BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 62, No. 1, 2014
DOI: 10.2478/bpasts-2014-0016

Real-time motion tracking using optical flow on multiple GPUs

S.A. MAHMOUDI[1], M. KIERZYNKA[2,3]*, P. MANNEBACK[1], and K. KUROWSKI[2]

[1] University of Mons, 20 Parc Sq., B7000 Mons, Belgium
[2] Poznań Supercomputing and Networking Center, 10 Noskowskiego St., 61-704 Poznań, Poland
[3] Poznań University of Technology, Poznań, 2 Piotrowo St., 60-965 Pozńań, Poland

Abstract. Motion tracking algorithms are widely used in computer vision related research. However, the new video standards, especially those in high resolutions, cause that current implementations, even running on modern hardware, no longer meet the needs of real-time processing. To overcome this challenge several GPU (Graphics Processing Unit) computing approaches have recently been proposed. Although they present a great potential of a GPU platform, hardly any is able to process high definition video sequences efficiently. Thus, a need arose to develop a tool being able to address the outlined problem.

In this paper we present software that implements optical flow motion tracking using the Lucas-Kanade algorithm. It is also integrated with the Harris corner detector and therefore the algorithm may perform sparse tracking, i.e. tracking of the meaningful pixels only. This allows to substantially lower the computational burden of the method. Moreover, both parts of the algorithm, i.e. corner selection and tracking, are implemented on GPU and, as a result, the software is immensely fast, allowing for real-time motion tracking on videos in Full HD or even 4K format. In order to deliver the highest performance, it also supports multiple GPU systems, where it scales up very well.

Key words: the Lucas-Kanade method, sparse optical flow, multiple GPU computations.

1. Introduction

Motion estimation algorithms have been at the core of various methods used in computer vision for many years. Object tracking, depth estimation, robot navigation [1] or even visual odometry [2] are only a few practical applications that have been developed owing to accurate motion estimation methods. They have been also used in surveillance systems tracking humans in public places, such as metro or airports, to identify possible abnormal behaviours and threats [3]. Motion estimation algorithms serve therefore as a common building block of some more complex routines and systems.

One of the most commonly used methods for the apparent motion estimation is the optical flow, initially described in 1950 by J.J. Gibson [4]. Since then a number of methods based on this technique have been developed, e.g. Horn and Schunck [5] or Lucas and Kanade [6] with the latter being regarded as more robust to the noise and capable of tracking even small motions. The high accuracy is, however, achieved at the expense of high computational complexity. Moreover, modern surveillance systems are nowadays more commonly equipped with high resolution cameras, but despite the increased computational burden are still expected to work in real-time. As a result, a need arose for high performance implementations of motion estimation algorithms.

In recent years, a lot of attention has been given to modern computational architectures, such as GPUs, that turned out to be very efficient in various fields of science, not necessarily related to computer graphics. They have been successfully used as accelerators for example in gas and oil industry [7, 8], medical imaging [9–11], bioinformatics [12–14], metaheuristics [15], or stencil-based computations [16, 17]. Nevertheless, the primary application of GPUs is still the image and video processing [18–21]. Yet, even though several approaches to the problem of motion estimation have been published lately, including those taking advantage of modern GPUs [22–25], they are either unable to handle high definition video streams or are limited to a single GPU and thus do not scale up well. Therefore, in response to presented needs we decided to implement a highly parallel multi-GPU version of the Lucas-Kanade algorithm for motion estimation. Experimental results show that our implementation is capable of handling video streams in Full HD or even 4K standard in real-time.

The remainder of the paper is organized as follows: the background of features detection and motion tracking methods is presented in Sec. 2. Related works are discussed in the third section. Section 4 describes the proposed algorithm for motion tracking using the optical flow. The fifth section presents our GPU and multi-GPU implementations of the method. Then, experimental results are given in Sec. 6 showing time comparisons and the overall performance of CPU, GPU and multi-GPU implementations. Finally, conclusions and future works may be found in Sec. 7.

2. Background

There are two main approaches to tracking algorithms that may be applied to a video stream. The first way it to compute the motion of each pixel within the video, which is often referred to as *dense* tracking. The other way is to apply the algorithm for selected pixels only, which in turn is known as

*e-mail: michal.kierzynka@man.poznan.pl

sparse tracking. The former method provides more detailed information about the image but is also more computationally intensive, and hence is not suitable to be considered for a high resolution stream processed in real-time. The latter approach tries to reduce the computational burden by tracking only the meaningful features of the video, e.g. corners, and thus allowing for easier interpretation of the results. This section outlines some of the key methods used for both corner detection and motion tracking.

2.1. Corner detection methods. Likewise contour detection, corner extraction presents a preliminary step of several computer vision methods. A corner is defined as an area that exhibits a high gradient value in multiple directions simultaneously. In literature, one can find three categories of corner detection methods, namely: contour, intensity and model based.

