Proceedings of the 2014 Federated Conference on
Computer Science and Information Systems pp. 709–717

DOI: 10.15439/2014F282
ACSIS, Vol. 2

Movement Tracking in Terrain Conditions Accelerated with CUDA

Piotr Skłodowski
Cybernetics Faculty at Military
University of Technology
ul. S. Kaliskiego 2,
00-908 Warsaw, Poland
Email: psklodowski@wat.edu.pl

Witold Żorski
Cybernetics Faculty at Military
University of Technology
ul. S. Kaliskiego 2,
00-908 Warsaw, Poland
Email: wzorski@wat.edu.pl

***Abstract*— The paper presents a solution to the problem of movement tracking in images acquired from video cameras monitoring outside terrain. The solution is resistant to such adverse factors as: leaves fluttering, grass waving, smoke or fog, movement of clouds etc. The presented solution is based on well known image processing methods, nevertheless the key was the use of an appropriate conduct procedure. In order to obtain a real-time system the CUDA technology was involved.**

I. Introduction

THE problem of movement detection in images [4] appeared relatively early [6]. The astronomy world struggled with objects detection in images [8] of the night sky acquired by telescopes long before the era of modern computers. In first systems images were alternately displayed in front of an operator who was able to perform detection of a motion celestial body. In such systems a natural subconscious human ability of movement detection was involved [9], [10].

Excluding cheap and simple movement detectors or sensors (passive infrared, ultrasonic, or microwave) the task of motion detection with the use of video cameras [11] is based on digital image processing [5]. Many present-day computer systems begin the work from the stage of a differential image of two images [14], which next undergoes a series of processes [16]. The detection of a movement [3] is not the only result – in modern systems a trajectory of a motion body can be determined [1] or even identification of the detected object may be performed [2], [22].

In this paper we consider the problem of object movement tracking [7] in images acquired from video cameras [12] monitoring outside terrain [13]. The assumption was to elaborate a solution resistant to such adverse factors as: leaves fluttering, grass waving, smoke or fog, movement of clouds etc. A set of well known image processing methods [19] is adopted, and the key was the use of an appropriate conduct procedure. In order to obtain a real-time system the CUDA technology was involved.

The CUDA (*Compute Unified Device Architecture*) technology appeared quite unexpectedly in 2007 as a result of new Nvidia's GPUs branded GeForce 8. CUDA gave the software developers direct access to the virtual instruction set and memory of the parallel computational elements in GPUs.

CUDA is a parallel computing platform and programming model [24] that makes using a GPU for general purpose computing simple and elegant. At present, there are two main CUDA architectures available: Fermi (see Fig. 1) and Kepler. The Maxwell architecture (20 nm technology node) is just about to be launched onto the market. From the programmer's point of view [25] the new architecture brings a set of features, both hardware and software, that is known as the compute capability of a device.

The idea of combining image processing methods or computer vision techniques with CUDA technology started relatively early [15] and going on, being very popular.

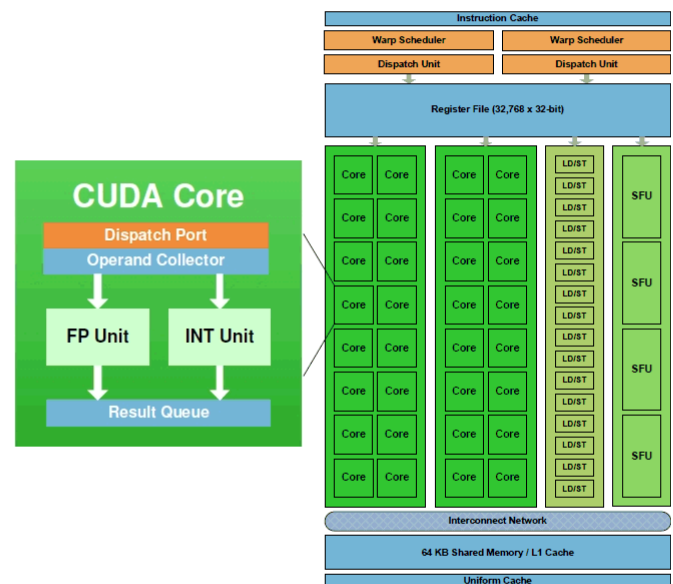


Fig. 1. CUDA core and Fermi SM (Streaming Multiprocessor) structure

978-83-60810-58-3/$25.00 © 2014, IEEE

This paper presents an announced earlier solution to movement detection and tracking, first elaborated using the Matlab environment, and finally independently implemented as the x86 and CUDA application.

The method was originally prepared for monitoring an airport's terrain, but for obvious reasons only neutral shots will be presented.

II. A Brief Presentation of the System

The used computer vision system consists of a PC equipped with a CUDA device (GTX 650 Ti, based on the Nvidia's Kepler architecture with compute capability 3.0), and an IP camera. It is supported by the Microsoft Visual Studio 2012 and CUDA 5.5 framework (the most important of it is the CUDA Toolkit component). Fig. 2 shows a visual scheme of the used computer vision system.

