VIDEO SURVEILLANCE APPLICATION BASED ON APPLICATION SPECIFIC VECTOR PROCESSORS

Roman Bartosinski, Martin Danek, Jaroslav Sykora, Lukas Kohout

Department of Signal Processing
Institute of Information Theory and Automation (UTIA) of the ASCR, v.v.i.
Pod Vodarenskou vezi 4, Praha, Czech Republic
{sykora, kohoutl, bartosr, danek}@utia.cz

Petr Honzik

CIP plus s.r.o.
Milinska 130, Pribram, Czech Republic
petr.honzik@cip.cz

ABSTRACT

Current video surveillance applications put higher demand both on processing power and personal privacy. This results in new video processing solutions being based on smart cameras. This paper presents a sample implementation of a system that implements core functions of a smart camera - motion detection and labelling - in an FPGA. The implementation is based on the data-flow ASVP platform extended with a number of selection operations that enable to implement constructs with conditional branching. Experimental performance results and power consumption data are presented for an actual implementation in the Xilinx SP605 board.

Index Terms— video surveillance, smart camera, FPGA, custom accelerators, vector processing

1. INTRODUCTION

Current video surveillance systems belong to two classes based on two technology concepts. The first and older concept is based on analog video systems that have been widely used up till now. The second more recent concept is based on the IP network technology, which enables much more variability in configuration and communication among all components connected in a network. IP-based video systems will slowly replace all analog video systems. In our approach we assume that in the future any network type will be available anywhere in the public and private space. All electronic appliances will be able to connect to each other over the network. Because of this reason in our design we need to consider autonomous equipment that can decide and connect to other devices in the network. In the future, as the number of monitored areas increases, there will be no chance to keep the current concept of video surveillance that relies on transferring complete image data from a big pool of cameras to one remote processing unit that processes all the data from the cameras as it would increase extremely the communication demands on the network, and its reliability is limited by the central processing node. Moreover, transferring all video data over a network brings in a big security risk. For this reasons we need to process video data and make decisions locally, and send only a minimum amount of data to the network to increase safety, processing capacity and dependability of the whole system. To sum it up, significant features of modern video surveillance systems are

- autonomous execution,
- local decision making,
- decentralization,
- increased privacy,
- cooperation with the outside world.

Privacy is one of todays biggest problems, and most video surveillance system cannot meet strict rules dictated by privacy of personal spaces. Current video surveillance systems collect, store and send image data, which all increases the potential for illegal privacy intrusion. Because of this reason current video surveillance systems cannot be used to detect dangerous situations in many places like private rooms or social facilities, and there we have to count only on personal involvement of persons present there. If we are able to process video data locally and send out just detected events, we will be able to increase the use use of video surveillance systems in these private spaces.

Modern video surveillance systems built to respect privacy issues must be based on *smart cameras*. A smart camera consist of an input device (mostly represented by a digital camera chip) and input video processing chain. These parts have crucial impact on the quality of outputs (events) generated by the system.

Figure 1 shows an example of an input video processing chain that we will consider in the remainder of this paper. The input video processing chain consists of several processes that can be seen as three processing levels [1]. Generally, the amount of data in the chain decreases and the algorithm complexity increases with a switch to a higher processing level.

This work has been supported from project SMECY, project number Artemis JU 100230 and MSMT 7H10001.

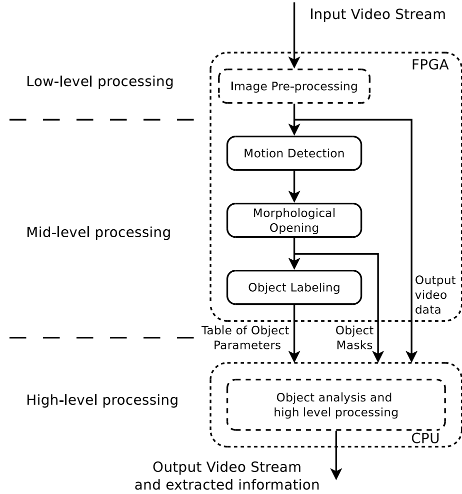

Fig. 1. Possible structure of an input video processing chain. Arrangement of component blocks in hardware resources used in our case.

Processes on the low and middle levels are usually where the data throughput bottleneck is. On the other hand, processes on the lower levels can be often parallelized.

