Hardware Acceleration of Background Modeling in the Compressed Domain

Popa, S., Crookes, D., & Miller, P. (2013). Hardware Acceleration of Background Modeling in the Compressed Domain. IEEE Transactions on Information Forensics and Security, 8(10), 1562-1574. DOI: 10.1109/TIFS.2013.2276753

Published in:
IEEE Transactions on Information Forensics and Security

Hardware Acceleration of Background Modeling in the Compressed Domain

Stefan Popa, Danny Crookes and Paul Miller

***Abstract*—In intelligent video surveillance systems, scalability (of the number of simultaneous video streams) is important. Two key factors which hinder scalability are: the time spent in decompressing the input video streams, and the limited computational power of the processor. This paper demonstrates how a combination of algorithmic and hardware techniques can overcome these limitations, and significantly increase the number of simultaneous streams. The techniques used are: processing in the compressed domain, and exploitation of the multi-core and vector processing capability of modern processors. The paper presents a system which performs background modeling, using a Mixture of Gaussians approach. This is an important first step in the segmentation of moving targets. The paper explores the effects of reducing the number of coefficients in the compressed domain, in terms of throughput speed and quality of the background modeling. The speed-ups achieved by exploiting compressed domain processing, multi-core and vector processing are explored individually. Experiments show that a combination of all these techniques can give a speed-up of 170 times on a single CPU compared to a purely serial, spatial domain implementation, with a slight gain in quality.**

***Index Terms*—video surveillance, background subtraction, compressed domain, hardware acceleration, multi-core, SSE**

I. Introduction

A key issue for intelligent video surveillance systems is that of scalability. Real-world deployment of video surveillance systems can involve hundreds, or even thousands, of cameras. Thus it is vital that video analytics are scalable with respect to both robust performance and also computability. Scalability is achieved for these systems by decomposing them into subsystems consisting of 8, 16, 32 or 64 IP cameras connected to a single sever. Thus, we argue that scalability can be achieved with respect to video analytics, if we can process the data in real-time produced by one of these subsystems. With regard to computational efficiency, an important consideration is the hardware/software architecture of the video analytics processing. The main commercial approach to achieve computational efficiency is the use of the Texas Da Vinci Multimedia DSP chipset. In the video surveillance research community, the use of GPUs for implementation of video analytics has been investigated. However, a drawback of this approach is the PCI bus bottleneck between the GPU and CPU. More recently, the availability of multi-core CPUs and vectorization has provided another option with respect to computational efficiency.

S. Popa (spopa01@qub.ac.uk), D. Crookes (d.crookes@qub.ac.uk) and P. Miller (p.miller@ecit.qub.ac.uk) are with the School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, UK (see: http://www.qub.ac.uk/).

Manuscript sent October 1, 2012

Another consideration with respect to scalability and computational efficiency is that the video data are transmitted and stored in the compressed domain. However, most video analytic algorithms operate in the spatial domain; therefore, the video data has to be decompressed. From a computational viewpoint, decompression of, say, 64 video streams is non-trivial. Furthermore, the presence of compression artifacts can also reduce performance. One approach to reducing this computational overhead, therefore, is to implement the video analytics in the compressed domain.

The first step in video analytics is usually some form of foreground detection based on background modeling. The gold standard in this regard, both commercially and in research, is the mixture of Gaussians (MoG) algorithm. Therefore, in this work we investigate the implementation of this algorithm in the compressed domain using multi-core and vectorization. Specifically, we set ourselves the challenge: Can we perform foreground detection on 64 streams of video in real-time on a single chip without compromising performance? The work reported in this paper tries to answer this.

II. Related Work

A. Background Modeling in the Compressed Domain

One of the most popular methods for extracting moving objects from a scene is background subtraction (BS). BS is one of the first low-level processing operations in virtually any intelligent video surveillance system, and it is the operation of identifying and segmenting moving objects in video frames by separating the still areas, called the background (BG), from the moving objects, called foreground (FG). Any BS algorithm first constructs a representation of the BG called the background model. Then, each subsequent frame is subtracted from this background model to give the resulting FG [1].

The main advantage of including a BS stage in a video surveillance system is the increase in performance of any higher-level video analysis. This is possible, for example, by focusing any tracking methods on the FG objects, or by narrowing the search window of any detection methods to the FG [2].

As shown in [3], background modeling (BM) algorithms can broadly be classified into three categories, depending on the compression features used: algorithms based only on Discrete Cosine Transform (DCT) coefficients [4]–[7]; algorithms based only on the motion vectors (MV) [8], [9]; and algorithms based on both [10]. Several studies have focused on BM algorithms using only DCT coefficients. In [4] the authors propose a framework that uses competing Hidden

Markov Models over small neighborhoods, which are capable of maintaining a valid background model. In this case, the small neighborhoods are the JPEG 8x8 blocks of DCT coefficients. For this framework, the main challenge is background initialization, which requires a large amount of time. Another drawback is that this framework is only well suited for indoor surveillance. In [5] the authors propose a fast and efficient adaptive method for modeling the background, which uses two features extracted from each 8x8 block of DCT coefficients in a JPEG frame. The first feature is the first DCT coefficient (also called the DC coefficient), and the second feature is a weighted sum of a few DCT low frequency coefficients (also called the AC coefficients). A Gaussian distribution is used to model each of the two features. In addition, further processing is used to handle variations in the environment and improve the segmentation. By combining the model built around the two features with different characteristics (the DC coefficient is more sensitive to changes in illumination and the low frequency AC coefficients are more sensitive to changes in texture), the authors claim that the model is robust to many of the problems linked with BS in video sequences representing outdoor scenes, such as gradual and sudden illumination changes, and small repetitive background movements. The authors of [6] use the same principle of modeling the background using two features extracted directly from each 8x8 block of DCT coefficients in a frame as part of a more complex tracking system. The main difference between the two is the choice of the second feature, which is represented by the first three AC coefficients in [6].

One of the most referenced papers addressing the problem of background modeling in the compressed domain, using only DCT coefficients, is [7]. Here, the authors present a framework composed of three algorithms for BM (Running Average, Median and Mixture of Gaussians) and a two-stage segmentation approach with pixel-level resolution. In the first stage, the background model is generated by one of the algorithms, which is further used to identify the block regions that are fully or partially occupied by a foreground object. In the second stage, the pixels from the partially occupied blocks are then processed (classified) in the spatial domain. Even though the processing is performed in the spatial domain for the partially occupied blocks, the authors prove through theoretical analysis and practical measurements that their algorithms in the compressed domain are several times faster than their spatial domain counterparts [11] and with better segmentation results.