Contour based methods extract edges before searching for the maximum curvature points along the contours. In some cases, they apply a polygonal approximation before selecting the intersection points [26]. In this category, Asada *et al.* [27] proposed an approach to detect corners for 2D objects from planar curves. The changes in curvatures are considered as points of interest. Mokhtarian *et al.* [28] developed a similar approach using inflexion points of a planar curve. The authors in [29] proposed to extract line segments from the image contours. The intersections of these segments present points of interest.

Intensity methods are based on computing the intensity function which represents the gray value variations between pixels of an image. In this category, Harris and Stephens [30] developed a corner detector which demonstrates a strong invariance to rotation, scaling, illumination variation, and image noise. It is based on the local auto-correlation function which measures the local changes of the signal with patches shifted by a small amount in different directions. Bouguet [31] proposed an efficient and simple implementation of Harris detector using several steps. A discrete predecessor of the Harris detector was described by Moravec [32], where the discreteness refers to the shifting of the patches.

Model based methods propose to fit a parametric intensity model to the image. They offer a high sub-pixel accuracy, but are limited to specific types of interest points such as L corners. In this category, Rohr *et al.* [33] described a junction model combined with a Gaussian filter. Parameters in this model (angle between the L corner and the x axis, the gray values, etc.) are adjusted to the image in order to detect L corners. This method was improved by Deriche and Blaszka in [34] by applying a new method for Gaussian smoothing in order to obtain better noise elimination. They presented also a faster convergence by the use of large image regions. Authors in [35] presented an approach to extract junctions using an optimal description of the signal.

The intensity based methods are the most widely used for corner detection since they do not require any information about contours nor types of points of interest. The work in [36] presents a comparison of classical techniques, and according to the presented results, the Harris corner detector, proposed in [30], has better performance than other detectors. This technique presents also a well-known robust solution for detecting points to track in a video stream, and therefore we exploit it in our implementation.

2.2. Motion tracking methods. Motion tracking methods try to estimate the displacement and velocity of features in a given video frame with respect to the previous one. They are considered necessary for several applications such as human behavior understanding, event detection or video indexation. These methods usually exploit different algorithms such as optical flow estimation [5], SIFT descriptors [37], block matching technique [38], etc. In this work we are more focused on the optical flow methods since they present a promising solution for human or car tracking even in noisy and crowded scenes or in case of small motions.

The optical flow represents a distribution of apparent velocities of movement of brightness pattern in an image. The method computes the spatial displacements of image pixels based on the constant light hypothesis which assumes that the properties of consecutive images are similar in a small region as shown in Eq. (1):

$$I(x, y, t) = I(x + \delta x, y + \delta y, t + 1), \tag{1}$$

where $\delta x, \delta y$ are displacements along horizontal and vertical axes, respectively; $I(x, y, t)$ is the gray level of the pixel (x, y) at moment t; $I(x + \delta x, y + \delta y, t + 1)$ is the gray level of the pixel $(x + u,\ y + v)$ at moment $t + 1$.

Based on this hypothesis (Eq. (1)), we obtain the following constraints:

$$\frac{dI}{dt} = 0, \tag{2}$$

$$\iff \frac{\delta I}{\delta x}\frac{d_x}{d_t} + \frac{\delta I}{\delta y}\frac{d_y}{d_t} + \frac{\delta I}{\delta t} = 0, \tag{3}$$

$$\iff Ix.u + Iy.v + It = 0, \tag{4}$$

where $u = \dfrac{d_x}{d_t}$, $v = \dfrac{d_y}{d_t}$.

Equation (4) presents the optical flow constraint with respect to movement. However, this is only one equation and the method needs to determine two unknown values (the two components of the movement vector for a given point). Therefore, in order to detect the displacement coordinates (x and y) for each pixel some additional hypotheses are introduced.

In this context, Horn and Schunck [5] introduced a global constraint of smoothness to estimate the optical flow over the whole image. It tries to minimize distortions in the flow and prefers solutions showing more smoothness. Indeed, this method supposes that the neighboring pixels should have similar velocities which means that the optical flow has a progressive variation. However, this method is hampered by its low efficiency in case of small motions. Lucas and Kanade [39] developed a robust local method for the optical flow estimation assuming that the flow is constant in a local neighborhood which enables to solve the basic optical flow Eq. (4) for all pixels in that neighborhood. The Lucas-Kanade method is also less sensitive to image noise than the point-wise methods.

However, in the case of uniform regions, it may provide inaccurate results (in terms of the optical flow). Authors in [40, 41] proposed a block matching algorithm for motion estimation in video sequences. The purpose of the latter method is to detect motion between two consecutive images in a blockwise sense. Each block (or region) from the current frame is matched with a corresponding block in the next image by shifting the current block over a neighborhood of pixels in the next frame. The matching is based on computing the distances between the gray values of two successive blocks, and the shift having the smallest total sum of distances represents the best match.

