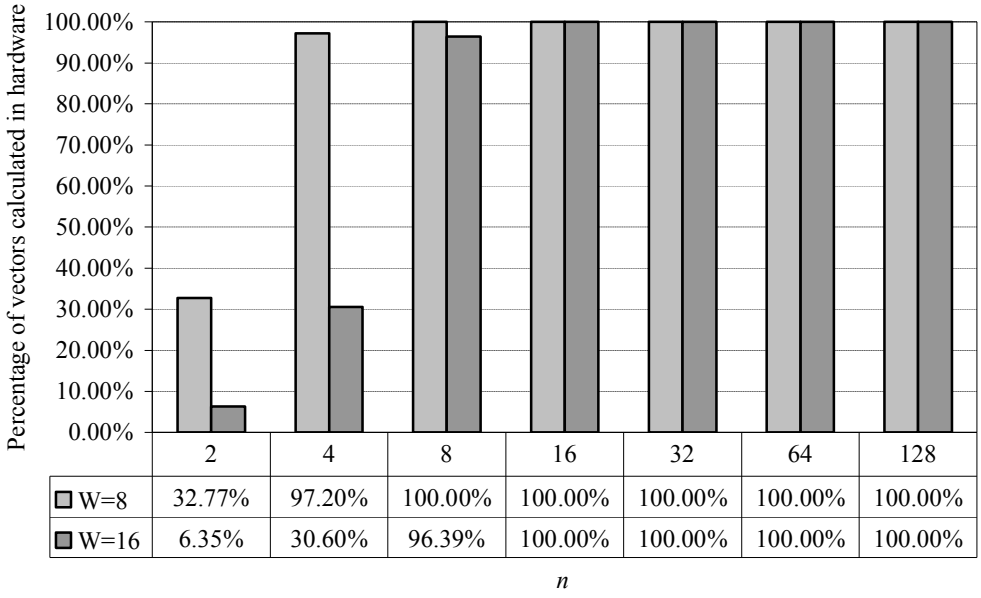
	2	4	8	16	32	64	128
W=8	32.77%	95.97%	100.00%	100.00%	100.00%	100.00%	100.00%
W=16	6.35%	23.04%	63.21%	88.63%	98.28%	99.92%	100.00%

Fig. 7. Percentage of vectors calculated in hardware for CIF video clips, using N_c = 256



	2	4	8	16	32	64	128
W=8	50.98%	98.35%	100.00%	100.00%	100.00%	100.00%	100.00%
W=16	13.24%	50.38%	86.27%	97.08%	99.62%	99.99%	100.00%

Fig. 8. Percentage of vectors calculated in hardware for QCIF video clips, using $N_c = 256$



	2	4	8	16	32	64	128
W=8	32.77%	97.20%	100.00%	100.00%	100.00%	100.00%	100.00%
W=16	6.35%	30.60%	96.39%	100.00%	100.00%	100.00%	100.00%

Fig. 9. Percentage of vectors calculated in hardware for CIF video clips, using $N_c = 512$

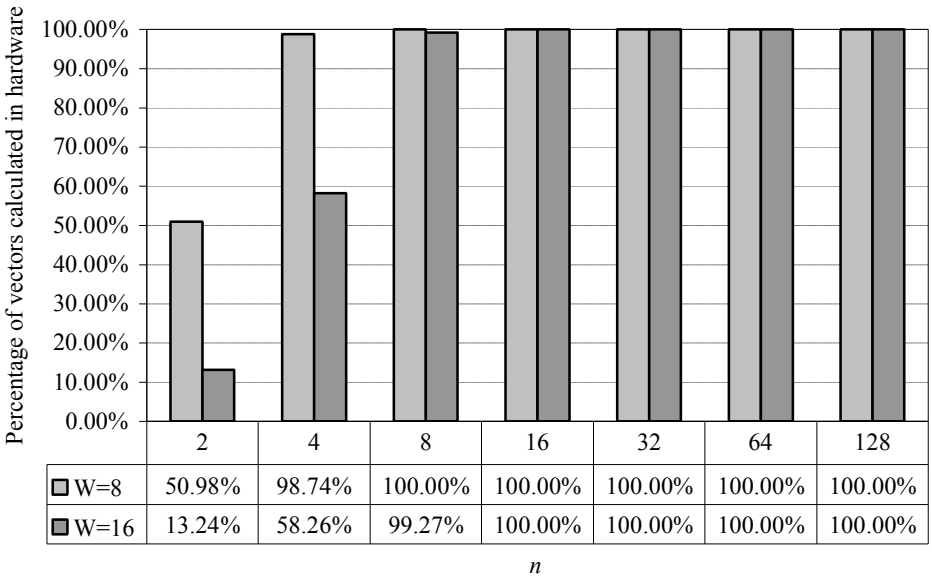


Fig. 10. Percentage of vectors calculated in hardware for QCIF video clips, using N_c = 512

