Computing and Informatics, Vol. 35, 2016, 1141–1159

EVALUATION OF MOG VIDEO SEGMENTATION ON GPU-BASED HPC SYSTEM

Mirosław Jabłoński, Jaromir Przybyło

AGH University of Science and Technology
Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering
Mickiewicza Ave., 30
30-059 Kraków, Poland
e-mail: {mjk, przybylo}@agh.edu.pl

Abstract. Automated and intelligent video surveillance systems play an important role in the modern world. Since the number of various video streams that must be analyzed concurrently grows, such systems can assist humans in performing tiresome tasks. In order to be effective, video surveillance systems have to meet several requirements: they must be accurate and able to process the received video stream in real-time. A robust system should not depend on lighting conditions, illumination changes and other sources of scene variation. A common component of surveillance systems is a module that performs background estimation and foreground segmentation. The MoG (Mixture of Gaussians) algorithm is a widely used statistical technique of video segmentation. The estimation process is time-consuming, especially for complex mixture models containing many components. The work presented here focuses on the performance evaluation of MoG algorithm aiming to assess feasibility of OpenCL-based processing of high resolution video on GPU accelerated platforms.

Keywords: MoG, Gaussian mixture model, OpenCL, GPU, video segmentation

Mathematics Subject Classification 2010: 62H35

1 INTRODUCTION

One of the frequently used algorithms in video-surveillance [3, 5] and multimedia systems [1] is MoG (Mixture of Gaussians) background modelling [15]. It allows

to track the evolution of the background and detect foreground objects moving across a scene. Numerous improvements of the original method have been proposed to enhance segmentation quality and reduce processing time. Most articles [2, 9] report implementations able to process consumer video formats in real-time when accelerated with FPGAs or CUDA framework on GPU. Latest work [6] shows FPGA and ASIC embedded implementations of GMM (Gaussian mixture model) processing HD video in real time. These were targeted for various platforms, manufacturing technologies and optimized for speed or silicon area.

In this paper, we present performance evaluation of MoG algorithm on GPU-based HPC computing system and commodity PC, aiming to assess feasibility of OpenCL-based processing of high resolution video.

2 BACKGROUND MODELLING USING MIXTURE OF GAUSSIANS

The MoG video segmentation algorithm belongs to statistical background modelling methods, where statistical variables are used to classify the pixels as foreground or background. The history of intensity values per each pixel is modelled as MoG (Mixture of Gaussians) corresponding to various image content (i.e. background, moving objects, shadows). In case, when each pixel is characterized by its gray level intensity or component color space (e.g. RGB, HSV, etc.), the probability of observing the particular pixel value is considered given by the formulas (1) and (2).

$$P(X_t) = \sum_{i=1}^{K} \omega_{i,t} \eta \left(X_t, \mu_{i,t}, \Sigma_{i,t}\right), \tag{1}$$

$$\eta \left(X_t, \mu, \Sigma\right) = \frac{1}{2\pi^{\frac{n}{2}} \left|\Sigma\right|^{\frac{1}{2}}} e^{-\frac{1}{2}(X_t-\mu_t)^T \Sigma^{-1} (X_t-\mu_t)} \tag{2}$$

where:

- K – number of distributions,
- X_t – pixel value as a function of time t,
- $P(X_t)$ – probability of observing particular pixel value,
- ω – weight,
- η – probability density function,
- $\mu_{i,t}$ – expectation (mean value) of i^{th} component at time t,
- n – number of X_t image channels: 1 for gray level, 3 for color video,
- Σ – covariance matrix.

Each pixel is characterized by a mixture of K Gaussians. In the original paper [15], authors assumed that RGB color components are independent and have the same variance, i.e. $\Sigma = \sigma^2 \mathbf{I}$. Every new pixel value, X_t, is then checked against existing K Gaussian distributions, until a match is found (pixel value is within

2.5 standard deviations of a distribution). Then, the background model – i.e. the parameters of the MoG's $(\omega, \mu_{i,t}, \Sigma)$ – is updated using modified online EM (Expectation Minimization) algorithm. The original Mixture of Gaussians algorithm has been improved by many authors. The extended survey of proposed extensions is presented in [2].

3 GENERAL PURPOSE COMPUTING WITH OPENCL

OpenCL exploits possibility of parallel computing with various processing elements. Primarily it was implemented for GPU devices. Later, implementations for CPUs have been added. Recently, its viability on FPGA platforms have been announced [4, 13]. OpenCL Application Programming Interface, data formats and dimensioning of data parallelism reflect features specific to GPU devices that are already supported natively by OpenGL standard dedicated to 2D and 3D graphics rendering.

Popular alternative for GPGPU computing is CUDA (Compute Unified Device Architecture environment) which is dedicated solely to NVidia GPU devices. It provides rich library of highly optimized procedures for general purpose computing. This is why it is frequently chosen for accelerating algorithms [7, 11]. Another framework for general purpose computing on GPU devices is DirectCompute library [8] that uses DirectX environment.