The low-level processing usually contains simple operations that are repeated for each pixels (e.g. color transformations, filtering). As these processes do not need any additional memory to store contextual data or they need to store only several common parameters, they can be implemented as high-speed pipelined IP cores in an FPGA.

On the other hand, mid-level processes (e.g. object segmentation and tracking) require more complex and iterative or recursive algorithms with additional contextual data. In general, mid-level algorithms still operate on the whole video data. More sophisticated algorithms need higher amount of contextual history data than can be extracted from a single input frame.

The core block in the mid-level processing of video surveillance applications is *motion detection*. This process creates a mask of moving objects from a sequence of time-delayed input frames. The mask is usually computed using statistical algorithms. The noise in such a result is usually cleaned in the next operation, traditionally *morphological opening* is employed. Then connected pixels are identified as objects by a *labeling* algorithm. The output from this processing level is a mask of foreground objects and a table with parameters of individually identified objects.

The high-level processes (e.g. object analysis and recognition) often involve complex sequential algorithms that usually execute on small regions determined in the middle level; these are best suited to be implemented in a CPU.

The paper is organized as follows: We begin by brief description of selected hardware platform in section 2, and continue with description of application in section 3. In section 4 details about implementation are described, and several results are presented and commented in section 4.3.

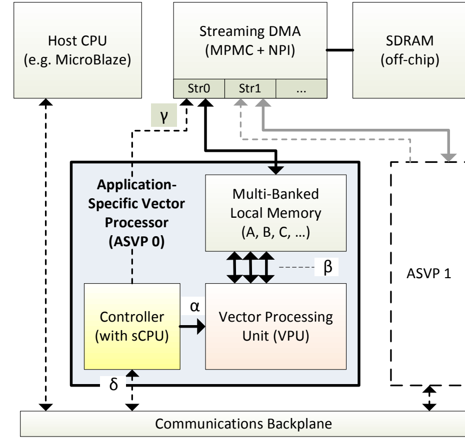

Fig. 2. A system-level organization of an ASVP-based core.

2. PLATFORM

The implementation platform used in this paper is the Application-Specific Vector Processor (ASVP) described in [2] and [3]. In traditional work-flows based on direct implementation in hardware description languages hardware accelerators must be recompiled and re-synthesized each time to generate a new FPGA configuration (bitstream) that can be verified. The drawback of the traditional approach is the long synthesis time caused mainly by a slow place&route process of the low-level tools. This also limits the end user to minor changes of the implemented algorithm such as tuning of coefficients. In our approach we abstract the custom accelerator into a specialized programmable architecture based on a network of programmable data-streaming computing nodes with local memory. To design for the platform, in the first step the high-level source code is analyzed and domain features are extracted. Based on the required domain features new computing kernels are implemented in the computing nodes or the existing ones are modified. The architecture is programmable by firmware to the extent that the sequencing of basic processing kernels can be changed without re-synthesizing the hardware, hence source code modifications that do not change the problem domain can be tested quickly for they require only fast firmware recompilation.

The system-level view is presented in Figure 2. Similar to the streaming architectures (Stanford Imagine [4], IBM Cell [5]), the execution control is hierarchical:

1. *Task scheduling* is executing in the Host CPU (e.g. MicroBlaze). Optional inter-core synchronization is handled by the Communications Backplane (δ).
2. *Scheduling of the vector instructions* is realized in a

Table 1. Required frame rate versus object motion speed.

Type of object	Speed	Frame rate
People	5 km/h	5 fps
Cars in a city	50 km/h	23 fps
Cars on a highway	130 km/h	60 fps

simple scalar control processor (sCPU) embedded in each ASVP core .The sCPU forms and issues wide instruction words (α) to the Vector Processing Unit (VPU).

3. *Data path multiplexing* and vector processing is realized autonomously in the VPU. The unit handles both the vector-linear and vector-reduction operations, as well as local memory banks access scheduling (β). The VPU contains Data Flow Unit (DFU) which performs vector operations.