B. Hardware Acceleration for Background Modeling

Several hardware-accelerated implementations for moving object detection and segmentation systems based on BM in the spatial domain have been developed. In [12], the authors implemented a BM algorithm based on an adaptive mixture of Gaussians (AGMM). Using an NVIDIA GPU with the Compute Unified Device Architecture (CUDA [13]) they achieved 18x acceleration compared with an implementation on an Intel multi-core CPU using multithreading. In the same paper, an implementation on IBMs Cell Broadband Engine Architecture (CBEA) achieved 3x the acceleration compared with the same Intel multi-core CPU benchmark. In [14], the authors implement a system that uses selective HOG-based features and BS based on a GMM for pedestrian detection. They report that using an Intel i7 and a low budget NVIDA GeForce 480GTX card, they are able to process 48 frames/second with high accuracy. Although the system does not achieve a significant speedup, the implementation is complex, and the BS represents only a part of the total computation. The authors of [15] implement only the AGMM algorithm. They report that the system can process close to 250 frames/second for high-resolution images, and close to 1600 frames/second for low-resolution images even when using a low-end NVIDIA GeForce 9600GT. Using the same GPU, they also report an acceleration of over 11x compared with a reference CPU implementation.

It is possible to conclude that, although there has been a significant amount of research in BM and higher level video analysis in the compressed domain, it is clear that the compressed domain has not been explored as extensively as the spatial domain with respect to algorithmic performance and hardware acceleration. To the best of our knowledge, there are no significant hardware accelerated implementations of BM algorithms in the compressed domain. This is the key novel contribution of the work reported in this paper.

Due to current industry practices, video surveillance footage is stored in the MJPEG compressed format in order to be accepted as evidence in law courts. In addition, all surveillance video cameras currently available on the market can stream images in the MJPEG compressed format. For these reasons, the research will be based on the MJPEG compressed format.

III. Background

A. MJPEG/JPEG Compression Format

MJPEG is a video compression format that takes advantage only of the pixel redundancies within a frame (spatial pixel redundancy). Each frame in a MJPEG video stream is a JPEG encoded image, which is independent of any previous or future frames; therefore, the compressed domain data available for processing in MJPEG compressed video streams is the DCT coefficients of the current frame [16].

The JPEG image compression standard [17] defines different modes of operation. Among the different modes of operation the most representative mode is the baseline JPEG mode, which is described next for a grayscale image.

First, the baseline JPEG encoder splits the image into 8x8 non-overlapping blocks of pixels and the pixels intensity is shifted from a range of 0:255 to a range of -127:127 in order to evenly distribute the values around zero. Further, each block is almost independently encoded, as in figure 1, using the following steps.

The forward type II two-dimensional discrete cosine transformation (2D-DCT type II) is first applied to the 8x8 block of pixels, resulting an 8x8 block of DCT coefficients. The top-left frequency component in the transformed block is called the DC coefficient and it is equivalent to the average intensity value of the pixels in the input block. The other coefficients

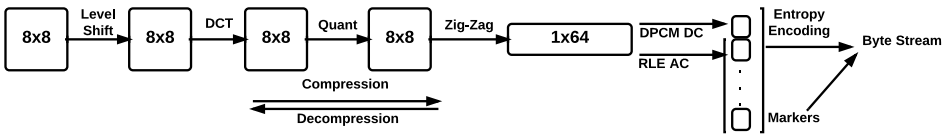


Fig. 1: JPEG Compression/Decompression Transformations

are called AC coefficients. Next, the 64 DCT coefficients are quantized using different quantization thresholds for different frequency components. This is the step that gives rise to the lossy nature of the JPEG compression. During quantization the high frequency coefficients are more often than not set to zero. By applying this transformation, the entropy encoding step will become much more effective, because it reduces the dynamic range of the DCT coefficients and, as a consequence, fewer bits are needed for representation. After quantization, the 64 DCT coefficients are rearranged in a zigzag order. In this way the low frequency coefficients, which, in contrast to high-frequency coefficients, are mainly non-zero, are grouped at the start of the array. The high frequency coefficients, which are usually zero, are grouped at the end of the array. This transformation is called zigzag reordering. The DC coefficients of the neighboring blocks tend to be highly correlated. The JPEG compression algorithm exploits this property and predicts the value of the current DC coefficient based on the value of DC coefficient of the previous block. This transformation is called DPCM DC (differential pulse-code modulation). After zigzag reordering and DPCM DC, the array containing the DCT coefficients, usually turns out to have some non-zero values scattered between zeros. Using this property, the JPEG format encodes the non-zero values and the number of previous zeros in just a pair of numbers. This transformation is called run-length encoding (RLE) of DCT coefficients. The entropy encoding transformation assigns variable length codes to pair of values created during RLE transformation, so that less probable pairs are represented by longer encoded binary codes and more probable pairs are represented by shorter encoded binary codes. JPEG uses Huffman encoding for entropy encoding. At the end, the entropy encoded data, the quantization table and the Huffman table are combined with markers to create the final byte stream (data stream). The markers are nothing more than delimiters for the different component parts of the byte stream. During the decompression, the JPEG decoder performs the inverse of the operations previously described in reverse order: starting with entropy decoding and finishing with inverse DCT transformation and inverse level shifting. Due to quantization, the decompressed image is not identical to the original one.

Because of the computation required to decompress an MJPEG video stream, processing the DCT coefficients directly is attractive. However, extracting the DCT coefficients from the video stream still requires a significant amount of processing, though inverse DCT is not required. The process is called partial decompression. The compressed domain representation of each frame is a 2D matrix of blocks, where each block comprises the DCT coefficients for the 8x8 block of pixels.

B. Background Modeling using Online Mixture of Gaussians

This section will introduce the Mixture of Gaussians (MoG) algorithm for which a hardware acceleration strategy will be presented later.

MoG is one of the most common BM algorithms in the literature. The algorithm received considerable interest from the moment it was introduced in [11]. The popularity of this algorithm is due to two important aspects. First, it is capable of modeling backgrounds containing non-static objects, such as tree branches or bushes moving in the wind (multi-model background). The algorithm can handle multi-model backgrounds because it follows the evolution of a set of K Gaussian distributions simultaneously (where K is a small number). Second, a recursive (or online) formulation of the algorithm can significantly reduce its complexity. However, the algorithm is still more complex than most of the non-statistical BM algorithms (simple algorithms like running-average, median, etc.). There are two notable drawbacks of MoG [2]. First, algorithm parameters require careful tuning. Second, for the case where the scene remains stationary for too long, the variance of the Gaussian distributions will become too small and any sudden change in the global illumination will force most of the scene to be classified as foreground.