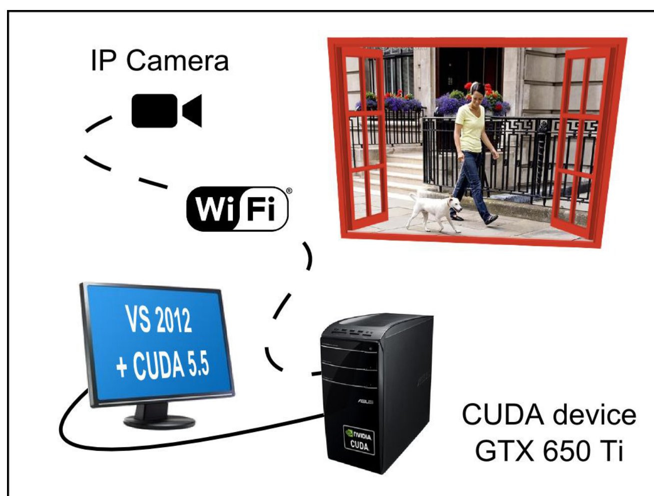


Fig. 2. Scheme of the used system

III. Matlab Implementation

The Matlab environment gives a possibility to elaborate the required procedure relatively fast. The amount of engineering tools included in the Matlab is impressive, nevertheless to obtain required speed the final implementation must be done with the CUDA technology.

A. Example source scenes

Source scenes have been acquired under various terrain and weather conditions. In this section two examples are presented (see Fig. 3). The first scene includes an object that is well visible, but also includes waving grass and clouds. The second scene is much more difficult, the object is comparatively small and there is a big tree with fluttering leaves. In both scenes a slight tilt effect is present between shots captured over a distance of a few seconds.

Fig. 3. The input source scenes

B. Compensation of the tilt effect

The initial obstacle is the tilt effect between shots, which may occur as a result of small vibration under the influence of wind or some mechanical reasons. To compensate the tilt one image is narrowed about a "frame" and matched with the second image in order to find a location with the smallest difference. Finally images of the scene are "framed" to guarantee the smallest difference between them. The source code in Fig. 4 gives details of the procedure. Fig. 5 shows (only) the cropping effect in the case of the second considered scene. The result will be visible in the case of difference images (the next section).

```
26    %IMAGE STABILIZATION by shift compensation
27 -  nop=5; %number of pixels (of the frame)
28 -  I1=I_in1(:,:,1);
29 -  I2=I_in2(:,:,1);
30 -  I2_framed=I2(nop+1:row-nop,nop+1:col-nop); %a "framed" image
31 -  d=255*row*col; bi=0; bj=0;
32    %searching for the smallest difference within range +-nop
33 -  for i=-nop:nop
34 -      for j=-nop:nop
35 -          I1_framed=I1(nop+1+i:row-nop+i,nop+1+j:col-nop+j);
36 -          difference=sum(sum(abs(I1_framed-I2_framed)));
37 -          if difference<d
38 -              bi=i; bj=j; d=difference;
39 -          end;
40 -      end;
41 -  end;
42 -  I1_framed=I1(nop+1+bi:row-nop+bi,nop+1+bj:col-nop+bj);
```

Fig. 4. Compensation of the tilt effect – the source code

Fig. 5. The cropping effect of the tilt compensation – frames are visible

C. Getting a difference image

The difference image generation [16] is the first processing stage for a scene. This approach is extremely popular in astronomy [17] and is commonly referred to as difference image analysis (DIA). Results (presented in negative) obtained for the considered scenes (after the tilt compensation) are visible in Fig. 6 and Fig. 7.

Fig. 6. The difference image for the first scene in Fig. 3

Fig. 7. The difference image for the second scene in Fig. 3

D. Removing unwanted artifacts

The received differenced images include moving objects as well as include some unwanted artifacts. In the case of the first scene (Fig. 6) a remnant of the tilt effect is still visible (e.g. contour of a building), and in the second scene (Fig. 7) a tree is well exposed. Some of the artifacts are heavy, what is shown in Fig. 8, which is a 3D visualization of the content of Fig. 7.

At the first glance the task of removing unwanted artifacts seems to be difficult. To solve the problem it is necessary to notice that tracked objects generate comparatively low frequencies and the unwanted artifacts rather high frequencies (see Fig. 8). As an outcome of many trials it turned out that erosion, a fundamental operation of morphological image processing [18], gives the best results.

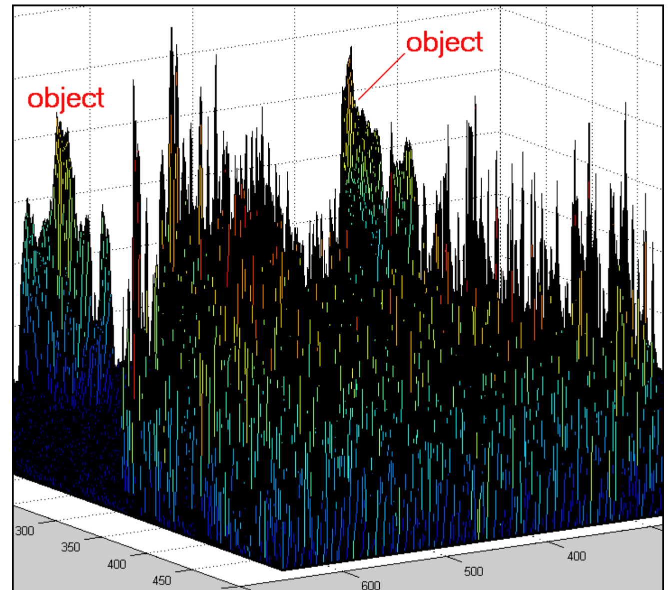


Fig. 8. A 3D visualization of the difference image from Fig. 7

The erosion operation is already available in Matlab as ready to use function (erode, imerode). Nevertheless, it was implemented "step by step" for gray-level images with the prospective x86 and CUDA implementations in mind. The source code in Fig. 9 shows details of the erosion implementation with a disk as the structuring element.

```
64      %erosion filter - "step by step"
65 -    SE=[  0     0     1     0     0;
66            0     1     1     1     0;
67            1     1     1     1     1;
68            0     1     1     1     0;
69            0     0     1     0     0]
70 -    [row col]=size(I_out1(:,:,1));
71 -    V=256*ones(5,5);
72 -    for i=1+2:row-3
73 -        for j=1+2:col-3
74 -            for k=-2:2
75 -                for l=-2:2
76 -                    if SE(k+3,l+3)==1
77 -                        V(k+3,l+3)=I_out1(i+k,j+l,1);
78 -                    end;
79 -                end;
80 -            end;
81 -            I_out2(i,j,1)=min(min(V));
82 -        end;
83 -    end;
84 -    I_out2(:,:,2)=I_out2(:,:,1);
85 -    I_out2(:,:,3)=I_out2(:,:,1);
```