B. Implementation

The simulation results led us to implement and evaluate two alternatives of the proposed system, using (N_c, n) = (256, 8) and (N_c, n) = (512, 16) respectively. As shown in the complexity analysis (Table I), the total number of cycles needed for the calculation of the feature vectors mostly depends on N_c and W. The use of N_c = 256 leads to faster processing than the use of N_c = 512 for a given block dimension W. Moreover, the use of larger blocks (W=16) results in higher overall performance, because the number of vectors is reduced.

The implementation results per configuration, including the FPGA area coverage, the maximum frequency and the required BlockRAMs are presented in Table II. The performance of the respective configurations in frames per second (fps) is presented in Table III. These tables also include the synthesis results for the XC2V4000-4 FPGA, demonstrating the performance increase achieved by using a next generation FPGA that provides multipliers, higher frequency potential and larger BlockRAMs.

TABLE II
IMPLEMENTATION RESULTS

N_c	N	Slices	Area	Frequency	BlockRAMs	FPGA Device
256	8	11625	60%	43.95 MHz	24	XCV2000E-6
512	16	16158	84%	35.75 MHz	40	XCV2000E-6
256	8	9332	40%	68.28 MHz	24	XC2V4000-4
512	16	15058	65%	56.24 MHz	24	XC2V4000-4

The software performance was measured on two workstations based on a 1GHz Athlon and on an Athlon XP 2800+ processor.

TABLE III
PERFORMANCE IN FRAMES PER SECOND (FPS)

Format	W (pixels)	FPGA XCV2000E-6		FPGA XC2V4000-4		Software	
		$n = 8$ $N_c = 256$	$n = 16$ $N_c = 512$	$n = 8$ $N_c = 256$	$n = 16$ $N_c = 512$	Athlon 1GHz	Athlon XP 2800+
CIF	8	74.44	35.89	115.64	56.46	5.3	8.57
	16	-	133.01	-	209.24	21.2	32.38
QCIF	8	297.75	143.58	462.57	225.88	21.30	36.03
	16	-	532.02	-	836.95	84.80	131.21

The results illustrate that the proposed FPGA-based system outperforms general-purpose processors for GLCM feature extraction from video frame blocks. Even though general-purpose processors have a significant frequency advantage, the parallel FPGA implementations result in higher overall performance. It should be noted that all calculations on the FPGA are performed using fixed-point arithmetic and no intermediate results are rounded or truncated. This results in the accurate calculation of the features, however a negligible error could be introduced by the fast logarithm approximation method used for the calculation of the entropy. It is also worth noting that the VirtexE FPGA (XCV2000E-6) retains its performance advantage compared to the AthlonXP processor, even though the latter was released four years later. The next generation Virtex2 FPGA (XC2V4000-4) displays a clear performance advantage over all other configurations, both hardware and software.

Comparison to state of the art architectures

Compared to the state of the art architectures presented in [19-23], the proposed architecture presents several significant advantages. The architecture proposed by Heikkinen

et al. [19] calculates two simple GLCM features without actually computing the GLCMs. This leads to a simpler design for the calculation of these two features, but the calculation of any other feature would involve substantial changes to the architecture, as the computation of the actual GLCM would be required.

The architecture presented by Tahir et al. [20] is implemented on a Celoxica RC-1000 board that includes a Xilinx XCV2000E-6 FPGA as well; however it uses a single FPGA core to calculate only the GLCMs and not the features. The solution given by the authors is to reprogram a second FPGA core explicitly for the feature extraction, which is much slower as it involves repetitive alternation of the FPGA core and transfers of the GLCM data from/to the memory banks. Each reprogramming of the FPGA core requires transferring more than 1.2 MB of data over the PCI bus, resetting the FPGA and possibly exchanging several control and status bytes in order to resume the calculation. This procedure incurs a delay of several tens of milliseconds, thus rendering real-time operation infeasible. The proposed architecture manages to implement both the GLCM and feature calculation on a single FPGA core by using a sparse representation of the GLCMs, which requires significantly fewer FPGA resources. Furthermore, in the system presented in [20] the pixels are read sequentially from the memory banks without the use of any buffering scheme, resulting in a throughput of only 4 pixels per clock cycle. This system was developed for the acceleration of feature extraction from multispectral images, but not for real-time computation. Indeed, its results illustrate that real-time performance is not achieved for the GLCM calculation and feature extraction.