Considering the case of a sequence of grayscale images, we have the following two situations. In the spatial domain, the input to a BM algorithm is, for each pixel position, a scalar value (one-dimensional feature vector) which is the pixel intensity at that location in the image. In the compressed domain, the input is, for each 8x8 block, a 64-dimensional feature vector which is obtained by concatenating the rows (or the columns) of the 8x8 block of DCT coefficients at that block location in the JPEG image. The underlying learning method used by the MoG algorithm to update the model parameters is not influenced by the type of feature vector used as input, and it can be applied to data coming from the spatial or compressed domain [7], [11]. The MoG algorithm uses a mixture of K Gaussian distributions to maintain a probability density function (PDF) for each feature vector (pixel intensity or block of DCT coefficients) in the current frame. This PDF is updated each frame.

As in [7] and [11], given the recent history $\{x_{t-n}, x_{t-n+1}, \ldots, x_{t-1}\}$ of a feature vector x_t, the distribution of the current feature vector is:

$$P\left(x_t|\theta_t\right) = \sum_{k=1}^{K} \omega_{k,t} * \eta_k\left(x_t, \mu_{k,t}, \sigma^2_{k,t}\right) \tag{1}$$

where $\theta_t = \left\{\omega_{k,t}, \mu_{k,t}, \sigma^2_{k,t}\right\}_{k=1}^{K}$ is the model parameter vector and $\omega_{k,t}, \mu_{k,t}, \sigma^2_{k,t}$ are the weight, mean and variance of the k^{th} Gaussian distribution, η_k, in the mixture and

$\left(\sum_{k=1}^{K} \omega_{k,t}\right) = 1$. The Gaussians are D-dimensional, where D is equal to the size of the feature vector x_t. To simplify the computation, as in [7], [11], we are making the following simplifications: $\Sigma_{k,t} = \sigma_{k,t}^2 * I$ (I is the unit matrix); although the Gaussians are D-dimensional, the variance is a scalar value (but the mean is a vector of length D).

The algorithm has two stages. The first stage is the binary classification of the current feature vector as BG or FG according to the background model; the second stage is the online update of the parameters using the current feature vector. The binary classification stage takes place as follows. The algorithm selects the first B most significant distributions to be part of the background model, where:

$$B = \underset{b}{\operatorname{argmin}} \left(\left(\sum_{k=1}^{b} \omega_{k,t} \right) > T \right) \tag{2}$$

and T is a user-defined threshold $(0 < T < 1)$. For the current feature vector x_t, a matching distribution $\hat{k}$ is sought using the following equation:

$$(x_t - \mu_{k,t-1})^T (x_t - \mu_{k,t-1}) < f * \sigma_{k,t-1}^2 \tag{3}$$

where f is a small deviation. If more than one matching distribution is found, the one closest to the feature vector is selected. This is done using the following equation:

$$\hat{k} = \min_{k} \left((x_t - \mu_{k,t-1})^T (x_t - \mu_{k,t-1}) / \sigma_{k,t-1}^2 \right) \tag{4}$$

The current feature vector x_t, is classified as BG if there is a matching distribution $\hat{k}$ and this matching distribution is part of the B selected distributions to represent the background $\left(\hat{k} \leq B\right)$. Otherwise, x_t is classified as FG.

Next, the online update stage takes place as follows. If a matching distribution $\hat{k}$ is found, the parameters of that distribution are updated as follows:

$$\begin{aligned} \omega_{\hat{k},t} &= (1-\alpha) * \omega_{\hat{k},t-1} + \alpha \\ \mu_{\hat{k},t} &= (1-\rho) * \mu_{\hat{k},t-1} + \rho * x_t \\ \sigma_{\hat{k},t}^2 &= (1-\rho) * \sigma_{\hat{k},t-1}^2 + \rho * \left(x_t - \mu_{\hat{k},t}\right)^T \left(x_t - \mu_{\hat{k},t}\right) \end{aligned} \tag{5}$$

where α is a user-defined learning rate $(0 < \alpha < 1)$ and ρ is a calculated learning rate:

$$\rho \approx \alpha / \omega_{\hat{k},t} \tag{6}$$

The parameters of the other distributions $\left(k \neq \hat{k}\right)$ are updated as follows:

$$\omega_{k,t} = (1-\alpha) * \omega_{k,t-1} \quad \mu_{k,t} = \mu_{k,t-1} \quad \sigma_{k,t}^2 = \sigma_{k,t-1} \tag{7}$$

If no matching distribution is found, the one with the smallest weight ($\omega_{k,t-1}$) is replaced by a new one with the following parameters:

$$\omega_{k,t} = \omega_0 \quad \mu_{k,t} = x_t \quad \sigma_{k,t}^2 = \sigma_0^2 \tag{8}$$

where ω_0 is a small weight and σ_0^2 a large variance.

After updating the parameters, the weights are renormalized, so that they still sum up to **1** and the distributions are ordered in descending order of their significance, $s_{k,t}$, where:

$$s_{k,t} = \omega_{k,t} / \sigma_{k,t}^2 \tag{9}$$

The reordering is necessary to ensure that only the relevant distributions are selected as part of the background model. A distribution is considered to be relevant when it has a large weight (evidence) and a small variance.

C. Hardware Acceleration

As mentioned in the introduction, real-world video surveillance systems can include deployments of a large number of cameras, which in turn generate a large amount of video data. Processing all this data in real-time requires a huge amount of computation. Modern CPUs commonly have multiple cores (typically 4-8) capable of parallel processing, where each core is equipped with short vector units capable of handling multiple data elements simultaneously. To handle the computation in a scalable and efficient manner, it is necessary to take advantage of all the resources available on the modern CPUs. To do so, the use of specific programming APIs and technologies is needed.

To use such CPUs to their full capacity, let us imagine a very simple image processing operation. The subtraction of two images, in a classic scenario, requires iterating over all elements in the two images and subtracting the pixel intensities at each location, using one core. By using, for example, a CPU with 4-cores and 4-ways SIMD processing units, the images are first split in four and the individual parts processed in parallel on the 4-cores; then each part is further split into vectors of four pixels which are processed simultaneously using the SIMD units. These optimizations can in theory give a speedup of 16x compared to a purely serial implementation.