Fig. 9. Matlab implementation of the erosion for gray-level images

The results obtained with the erosion are shown in Fig. 10 and Fig. 11. The objects are still well visible and the artifacts are predominantly filtered.

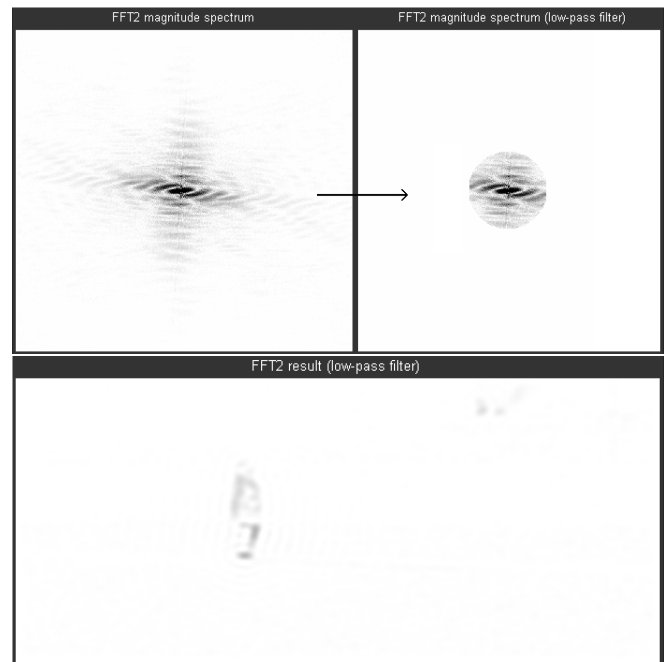
Fig. 10. The result after erosion for the first scene (compare Fig. 6)

Fig. 11. The result after erosion for the second scene (compare Fig. 7)

E. Exposing objects

The use of erosion filtering was beneficial to objects detection. It turned out that objects can be further exposed with the use of low-pass filtering. There are two possibilities: simple spatial filtering (neighborhood averaging) with a large mask 7x7 or just the standard transform FFT2. The second tool is faster and already available in majority of programming environments (including CUDA). In the case of Matlab we have two-dimensional convolution conv2 and set of tools for two-dimensional discrete Fourier transform: fft2, ifft2, fftshift.

The process of FFT2 filtering is shown in Fig. 12, and a 3D result for the first scene is visible in Fig. 13, and for the second scene is presented in Fig. 14.



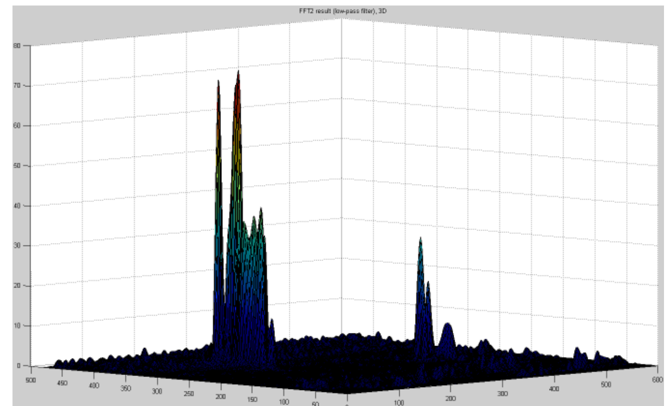


Fig. 12. The use of the FFT2 for the first scene (compare Fig. 10)

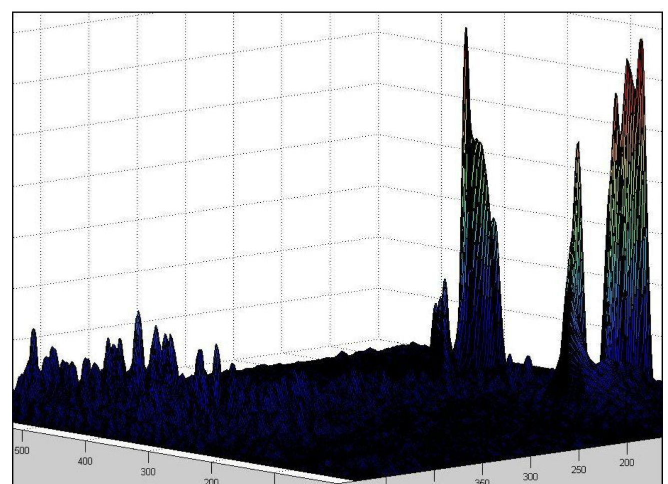


Fig. 13. The result of FFT2 for the first scene (see Fig. 12)

Fig. 14. The result of the FFT2 for the second scene (see Fig. 11)

F. Binarization and the decision

The results visible in Fig. 13 and 14 are rather satisfying. The last stage before the final decision about movement detection is binarization. The best results of binarization were received for threshold from the range of **20-50** of gray levels. The results of binarization are presented in Fig. 15.