An early architecture proposed by our research group [21] performs best when the input image is divided into highly overlapping blocks. The implementation includes computational units, such as set-associative GLCM calculation units and logarithm approximation circuit in a single feature calculation unit, that are unused for large periods of time if the blocks are non-overlapping, rendering the high FPGA area utilization unnecessary and inefficient. Furthermore, the preprocessing of the input image blocks involves a replication scheme for the pixels in the memory banks, increasing the memory requirements by four times.

In another architecture proposed by our research group [22], the hardware only calculates the GLCMs, however the features are not calculated on the FPGA, leaving this task for the supporting software. The GLCMs are not generally calculated using sparse set-associative arrays. Sparse arrays are used only for small input blocks (W=16 and N_g=32). The input blocks are read and processed at a rate of 20 pixels per clock cycle by replicating the input image blocks in the memory banks. The method that allows the maximization of input bandwidth in [22] is the preprocessing of the input image blocks, by replicating and packing the pixels in a way that allows keeping the calculation units busy at all times. This results in efficient utilization of the FPGA resources, but inefficient memory utilization, as the preprocessed input image blocks require four times more memory than the original image blocks. The main drawback of [22] as compared to the proposed system is the requirement of transferring large amounts of data through the PCI bus, such as the replicated pixels and the computed GLCMs.

The system proposed by our research group in [23] is a preliminary version of the proposed system, which handles the feature extraction in hardware. Thus, it gains a performance increase compared to [22], which leads to real-time calculation of texture features. However, it does not employ a more efficient memory utilization scheme, such as the circular buffer, but it relies on data replication for the maximization of input bandwidth and processing throughput. However, such a redundancy due to data replication leads to high memory capacity requirements and redundant transfers of data over the PCI bus.

In contrast to the above state of the art architectures, the proposed architecture combines an efficient usage of the memory banks and the FPGA resources. A buffering scheme on the FPGA ensures a high processing throughput of 20 pixels per cycle, while the read rate from the memory banks is only 4 pixels per cycle and no data is replicated. By avoiding data replication, the memory capacity and memory bandwidth requirements are reduced by four times, while retaining high processing throughput. Taking into account the symmetry and the sparseness of the GLCM, a significant reduction in the required FPGA slices is achieved, enabling the implementation of both the GLCM and feature calculation in a

single FPGA core and eliminating the overhead associated with transferring the GLCMs from/to the memory banks and reprogramming the FPGA. Furthermore, the exploitation of the sparseness of the GLCM using set-associative arrays allows the proposed system to achieve high performance for N_g=64 instead of N_g=32 used in architectures [20,22], even though the number of elements of the GLCM is quadrupled. The use of N_g=64 in architectures that use dense representations of the GLCM [20,22] is infeasible on the particular FPGA as it would necessitate a quadruplication of the FPGA resource requirements. Additionally, the organization of the proposed architecture enables the exploitation of the full potential of the FPGA for parallel computations, by avoiding any idle states for the system units, in the case of non-overlapping blocks.

6. Conclusions

In this paper a novel system capable of performing real-time texture analysis of video frames was proposed. It is capable of calculating a total of 64 features comprising of four 16-dimensional GLCM feature vectors from four video frame blocks in parallel. The hardware is based on FPGA technology and it is capable of performing fast integer and fixed point operations, which include the computation of many GLCMs in parallel and the computation of GLCM features. An algorithm for the approximation of the logarithm, which is required for the computation of the entropy feature, has been included within the hardware architecture. A buffering scheme ensures a high processing throughput, while maintaining low memory bandwidth requirements. The software supports the hardware by managing the video frame transfers from/to the hardware and by performing only supplementary floating point operations.

The proposed system was tested on standard test video clips encoded in CIF and QCIF formats, and demonstrated real-time performance for video texture analysis. Its performance exceeds the PAL/NTSC frame rate requirements [38], providing the potential of performing additional video frame processing tasks e.g. efficient discrete wavelet transform

[39]. The evaluation procedure showed that the proposed system is capable of performing GLCM feature computations much faster than software running on modern workstations, thereby making it suitable for replacing software implementations in systems requiring real time extraction of GLCM features from video frames [24]. Furthermore, its fundamental units can be used in any other hardware application that requires similar components, for example the sparse matrix representation or the accurate and efficient logarithm estimation.