At present, OpenMP [18] is the industry standard for parallel programming of shared memory multiprocessors. Multicore CPUs are considered to be shared memory multiprocessors. OpenMP uses the shared memory model to represent the interaction between threads and the memory, and uses the fork/join execution model to manage multithreaded execution [19].

The shared memory model has the following characteristics. One or more threads can run on a processor/core, but all threads have access to the shared memory. Data can exist as shared (in which case it can be accessed by all threads) or private (in which case it can be accessed only by the thread that owns it). Data transfers between threads and the private or shared memory is transparent, but the synchronization is mostly explicit (i.e. it is the programmers responsibility).

The fork - join execution model has the following characteristics. The processing starts with a single thread, called the master thread, and it runs sequentially until a parallel region construct is encountered. When this happens, the master thread will create a team of parallel threads (fork), and will divide the execution of the code inside the parallel region among the team of threads. After a team of threads finishes the code inside the parallel region, they synchronize and terminate (join), leaving the master thread to continue the program sequentially.

From the programmer's point of view, OpenMP is accessible in C/C++. It is composed of a set of compiler directives, a library of support functions and a set of environment variables.

Furthermore, each core commonly has both scalar units and short vector units, so vectorization is a form of parallelism

available at the core level and the individual CPU cores can achieve vectorization independently. The vector units are single instruction, multiple data (SIMD) processing units and can process multiple data elements using the same instruction (data-level parallelism). The use of SIMD units should accelerate the calculations by a factor equal to the length of the unit (for example, a 4-way SIMD processing unit should in theory accelerate the calculations by a factor of four).

On the Intel and AMD CPU micro-architectures the vector processing units use vector registers of 128 bits, which are able to handle the following data types: integer (usually 8 x 16-bits or 4 x 32-bits) and float (4 x 32-bits or 2 x 64-bits). This technology is called Streaming SIMD Extensions Technology (SSE) [20].

Some modern compilers can auto-vectorize scalar code to some extent, but usually just for simple cases. For complex cases, programmers have to explicitly use the vector data types and write code in a SIMD manner. From the programmer's perspective, SSE programming can be done in C/C++ using compiler intrinsics (which are compiled to one or a small set of assembly instructions), or directly in assembly language [20].

There are three important aspects that anyone should be aware of when developing algorithms using SSE [20]–[23]: the alignment of data in memory, the layout of data in memory, and the instructions available.

The best performance for accessing data in memory using the SSE instructions is achieved when the memory address of the accessed data (usually a 4-element vector) is divisible by 16 (16-byte aligned). The performance impact of misaligned memory accesses and methods to avoid them have been previously studied in [23].

When vectorizing code which is processing an array of structured objects (e.g. an array of grouped Gaussian parameters), the recommended layout for data is to have a separate array for each component of the structure (e.g. an array of means, an array of variances, etc.) This is called Structure of Arrays (SoA). An alternative layout is the more 'natural' (from an object oriented point of view) layout of a single array of structures. This is called Array of Structures (AoS). The advantage of SoA is that it keeps the homogeneous data components together in memory. This makes it easy to load and process them with the same instruction (or set of instructions). The performance impact of using different data memory layouts has been previously studied in [22]. For example, an RGB image using SoA layout is represented by three memory areas, where each memory area stores the data of one channel for all pixels. In contrast, using AoS layout, just one memory area is required, but each pixel occupies three consecutive memory locations, one for each channel (R, G, B).

The SSE vector instructions can be grouped into categories [20], such as: load/store, arithmetic, logic, comparisons, miscellaneous (min/max, type conversions, rounding, cache control, etc.) and application specific (fast-block difference, etc.) and they are closely tied to the type of the vector elements. For most float vector instructions, SSE offers a corresponding scalar version. These should not be confused with the classic (x86) scalar instructions because they operate on the vector registers (even though they are applied only on the first element of the vector register). Instructions for different data types, and even different instructions for the same data type, can be serviced by different parts of the processor. In fact, modern CPUs can issue up to six instructions per clock cycle, which increases the chances of executing instructions in parallel if there are no data dependencies between the operands of consecutive instructions. SSE is IEEE floating-point compliant, except for the reciprocal and square root reciprocal instructions. Division is one of the most expensive SSE instructions. The two instructions have been introduced to speed up algorithms involving division where high precision is not the first requirement. Another feature of the SSE instruction set is that it can handle conditional code without having to individually process each element in a vector register. This is possible by using a combination of logical SSE instructions or the newer $blend$ instruction. The $blend$ instruction takes two vector registers and selects their elements according to a mask. Usually the mask comes from a vector comparison instruction.

IV. Accelerating the Mixture of Gaussians in the Compressed Domain

To accelerate multi-stream background subtraction, there are two forms of parallelism to be exploited: multi-core (at the high level) and vector processing (low level). Our approach to exploiting multiple cores is straightforward: each input video stream is processed by a separate thread, with no need for communication with other threads. We do not partition the data for a video stream across multiple threads. So there is no significant programming effort involved in exploiting multi-core parallelism. Each thread performs the partial decompression followed by background modeling on a complete video stream.

To exploit vector processing, our approach is to vectorize the processing of Gaussian mixtures, rather than processing image blocks in parallel. Since it is sufficient to use a mixture of four Gaussians, and given the size of SSE vector registers, it is appropriate to vectorize the processing of the Gaussians. Further, since each Gaussian has three parameters (plus sometimes a fourth index - see below), we also have the option of using either a SoA approach or, if need be, an AoS approach.

One further possible dimension for increasing the amount of parallelism is to reduce the precision of the Gaussian components from 32 bit float to 16 bit fixed point. This would in theory provided a further speed up of 2x. However, having explored this, it was discovered that the increased complexity of the coding for 16 bit representation meant that the overall performance was actually slower than for processing 32 bit floats. In what follows, we therefore do not consider the question of precision any further.

Because of the complexity of SSE programming, the remainder of this section focuses on the SSE accelerated implementation of the MoG algorithm in the compressed domain.

As previously specified, the MoG algorithm in the compressed domain classifies and models the PDF of an entire block of DCT coefficients simultaneously using a mixture of

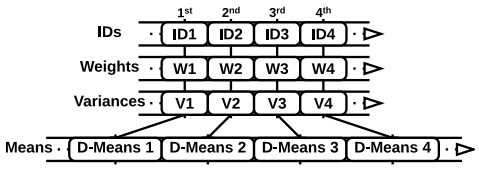


Fig. 2: The data layout of the model parameters in the compressed domain.