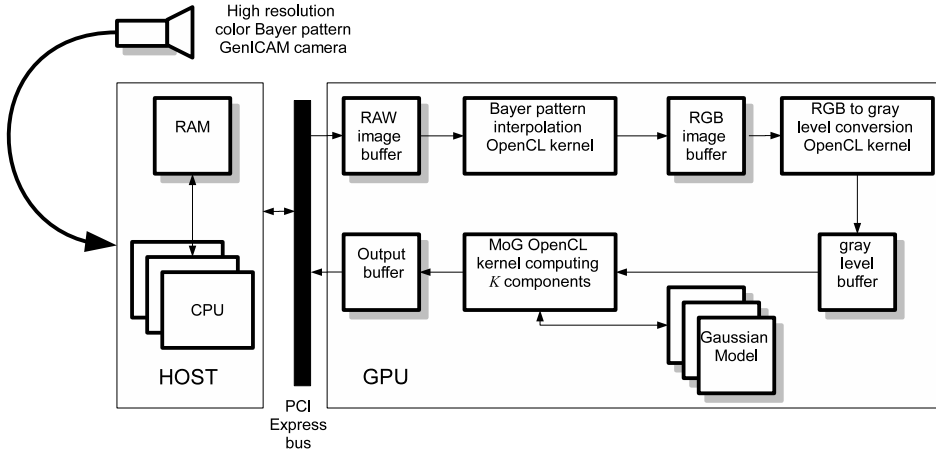
OpenCL computing model assumes execution of OpenCL kernel on number of parallel, so called, computing devices which may be of the following types: GPU, multicore CPU, multicore DSP or FPGA. OpenCL kernels that use generic data types and basic features are portable across multiple types of computing devices. However, there are some dedicated data types and built-in functions supported solely by GPUs, due to their specific architecture, e.g. texturing units, samplers, etc. that are not present in generic CPU architectures.

GPU pipeline is well suited for processing rasterized images, both artificial and real ones. In OpenGL, this final stage of 3D graphics rendering is called pixel shading. It is worth mentioning, that prior GPGPU computing interfaces were standarized in form of CUDA, OpenCL or DirectCompute frameworks, GPU shader units were used for accelerating computer vision [17]. The algorithm functionality was implemented in a form of pixel shader programs operating on textures containing image pixels. Both CUDA and OpenCL implement the concept of shaders in a form of so called kernels executed in concurrent threads by multiple processing elements of a computing device.

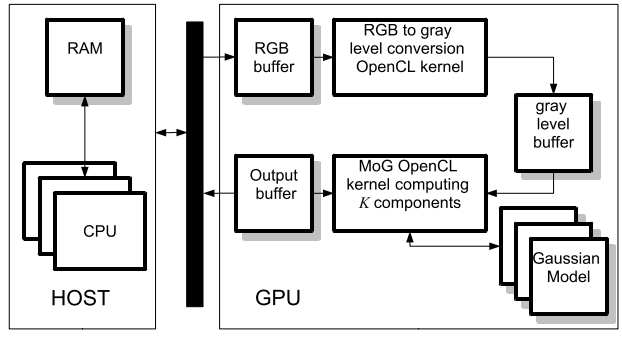
4 IMPLEMENTING VIDEO-SEGMENTATION ON GPU USING OPENCL

The implementation described in this paper is based on data-oriented OpenCL kernels designed for execution on parallel processing elements of GPU devices. The software comprises also single-threaded sequential, C++ based implementation able

to operate on CPU and consumer devices for reference and comparison. In both implementations, host CPU is responsible for data acquisition, video task control and measurement. Two diagrams of data processing flows that were used in the work, are presented in Figure 1. Initial setup (Figure 1 a)) consisted of full data pipeline that acquired video signal from advanced industrial 5 Mpixel color video camera via GigaBit Ethernet interface in Bayer pattern format.



a)



b)

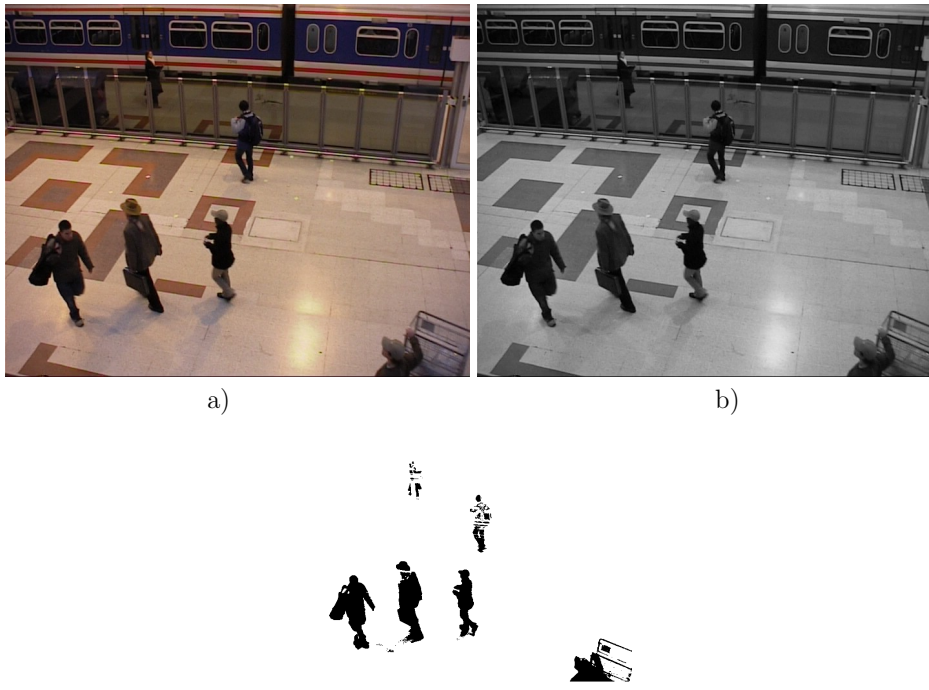
Figure 1. Video segmentation – OpenCL data flow and memory usage; a) Full video data path for PC based system OpenCL-accelerated system, b) OpenCL data path for performance evaluation

JAI BB500-GE GigaBit Ethernet camera device was used to acquire high resolution video in real time for design and initial verification of OpenCL kernel for data processing in GPU pipeline. Preliminary interpolation of raw video data was necessary to obtain RGB video data and to produce gray level video signal for segmenation.