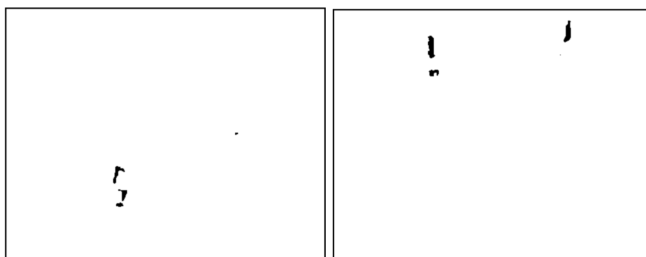


Fig. 15. The result of binarization for the considered scenes

The final decision about the movement detection is based on percentage size of objects in frames. In the case of Fig. 15 the percentage sizes of objects are respectively: 0,15% and 0,32%. The established threshold for the elaborated method is **0,1%**.

IV. X86 IMPLEMENTATION

The x86 implementation of the elaborated method has been made in C++ with the use of Visual Studio 2012. To speed up the implementation process the well known and free library called OpenCV was used. The library contains a set of ready to use computer vision algorithms (e.g.: linear filtering, cosine transform) as well as basic image processing functions (read/write images, conversion). Custom implementation has been made only for elements that are not included in the OpenCV or those which are poor optimized for the considered application.

C++ language has been chosen for both: x86 and CUDA implementation. Thanks to that it was possible to use exactly the same template project and therefore the execution is not overwhelmed by any language runtime.

A. Example source scenes

Source scenes were captured from an IP camera, flipped horizontally and then converted to 8 bits gray-scale images.

B. Compensation of the tilt effect

The tilt reduction function has been implemented independently in accordance with proposed algorithm and shown in following listing (Fig. 16). This function requires two images which we call “previous” and “current” frame. The previous frame is the frame captured first.

The only parameter required is *nop*. The *nop* stands for number of pixels. In our implementation we used constant value 5 which means that the previous frame is cropped by 5 pixels from all sides and the current frame is centered this way that the tilt effect (the images deference) to the previous frame is the smallest.

```
37 void Movement::tilt_reduction()
38 {
39     int iw = this->iw - 2*nop,
40         ih = this->ih - 2*nop,
41         is = iw * ih;
42
43     // crop the previous frame
44     Mat roi_im1 = im1(Rect(nop, nop, iw, ih));
45
46     int by = 0, bx = 0;
47     int min = is*256;
48     Npp8u *px;
49
50     // find the best position in the current frame
51     // where tilt is the lowest
52     for (int y = 0; y < nop*2+1; y++)
53         for (int x = 0; x < nop*2+1; x++)
54         {
55             Mat a = abs(roi_im1 - im2(Rect(x, y, iw, ih)));
56             Scalar ssum = sum(a);
57             int sum = ssum[0];
58             if (sum < min)
59             {
60                 by = y;
61                 bx = x;
62                 min = sum;
63             }
64         }
65
66     Mat tmp_im = Mat::ones(this->ih, this->iw, CV_8U);
67     Mat tmp_roi = tmp_im(Rect(nop, nop, iw, ih));
68
69     // set "the previous frame"
70     roi_im1.copyTo(tmp_roi);
71     tmp_im.copyTo(im1);
72
73     // set "the current frame"
74     Mat roi_im2 = im2(Rect(bx, by, iw, ih));
75     roi_im2.copyTo(tmp_roi);
76     tmp_im.copyTo(im2);
77 };
```

Fig. 16. Compensation of the tilt effect in C++

C. Getting a difference image

The pixels from previous frame are subtracted from current frame then provided as an argument of *abs* function. The difference image is getting very easy using a ready function *abs* from OpenCV library and is coded as one line in movement detection function (see Fig. 17).

```
22 int Movement::detection(Mat &frame, Mat &result)
23 {
24     frame.copyTo(im2);
25
26     tilt_reduction();
27     diff = abs(im1 - im2);
28     erosion();
29     low_pass_filter();
30     binarization();
31
32     im2.copyTo(im1);
33     diff.copyTo(result);
34     return 0;
35 }
```

Fig. 17. Movement detection in C++

D. Removing unwanted artifacts

To remove unwanted artifacts that might still persist in the processed image the erosion operator is applied. This has been made using our own implementation because we found it much faster than the option provided by OpenCV. The structuring element used in our implementation is disk inscribed in 5x5 matrix (see Fig. 18).

```
void Movement::erosion()
{
    unsigned disk[5][5] = {
        { 0, 0, 1, 0, 0 },
        { 0, 1, 1, 1, 0 },
        { 1, 1, 1, 1, 1 },
        { 0, 1, 1, 1, 0 },
        { 0, 0, 1, 0, 0 }
    };

    Mat out = Mat::zeros(ih, iw, CV_8U);
    Npp8u *data = (Npp8u *) diff.data,
        *out_d = (Npp8u *) out.data;

    for (int iy = 0; iy < ih-4; iy++)
        for (int ix = 0; ix < iw-5; ix++)
        {
            int min = 255, val;
            for (int dy = 0; dy < 5; dy++)
                for (int dx = 0; dx < 5; dx++)
                {
                    if (disk[dy][dx] == 1)
                    {
                        val = *(data + ((iy + dy) * iw + ix + dx));
                        if (val < min)
                        {
                            min = val;
                        }
                    }
                }
            *(out_d + (iy + 2) * iw + ix + 2) = min;
        }

    out.copyTo(diff);
};
```