The Lucas-Kanade method is the most widely used variant of the optical flow estimation since it presents a local approach providing accurate results in many cases. It is also less sensitive to image noise, and allows for tracking even small motions. Due to these properties, we propose to exploit the Lucas-Kanade method in our work. In the literature, several optical flow based techniques may be found applied to different problems. In [42], for instance, the authors estimate velocity of ground vehicles. The objective is to compute this estimation automatically from video sequences acquired with a fixed camera. The vehicle motion is detected and tracked over the frames using the Horn and Schunck optical flow method [5]. Andrade *et al.* [43] developed a method for modeling normal behavior in order to detect abnormal events. This solution combines the optical flow vectors, Hidden Markov Models (HMM) [44], spectral clustering and principal component analysis for detecting crowd emergency scenarios. Authors in [45] proposed a method for human face tracking in unconstrained videos, based on TLD (Tracking-Learning-Detection) approach, while [46] presented some tracking techniques using multiple cameras. There are also some works in [47] for detecting abnormal situations in crowded scenes by analyzing the motion aspect instead of tracking subjects one by one. These works present only a small fraction of possible applications for the optical flow methods, and we may certainly conclude here that there is a real need for this kind of efficient implementations in the market.

3. Related works

Most of the image and video processing algorithms apply similar or even the same computations to many pixels. This fact makes these algorithms well adapted for acceleration on GPU by exploiting its processing units in parallel. In the case of GPU-accelerated optical flow motion tracking algorithms, we may distinguish two categories of related works. The first presents so called dense optical flow which tracks all frame pixels without selecting any features. In this context, [25] proposed a GPU implementation of the Lucas-Kanade method for the optical flow estimation. The software was programmed using the CUDA library (Compute Unified Device Architecture) to compute dense and accurate velocity field at about 15 frames per second (FPS) for the image resolution of 640×480. Authors in [23] presented the CUDA implementation of the Lucas-Kanade optical flow method with a real-time processing (25 FPS) of low resolution videos (316×252). This method produces dense displacement field based on a straightforward processing procedure.

The second category includes software tools tracking selected image features only. Sinha *et al.* [22] developed a GPU implementations of the popular KLT feature tracker [48] and the SIFT feature extraction algorithm [37]. This was developed with the OpenGL/Cg libraries allowing to extract about 800 features from 640×480 video at 10 FPS which is approximately 10 times faster than the corresponding CPU implementation. Their software may also track a thousand features in real-time (30 FPS) in a video of resolution 1024×768, which is around 20 times faster than in the case of its CPU version. In [49] the authors proposed a GPU-based block matching technique using OpenGL. This implementation offered a real-time processing of 640×480 video with a speedup of 3 compared to its CPU version. Sundaram *et al.* [50] developed a method for computing point trajectories based on a fast GPU implementation of the optical flow algorithm that tolerates fast motion. This parallel implementation runs at about 22 fps, which is 78 times faster faster than its CPU version.

However, despite their great speedups, none of the abovementioned GPU-based software tools can provide real-time processing of high definition videos (HD/Full HD). Moreover, they are not well adapted for exploiting multiple GPUs simultaneously. Our contribution focuses on the development of a real-time motion tracking method using the optical flow on multiple GPUs. The proposed method includes both feature detection and motion tracking steps which are implemented entirely on GPU. We contribute also by exploiting effectively multiple GPUs, with an efficient management of different kinds of GPU memory to obtain fast access to frame pixels. As a result, our implementation is able to perform a real-time motion tracking on Full HD or even 4K standard videos.

4. Motion tracking algorithm

Before presenting the implementation details of our GPU-based software tool for the optical flow motion tracking, we describe in this section the consecutive steps of the developed algorithm. The approach consists of three main steps: features detection, the optical flow based features tracking and static features removing.

4.1. Features detection. The first step of the proposed method is to detect features that are good to track, i.e. corners. To achieve this, we have exploited the Bouguet's corner extraction technique [31], based on the Harris detector [30]. This is a very efficient method thanks to its invariance to rotation, scaling, brightness and noise. It is based on five steps: spatial derivatives computation, eigenvalues computation, maximum eigenvalue selection, small eigenvalues removing and eigenvalues selection, all of them briefly described below.

1. **Spatial derivatives computation:** this step consists of computing the matrix G of spatial derivatives for each pixel

using the Eq. (6). This 4-element matrix (2×2) is calculated with the spatial derivatives I_x, I_y computed using the Eq. (5).

$$I_x(x,y) = \frac{I(x+1,y) - I(x-1,y)}{2}, \quad I_y(x,y) = \frac{I(x,y+1) - I(x,y-1)}{2}, \tag{5}$$

$$G = \begin{pmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{pmatrix}. \tag{6}$$

2. **Eigenvalues computation:** based on the matrix G, for each pixel we calculate two eigenvalues of which only the greater is kept.
3. **Maximum eigenvalue selection:** once all the eigenvalues are calculated, the algorithm retrieves the maximum value.
4. **Small eigenvalues removing:** the search for small eigenvalues is performed by comparing the eigenvalue of each pixel with the maximum eigenvalue. If this absolute value is lower that 5% of the maximum value, the pixel is excluded.
5. **Corner selection:** for each image area (of predefined size) the algorithm extracts one pixel having the largest eigenvalue. The selected pixels represent points of interest (the final corners).