In order to measure performance of OpenCL kernels on various platforms, dedicated simple data flow has been arranged as shown in Figure 1 b). In this scheme, video data for testing is stored on the hard drive of the host computer. This solution enables easy emulation of various video formats: high resolution video cameras and low cost general purpose imaging devices.

$$\eta(x_t, \mu, \sigma) = \frac{1}{\sigma\sqrt{(2\pi)}} e^{-\frac{(x_t-\mu_t)}{2\sigma^2}}. \tag{3}$$

Sample low resolution video frames (720×576) are shown in Figure 2 a)–c). For the purpose of performance evaluation in this work, gray level videos ($n = 1$) have been analyzed, thus probability density function (2) was reduced to formula (3).



a) b)

c)

Figure 2. Sample video data: frame No. 1050 – video sequence **A** [20]; a) Source RGB image b) Gray level image c) Segmentation results

MoG algorithm is time consuming due to intensive memory transactions necessary to update estimated properties of probability density function (2) for each of K components assigned to image pixels. Number of mixture parameters, along with size of input and output image, determine amount of memory needed for video

segmentation. Each of K declared distributions needs 3 parameters to describe the Gaussian model (2): weight $\omega_{i,t}$, mean value $\mu_{i,t}$, and variance $\sigma_{i,t}$. All must be read and written back in order to track changes and update Gaussian mixture models. Estimate of mixture model size for video frame $M \times N$ is about $MNK12$ bytes, assuming 4-byte floating point precision. It makes 576 MB ($K = 10$) or 289 MB ($K = 5$) for high resolution $2\,456 \times 2\,048$ video. Extra buffers are necessary for input, output and gray level conversion. Device-specific OpenCL memory buffer constraint must be considered in order to avoid allocation and transfer errors.

Sample OpenCL kernel implementing image convertion from RGB to gray level is presented in source code listing 1. Kernel definition consists of sampler declaration for image data adressing, constants and kernel body. Listing 2 shows part of kernel performing video segmentation of gray level video stream. Presented source code lines compute probability density function in accordance with formula (3) and update MoG matching component. Kernel body is executed concurrently by multiple GPU processing elements available on OpenCL Device.

```
__constant sampler_t smp =
    CLK_NORMALIZED_COORDS_FALSE | CLK_FILTER_NEAREST |
    CLK_ADDRESS_CLAMP_TO_EDGE;

__constant float4 coeff = { 0.299f, 0.587f, 0.114f, 0.000f };

__kernel void rgb_to_gray(  __read_only image2d_t src,
                            __write_only image2d_t dst)
{
  const int2 gid = {get_global_id(0), get_global_id(1)};
  const int2 size = {get_image_width(src),
    get_image_height(src)};

  float4 rgb = read_imagef(src, smp, gid);
  float gray = dot(coeff, rgb);

  write_imagef(dst, gid, (float4) gray);
}
```

Listing 1. Color RGB to gray level conversion kernel

Several optimization techniques were applied and OpenCL specific constructs were used to obtain results presented in the following Section 5. Amongst all, compilation level optimisation based on loop unrolling pragma should be mentioned as it reduced kernel execution time significantly. Moreover, `private` local memory was used to replicate parameters $\omega_{i,t}$, $\mu_{i,t}$, $\sigma_{i,t}$ (`weight`, `mean`, `var`) of each of K Gaussian processes assigned to particular pixel of video frame. Time of data transfer

```
__constant sampler_t smp =
  CLK_NORMALIZED_COORDS_FALSE |    CLK_FILTER_NEAREST |
  CLK_ADDRESS_CLAMP_TO_EDGE;

__kernel void mog_segment(  __read_only image2d_t frame,
     __write_only image2d_t dst,  __global float* mog_buf,
     __constant MogParams* params, const float alpha)
{
  const int2 g_id = { get_global_id(0), get_global_id(1) };
  const int2 size = { get_image_width(frame),
    get_image_height(frame) };

  float pix = read_imagef(frame, smp, g_id).x * 255.0f;
  int pdf_match = -1;

  //... initialize kernel and read MoG parameters
  //... find matching MoG component

  #pragma unroll nmixtures
  for(int i = 0; i < K; ++i)
  {
    if(i == pdf_match)
    {
        float d = pix - mean[i];
        float r = a/native_sqrt(2_PI*var[i])
          *native_exp(-0.5f*d*d/var[i]);
        weight[i] = weight[i] + a*(1.0f - weight[i]);
        mean[i] = mean[i] + r*diff;
        var[i] = max(params->minVar,
          var[i] + rho*(d*d - var[i]));
      }
      else
      {
        weight[i] = (1.0f - a)*weight[i];
      }
  }

  //... normalize, sort and save MoG parameters
  //... compute result pixel

  write_imagef(dst, g_id, (float4) result);
}
```

Listing 2. MoG kernel – probability density calculus

from global GPU memory to local memory and back was still significant, but much less than subsequent transaction to global memory whilst searching for matching distribution and sorting after parameter update.

Also `image2d` data structure, instead of initial 1D `buffer` object, was used to store and access input and output image pixels. This made kernel code compact and allowed to avoid excessive computations of 2D image coordinates. The important feature of this method of video data access was usage of samplers that are specific to GPU architectures. As samplers are not supported by OpenCL drivers for multicore CPU, designed kernel cannot be executed on CPU based system.

5 RESULTS

The experimental part of the work is comprised of two tests. Parallel OpenCL module was evaluated on GPU devices: MoG and conversion of RGB video to gray level were both implemented in form of two separate OpenCL kernels (see Figure 1). C++ implementation was executed by CPUs of the same computers for reference and comparison. In the CPU part, dedicated functions for gray-level conversion and MoG algorithm from OpenCV v2.4 library [19] have been used in single-threaded version. For each configuration, single core of single CPU was used for sequential processing. Performance tests were executed on two hardware configurations:

hw1: HPC cluster node, Nvidia Tesla M2090, Intel Xeon E5645 running at 2.4 GHz,

hw2: PC workstation, Nvidia GeForce GTX670, Intel i5 3570 running at 3.4 GHz.