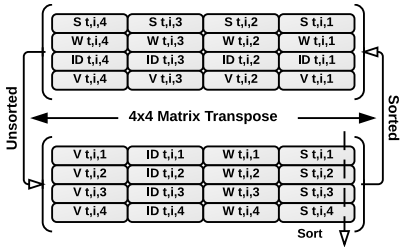


Fig. 3: Data transformations and sorting.

K Gaussian distributions. Therefore, the parameters for one Gaussian distribution in a mixture are: one weight (which represents the probability of a block to match that particular distribution, and is calculated based on previous observations), one variance and a set of D-means (one for each DCT coefficient in the block). The number of Gaussian distributions in a mixture has been set to four $(K = 4)$. This is (more than) adequate algorithmically, and matches the hardware constraints imposed by the size of the vector registers (128-bits). However, we note that one change to the detail of the algorithm, introduced to suit vectorization, is that we always maintain four Gaussians, instead of a dynamically varying number up to four. In order to avoid some expensive memory copy operations in the later steps of the algorithm, a new index parameter, ID, is added to each distribution (see step 6 below for more details). These IDs will be used to access the correct set of means in the means memory area corresponding to a particular mixture. The IDs of the distributions in a mixture are initialized to $(0, 1, 2, 3)$. The data layout used for the background model is SoA (figure 2).

Before processing a DCT block, the weights, IDs and variances of all distributions in the associated mixture are loaded in three vector registers. Due to these loads the CPU will (automatically) load these values also in the level 1 (L1) cache memory. The four sets of means corresponding to a mixture are loaded only when needed in the later steps. The parameters are saved back in the memory after their values have been updated and the process is repeated for the next block. The result of the assessment of the current block is a value which represents the binary classification of the block as FG or BG. In practice, the update steps for a block are as follows.

Step 1. Identify how many distributions make up the background model, according to equation 2. The result is an index, idx_1, with a value between zero and three. The distributions are already ordered according to their significance. The algorithm loads the weight of one distribution at a time (using SSE scalar load - $movss$), adds it to a sum (using SSE scalar add - $addss$) and compares the sum with the matching threshold T (using SSE scalar compare - $comiss$).

Step 2. Identify the closest distribution to the current block using equations 3 and 4. The result of this step is an index, idx_2, with value between zero and three if the current block matches any of the distributions, or **-1** otherwise. The algorithm computes the distances between the current block and all the distributions in the mixture simultaneously using vector instructions. The computation of one of the distances is equivalent to processing a scalar product between a vector and itself, which is a classic reduction problem, where the vector is the result of the difference between the current DCT block and the means set of one of the distributions. The distances can be processed simultaneously because the DCT block is read only once, and used for all distances. These distances are then stored in memory. Further, using the SSE scalar instructions, a short algorithm is implemented to find the index of the closest distribution accordingly to the already computed distances (which is a typical "minimum" problem). **Step 3.** Classify the current block of DCT coefficients as BG or FG using the indices computed in step 1 (idx_1) and step 2 (idx_2). The pixel is classified as BG if $idx_2 \geq 0$ and $idx_1 \geq idx_2$.

Step 4. Update all the distribution parameters. How this is done depends on the index computed in step 2 (idx_2), which identifies the closest distribution. There are two cases:

Step 4.a. If there is a matching distribution $(idx_2 \geq 0)$, then its parameters are updated according to equations 5 and 6. The parameters of the remaining distributions are updated according to equation 7. Specific SSE programming techniques are used to avoid "branching" code. First, the weights and variances are updated using vector instructions twice, once according to equation 5 and once according to 7. Then, the index from step 2 (idx_2) is used to build the mask used by the vector Blend instruction (see III.C for details) to select the correct updated parameters for each distribution. This procedure adds more computation but avoids "branching" code. Note that the update of the means is done last and is highly efficient, because it is performed on the full length of the vector registers (see SoA layout, III.C).

Step 4.b. If there is no matching distribution, the distribution with the smallest weight is replaced by a new one according to equation 8. For this, the distribution containing the smallest weight is identified (again using SSE scalar instructions) and then its parameters are reinitialized. An important note for step 4(a and b) is that the selection of the correct set of means associated with a particular distribution in a mixture is done using the ID parameter which we have added earlier to each distribution (see step 6).

Step 5. Renormalize the weights to sum up to **1**. For this, specific SSE programming techniques are used in order to handle the horizontal operation (sum) on the vector register containing the weights. The vector division necessary for renormalization $(divps)$ has been replaced with a vector multiplication $(mulps)$ between the weights register and the vector reciprocal $(rcpps)$ of the previously calculated sum.

Step 6. Reorder the distributions based on their significance. Here, few implementation details deserve mention. The sig-

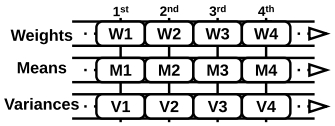


Fig. 4: The data layout used for the model parameters in the spatial domain.

nificance values, $s_{k,t}$, are computed according to equation 9 using SSE vector instructions (here the vector division has been replaced). Further (figure 3), to prepare for reordering all distributions in a mixture, a 4x4 matrix transposition operation is required. This transformation will reorganize the parameters of the unsorted distributions from SoA to AoS format. The operation can be done using only vector registers (without accessing the memory) and some specific SSE instructions for reordering the data in these registers. For the actual sorting we use a 4-ways sorting network [24] which offers an optimal number of comparisons. The sorting network implementation requires SSE scalar instructions for handling the conditions and vector instructions to swap a full register at a time ($movps$). After sorting, another 4x4 matrix transposition operation is necessary to switch the parameters back to oA format. This implementation turns out to be highly efficient because no memory operations are required and the instructions involved in the transposition are low latency. But the most important optimization here is the introduction of the ID parameter to each distribution. The introduction of this parameter eliminated the necessity to move around the large sets of means associated with the D-dimensional Gaussian distributions during the sorting operation.