4.2. Optical flow feature tracking. Once the corners are selected, we track them within the next frame using the optical flow technique. We exploit the Lucas-Kanade [31] algorithm for the optical flow estimation. As pointed out in the Background section, this method is well-known for its high efficiency, accuracy and robustness. The algorithm consists of seven steps that are briefly described below.

- **Step 1: Pyramid construction:** in the first step, the algorithm computes a pyramid representation of images I and J which represent two consecutive images from the video. A conversion to grayscale level is applied to these two images representing the pyramid level 0. The rest of pyramid levels are built in a recursive fashion by applying a Gaussian filter. Each pyramid level is represented by the same image as in the previous level, but with a reduced resolution (half). The image resolution for each level L is calculated with Eqs. (7) and (8).

$$Width^L = \frac{Width^{L-1} + 1}{2}, \tag{7}$$

$$Height^L = \frac{Height^{L-1} + 1}{2}. \tag{8}$$

Once the pyramid is constructed, a loop is launched that starts from the smallest image (the highest pyramid level) and ends with the original image (level 0). Its goal is to propagate the displacement vector between the pyramid levels. Note that this vector is initialized with 0 values, and will be calculated during next steps.

- **Step 2: Pixels matching over levels:** for each pyramid level (described in the previous step), the new coordinates of previously detected corners (see Subsec. 4.1) are calculated. This computation is performed with Eq. (9).

$$x^L = \frac{x}{2^L}, \qquad y^L = \frac{y}{2^L}. \tag{9}$$

As an example, the pixel having the coordinates (204, 100) at level 0 has the coordinates (102, 50) at level 1, and (51, 25) at level 2.

- **Step 3: Local gradient computation:** in this step, the matrix of spatial gradient G is computed for each corner (point of interest) of the image I, using Eq. (12). This matrix of four elements (2×2) is calculated based on the spatial derivatives I_x and I_y computed using Eqs. (10) and (11).

$$I_x(x,y) = \frac{I^L(x+1,y) - I^L(x-1,y)}{2}, \tag{10}$$

$$I_y(x,y) = \frac{I^L(x,y+1) - I^L(x,y-1)}{2}. \tag{11}$$

The computation of the gradient matrix takes into account the area (window) of pixels which is centered on the point to analyze (track). The size of the window depends on image type and size. Generally, size of 7×7 or 9×9 is a commonplace.

$$G = \sum_{x_i = x^L - w}^{x^L + w} \sum_{y_i = y^L - w}^{y^L + w} \cdot \begin{bmatrix} I_x^2(x_i,y_i) & I_x(x_i,y_i) I_y(x_i,y_i) \\ I_x(x_i,y_i) I_y(x_i,y_i) & I_y^2(x_i,y_i) \end{bmatrix}. \tag{12}$$

In Eq. (12), the size of the window is: $(2w+1) \times (2w+1)$.

- **Step 4: Iterative loop launch and temporal derivative computation:** in this step, a loop is launched and iterated until the difference between the two successive optical flow measures (calculated in the next step), or iterations, is higher than a defined threshold.
Once the loop is launched, the computation of the temporal derivatives is performed using the image J (second image) based on Eq. (13).

$$I_t(x,y) = I^L(x,y) - J^L(x + g_x + v_x, y + g_y + v_y). \tag{13}$$

This derivative is obtained by the subtraction of each point (corner) of the image I (first image) and its corresponding corner in the image J (second image). The values of g_x and g_y, initialized to zero, present the displacement estimation which is then propagated between successive pyramid levels. The values of v_x and v_y present the corrections of displacement estimation computed and propagated within the iterative loop in Step 6.

- **Step 5: Optical flow computation:** the optical flow measure is calculated using the gradient matrix (cf. Step 3) and the shift vector $\overline{b}$. This vector represents the sum of temporal derivatives (in a window of size $2w + 1$) as shown in Eq. (14).

$$\overline{b} = \sum_{x_i = x^L - w}^{x^L + w} \sum_{y_i = y^L - w}^{y^L + w} \cdot \begin{bmatrix} I_t(x_i, y_i) & I_x(x_i, y_i) \\ I_t(x_i, y_i) & I_y(x_i, y_i) \end{bmatrix}. \tag{14}$$

Then, the measure of optical flow $\overline{n}$ is calculated by multiplying the inverse of the gradient matrix G by the shift vector $\overline{b}$ (Eq. (15)).

$$\overline{n} = G^{-1}\overline{b}. \tag{15}$$

- **Step 6: Estimation correction and end of the iterative loop:** during this step, a correction of estimation Eq. (16) is applied before the results are propagated to the next iteration of the iterative loop. v_x and v_y are initialized to zero.

$$v_x = v_x + n_x, \qquad v_y = v_y + n_y. \tag{16}$$

  We may distinguish two ways of the iterative loop termination (launched in Step 4). The first is when the algorithm reaches the last iteration (the maximum number of iterations). The second case occurs when the measured correction is smaller than the defined threshold.
- **Step 7: Result propagation and end of the pyramid loop:** In this step the current results are propagated to the lower level using Eq. (17).

$$g_x = 2(g_x + v_x), \qquad g_y = 2(g_y + v_y). \tag{17}$$

  Once the algorithm reaches the lowest pyramid level (the original image), the pyramid loop (launched in the first step) is stopped. The vector $\overline{g}$ represents the final optical flow value of the analyzed corner.