Two video sequences (each containing 1000 frames) were used for benchmarking both implementations on the two paltforms:

A. Low resolution (720×576) color video sequence [20],
B. High resolution ($2\,456 \times 2\,048$) color video sequence captured with JAI BB500-GE camera.

All software modules, including data acquisition, visualization and time measurement were written in C++ language with use of OpenCL SDK, custom intermediate level OpenCL wrapper [16] and OpenCV library compiled without parallel processing support. Detailed analysis of output data and functional verification was performed with MATLAB tool. Platform specific executable files were compiled from the same source files for target operating systems of PC workstation and HPC cluster.

Summary of results collected when processing selected video data sets on both platforms **hw1** and **hw2** were presented in Table 1 and in Figure 3. Speedup S of GPU over CPU on HPC platform exceeded value of 8 for regular mixture $K = 5$ and low resolution video sequence **A**. It reached value of 5 for high resolution video sequence **B**. Average GPU computing time and CPU computing time were used for speedup estimation $S = \frac{T_{CPU}}{T_{GPU}}$. T_{CPU} corresponds to average CPU processing time,

T_{GPU} consists of two components: execution of OpenCL kernels (MoG algorithm and color to gray level conversion) and video data transfers from host device to global memory of GPU device and vice versa.

HW Config.	Frame Size	K	T_{GPU}	T_{CPU}	S	Theoretical Throughput of GPU Implementation	
			ms	ms		FPS	Mpixel/s
hw1	720×576	**5**	2.01	17.67	8.8	498	197
		10	3.91	24.84	6.4	256	101
	$2\,456 \times 2\,048$	**5**	27.03	212.19	7.9	37	177
		10	53.19	265.18	5.0	19	90
hw2	720×576	**5**	1.79	9.23	5.2	559	221
		10	3.36	12.03	3.6	298	118
	$2\,456 \times 2\,048$	**5**	27.11	103.48	3.8	37	177
		10	47.32	137.97	2.9	21	101

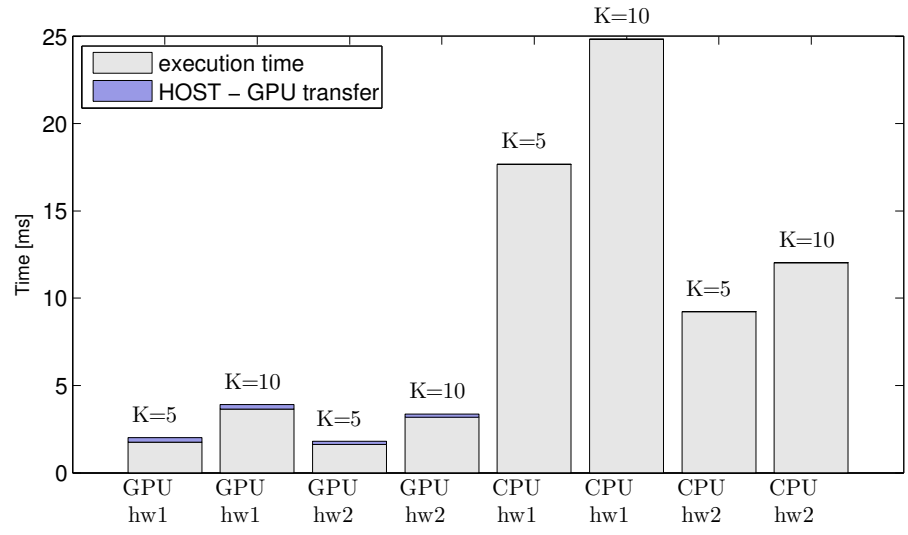
Table 1. Performance evaluation results (per 1 video frame)

Theoretical throughput was derived directly from GPU computing time without considering video acquisition, foreground post-processing, UI procedures nor other CPU overhead that make effective FPS (frame per second) rate. Throughput expressed in Mega-pixels allows to estimate number of pixels processed within a second assuming $1\,\text{Mpixel} = 1\,024 \times 1\,024$ pixels.

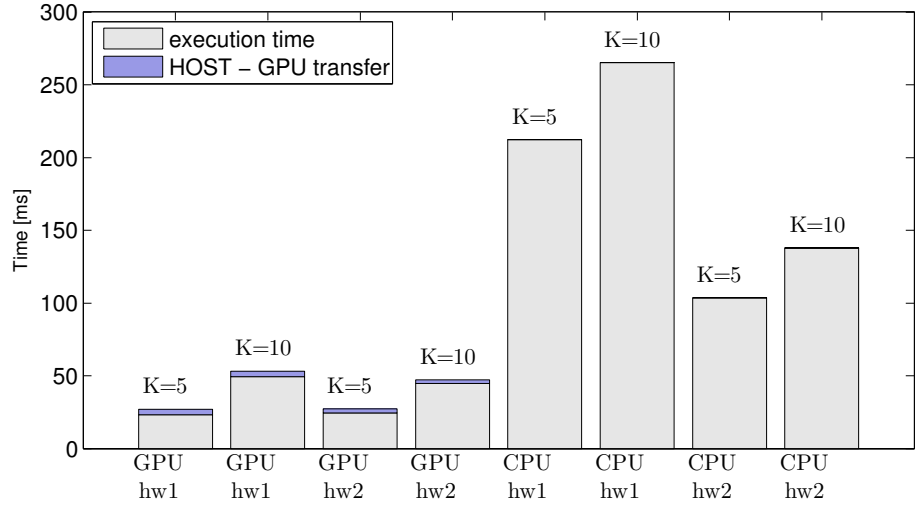
An impact of data transfers on average computing time on GPU devices and CPU processing times is presented in Figure 3. For low resolution video and $K = 5$, writing color video to GPU buffer and reading foreground segmentation results takes in average about 14 % of overall T_{GPU} computing time on **hw1** HPC platform configuration and about 10 % on **hw2** commodity PC. This fractions are reduced to halves for extended mixtures $K = 10$, since kernel execution time is proportional to K, thus amount of data to be transferred is constant.