For benchmarking reasons an SSE optimized version of the MoG algorithm in the spatial domain was also implemented. Now, the parameters for one Gaussian distribution in a mixture are: one weight, one mean and one variance per pixel. The ID parameter is not necessary anymore. Due to the same hardware constraints the number of Gaussian distributions in a mixture has also been set to four ($K = 4$). The data layout used for the background model remains SoA (figure 4). Although the SSE implementations of the MoG algorithm for the two domains seem very similar, a few differences are visible straight away. The compressed domain implementation is using a very compact memory representation of the model parameters, which is possible because it keeps the means of all distributions of a mixture associated with a block of DCT coefficients in a contiguous memory block. This makes their update, during step 4, much more efficient. Also, the compressed domain implementation has to maintain a lot fewer parameters. As already mentioned in [7], for a 8x8 pixel block, the algorithm uses $192 * K$ parameters in the spatial domain compared with just $66 * K$ in the compressed domain (K represents the number of distributions in a mixture). The ID parameter is not included in these numbers because it does not require updating.

TABLE I: Decompression Speed Results Per Frame

	1 Core	4 Cores
Libjpeg v8c (MC)	8 ms	1.82 ms
IJG v6b (MC)	7.2 ms	1.6 ms
IJG v6b (IC)	7 ms	1.54 ms
IJGmod v6b (IC & IPP)	3.72 ms	0.86 ms
UIC (IC & IPP)	3 ms	0.82 ms
Partial Decoder (IC & IPP)	0.84 ms	0.19 ms

V. Experiments and Results

A. Results for Partial Decompression

In the spatial domain, a frame is represented as a 2D matrix of pixels. To get access to the spatial pixel data in a video frame, the video stream has first to be fully decompressed. In the compressed domain, a frame is represented as a 2D matrix of blocks of DCT coefficients. This representation of a frame can be extracted from a video stream without fully decompressing the stream. Before any computation can begin, the video sequences need to be partially decompressed (if processing in the compressed domain) or fully decompressed (if processing in the spatial domain).

To evaluate the partial decompression, its performance was compared with five different full decompression implementations. For the latter, three different libraries were used: the standard Libjpeg v8c, Intel's Libjpeg-based library (IJG v6b), and Intel's proprietary library UIC which uses the Intel Performance Primitives Library (IPP). Two different compilers were also used: Microsoft's C++ Compiler (MC), and Intel's C++ Compiler (IC). Our MJPEG partial decoder makes use of the IPP library for Huffman decoding. The tests were carried out by simulating four camera streams, by replicating the input video sequence. We used the PETS2001 video sequence, which comprises 2688 frames, each of size 768x576 pixels. All images were first loaded into memory (decompression times do not include the time to read the images from the hard drive). The CPU used for these tests is an Intel i7 960 4-core CPU. For the multithreaded implementation, OpenMP was used to launch 4 threads, mapped to the 4 cores.

Table I shows the decompression times per frame, for the five full decompression versions, and for the partial decompression. We can draw the following conclusions from the table.

The fastest full decompression version is UIC. The use of IPP approximately doubles the performance. The speed of partial decompression is 4.3 times that of the fastest full decompression version. At a frame rate of 25 frames per second, the time for partial decompression equates to the ability to process 210 simultaneous video streams (not allowing for the input of the streams to memory). When processing fewer video streams (e.g. 64), this allows more time for processing tasks such as background modeling.

B. Hardware Acceleration results for Background Modeling

Experiments on the PETS2001 test video sequence showed that, on average only 14-16% of the DCT coefficients are non-zero in the entire video sequence, and most DCT blocks have

TABLE II: Processing Speed Results Per Frame

		1 Core	4 Cores
Spatial Domain	**Scalar**	23.342 ms	6.485 ms
		1x	3.6x
	Vector	17.493 ms	4.714 ms
		1.3x	4.9x
Compressed Domain - 64	**Scalar**	11.858 ms	3.082 ms
		1.9x	7.5x
	Vector	0.930 ms	0.616 ms
		25x	37.9x
Compressed Domain - 32	**Scalar**	5.854 ms	1.551 ms
		3.9x	15x
	Vector	0.566 ms	0.342 ms
		41.2x	68.2x
Compressed Domain - 16	**Scalar**	2.927 ms	0.781 ms
		7.9x	29.8x
	Vector	0.438 ms	0.137 ms
		53.3x	170.3x

32 or fewer non-zero DCT coefficients. The number of non-zero DCT coefficients per block depends on two factors. First, the quantization table used during compression; a stronger quantization table will increase the number of zero DCT coefficients. Second, the complexity and level of activity in the scene; for example, in the PETS2001 video sequence, the sky areas can be encoded with just one non-zero DCT coefficient per block, the grass areas are typically encoded with 6-10 non-zero DCT coefficients per block, and the regions covered by the cars are encoded with over 10 DCT coefficients per block. In general, blocks containing many edges or complex texture patterns are encoded using a larger number of DCT coefficients. These experiments led us to test the algorithm in the compressed domain using 64, 32 and 16 coefficients per block.

Table II presents the execution times for the spatial domain implementation vs. compressed domain implementation for three different numbers of DCT coefficients per block (64, 32, 16). For each case, it shows execution times for a single core and a 4-core implementation, first using scalar (non-vector) and then using vector implementations of the algorithm. The tests were run on the same custom video sequence described in the previous section, and under the same conditions.

The results show that the multi-core, vectorized versions of the three compressed domain implementations (using 64, 32 and 16 DCT coefficients) outperform the purely serial spatial domain version by factors 5x, 38x and an impressive 170x respectively. The speed up is due to three factors: the data reduction in the model, the use of the SIMD processing units, and the use of multiple cores. The data reduction counts for 45%, the use of SIMD processing units counts for 37% and the use of multi-cores contributes the remaining 18% from the total acceleration. The higher than expected contribution of the SIMD processing units is partly due to the manually optimized register management when programming at the SSE level, in a way which the compiler cannot yet achieve.