Finally, the result of tracking n features (corners) is a set of n vectors as shown in Eq. (18):

$$\Omega = \{\omega_1 \ldots \omega_n \mid \omega_i = (x_i, y_i, v_i, \alpha_i)\}, \tag{18}$$

where x_i is the x coordinate of the feature i; y_i is the y coordinate of the feature i; v_i represents the velocity of the feature i; α_i denotes motion direction of the feature i.

4.3. Static features removing. The final step removes the static and noisy features. Features (points of interest) having the velocity equal to zero (i.e., $v_i = 0$) are considered as static. Noisy features are the isolated features that have a relatively large difference in angle α_i and velocity v_i values compared to their nearest neighbors due to tracking calculation errors.

5. Implementation of the algorithm

As pointed out in the previous sections, a graphics processing unit makes an effective tool for improving the performance of image and video processing algorithms. This section presents our proposed GPU and multi-GPU versions of the optical flow based feature tracking method in three parts. The first describes our development scheme for the video processing on GPU. The second part presents the GPU implementation of the proposed feature tracking method. The multi-GPU implementation is described in the last part.

5.1. Development scheme for the video processing on GPU. The proposed scheme is based upon CUDA for parallel computing and OpenGL for visualization. It consists of three steps: loading video frames on GPU, CUDA parallel processing and OpenGL visualization (see Fig. 1).

1. **Loading video frames on GPU:** we start with reading and decoding the video frames using the OpenCV library[1]. We copy the current frame on a device (GPU) that processes it in the next step.
2. **CUDA parallel processing:** before launching the parallel processing of the current frame, the number of GPU threads in the so called blocks and grid has to be defined, so that each thread can perform its processing on one or a group of pixels in parallel. This enables the program to process the image pixels in parallel. Note that the number of threads depends on the number of pixels. Once the number and the layout of threads is defined, different CUDA functions (kernels) are executed sequentially, but each of them in parallel using multiple CUDA threads.
3. **OpenGL visualization:** the current image (result) can be directly visualized on the screen through the video output of GPU. Therefore, we use the OpenGL graphics library that allows for fast visualization, as it can operate on the already existing buffers on GPU, and thus requires less data transfer between host and device memories. Once the visualization of the current image is completed, the program goes back to the first step in order to load and process the next frames of the video.

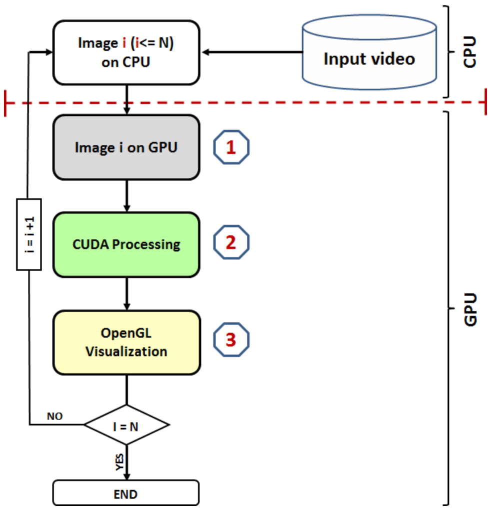

Fig. 1. Development scheme for the video processing on GPU (N denotes the total number of frames)

[1]OpenCV Computer Vision Library: `http://opencv.org`

5.2. GPU-based motion tracking using the optical flow. Based on the aforementioned scheme, we proposed the GPU implementation of the motion tracking method using both the Harris corner detection and the Lucas-Kanade algorithm for the optical flow. This solution offers efficient processing in terms of the quality of motion tracking and also improved performance thanks to the exploitation of GPU computing units in parallel. Our GPU implementation can be described in two main steps: corner detection and corner tracking, both performed on GPU.

Corner detection. We developed a GPU implementation of the Bouguet's corner extraction method [31], which in turn is based on the Harris detector [30]. Our approach parallelizes its five steps on GPU (Fig. 2). The GPU implementation of the first step (spatial derivatives computation) is based on the parallel processing of pixels using the GPU grid of threads with the number of threads equal to the number of pixels. Each thread computes the spatial derivatives of one pixel using Eq. (5). Then, each thread calculates the matrix G for each image pixel by applying Eq. (6). The values of the neighboring pixels (left, right, top and bottom) of each image point are loaded into the GPU shared memory since these values (neighbors) are required for the computation of spatial derivatives. In the second step (eigenvalues computation), the algorithm computes the eigenvalues in parallel over image pixels based on the G matrix. In this case, we also use the GPU grid of threads with the number of threads equal to the number of pixels.