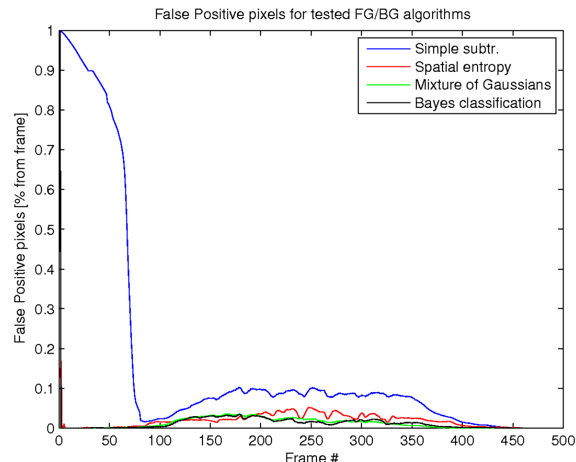

Fig. 3. False positive error frequency for a testing video sequence.

3. APPLICATION

3.1. Requirements

We have performed several tests to measure the speed of video processing required in real-world situations. The main parameter for video surveillance is the frame rate of the video processing block and video surveillance camera. The minimal frame rate of the video surveillance camera can be set according to the setting. 5 fps is valid for common use in public spaces with people walking. In the case of moving cars or objects moving faster than pedestrians we need a higher frame rate. From our tests and experience it is desirable that in two subsequent frames the distance objects travel be 0.3m for people and 0.6m for cars. Table 1 shows the frame rate in three most common scenarios.

The next parameter is the minimal required resolution of the input video. The increasing demands on video quality imply the increase in the resolution of the input video stream. Currently cameras with the full HD resolution are often used as input of video stream. Hence we need to consider image resolutions starting at 640x480 pixels.

Another strong requirement is that the used algorithm must not exhibit faulty behaviour with varying external conditions such as time-varying lighting. The implemented algorithms should be adjustable by the user.

The main idea is that the application should respect privacy issues and provide image information only if a monitored event occurs.

To fulfil the stated requirements and in line with [1] we have selected the FPGA technology for implementating the low and mid-level processes (shown in Figure 1); the high-level analyses will be executed in a CPU.

As it is not efficient to design, implement and debug complex function cores directly on an FPGA (i.e. in a hardware description language), we have developed and analyzed all algorithms in Matlab, and then we have implemented them in hardware. The next subsections analyze algorithms used in the processing blocks in our video chain.

3.2. Motion Detection

The motion detection process detects changes in an input video stream and returns mask of presently changed regions in the frame. This process is often called foreground-background segmentation, where static regions are marked as background and changes as foreground objects. There are many algorithms for motion detection, from simple background subtraction to complex algorithms which are based on statistical models of the background. In general, simpler algorithms are faster and need smaller amount of memory to store their contextual data. More robust and sophisticated algorithms are more computationally intensive and require higher amount of data memory. All algorithms use one or more parameters (mainly thresholds) that are adjustable and must be set by the user, but the quality of their values is crucial to get useful results. These parameters are commonly set experimentally.

We have analysed several algorithms for their computation complexity, data consumption and robustness. In the analysis we used the settings recommended in papers [6], [7], [8] where the algorithms have been described. The algorithms and the required memory size for their contextual data are listed in Table 2. The table also contains calculated sizes for the recommended parameters and color input video with resolution 640x480 pixels. Figure 3 demonstrates the basic quality of analysed algorithms, it shows the number of false positive errors each algorithm produced in relation to the video resolution. The algorithms have been tested on artificial video sequences prepared in a scene modelling program.

Table 2. The analyzed algorithms for motion detection and memory required to store their contextual data normalized wrt the size of the input video frame. The last column shows the size of contextual data for the recommended settings and video resolution 640x480.

Algorithm	Normalized size of contextual data	Size for 640x480
Simple background subtraction	1	900 kB
Spatial temporal entropy image [6]	N+1/3 (for N quantization levels)	23.7 MB
Mixture of Gaussians [7]	8*K (for K Gaussian models)	37.5 MB
Modified Mixture of Gaussians	5*K (for K Gaussian models)	23.4 MB
Bayes classifications [8]	3*N+4*M+3 (for N color vectors and M co-occurrence vectors)	247.5 MB

Except simple background subtraction all tested algorithms require floating point arithmetic.

Based on the analysis we have selected the algorithm based on the Mixture of Gaussians (*MoG*) as a compromise between computational complexity, required memory for contextual data, used operations and the possibility to transform the algorithm to a vector form. The selected algorithm has been modified to decrease its memory requirements. The algorithm is based on [7] without shadow detection. This algorithm belongs to the class of *pixel motion detection* algorithms which consider all pixels as independent.

The algorithm is executed for each new frame from an input video in RGB or another three-component color space. The output of the algorithm is a mask of foreground objects in the frame.