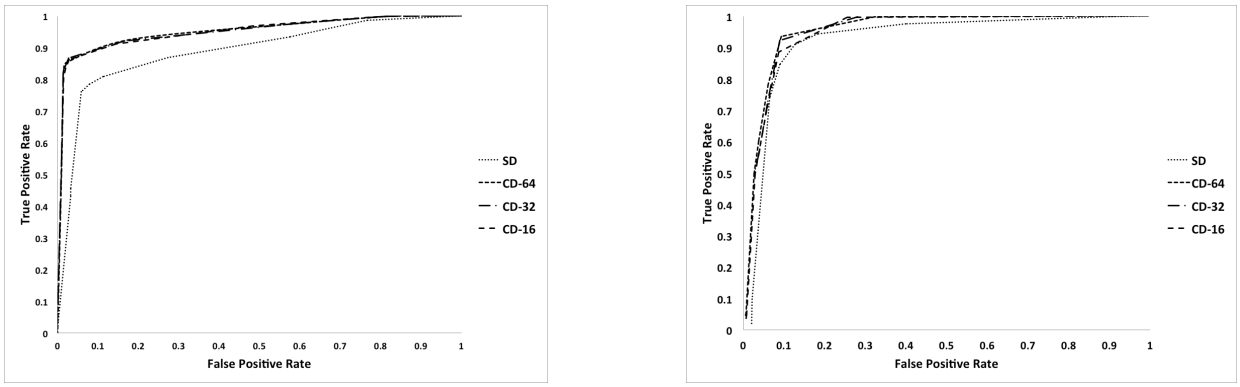
With a frame rate of 25 frames per second, the optimized implementation of the algorithm in the compressed domain using 16 DCT coefficients per block equates to the ability to perform background modeling on 292 video streams in real-time. This is equivalent to processing one frame every 0.14 ms. Further, by using the results from the previous section (where it was shown that a JPEG image can be partially decompressed in around 0.19 ms) it can be seen that, by using the same implementation, it is possible to perform combined partial decompression and background modeling on an equivalent of 122 video streams in real-time.

An additional investigation was carried out to evaluate the quality of the foreground detection, by measuring the impact on the detection rate of the MoG algorithm when moving from the spatial to the compressed domain, and when reducing the number of DCT coefficients from 64 to 32 and 16. Because accuracy could potentially be data dependent, two very different video sequences were used for these tests: PETS2001, a typical outdoor video surveillance sequence; and BusSeq, a very challenging video surveillance sequence taken inside a bus, with moving background seen through the windows. These were manually ground truthed. As can be seen from the receiver operating characteristics (ROC) curves of the two video sequences (figures 5a and 5b), there is no overall loss of quality when moving from the spatial domain to the compressed domain; indeed, there is actually a small improvement in the detection rate, especially in PETS2001. Figures 6 and 7 show that the block-based approach also filters the large amount of pixel-level misclassifications in the spatial domain method. This suggests it may be safe to process the sequences directly in the compressed domain using as few as 16 DCT coefficients, and hence obtain the 170x acceleration.

VI. Conclusions

The implementation of background modeling presented in this paper uses compressed domain processing with a reduced number of DCT coefficients, plus hardware acceleration, to achieve a speed-up of up to 170x compared to a purely serial implementation of the same algorithm in the spatial domain. At 25 fps, this speed up equates to the ability to segment in real-time around 122 video streams (including the time for partial decompression), which exceeds our initial target of 64 simultaneous video streams.

The overall speed-up is the result of the combination of several different optimizations, which all complement each other. By processing at the block level instead of the pixel level, the number of modeling parameters is immediately reduced by almost a third. Also, the number of coefficients per block can be reduced from 64 to 16 while maintaining segmentation quality. Together, these algorithmic optimizations contribute a speed up of approximately 8x. The subsequent individual contributions of the two forms of parallelism depend on the order in which they are applied. Choosing vectorization first, this contributes a further speed-up of 6.7x. This is greater than the expected 4x maximum because the low level of programming required leads to reduced internal data transfers. The subsequent use of multi-core (four core) contributes a further 3.2x. It was noted that, in the spatial domain implementation, the use of the SIMD units provided only 1.3x speed up, because of the low computation to data ratio. This shows the interesting result that operating in the compressed domain can increase the potential for hardware acceleration.



(a) PETS2001 ($\alpha = 0.001, \quad T = 0.75$)

(b) BusSeq ($\alpha = 0.005, \quad T = 0.8$)

Fig. 5: ROC Curves - SD for spatial domain and CD for compressed domain (using 64, 32, 16 coefficients).

Fig. 6: PETS2001, from left to right and top to bottom: the original frame (frame no. 864), the ground truth, the result in spatial domain, the result in compressed domain using all 64 DCT coefficients, using 32 coefficients and using 16 coefficients.

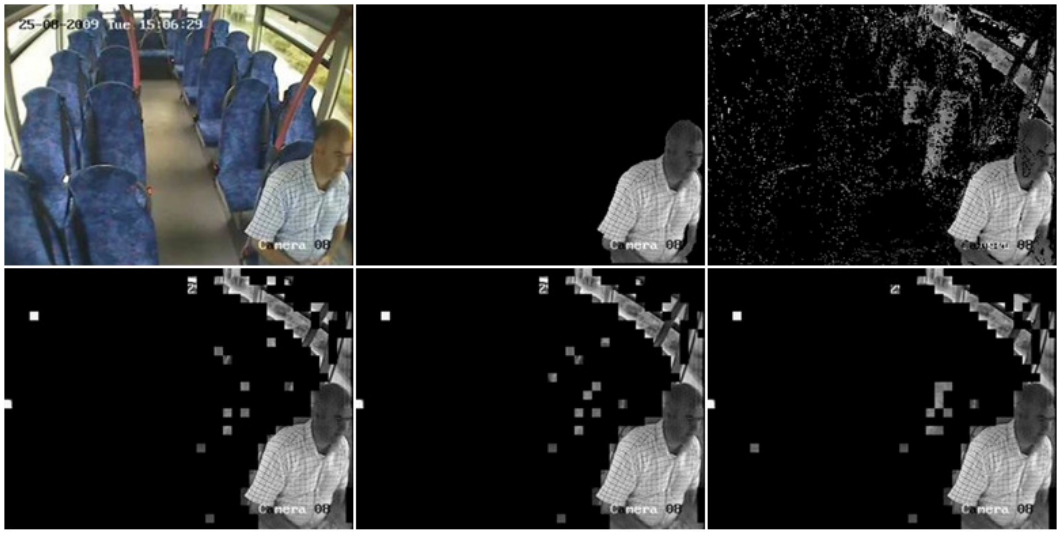


Fig. 7: BusSeq, from left to right and top to bottom: the original frame (frame no. 121), the ground truth, the result in spatial domain, the result in compressed domain using all 64 DCT coefficients, using 32 coefficients and using 16 coefficients.

From the point of view of programming effort, the use of multi-core required minimal effort. On the other hand, the use of SIMD processing was by far the most difficult task. It required very considerable programming effort and a steep learning curve because of the low level, architecture dependent nature of the programming. The effort in implementing the compressed domain processing lay somewhere between these two steps, once the partial decompression code was available.

For the case of background modeling at least, the speed up obtained did not come at the expense of a loss in segmentation quality when moving from the spatial to the compressed domain.