Once the eigenvalues are calculated, the algorithm extracts the maximum value (the third step), which is computed on GPU using the CUBLAS library[2]. The fourth step of the corner detection method, that is the search for small eigenvalues, is performed in the way that each GPU thread compares the eigenvalue of its corresponding pixel with the maximum eigenvalue. If this value is less than 5% of the maximum value, the pixel is excluded. In the last step, we proposed to assign a GPU thread to a group of pixels representing an area (with adjustable size, by default 10×10 pixels). This enables each thread to detect the maximum eigenvalue in its region. The pixels having these values extracted represent the detected corners.

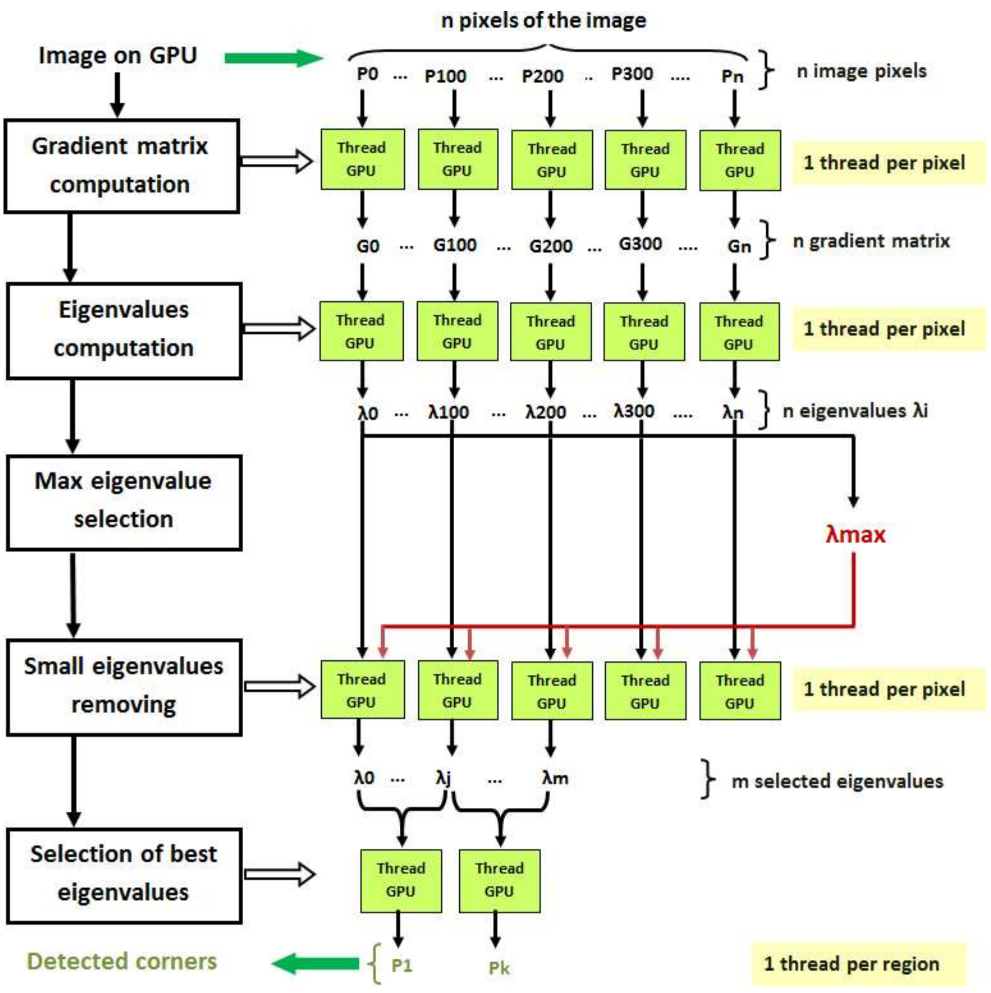

Fig. 2. GPU implementation of the corner detection phase

[2]NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) library: `https://developer.nvidia.com/cuBLAS`

Corner tracking. Once the corners are detected, the algorithm tracks them using the optical flow estimation based on the Lucas-Kanade approach. Thus, we calculate the optical flow vectors for previously selected corners only, and not for all frame pixels. The parallel implementation of this method uses a similar technique as for the corner detection. The steps described in Subsec. 4.2 are executed in parallel using the CUDA library such that each GPU thread applies its instructions (among the seven steps) on one previously detected frame corner. Therefore, the number of selected GPU threads is equal to the number of corners. Since the algorithm looks at the neighboring pixels, for a given corner, the images, or pyramid levels to be more precise, are kept in the texture memory. This allows for faster access within the 2-dimensional spatial data. Other data, e.g. the arrays with computed displacements, are kept in the global memory, and are cached in the shared memory if needed. The software was optimized for the Fermi architecture and many low-level optimizations, like the number of threads within a block or the actual location of different variables, were selected empirically. Moreover, the great advantage of the Fermi architecture is the L1/L2 cache that allows for an efficient data fetch in case of recurring reads of consecutive elements from a cache line. This feature also contributed to the acceleration of parts of the code.