The decisions if pixels from the current frame represent foreground or background depend on statistical models and their mixture for each pixel. Each pixel is modeled by a mixture of K strongest Gaussians models of the background in the pixel (K=4 in our implementation). Each Gaussian model defined by its mean value and variance represents one state (color) of a pixel. Each model also contains a parameter *weight* which represents how often the particular model classified the pixel as the background.

Four models are sufficient to describe basic and repeating changes of the scene background such as trees in air, moving escalators.

The algorithm tests if the current pixel value belongs to one of the already existing pixel models. It tests models sequentially from the strongest to the weakest one. If there is an appropriate model, the model is updated with the new value (the mean value and variance are updated and the weight of the model is increased). The model is updated with a given learning rate to perform progressive adaptation to new conditions. If none of the models describes the pixel, the weakest model is replaced with a new one which is built based on the current pixel value and with the default weight. After updating the models, they are reordered from the strongest to the weakest one, and their weights are normalized. Then the algorithm tests if the weight of the model (or mixture of stronger models) that describes the new pixel value is higher than the user's threshold; if so, the pixel is classified as background. Otherwise it is classified as a foreground object.

3.3. MoG Implementation

In its basic form the algorithm is suitable for sequential processing. For processing on the proposed ASVP architecture and with respect to the requirements it has been modified and transformed to a vector form. The main parts changed are conditional branching, reordering of the models, replacement of the operations divide and square root.

Each model in the original version is described by its mean value and variance in each color channel, weight and sort key. Thus each model is represented by eight single precision floating point (FP) numbers. In our modified version of the algorithm we reduced the model data to weight, mean value in each color channel and a common sum of variances, which is five FP numbers that equal to the reduction of the transferred contextual data by about 37,5%.

In the original version the operation *sqrt* was used to compute sort keys as a relation between the weight and variances of a model. Our modified algorithm uses only the weight to find the strongest and the weakest models. The modified version does not use the division to normalize weights; this is possible because the algorithm is recursive, and the weights of the models are updated in each step (with each new frame) by multiplying them with the forgetting factor that is always smaller then 1, thus the sum of weights for a given pixel cannot be greater than the number of models (four in our case).

The behaviour of both the original and the modified algorithms can be tuned with several parameters: background threshold, variance threshold, learning rate, initial weight, and initial variance.

The modifications of the algorithm have impact on the behaviour of the algorithm and its convergence, but the impact can be partially compensated by slightly adjusting parameters of the algorithm. The modified algorithm returns results with a slightly higher error constituent, it returns about 1% more false positive and about 1% fewer false negative errors.

3.4. Morphological Opening

The second block in the mid-level processing (Figure 1) performs morphological opening. It cleans the mask of foreground objects from the smallest objects which are mostly manifestations of noise or statistical errors. The opening algorithm is the dilation of the erosion of an input image. These operations are well known, and their description is included

in many image processing textbooks (see for example [9]).

3.5. Object Labelling

The input to most object labelling algorithms is a binary mask of objects. Some of the more complex algorithms use both the binary mask and the original colour (or greyscale) image as their input. Each algorithm produces either a color mask where each separate object has a unique color, or it produces a table of objects with their description. The minimal description should contain a bounding box of each object (object position and its size) and its center of mass. To generate a colour mask object colouring must be executed in two sequential steps. In the first step the algorithm marks objects with consecutive colours and produces a table of equivalent colours. In the second step the objects are re-mapped to the colour that described the particular object in the previous step. After that other characteristics can be computed for each object. Common labelling algorithms based on colouring the input image is processed by lines, pixel by pixel. The 8-neighbourhood is used in most cases while the 4-neighbourhood is used rarely. If a pixel is neighbouring with another pixel that has already been assigned a colour, the pixel is coloured with the same colour, otherwise a new colour is assigned to the pixel. If the pixel has more pixels with different colours in its neighbourhood, the algorithm selects one colour and saves all the other colours to a temporary table of colours assigned to each object.

If we need only the basic characteristics (bounding box, center of mass, volume), we can use a simplified algorithm that colours objects and computes their characteristics in just one step. All these characteristics can be computed and updated recursively, therefore the labelling algorithm can also compute the other characteristics. The algorithm is based on a static table of object characteristics, where one of the characteristic is colour used in the algorithm, and objects are coloured with more colours saved in the table separately.