Fig. 18. Erosion function in C++

E. Exposing objects

The last operation applied to the image before movement detection is convolution with 7x7 kernel of all ones.

```
void Movement::low_pass_filter()
{
    Mat kernel = Mat::ones(lowPassFilter, lowPassFilter, CV_64F), tmp1, tmp2;
    double mean = double(lowPassFilter * lowPassFilter);
    diff.convertTo(tmp1, CV_64F, 1./255);
    filter2D(tmp1, tmp2, -1, kernel);
    tmp2 = tmp2 / mean;
    tmp2.convertTo(tmp2, CV_8U, 255);
    tmp2.copyTo(diff);
}
```

Fig. 19. Low-pass filtering in C++

F. Binarization and the decision

```
void Movement::binarization()
{
    Mat to_compare = Mat::ones(ih, iw, CV_8U) * threshold;
    Mat out;
    compare(diff, to_compare, out, CMP_GT);
    out.convertTo(out, CV_8U);
    out.copyTo(diff);
}
```

Fig. 20. Binarization in C++

The result of all previous steps is still an grayscale image. Applying the threshold we got the final binarized image ready for the final step. This has been also achieved using one line ready to use function (see Fig. 20).

V. CUDA IMPLEMENTATION

Most of operations in proposed algorithm are available in Nvidia Performance Primitives [26]. The NPP is a collection of GPU-accelerated functions for image, video and signal processing. The library is freely available as a part of the CUDA Toolkit.

A. Use of the CUDA device structure

The only function that needed to be implemented independently was the tilt reduction. We couldn't match any function from NPP that would help us to achieve desired results therefore an own kernel has been implemented.

Although CUDA device allows to organize threads in 3D structure, 2D structure was enough. The X and Y axes responds to the position of pixels in the image. Block Index address pixels from “the previous” frame. Pixels from “the current” frame are further offset by the Grid Index. That makes two regions of interest (ROI) for each kernel iteration as shown in Fig. 21.

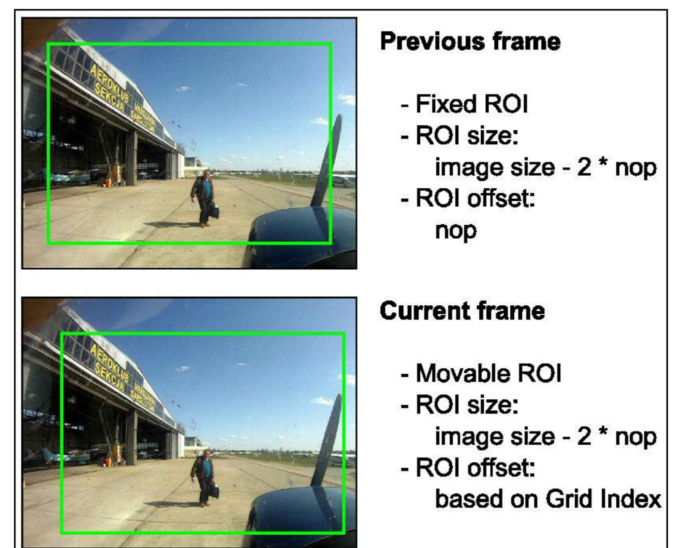


Fig. 21. Tilt reduction with regions of interest (ROI) in CUDA

Grid Size corresponds to *nop* parameter from the x86 implementation. Block Size is a parameter chosen empirically and have to be power of 2 for further reduction process. The source code of the CUDA kernel is presented in Fig. 22.

```
13   __global__ void tilt_reduction_kernel(Npp8u *im1, Npp8u *im2,
14                                         int iw, int ih, int nop, int *ret)
15   {
16       int iww = iw - 2*nop,
17           ihh = ih - 2*nop;
18
19       int thread = threadIdx.y * blockDim.x + threadIdx.x;
20
21       __shared__ int sums[256];
22       sums[thread] = 0;
23
24       int offset1, offset2;
25       int sum = 0;
26
27       int iy = threadIdx.y,
28           ix = threadIdx.x;
29
30       while (ix < iww && iy < ihh)
31       {
32           // Offset in "the previous" frame
33           offset1 = (iy + nop) * iw + ix + nop;
34           // Offset in "the current" frame
35           offset2 = (iy + blockIdx.y) * iw + ix + blockIdx.x;
36
37           // Check the pixels difference
38           int value = abs(*(im1 + offset1) - *(im2 + offset2));
39           // Sum all differences for given ROI
40           sum += value;
41
42           ix += blockDim.x;
43           if (ix >= iww)
44           {
45               ix = threadIdx.x;
46               iy += blockDim.y;
47           }
48       }
49       sums[thread] = sum;
50       __syncthreads();
51
52       // Reduction
53       for (int i = blockDim.x * blockDim.y/2; i > 0; i /= 2)
54       {
55           if (thread < i)
56           {
57               sums[thread] += sums[thread + i];
58           }
59           __syncthreads();
60       }
61
62       // Save the result for given ROI
63       if (thread == 0)
64       {
65           int return_offset = blockIdx.y * gridDim.x + blockIdx.x;
66           *(ret + return_offset) = sums[0];
67       }
68   }
```

Fig. 22. The source code for the CUDA kernel

B. CUDA implementation supported by the NPP library

The use of NPP library is relatively simply. The major difficulty is a preparation of the image data accordingly to NPP requirements. The NPP supports variety of data. Pixels may be provided as 8, 16 or 32 bits signed or unsigned integers or 32 bits floating point numbers. Unfortunately some functions don't support all data types. The choose should be made base on a function availability that need to be used.