Usually, in practical applications, number of mixtures K does not exceed value of 4. Some implementations adjust number of mixtures automatically in order to reduce computational complexity and memory usage. In most cases, no significant change in segmentation result can be observed for large number of mixtures. The value $K = 10$ was selected in our experiments to measure impact of calculus complexity on video processing time.

Other related research reported in the recent articles usually provide quantitative results of processing widely used HD video format, i.e. resolution $1\,920 \times 1\,080$. In [6] authors proposed the FPGA implementation of the GMM algorithm which is able to process HD video stream in real-time. The implementations on Virtex6 and Virtex5 without pipeline levels were able to process 41 and 38 FPS, respectively. Using one level of pipeline, both the Virtex5 and Virtex6 implementations are able to process more than 60 FPS. In [18] CUDA-based GPU implementation of background subtraction MoG algorithm is presented that surpasses real-time processing for full HD (1080p 60 Hz). Our OpenCL MoG implementation is also able to process video



a)



b)

Figure 3. Comparison of average processing times; a) Average processing and transfer times for low resolution video sequence A, b) Average processing and transfer times for high resolution video sequence B

stream in real-time. For video resolution higher than HD ($2\,456 \times 2\,048$) and typical MoG parameters (that is $K = 5$), the average FPS is about 37. For lower resolutions, corresponding to SD (720×576), the throughput of our OpenCL implementation is larger than 400 FPS.

When comparing our video sequence **B** to HD format one can notice that number of pixels is about 2.4 times larger. Therefore, when discussing results, throughput expressed in Mpixel/s unit should be used instead of number of frames per second (FPS). The lowest frame rate we obtained in OpenCL implementation was measured for sequence **B** with regular parameter $K = 5$: 177 Mpixel/s (37 FPS). Corresponding value for reference [6] is 81 Mpixel/s (38 FPS) and 119 Mpixel/s (60 FPS) for reference [18].

5.1 Video-Frame Processing Time Analysis

Video processing algorithms, in order to reach technology readiness level, must at least meet real-time constraints that are usually defined by data rate of input video or camera, expressed in frames per second (FPS). The system that conforms this requirements should not drop any video frames. For video segementation studied in this work it is crucial because it impacts segmentation result quality. Loosing particular video frame does not only cause the lack of segmentation result but it also might introduce severe noise into statistical models of pixels that would produce artifacts in forthcoming frames.

At protocol and systems levels, particularly in complex parallel systems, fulfillment of this requirement can be guaranteed by an appropriate value of QoS (Quality of Service) metrics [14]. In this work however, we only focus on low level performance evaluation. Hardware applications (ASIC or FPGA) (e.g. [6, 9]) that use fine and medium grain pipelining, usually exhibit fixed latency and throughput, so frame processing time in most cases does not depend on frame data content or other factors.

In software solutions, based on general purpose CPU (also accelerated by GPU), under control of multitasking operating system, frame processing time can be impacted by several factors, e.g. system workload, hardware configuration, operating system configuration, etc. In such system it should be treated as random. Outliers exceeding inter-frame delay can introduce severe disturbance, therefore, average processing time is not good estimate for performance evaluation for real-time application. This applies to real-time video surveillance systems and also to advanced algorithms used for modeling 3D environment. For example, in realistic modeling of sound waves propagation [12], excessive computing time can result in noticable artefacts once data frames are not provided within proper period of time for rendering. In video-surveillance system, loss of input data may result in false detection.

Both CPU-based sequential reference model of background estimation and parallel GPU-accelerated implementation are prone to variations of frame processing time. Therefore, when measuring efficiency of presented video processing systems, beside computing average FPS already presented in Table 1, we took into considera-

tion also variations of processing time that can disturb mixture model through loss of input data. Figures 4 a)–b) and 4 e)–f) show diagrams of processing time, each consisting of 1 000 samples, one per single frame of examined video sequences. The same set of data was used to render box-plot charts (Figure 4) and contributed to average processing time T_{mean} values and also to standard deviation σ_T presented in Table 2. Additionally, minimum processing time T_{min}, maximum processing time T_{max} and variability factor defined as follows, has been appended: $V = 100\,\% \frac{T_{max} - T_{min}}{T_{mean}}$.

On each of box-plots, the central marks correspond to median values, the edges of boxes are the 25$^{\text{th}}$ and 75$^{\text{th}}$ percentiles, the whiskers extend to the most extreme data points that are not considered outliers. Outliers are plotted individually. The most distant outliers T_{min} and T_{max} contribute to V which was 16 % and 134 % for GPU and CPU, accordingly for low resolution video and $K = 5$. The same metric applied to GPU transfer time rendered variability V in range 15 %–68 %. It is negligible, however, due to low contribution of data transfer time to overall computing time (Figure 3). It should be noticed that values of V do not follow changes of σ_T which is the metrics usually applied for analysis of statistical data.

It is worth mentioning that no specific configuration of cluster node has been applied. This means that processing times of particular frames may be treated as random, due to CPU load introduced by other tasks executed. Activity of host CPU during execution of OpenCL Kernels is limited mainly to dispatching transfers to Direct Memory Access subsystem and collecting measurement data. Therefore in this case, the influence of other tasks can be considered lower.