The output of the algorithm is a table that contains the bounding box, mass center, object volume and colour of each object.

4. IMPLEMENTATION AND RESULTS

The ASVP approach first constructs a programmable architecture customized for a given application, then employs software techniques to develop firmware that implements the algorithm.

In this methodology we need to describe the algorithm as a sequence of vector operations running in the computing nodes because the synthesized hardware must support all the required operations. Then the algorithm is written as a firmware with a sequence of set-up and computing operations in the VPU. The firmware also controls data transfers between the off-chip shared memory and local memories

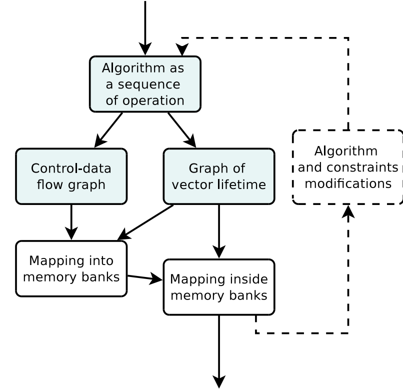

Fig. 4. Diagram of mapping process.

through the streaming DMA. Because the ASVP uses dual-port local memories, operation processing and data transfers from or to the shared memory can overlap. The data throughput can be maximized by optimizing this overlap. The speed of computation and data transfer minimization depends on the mapping of the data vectors to the accelerator local memory. The best performance of the accelerator is achieved for long data vectors that minimize data transfers between the local and shared memories.

The mapping problem can be solved through colouring of the data-dependency graph constructed from the sequence of operations and their variables. As the graph colouring is an NP-complete problem, the mapping is solved by heuristic algorithm.

If a solution is not found or if we want to optimize vector lengths, we can either insert the *VCOPY* operation to the operation sequence or use the time domain access scheduling in the VPU crossbar or try a different colouring.

Figure 4 depicts the mapping process. After all variables are mapped to the local memories, the data transfer schedule can be derived and a firmware program with operations and data transfers can be designed. In the future the firmware will be generated automatically from the input sequence of vector operations.

The whole video processing chain is developed and implemented on a Xilinx SP605 development board with a low cost, low power Xilinx Spartan 6 FPGA. The MicroBlaze soft-core processor is employed as the host CPU. Each algorithm is implemented in a separate ASVP. The entire system runs at 50MHz except the data flow units that run at 100MHz. All accelerators are synthesized with four memory banks with 1024x32bit words.

4.1. Foreground/Background Segmentation

The architecture of the accelerator contains small local memories relative to the size of the input frame, therefore each frame must be processed in tiles with *N* pixels. In our im-

plementation, each tile contains 50 pixels. The size of the tile is the maximal possible in order to reduce data transfers. Mappings and lifetimes of variables in the implemented MoG algorithm are shown in Figure 5. In the figures, the X axis represents the time in the execution steps of the algorithm, and the vertical axis corresponds to the offset in memory. The second memory bank (memory B in Figure 5) is occupied mainly by the pixel models of the background. Our version of the algorithm uses 4 models for each pixel. Each model is defined by its mean value for R,G,B color components, common variance (placed in the first memory bank) and model weight.

The input data and models for pixels in each tile must be transferred from the shared off-chip memory to the local memories. Then the accelerator executes the modified MoG algorithm for all pixels in the tile treated as vectors, then the updated models are saved back to the shared off-chip memory.

We have modified the algorithm to eliminate conditional branches, hardware-expensive operations (division, square root) and reordering. We also had to extend the basic ASVP platform with a number of instructions described below (also see Table 3). Branches are reimplemented so that the algorithm speculatively computes both possibilities (branch taken and not taken), and with a special operation *VSELECT* it selects the proper result from both branches according to the condition. Reordering of the models has been replaced with a selection of the strongest and the weakest models from all pixel models. The operations *INDEXMAX* and *INDEXMIN* have been added for this reason. These operations return the integer index of the element where the maximal/minimal value is located. The integer index can be used in subsequent operations that work only with a selected model.

Another example of application-specific vector instructions required by the MoG algorithm is the group of VCONVR / VCONVG / VCONVB instructions. These instructions take a 32-bit word that represents one pixel, extract a given 8-bit colour (R, G, or B), and convert the colour to a floating-point value. The VCMPLT (compare-less-than) operation compares two vectors element-wise, and returns a vector of boolean values. The VSELECT operation is a vectorized conditional ternary operator as defined in the C language.