What one need to remember is that the NPP is mainly C library. It is a reason that some features like function overloading are not available. One need to use functions that exactly match parameters types. To help to recognize functions a special function name convention has been introduced. Each NPP function begins with nppi. The data type that the function is dedicated for might be distinguish by its suffix. For example suffix R indicates the primitive operates only on a rectangular. Suffix I indicates that the primitive works "in-place". This is well described in the NPP documentation [26].

The image that is passed to the NPP is always described by three parameters: pointer to the image, image size (as ROI), and line step. Pointer to the image has to be the CUDA device pointer. Line step is the number of bytes between successive rows in the image. Fig. 23 shows the use of the NPP library.

```
166  void Movement::erosion()
167  {
168      NppiSize oMaskSize = { 5, 5 };
169      NppiPoint oAnchor = { 3, 3 };
170      nppiErode_8u_C1R(pdDiff, iw, pdTmp, iw, oSizeROI,
171          pdErodeMask, oMaskSize, oAnchor);
172      swap(&pdTmp, &pdDiff);
173  };
174
175  void Movement::low_pass_filter()
176  {
177      NppiSize oKernelSize = { lowPassFilter, lowPassFilter };
178      NppiPoint oAnchor = { lowPassFilter, lowPassFilter };
179      NppiSize oDstSizeROI = { iw + lowPassFilter, ih + lowPassFilter };
180      int nDstStep = oDstSizeROI.width;
181      int offset = lowPassFilter - lowPassFilter / 2;
182
183      nppiCopyReplicateBorder_8u_C1R(pdDiff, iw, oSizeROI, pdTmp,
184          nDstStep, oDstSizeROI, offset, offset);
185      nppiFilter_8u_C1R(pdTmp, nDstStep, pdDiff, iw, oSizeROI,
186          pdLPKernel, oKernelSize, oAnchor, lowPassFilter * lowPassFilter);
187  }
188
189  void Movement::binarization()
190  {
191      nppiCompareC_8u_C1R(pdDiff, iw, (Npp8u) threshold, pdTmp,
192          iw, oSizeROI, NPP_CMP_GREATER);
193      swap(&pdTmp, &pdDiff);
194  }
195
196  void Movement::swap(Npp8u **a, Npp8u **b)
197  {
198      Npp8u * tmp = *a;
199      *a = *b;
200      *b = tmp;
201  }
```

Fig. 23. CUDA implementation using the NPP library

VI. MOVEMENT TRACKING

A basic extension to the issue of movement detection is the problem of object tracking. The simplest way of tracking can be performed by drawing a trajectory (a path) for the detected object as shown in Fig. 24-26.