Figure 3 summarizes the GPU implementation of the Lucas-Kanade algorithm applied to a set of points of interest. An example output from the algorithm is presented in Fig. 4.

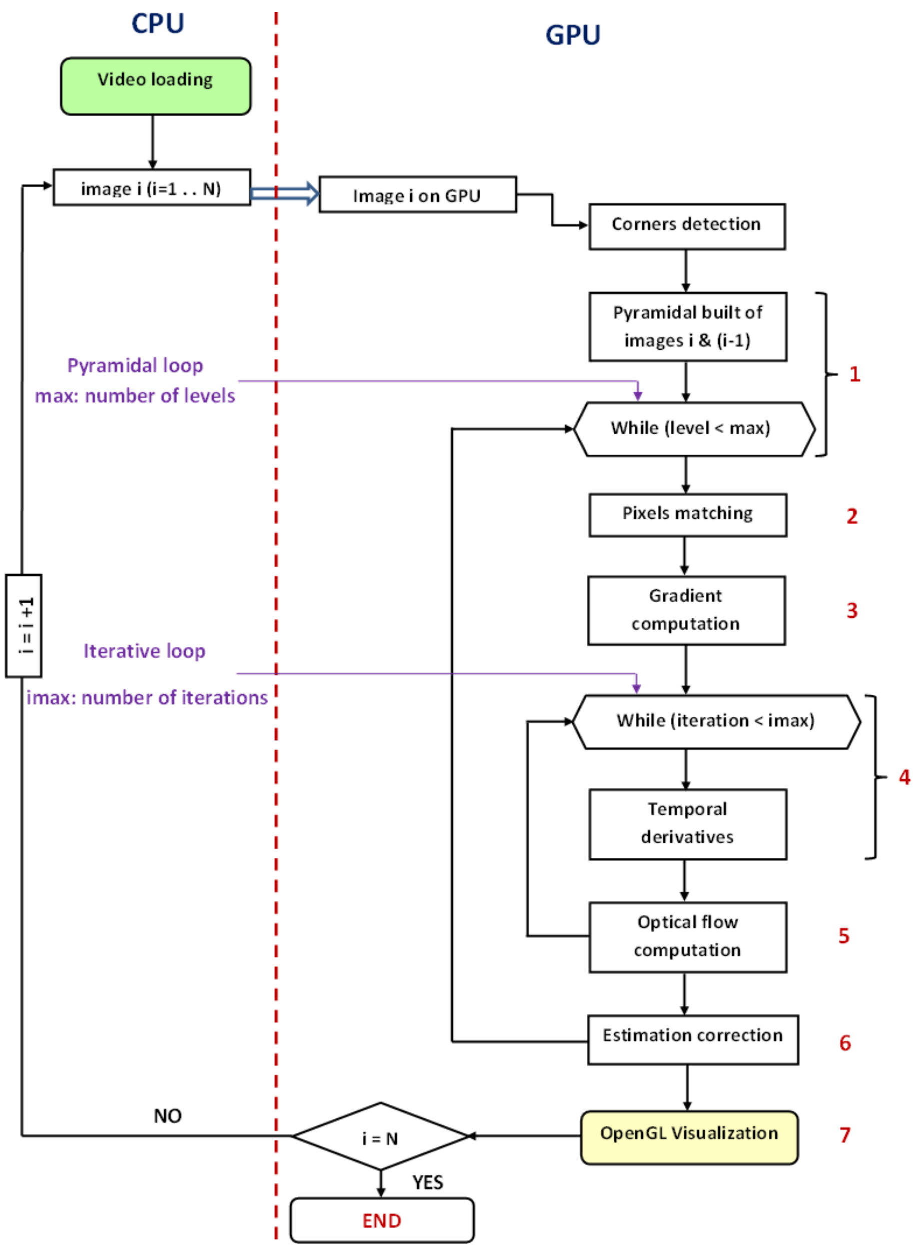

Fig. 3. A GPU implementation of the Lukas-Kanade tracking method (N denotes the number of video frames)