Given a set of operations and their high-level specifications in Table 3, the hardware implementation of the customized DFU can be generated. Currently this is done mostly manually in VHDL, however, it should be possible to synthesize the DFU automatically in a tool (this is our future work).

4.2. Morphological Opening

In our implementation, we want to be able to set the size of the structure element for morphological operations. Therefore we use a 3x3 square with the origin in its center as a structure element. It has a feature that a sequence of N repetitions of morphological operation with the basic structure element can be substituted for an operation with an N times

Table 3. Specific operations implemented in the DFU for motion detection.

Operation	Definition
VMAX	$A_0 \leftarrow \Psi_{\mathrm{Max}}(B_i)$
VMIN	$A_0 \leftarrow \Psi_{\mathrm{Min}}(B_i)$
INDEXMAX	$A_0 \leftarrow Arg\{\Psi_{\mathrm{Max}} B_i\}$
INDEXMIN	$A_0 \leftarrow Arg\{\Psi_{\mathrm{Min}} B_i\}$
VCMPLT	$A_i \leftarrow (B_i < C_i)\ ?\ T : F$
VSELECT	$A_i \leftarrow (B_i \neq 0)\ ?\ C_i : D_i$
VGTE	$A_i \leftarrow (B_i < C_i) ? C_i : B_i$
VBOR	$A_i = bitwiseOR(B_i, C_i)$
VBNOT	$A_i = bitwiseNOT(B_i)$
VCONVR	$A_i \leftarrow int2float((B_i >> 16) \& \mathrm{0xFF})$
VCONVG	$A_i \leftarrow int2float((B_i >> 8) \& \mathrm{0xFF})$
VCONVB	$A_i \leftarrow int2float(B_i \& \mathrm{0xFF})$

Table 4. Operations implemented in the DFU for morphological opening. AND and OR are logical operations, i.e. the result of the AND operation is TRUE if all operands are TRUE.

Operation	Definition
VCOPY	$A_i \leftarrow B_i$
VAND3H	$A_i \leftarrow and(B_i, B_{i+1}); i = 0$
	$A_i \leftarrow and(B_{i-1}, B_i, B_{i+1}); i \in (0, N-1)$
	$A_i \leftarrow and(B_{i-1}, B_i); i = N-1$
VAND3V	$A_i \leftarrow and(B_i, C_i, D_i)$
VOR3H	$A_i \leftarrow or(B_i, B_{i+1}); i = 0$
	$A_i \leftarrow or(B_{i-1}, B_i, B_{i+1}); i \in (0, N-1)$
	$A_i \leftarrow or(B_{i-1}, B_i); i = N-1$
VOR3V	$A_i \leftarrow or(B_i, C_i, D_i)$

bigger structuring element.

The opening operation consists of the erosion and the dilation operations. We work with binary images, that is in binary morphology. This means we can replace the erosion operation with the logical *AND* among neighbouring elements, and similarly we can replace the dilation operation with the logical *OR* among neighbouring elements.

To maximize the ASVP utilization we have implemented four new operations in the VPU. These operations allow the VPU to treat each line in the image as one vector, each line will be read from and written to the shared memory only once. As the morphological operations described here are fixed-point by nature, they are executed in a separate computing node that implements just these new operations to save resources (compared to the floating-point computing nodes with operations described in the previous section).

Table 4 shows the instructions implemented in the DFU that are used in the Opening function. Operations VAND3H and VOR3H get three consecutive elements from an input vector (line of image) and set the corresponding element in the output vector if all three elements are set (VAND3H) or one of them is set at least (VOR3H). On the contrary, operations VAND3V and VOR3V process tree elements with the same index from three vectors (consecutive lines of the input image).