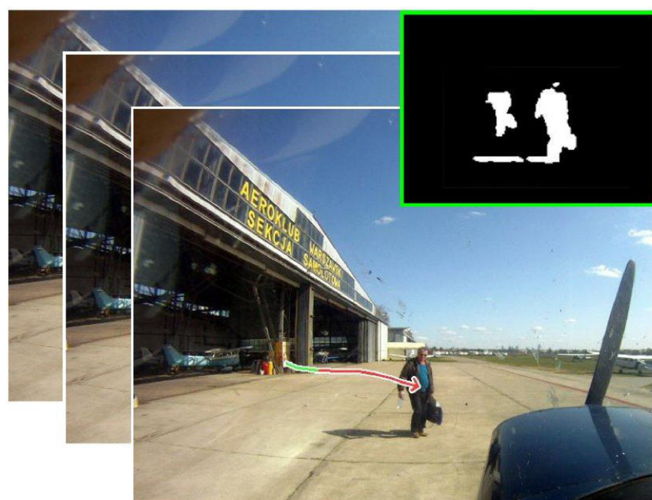


Fig. 24. An example of movement tracking in terrain conditions

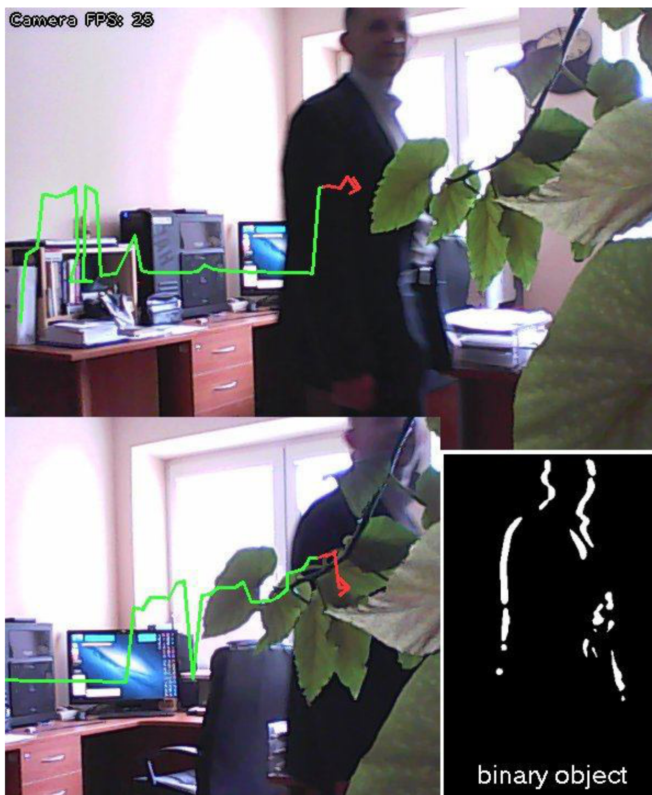


Fig. 25. An example of movement tracking inside a room

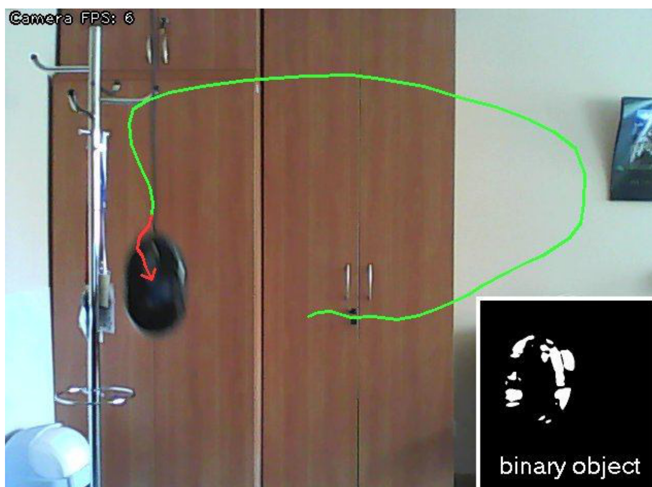


Fig. 26. An example of movement tracking of a small object (a mouse)

The suggested method allows to track only one object that is being detected. The object to be detected must occupy more than **0.1%** space of the binarized image. If that happens the object is surrounded by a rectangle and then center of its mass is calculated. If movement is detected in following frames then the track is plotted by joining calculated centers.

Aside of common template project, both implementations use the same algorithm for the considered stage, i.e. movement tracking. This is because of the algorithm simplicity which cause the CUDA implementation unnecessary. Thus, the function is common for x86 and CUDA implementation (see Fig. 27).

```
21 void process_result(Mat &result, t_detected_object &obj)
22 {
23     int iw = result.cols, ih = result.rows, is = iw * ih;
24     Npp8u * im = (Npp8u *) result.data;
25
26     obj.pt1.x = iw;
27     obj.pt1.y = ih;
28     obj.pt2.x = -1;
29     obj.pt2.y = -1;
30
31     int cnt = 0;
32     for (int y=0; y<ih; y++)
33         for (int x=0; x<iw; x++)
34         {
35             int offset = y * iw + x;
36             if (*(im + offset))
37             {
38                 cnt++;
39                 if (obj.pt1.x > x) obj.pt1.x = x;
40                 if (obj.pt1.y > y) obj.pt1.y = y;
41                 if (obj.pt2.x < x) obj.pt2.x = x;
42                 if (obj.pt2.y < y) obj.pt2.y = y;
43             }
44         }
45
46     if ((double) cnt / is > 0.001)
47     {
48         obj.detected = cnt;
49         obj.center.x = obj.pt1.x + (obj.pt2.x - obj.pt1.x) / 2;
50         obj.center.y = obj.pt1.y + (obj.pt2.y - obj.pt1.y) / 2;
51     }
52     else
53     {
54         obj.detected = 0;
55     }
56 }
```

Fig. 27. Movement tracking in C++

VII. CONCLUSION

There are two gains of the performed work that are fully concordant to the title of this paper: the elaboration of the method of movement tracking and its implementation in CUDA. The elaborated method can be described as a sequence of actions, what is shown in Fig. 28.

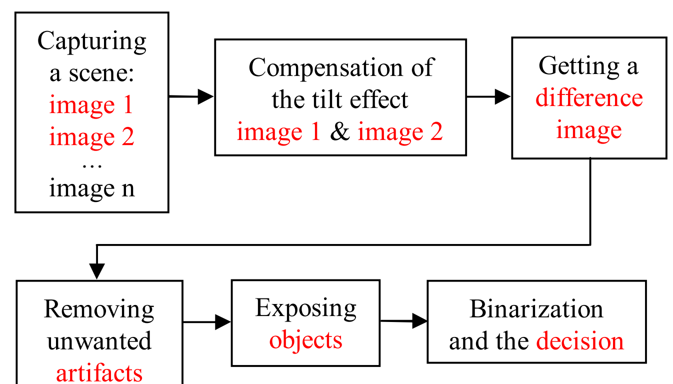


Fig. 28. A block diagram of processes for the elaborated method

CUDA and x86 implementations of the method were examined in details and optimized to receive the best performance. Performed benchmarks concerned only on selected portion of the source code directly responsible for

movement detection and tracking. Functions common for both implementations has been omitted in benchmarking.

The presented solution shows a significant performance difference between the implementation for x86 and the massively parallel implementation in CUDA. Both implementations give the same final result – confirmation that solution is correct. Performed benchmarks demonstrated substantial acceleration thanks to CUDA implementation which is suitable for a **real-time system**. It was possible to reach the speed of **25+ fps** for resolution **640x480**, at least **10 times faster** then in the case of x86 implementation. The upper limit velocity of tracked objects for the elaborated method is **4 m/s**. It is the outcome of the distance between adjacent frames. The lower limit velocity of tracked objects can be widely adjusted by the distance between analyzed frames.

In order to enrich the method an extension about identification of the detected object may be added using a method similar to one described in [23]. Another challenge is the problem of tracking multiple independent objects [20], [21].