The partial decompression is over 4x faster than full decompression using the highly optimized JPEG decoder from Intel, which on its own represents an important speed up for any intelligent video surveillance system working in the compressed domain.

Future work will extend the foreground detection in the compressed domain to human detection and tracking for a multi-camera system.

REFERENCES

[1] M. Cristani, M. Farenzena, D. Bloisi, and V. Murino, "Background Subtraction for Automated Multisensor Surveillance: A Comprehensive Review," *EURASIP Journal on Advances in Signal Processing*, vol. 2010, pp. 1–24, 2010.

[2] H. Hassanpour, "Video Frames Background Modeling: Reviewing the Techniques," *Journal of Signal and Information Processing*, vol. 02, no. 02, pp. 72–78, 2011.

[3] S. Primechaev, A. Frolov, and B. Simak, "Scene Change Detection Using DCT Features in Transform Domain Video Indexing," in *Systems, Signals and Image Processing, 2007 and 6th EURASIP Conference focused on Speech and Image Processing, Multimedia Communications and Services. 14th International Workshop on*, 2007, pp. 369–372.

[4] M. Lamarre and J. Clark, "Background subtraction using competing models in the block-DCT domain," *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, vol. 1, pp. 299 – 302, 2002.

[5] S. Schwartz, "A Transform Domain Approach to Real-Time Foreground Segmentation in Video Sequences," *Proceedings. (ICASSP '05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, vol. 2, pp. 685–688, 2005.

[6] L. Dong and S. Schwartz, "DCT-Based Object Tracking in Compressed Video," in *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, 2006, pp. 665–668.

[7] W. Wang, J. Yang, and W. Gao, "Modeling Background and Segmenting Moving Objects from Compressed Video," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 18, no. 5, pp. 670–681, 2008.

[8] J. Meng, Y. Juan, and S. Chang, "Scene change detection in a MPEG compressed video sequence," *IS&T/SPIE Symposium ...*, vol. 2419, no. February, pp. 1–12, 1995.

[9] Y. Chen and I. Bajic, "Compressed-domain moving region segmentation with pixel precision using motion integration," in *Communications, Computers and Signal Processing, 2009. PacRim 2009. IEEE Pacific Rim Conference on*, 2009, pp. 442–447.

[10] F. Porikli and F. Bashir, "Compressed Domain Video Object Segmentation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 20, no. 1, pp. 2–14, Jan. 2010.

[11] C. Stauffer and W. Grimson, "Adaptive background mixture models for real-time tracking," in *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*. IEEE Comput. Soc, pp. 246–252.

[12] M. Poremba, Y. Xie, and M. Wolf, "Accelerating adaptive background subtraction with GPU and CBEA architecture," in *2010 IEEE Workshop On Signal Processing Systems*. Ieee, Oct. 2010, pp. 305–310.

[13] NVIDIA, "CUDA C Programming Guide." [Online]. Available: http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation

[14] D. Weimer, S. Kohler, C. Hellert, K. Doll, U. Brunsmann, and R. Krzikalla, "Gpu architecture for stationary multisensor pedestrian detection at smart intersections," *2011 IEEE Intelligent Vehicles Symposium (IV)*, no. Iv, pp. 89–94, Jun. 2011.

[15] V. Pham, P. Vo, V. T. Hung, and L. H. Bac, "GPU Implementation of Extended Gaussian Mixture Model for Background Subtraction," in *2010 IEEE RIVF International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future (RIVF)*. Ieee, Nov. 2010, pp. 1–4.

[16] Iain E. Richardson, *Video Codec Design: Developing Image and Video Compression Systems*, 2002.

[17] ISO, "ISO/IEC 10918-1:1993(E) and CCITT Recommendation T.81 (1992 E)." 1993. [Online]. Available: http://www.w3.org/Graphics/JPEG/itu-t81.pdf

[18] OpenMP - Committee, "OpenMP Specifications." [Online]. Available: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[19] R. V. D. P. B. Chapman, G. Jost, *Using OpenMP, Portable Shared Memory Parallel Programming*, 2007.

[20] Intel, "Intel 64 an IA-32 Architectures and Software Development Manuals." [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[21] Alex Klimovitski, "SSE/SSE2 Toolbox - Solutions for Real-Life SIMD Problems," 2001. [Online]. Available: http://www.thomasdideriksen.dk/misc/SIMD/sse_in_real_applications.pdf

[22] W. Eckhardt and A. Heinecke, "An efficient vectorization of linked-cell particle simulations," *Proceedings of the 9th conference on Computing Frontiers - CF '12*, p. 241, 2012.

[23] M. Alvarez and E. Salami, "Performance impact of unaligned memory operations in SIMD extensions for video codec applications," *Performance Analysis of ...*, 2007.

[24] T. Furtak, J. N. Amaral, and R. Niewiadomski, "Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms," *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures - SPAA '07*, p. 348, 2007.

Stefan Popa obtained his BSc in Robotics from University POLITEHNICA of Bucharest in 2006, and his MSc in Computer and Electronic Security from Queen's University Belfast in 2011. He is currently studying for a PhD in the Institute for Electronics, Communications and Information Technology (ECIT), Queens University Belfast. His current research interests are image processing in the compressed domain, high performance video processing, and video surveillance.

Prof. Danny Crookes was appointed Professor of Computer Engineering in 1993 at Queens University Belfast, and was Head of Computer Science from 1993-2002. He is currently Director of Research for Speech, Image and Vision Systems at ECIT. His research interests include the use of novel architectures (including FPGAs, multi-core and GPUs) for high performance video and speech processing. Danny Crookes has over 200 scientific papers in journals and international conferences. He is currently involved in projects in automatic shoeprint recognition, speech separation and enhancement, and high performance processing of 4D confocal microscopy imagery.

Dr. Paul Miller is a Senior Lecturer in the School of Electronics, Electrical Engineering and Computer Science at Queens University Belfast (QUB). He is also Research Director of the Intelligent Surveillance Systems group in Centre for Secure Information Technology. He has published over sixty papers in image and video analysis, including a best paper award for his work on object recognition. He received his PhD in Optical Image Processing from QUB in 1989. He worked as a senior research scientist at the Defence, Science and Technology Organisation, Australia where he led a team on unmanned aerial surveillance systems. Since returning to academia he has continued to work in video analytics for both defence and civilian CCTV applications, and also bio-medical image analysis. His research interests include image restoration, segmentation, multi-camera tracking and gender/age profiling of subjects in video.