HW Config.	Frame Size	K	T_{mean} ms	T_{min} ms	T_{max} ms	σ_T	V %	Device
hw1	720×576	**5**	17.67	17.18	19.94	0.20	16	sequential: 1 CPU core
		10	24.84	21.54	38.24	5.50	67	
	$2\,456 \times 2\,048$	**5**	212.19	209.70	233.73	1.51	11	
		10	265.18	262.69	300.96	3.63	14	
hw2	720×576	**5**	9.23	8.56	20.93	1.59	134	
		10	12.03	11.28	26.31	1.66	125	
	$2\,456 \times 2\,048$	**5**	103.48	101.94	141.93	1.41	39	
		10	137.97	136.13	178.05	1.74	30	
hw1	720×576	**5**	1.75	1.73	2.01	0.01	16	parallel: OpenCL GPU device
		10	3.65	3.60	3.91	0.02	80	
	$2\,456 \times 2\,048$	**5**	23.21	23.06	25.84	0.15	12	
		10	49.37	49.16	51.91	0.16	60	
hw2	720×576	**5**	1.62	1.60	1.65	0.01	3	
		10	3.19	3.14	3.23	0.01	3	
	$2\,456 \times 2\,048$	**5**	24.41	24.15	26.97	0.10	12	
		10	44.62	44.32	45.55	0.09	30	

Table 2. MoG execution time and variability

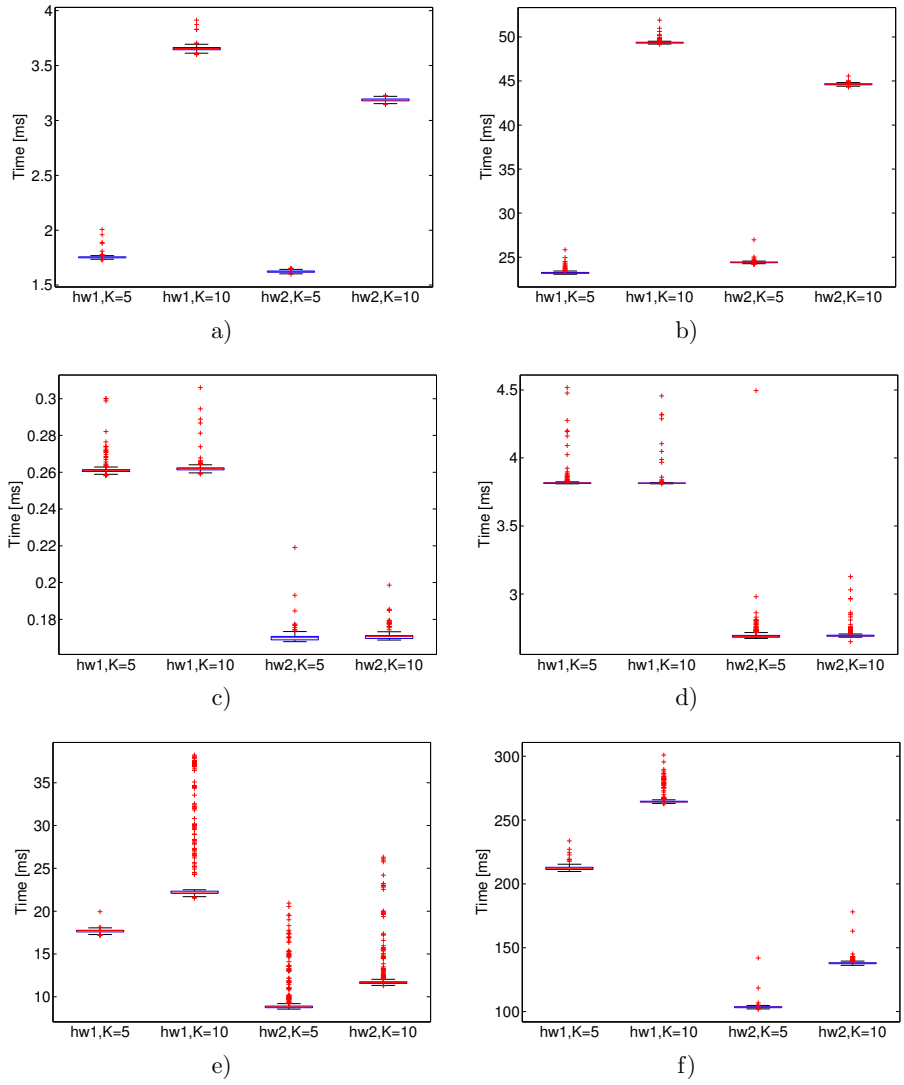


Figure 4. Boxplots: MoG execution and transfer time; a) OpenCL, low resolution video sequence **A**, b) OpenCL, high resolution video sequence **B**, c) GPU transfer, low resolution video sequence **A**, d) GPU transfer, high resolution video sequence **B**, e) CPU, low resolution video sequence **A**, f) CPU, high resolution video sequence **B**

5.2 Functional Verification

GPU implementation is slightly different functionally from sequential OpenCV CPU implementation, therefore comparison of segmentation results has been performed. Figure 5 a)–b) shows normalized sum of absolute differences between GPU and CPU for each output video frame. Maximum value of normalized $SAD = 1\,\%$ was observed for video sequence **A**. Segmentation output and absolute difference image of GPU and CPU result is presented in Figure 6. Mismatching output pixels (Figure 6 c)) appear mainly on boundaries of moving objects and in area of casted shadows. Another difference between the two implementations, that one should be aware of, is video frame latency introduced by GPU pipeline architecture. The first output video frame should be dropped as it does not contain valid data. For CPU implementations no frame latency is observed because execution should be treated asynchronous when compared to OpenCL application programming interface.