7. Acknowledgement

This work was realized under the framework of the Operational Program for Education and Vocational Training Project "Pythagoras" cofunded by European Union and the Ministry of National Education of Greece.

References

[1] Y. Deng and B. S. Manjunath, "Unsupervised Segmentation of Color-Texture Regions in Images and Video," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 23, no. 8, pp. 800-810, Aug. 2001.

[2] J. Kim and T. Chen, "Multiple Feature Clustering for Image Sequence Segmentation," Pattern Recognition Letters, vol. 22, pp. 1207-1217, 2001.

[3] D. K. Iakovidis, D. E. Maroulis, S. A. Karkanis, and I. N. Flaounas, "Color Texture Recognition in Video Sequences using Wavelet Covariance Features and Support Vector Machines," in Proc. 29th EUROMICRO Conference, pp. 199-204, Antalya, Turkey, Sept. 2003.

[4] E. Ozyildiz, N. Krahnstöver, and R. Sharma, Adaptive Texture and Color Segmentation for Tracking Moving Objects, Pattern Recognition, vol. 35, no. 10, pp. 2013-2029. Oct. 2002.

[5] A. Shahrokni, T. Drummond, and P. Fua, "Texture Boundary Detection for Real-time Tracking." In Proc. ECCV, vol. 2, pp. 566-577, 2004.

[6] Y. Wang, J.-G. Kim, and S.-F. Chang, "Content-based Utility Function Prediction for Real-time MPEG-4 Transcoding,", in Proc. IEEE International Conference on Image Processing, Barcelona, Spain, 2003.

[7] L. D. Soares, and F. Pereira, "Adaptive Shape and Texture Intra Refreshment Schemes for Improved Error Resilience in Object-Based Video Coding," *IEEE Trans. on Image Processing*, vol. 13, pp. 662-676, May 2004.

[8] R.M. Haralick, K. Shanmugam and I. Dinstein, "Textural features for image classification," *IEEE Trans. Systems, Man and Cybernetics*, vol. 3, pp. 610-621, 1973.

[9] B. Julesz, "Texton gradients: the texton theory revisited," *Biol. Cybern.*, vol. 54, pp. 245-251, 1986.

[10] S. A. Karkanis, D. K. Iakovidis, D. E. Maroulis, D. A. Karras and M. Tzivras, "Computer Aided Tumor Detection in Endoscopic Video using Color Wavelet Features," *IEEE Trans. on Information Technology in Biomedicine*, vol. 7, pp. 141-152, 2003.

[11] A. Baraldi and F. Parmiggiani, "An Investigation of the Textural Characteristics Associated with Gray Level Cooccurrence Matrix Statistical Parameters," *IEEE Trans. Geoscience and Remote Sensing*, vol. 33, no. 2, pp. 293-304, 1995.

[12] R.M. Haralick, "Texture Measures for Carpet Wear Assessment," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 10, no. 1, pp. 92-104, 1988.

[13] A. Latif-Amet, A. Ertuzun and A. Ercil, "An Efficient Method for Texture Defect Detection: Sub-band Domain Co-occurrence Matrices," *Image and Vision Computing*, vol. 18, pp. 543-553, 2000.

[14] R. Fablet, and P. Bouthemy, "Motion Recognition using Non Parametric Image Motion Models Estimated from Temporal and Multiscale Cooccurrence Statistics," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 25, no. 12, pp. 1619-1624, Dec. 2003.

[15] D. Smutek, R. Sara, P. Sucharda, T. Tjahjadi, M. Svec, "Image texture analysis of sonograms in chronic inflammations of thyroid gland", Ultrasound in Medicine and Biology, 29 (11), pp. 1531-1543, 2003.

[16] N.K. Ratha, “A real-time matching system for large fingerprint databases”, IEEE Transactions on Pattern Analysis and Machine Intelligence, 18 (8), pp. 799-813, 1996.

[17] D. Nguyen, D. Halupka, P. Aarabi, A. Sheikholeslami, “Real-time face detection and lip feature extraction using field-programmable gate arrays”, IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, 36 (4), pp. 902-912, 2006

[18] L. Kotoulas, I. Andreadis, “Real-time computation of Zernike moments”, IEEE Transactions on Circuits and Systems for Video Technology, 15 (6), pp. 801-809, 2005

[19] K. Heikkinen, and P. Vuorimaa, “Computation of Two Texture Features in Hardware,” in Proc. 10th International Conference on Image Analysis and Processing, pp. 125-129, Venice, Italy, Sept. 1999.