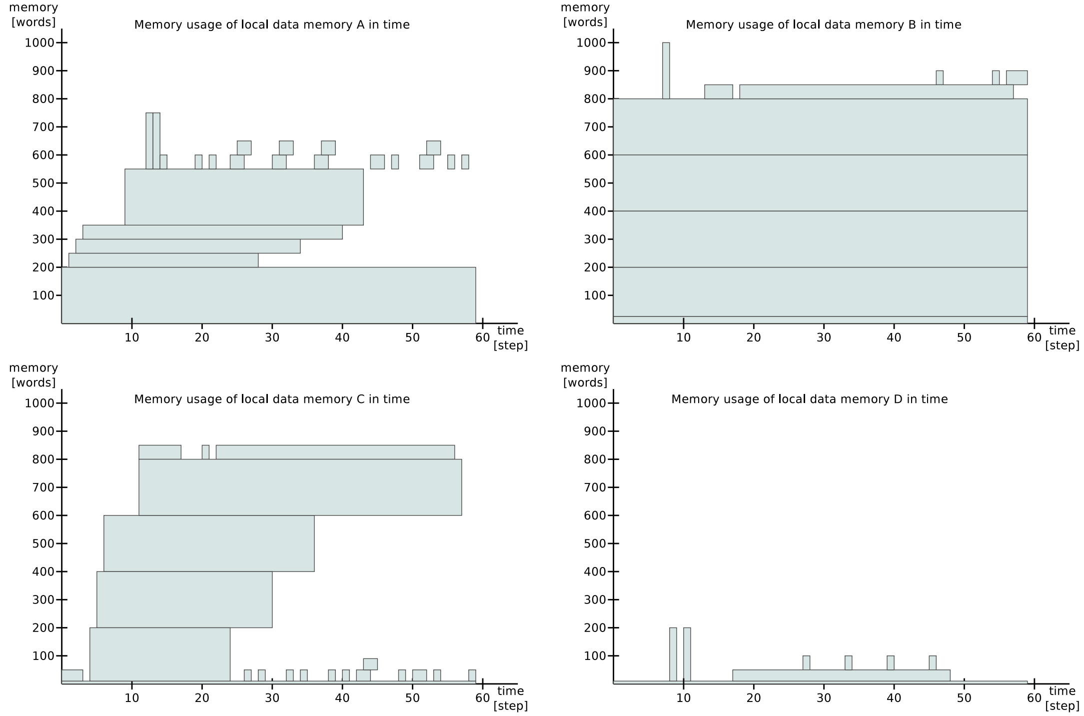

Fig. 5. Mapping of variables in local memories. Each box represents one variable.

Figure 6 shows the firmware pseudocode for one part of the opening function. It performs the erosion operation on an input image in $InputBuffer$ (resolution $W \times H$) with a full square structure element of size $1 + 2 * N$.

Fig. 6. ASVP firmware pseudocode for the erosion operation used in opening function.

```
SetReadingFrom(InputBuffer)
SetWritingTo(TmpBuffer1)
for (cn=0:N-1) {
 /* initiate */
 Line1 = VCOPY(cZero),W
 ReadLine(Out),W
 Line2 = VAND3H(Out),W
 ReadLine(Out),W
 Line3 = VAND3H(Out),W
 WriteLine(Line1),W
 /* process entire image */
 for (y=1:H) {
  Line1 = VCOPY(Line2),W
  Line2 = VCOPY(Line3),W
  ReadLine(Out),W
  Line3 = VAND3H(Out),W
  Out = VAND3V(Line1,Line2,Line3),W
  WriteLine(Out),W
 }
 Line3 = VCOPY(cZero),W
 WriteLine(Line3),W

 if (cn & 1) { /* switch buffers */
  SetReadingFrom(TmpBuffer2)
  SetWritingTo(TmpBuffer1)
 } else {
  SetReadingFrom(TmpBuffer1)
  SetWritingTo(TmpBuffer2)
 }
}
```

4.3. Results

The data path of the implementation is shown in Figure 7. The labelling process is implemented in software executed in MicroBlaze in the current version.

As mentioned above the video application has been implemented in the system on a chip with hardware compute cores on a Xilinx SP605 development board. The board contains power management chip accessible through PMBus. Power consumption has been measured by this chip on power rail that supplies power to the FPGA internal logic.

The average power consumption of the entire system on a chip with disabled accelerators is $Pavg = 433.2mW$. Table 5 contains performance data for the motion detection algorithm executed on one, two and three floating-point compute cores for an input video stream with resolution of 640x480 pixels. It is shown that the time required to process one frame is a multiple of the number of used compute cores, but the consumed energy is the same. The initial requirements for the minimal frame rate 5 FPS can be reached with three cores running in parallel; each core computes one-third of each frame.