6 CONCLUSIONS AND FUTURE WORK

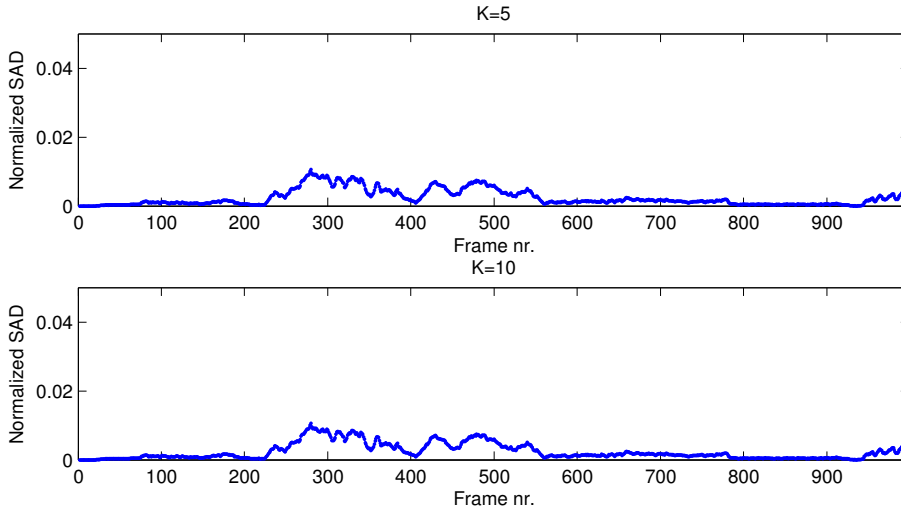
Presented OpenCL implementation enables high-resolution and high-frame rate segmentation on GPU devices. Speedup versus sequential implementation was in range: 2.9–8.8. However, the more important advantage is that GPU acceleration reduced computing time variability for both platforms – workstation and HPC cluster node. This parameter impacts system robustness, particularly when multiple video streems must be processed in real-time.

Obtained throughput, expressed in Mpixel/s unit was larger than in reffered FPGA [6] and GPU [18] implementations. Another advantage of presented implementation is that, in general, OpenCL kernels can be executed on GPU devices produced by various vendors and also on multicore-CPUs and parallel computers. CPUs however do not support all OpenCL extensions utilized in presented kernels.

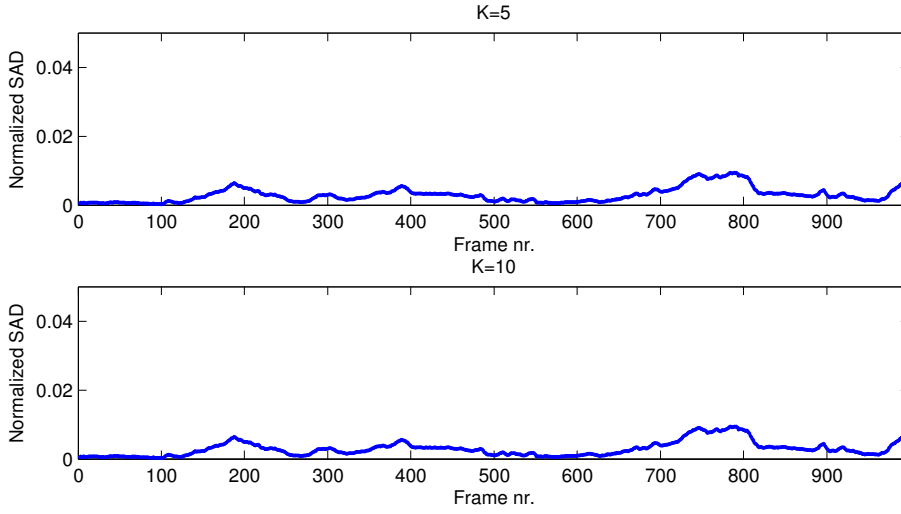
In order to reduce impact of data transfer overhead between CPU and GPU, it is planned to extend video processing flow executed on GPU by adding more pre-processing and post-processing operators. These would be used for input data conditioning (e.g. [10]) and further stages of event detection and recognition. In this case qualitative verification is also inevitable, so it is planned to compare results of presented solution with reference implementations for various data sets and wider range of MoG parameters (number of Gaussian models K and other parameters, etc.).

Evaluation of 1-channel MoG segmentation algorithm was presented in this article. It is planned in the further research to implement and evaluate 3-channel MoG operating on RGB or other color spaces. The increase of throughput is also expected after optimizations and executing kernels on multiple GPU boards.

Further gains can be achieved after thorough kernel profiling and optimizations that would allow to reduce data transfers between GPU and its global memory. Small kernels corresponding to simple image processing operators (e.g. conversion



a)



b)

Figure 5. Comparison of segmentation results – sequential CPU versus parallel OpenCL: normalized sum of absolute differences in binary output image frames; a) Low resolution video sequence **A**, platform **hw1**, b) High resolution video sequence **B**, platform **hw1**

Figure 6. Functional comparison: GPU OpenCL versus CPU OpenCV implementation, frame No. 280 – video sequence **A** [20]; a) Source RGB image, b) CPU segmentation result, c) CPU – GPU: differences

to gray level) can be merged into more sophisticated ones (i.e. MoG operator), thus they would share memory access units of processing elements. This expectation is justified by the observation collected in this research during kernel development and testing. It has been noticed that oposite to transfer versus computing time statistics presented in Figure 3, transfer time between OpenCL processing element and global GPU memory contributes to overall GPU excution time significantly. Thus low complexity computation may be performed "on the fly" without scheduling dedicated kernel, extra memory transcation nor dedicated memory buffer object. This, however, requires careful dealing, as possibly kernel size extension may also impact overall execution time. There is also a field for optimization in evalution of costly expression (2) that contains inverted square root. It can be replaced by equivalent multiplication of component obtained with built-in OpenCL functions.

OpenCL 1.1 library has been used in the software modules, however OpenCL 1.2 and 2.0 specifications have been already published [21] and are being implemented by GPU vendors. These introduce new functionalities, e.g. memory access adequate for image data handling and utilisation of GPU specific hardware.

Acknowledgements

The work presented in this paper was supported by the Ministry of Science and Higher Education of the Republic of Poland, grant No. 11.11.120.612. This research was also supported in part by PL-Grid Infrastructure.

REFERENCES

[1] El Baf, F.—Bouwmans, T.—Vachon, B.: Comparison of Background Subtraction Methods for a Multimedia Learning Space. In: Sérgio, M., de Faria, M., Amado Assuncao, P. A. (Eds.): SIGMAP, INSTICC Press, 2007, pp. 153–158.