REFERENCES

[1] M. Andriluka, S. Roth, B. Schiele, *People-tracking-by-detection and people-detection-by-tracking*, Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2008, pp. 1-8. http://dx.doi.org/10.1109/CVPR.2008.4587583

[2] A. Bugeau, P. Perez, *Detection and segmentation of moving objects in highly dynamic scenes*, IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2007, pp. 1-8. http://dx.doi.org/10.1109/CVPR.2007.383244

[3] S. Dasiopoulou, V. Mezaris, I. Kompatsiaris, V. K. Papastathis, M. G. Strintzis, *Knowledge-assisted semantic video object detection*, IEEE Transactions on Circuits and Systems for Video Technology Vol. 15, (10) 2005, pp. 1210–1224. http://dx.doi.org/10.1109/TCSVT.2005.854238

[4] Guofeng Zhang, Jiaya Jia, Wei Xiong, Tien-tsin Wong, Pheng-ann Heng, Hujun Bao: *Moving object extraction with a hand-held camera*, IEEE International Conference on Computer Vision, 2007, pp. 1-8. http://dx.doi.org/10.1109/ICCV.2007.4408963

[5] M. Heikkila, M. Pietikainen, *A texture-based method for modeling the background and detecting moving objects*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 28, (4) 2006, pp. 657–662. http://dx.doi.org/10.1109/TPAMI.2006.68

[6] R. Jain, H. Nagel, *On the analysis of accumulative difference pictures from image sequence of real world scenes*. IEEE Trans. Pattern Anal. Machine Intell., Vol. 1 (2) 1979, pp. 206-214. http://dx.doi.org/10.1109/TPAMI.1979.4766907

[7] Alper Yilmaz, Omar Javed, Mubarak Shah, *Object Tracking: A Survey*. ACM Computing Surveys, Vol. 38, No. 4, Article 13, Publication date: December 2006. http://doi.acm.org/10.1145/1177352.1177355

[8] K. J. Meech, *Astronomical image processing - applications to ultra-faint imaging of small, moving, solar system bodies: comets and near-Earth-objects*. Intelligent Processing and Manufacturing of Materials, Vol. 1, 1999. http://dx.doi.org/10.1109/IPMM.1999.792520

[9] G. Jahn, J. Wendt, M. Lotze, F. Papenmeier, M. Huff, *Brain activation during spatial updating and attentive tracking of moving targets*. Brain & Cognition, 78, 2012, pp. 105-113. http://dx.doi.org/10.1016/j.bandc.2011.12.001

[10] J. Ericson, J. Christensen, *Reallocating attention during multiple object tracking*. Attention, Perception & Psychophysics, 74, 2012, pp. 831-840. http://dx.doi.org/10.3758/s13414-012-0294-z

[11] K.A. Patwardhan, G. Sapiro, V. Morellas, *Robust foreground detection in video using pixel layers*, IEEE Transactions on Pattern Analysis and Machine Intelligence Vol. 30, (4) 2008, pp.746-751. http://dx.doi.org/10.1109/TPAMI.2007.70843

[12] Y. Wang, J.F. Doherty, R.E. Van Dyck, *Moving object tracking in video*. In proceedings of 29th Applied Imagery Pattern Recognition Workshop, 2000, pp. 95-101. http://dx.doi.org/10.1109/AIPRW.2000.953609

[13] D. Hurych, K. Zimmermann, T. Svoboda, *Fast Learnable Object Tracking and Detection in High-resolution Omnidirectional Images*. VISAPP, 2011, pp.521-530.

[14] Hironori Sumitomo, *Monitoring camera system, monitoring camera control device and monitoring program recorded in recording medium*. US 20030185419 A1, 2003.

[15] Z. Yang, Y. Zhu, and Y. Pu., *Parallel Image Processing Based on CUDA*. International Conference on Computer Science and Software Engineering 2008, Vol. 3, pp. 198–201. http://dx.doi.org/10.1109/CSSE.2008.1448

[16] Aisaka, *et al.*, *Image processing apparatus and method, and program*. United States Patent 8,577,137, November 5, 2013.

[17] D. M. Bramich, Keith Horne, M. D. Albrow, *et al.*, *Difference image analysis: extension to a spatially varying photometric scale factor and other considerations*. Monthly Notices of the Royal Astronomical Society, Volume 428, Issue 3, 2013, p.2275-2289. http://dx.doi.org/10.1093/mnras/sts184

[18] Frank Y. Shih, *Image Processing and Mathematical Morphology: Fundamentals and Applications*, CRC Press, 2009. http://dx.doi.org/10.1201/9781420089448

[19] Frank Y. Shih, *Image Processing and Pattern Recognition: Fundamentals and Techniques*, IEEE Press, 2010. http://dx.doi.org/10.1002/9780470590416

[20] P. Cavanagh, G. A. Alvarez, *Tracking multiple targets with multifocal attention*. Trends in Cognitive Sciences, 9, 2005, pp. 349-354. http://dx.doi.org/10.1016/j.tics.2005.05.009

[21] G. d'Avossa, G. Shulman, A. Snyder, M. Corbetta, *Attentional selection of moving objects by a serial process*. Vision Research, 46, 2006, pp. 3403-3412. http://dx.doi.org/10.1016/j.visres.2006.04.018

[22] W. Żorski, *Application of the Hough Technique for Irregular Pattern Recognition to a Robot Monitoring System*. Proceedings of the 11th IEEE International Conference MMAR 2005, pp.725-730.

[23] W. Żorski, K. Murawski, *Irregular patterns learning and matching in an example vision system*. Proceedings of the 18th IEEE International Conference MMAR 2013, pp.645-649.

[24] NVIDIA corporation, *CUDA C Programming Guide*, July 2013, PG-02829-001_v5.5: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[25] NVIDIA corporation, *CUDA C Best Practices Guide*, July 2013, DG-05603-001_v5.5: http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

[26] NVIDIA corporation, NVIDIA Performance Primitives (NPP), Version 4.0, 2014: http://docs.nvidia.com/cuda/pdf/NPP_Library.pdf