Table 6 shows average of consumed energy for processing one frame in opening function. The current implementation

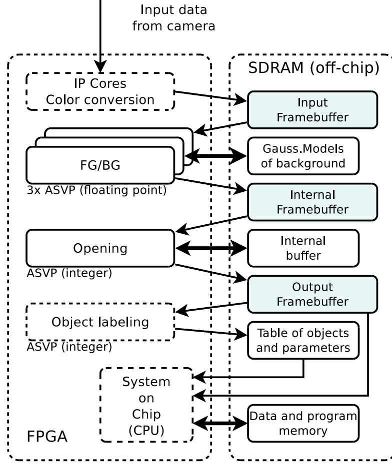

Fig. 7. The data path of the video chain.

Table 5. Motion detection. Implementation parameters measured on Xilinx SP605.

Parameter	Number of used ASVP		
	1	**2**	**3**
$FPS[s^{-1}]$	2.07	4.14	6.21
$P_{SoC}[mW]$	460.26	487.58	509.68
$P_{ASVP}[mW]$	27.061	54.38	76.48
Computation of one frame			
$t[s]$	0.436	0.218	0.146
$E[mJ]$	11.81	11.87	11.13

of opening function reaches frame rate 11.51 FPS. Consumed energy is 1.67 mJ for one frame and it is only one tenth of energy consumed by motion detection operation.

5. CONCLUSIONS

In this paper we have described an implementation of a video processing chain to be used in smart camera-based video surveillance applications. The implementation is based on the Xilinx SP605 development board. The implementation of the video processing chain is based on the ASVP platform, with the MicroBlaze running at 50MHz and the computing nodes running at 100MHz. Performance results for motion detection and morphological opening have been shown to meet the initial design requirements, the power consumption data indicate that our implementation is suitable for low-power embedded devices.

Table 6. Morphological opening process. Implementation parameters measured on Xilinx SP605.

Parameter	Value
$FPS[s^{-1}]$	11.51
$P_{SoC}[mW]$	452.419
$P_{ASVP}[mW]$	19.219
Computation of one frame	
$t[s]$	0.0868
$E[mJ]$	1.67

6. REFERENCES

[1] D. Fábio Real and François Berry, "Smart Cameras: Technologies and Applications," in *Smart Cameras*, Ahmed N. Belbachir, Ed., pp. 35–50. Springer US, 2010.

[2] Jaroslav Sykora, Lukas Kohout, Roman Bartosinski, Leos Kafka, Martin Danek, and Petr Honzik, "The Architecture and the Technology Characterization of an FPGA-based Customizable Application-Specific Vector Processor," in *Proceedings of the 2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits and Systems*. 2012, DDECS '12, pp. 62–67, IEEE.

[3] Jaroslav Sykora, Lukas Kohout, Roman Bartosinski, Leos Kafka, Martin Danek, and Petr Honzik, "Reducing Instruction Issue Overheads in Application Specific Vector Processors," in *Proceedings of the 15th Euromicro Conference on Digital System Design*. 2012, DSD '12, pp. 600–607, IEEE Computer Society.

[4] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo López-Lagunas, Peter R. Mattson, and John D. Owens, "A bandwidth-efficient architecture for media processing," in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, Los Alamitos, CA, USA, 1998, MICRO 31, pp. 3–13, IEEE Computer Society Press.

[5] H. Peter Hofstee, "Heterogeneous Multi-core Processors: The Cell Broadband Engine," in *Multicore Processors and Systems*, Stephen W. Keckler, Kunle Olukotun, and H. Peter Hofstee, Eds., Integrated Circuits and Systems, pp. 271–295. Springer US, 2009.

[6] Guo Jing, Chng E. Siong, and D. Rajan, "Foreground motion detection by difference-based spatial temporal entropy image," in *TENCON 2004. 2004 IEEE Region 10 Conference*, 2004, vol. A, pp. 379–382 Vol. 1.

[7] P. Kaewtrakulpong and R. Bowden, "An Improved Adaptive Background Mixture Model for Realtime Tracking with Shadow Detection," 2001.

[8] Liyuan Li, Weimin Huang, Irene Y. H. Gu, and Qi Tian, "Foreground object detection from videos containing complex background," in *Proceedings of the eleventh ACM international conference on Multimedia*, New York, NY, USA, 2003, MULTIMEDIA '03, pp. 2–10, ACM.

[9] M. Sonka, V. Hlavac, and R. Boyle, *Image Processing, Analysis, and Machine Vision*, Chapman & Hall, 2 edition, 1998.