[2] Bouwmans, T.—El Baf, F.—Vachon, B.: Background Modeling Using Mixture of Gaussians for Foreground Detection – A Survey. Recent Patents on Computer Science, Vol. 1, 2008, pp. 219–237.

[3] Chmiel, W.—Kwiecień, J.—Mikrut, Z.: Realization of Scenarios for Video Surveillance. Image Processing and Communications, Vol. 17, 2013, pp. 231–240.

[4] Czajkowski, T. S.—Aydonat, U.—Denisenko, D.—Freeman, J.—Kinsner, M.—Neto, D.—Wong, J.—Yiannacouras, P.—Singh, D. P.: From OpenCL to High-Performance Hardware on FPGAs. 2012 22nd International Conference on Field Programmable Logic and Applications (FPL), 2012, pp. 531–534.

[5] Devasena, C. L.—Revathí, R.—Hemalatha, M.: Video Surveillance Systems – A Survey. IJCSI International Journal of Computer Science Issues, Vol. 8, 2011, No. 4, pp. 635–642.

[6] Genovese, M.—Napoli, E.: ASIC and FPGA Implementation of the Gaussian Mixture Model Algorithm for Real-Time Segmentation of High Definition Video. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 22, 2014, No. 3, pp. 537–547.

[7] Idzenga, T.—Gaburov, E.—Vermin, W.—Menssen, J.—De Korte, C.: Fast 2-D Ultrasound Strain Imaging: The Benefits of Using a GPU. IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control, Vol. 61, 2014, No. 1, pp. 207–213.

[8] Ilgner, R. G.—Davidson, D. B.: A Comparison of the FDTD Algorithm Implemented on an Integrated GPU Versus a GPU Configured as a Co-Processor. 2012 International Conference on Electromagnetics in Advanced Applications (ICEAA), 2012, pp. 1046–1049.

[9] Kryjak, T.—Gorgoń, M.: Real-Time Implementation of Moving Object Detection in Video Surveillance System Using FPGA. Computer Science, Vol. 15, 2011, No. 3, pp. 149–163.

[10] Kryjak, T.—Gorgoń, M.: Pipeline Implementation of Peer Group Filtering in FPGA. Computing and Informatics, Vol. 31, 2012, No. 4, pp. 727–741.

[11] Monteiro, E.— Vizzotto, B.—Diniz, C.—Maule, M.—Zatt, B.—Bampi, S.: Parallelization of Full Search Motion Estimation Algorithm for Parallel and Distributed Platforms. International Journal of Parallel Programming, Vol. 42, 2014, No. 2, pp. 239–264.

[12] Pałka, S.—Głut, B.—Ziółko, B.: Visibility Determination in Beam Tracing with Application to Real-Time Sound Simulation. Computer Science, Vol. 15, 2014, No. 2, pp. 197–215.

[13] Shagrithaya, K.—Kepa, K.—Athanas, P.: Enabling Development of OpenCL Applications on FPGA Platforms. 2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), June 2013, pp. 26–30.

[14] Słota, R.—Nikolow, D.—Skałkowski, K.—Kitowski, J.: Management of Data Access with Quality of Service in PL-Grid Environment. Computing and Informatics, Vol. 31, 2012, No. 2, pp. 463–479.

[15] Stauffer, C.—Grimson, W.: Adaptive Background Mixture Models for Real-Time Tracking. IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 1999, pp. 246–252.

[16] Swierk, K.: Parallel Implementation of Complex Algorithms Using Graphic Processing Units. M.Sc. Thesis, AGH University of Science and Technology, 2013 (in Polish).

[17] Zhang, Q.—Eagleson, R.—Peters, T. M.: GPU-Based Visualization and Synchronization of 4-D Cardiac MR and Ultrasound Images. IEEE Transactions on Information Technology in Biomedicine, Vol. 16, 2012, No. 5, pp. 878–890.

[18] Zhang, C.—Tabkhi, H.—Schirner, G.: A GPU-Based Algorithm-Specific Optimization for High-Performance Background Subtraction. International Conference on Parallel Processing, Minneapolis, 2014.

[19] Zivkovic, Z.: Improved Adaptive Gaussian Mixture Model for Background Subtraction. Proceedings of the 17th International Conference on Pattern Recognition (ICPR 2004), 2004, Vol. 2, pp. 28–31.

[20] PETS 2006 Benchmark Data, DATASET S1 (TAKE 1-C), scenario: left luggage, 2006. `http://www.cvg.rdg.ac.uk/PETS2006/data.html` (access time 2014.01.10).

[21] Khronos OpenCL Registry, 2012. `http://www.khronos.org/registry/cl/` (access time 2014.09.27).

Mirosław Jabłoński works in the Department of Automatics and Bioengineering at AGH University of Science and Technology in Krakow, Poland. He received his M.Sc. and Eng. degrees in electronics and telecommunication (2001) and Ph.D. (degree with honors) in automatics and robotics (2009), both from Faculty of Electrical Engineering, Automatics, Computer Science and Electronics of the AGH-UST. He worked as research and teaching assistant at the Laboratory of Biocybernetics in the Department of Automatic Control at AGH-UST (2001–2009) and since 2010 as Assistant Professor. He reviews for international conferences. Efficient parallel computing techniques for image processing, analysis and video-detection have been within the scope of his research interest.

Jaromir Przybyło graduated with an outstanding proficiency, with M.Sc., Eng. degree in 2000 from the Faculty of Electrical Engineering, Automatics, Computer Science and Electronics, AGH-UST. He received his Ph.D. degree (with honours) in 2008 in computer science: biocybernetics and biomedical engineering. Since 1999 he has worked at the AGH-UST Institute of Automatics as a research and teaching assistant, since 2009 as Assistant Professor.