[20] M.A. Tahir, A. Bouridane, F. Kurugollu, “An FPGA Based Coprocessor for GLCM and Haralick Texture Features and their Application in Prostate Cancer Classification”, Anal. Int. Circ. Signal Process. 43 (2005) 205-215

[21] D.G. Bariamis, D.K. Iakovidis, D.E. Maroulis, S.A. Karkanis, “An FPGA-based architecture for real time image feature extraction”, Proceedings - International Conference on Pattern Recognition, pp. 801-804, vol. 1, 2004.

[22] D.K. Iakovidis, D.E. Maroulis, and D.G. Bariamis, “FPGA Architecture for Fast Parallel Computation of Cooccurrence Matrices,” Microprocessors and Microsystems, vol. 31, issue 2, pp. 160-165, 2007

[23] D. Bariamis, D.K. Iakovidis, D. Maroulis, “Dedicated hardware for real-time computation of second-order statistical features for high resolution images”, Lecture Notes in Computer Science, 4179 LNCS, pp. 67-77, 2006.

[24] D.E. Maroulis, D.K. Iakovidis, S.A. Karkanis, D.A. Karras, “CoLD: A versatile detection system for colorectal lesions in endoscopy video-frames”, Computer Methods and Programs in Biomedicine, 70 (2), pp. 151-166, 2003.

[25] S. Theodoridis, and K. Koutroumbas, Pattern Recognition, Academic press, San Diego, 1999.

[26] S. Karkanis, G. D. Magoulas and N. Theofanous, “Image Recognition and Neuronal Networks: Intelligent Systems for the Improvement of Imaging Information,” Minimal Invasive Therapy and Allied Technologies, vol. 9, pp. 225-230, 2000.

[27] K. C. Chang, Digital Systems Design with VHDL and Synthesis, IEEE Computer Society, 1999.

[28] Celoxica Corporation, http://www.celoxica.com.

[29] J. L. Hennesy, and D. A. Patterson, Computer Architecture, A Quantitative Approach, Morgan Kaufmann, May 2002.

[30] J. A. Starzyk, and Y. Guo, "An Entropy-based Learning Hardware Organization Using FPGA," in Proc. Southeastern Symposium on System Theory, pp. 1-5, Athens, OH, 2001.

[31] J. E. Volder, “The CORDIC Trigonometric Computing Technique”, IRE Transactions on Electronic Computers, EC-8, pp. 330-334, 1959

[32] R. Andraka, “A Survey of CORDIC Algorithms for FPGA Based Computers” in Proc. of the 1998 CM/SIGDA Sixth International Symposium on FPGAs, Monterey, CA, pp.191-200, Feb. 1998.

[33] D. M. Mandelbaum, and S. G. Mandelbaum, “A Fast, Efficient Parallel-Acting Method of Generating Functions Defined by Power Series, Including Logarithm, Exponential, and Sine, Cosine,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 7, no. 1, pp. 33-45, Jan. 1996.

[34] J.N. Mitchell Jr., “Computer Multiplication and Division Using Binary Logarithms,” IRE Trans. Electronic Computers, vol. 11, pp. 512-517, Aug. 1962.

[35] S.L. SanGregory, R.E. Siferd, C. Brother and D. Gallagher, “A Fast, Low-Power Logarithm Approximation with CMOS VLSI Implementation,” Proc. IEEE Midwest Symp. Circuits and Systems, Aug. 1999.

[36] M. Combet, H. Zonneveld, and L. Verbeek, “Computation of the Base Two Logarithm of Binary Numbers,” IEEE Trans. Electronic Computers, vol. 14, pp. 863-867, Dec. 1965.

[37] E.L. Hall, D.D. Lynch, and S.J. Dwyer III, “Generation of Products and Quotients Using Approximate Binary Logarithms for Digital Filtering Applications,” IEEE Trans. Computers, vol. 19, pp. 97-105, Feb. 1970.

[38] Y. Wang, J. Ostermann and Y. Zhang, "Digital Video Processing and Communications", Prentice Hall, 2001.

[39] Y. Zeng, G. Bi, and A. C. Kot, “Combined Polynomial Transform and Radix-*q* Algorithm for MD Discrete W Transform,” *IEEE Trans. on Signal Processing*, vol. 49, no. 3, pp. 634-641, Mar. 2001.