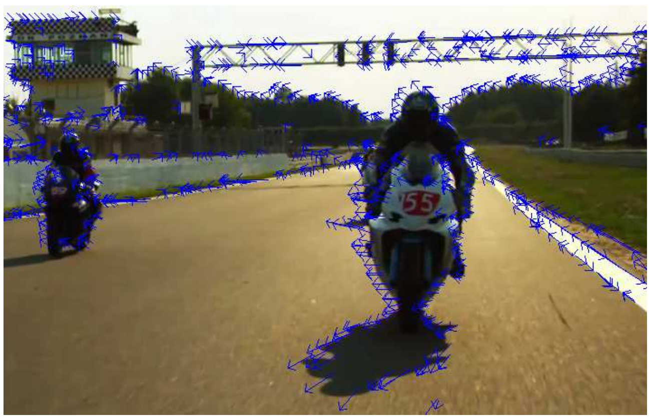
Fig. 4. An example part of the Full HD video frame with characteristic points detected with the Harris corner detector and then tracked with the Lucas-Kanade method. Displacements are marked with arrows. Note that the arrows located on the static objects like trees or a building are there as a result of moving camera

5.3. Multiple GPU implementation. The nature of the video processing algorithms is usually highly parallel. This also applies to the problem of corner detection as well as motion tracking. Therefore, apart from implementing described methods on a single GPU, we also prepared a version that takes advantage of multiple GPU systems that nowadays are becoming commonplace. The program, once launched, first detects the number of GPUs available in the system, and initializes all of them. In the corner detection phase, the input image frame is first uploaded to each GPU. Although it seems to be redundant to have the whole image on each GPU, the data is needed there in a later phase. Nevertheless, the frame is virtually divided into equally-sized subframes along y dimension. Once the image data is available, each GPU is responsible for detecting corners within its part of the frame. In the next phase, that is in the Lucas-Kanade method, each GPU computes the pyramid. This is to ensure, that each unit has then access to the whole pyramid, which is relatively easy to compute, comparing to the potential burden of copying its structure via PCI-E connection. Once the pyramid is computed, each GPU tracks previously detected corners according to the Lucas-Kanade algorithm. If visualization is enabled, then at the end of the computations for a given frame, all the results, namely tracked points together with corresponding displacements, need to be copied to the GPU which is in charge of displaying. This, however, is a fast operation since contiguous memory space is always transferred.

6. Results

6.1. Time comparisons of the Lucas-Kanade implementations. In order to evaluate the performance of our software we decided to compare it to the very recent and up to our best knowledge the fastest implementation of the Lucas-Kanade algorithm, i.e. this available in OpenCV 2.4. It is worth noting that OpenCV provides both CPU and GPU versions of the algorithm, so additional insight into the difference in performance between these two is presented as well. The optical flow method has three primary parameters, which define how much computational burden is actually imposed. These are: the number of pyramid levels, the number of iterations within each pyramid level and finally the window size. To make the comparison clear and fair we measured the performance of the methods by changing the value of parameters one by one. This was applied to all the parameters in turn. The standard values were as follows: the number of pyramid levels – 4, the number of iterations – 3 and the window size – 5×5. The video sequence used in the test had 4850 frames, each of resolution 1920×1080 pixels. The results present the time in seconds needed by each program to perform the optical flow algorithm solely, that is excluding the time spent for video decoding, its conversion to the grayscale, corner detection and results visualization. It is worth noting that in order to ensure each implementation tracked the same points, our own implementation of Harris corner detector was used in each case.

The tests were run on the following hardware:

- CPU: Intel Core 2 Quad Q8200, 2.33GHz,
- GPU: 2×NVIDIA GeForce GTX 580 with 1.5GB of RAM,
- RAM: 8GB,
- OS: 64-bit Linux.

Figure 5 presents the time needed by each implementation of the Lucas-Kanade method to process an example movie in the Full HD standard. The results depend here on the number of algorithm's refining iterations at each pyramid level. The OpenCV CPU-based method is the slowest one, whereas our own implementation running on two GPUs is the fastest. One may notice that the time used by OpenCV does not grow as fast as in case of our implementation. The reason is that OpenCV checks whether additional iteration improves the solution and exits if not, while our application does not include this feature. However, for the most commonly used value of this parameter in practice, i.e. 3, our implementation outperforms those from OpenCV.

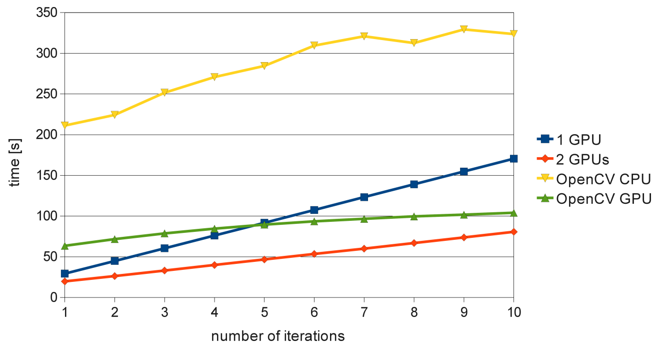

Fig. 5. Time needed to perform the optical flow algorithm depending on the number of iterations within each pyramid level

Figure 6 depicts how the window size parameter influences the computational time of different implementations. For small windows our method is the fastest one. Yet, when large windows are used for computations the GPU-based OpenCV is faster. Moreover, this method seems to be almost invariant to the window size. This is not an entirely clear result to us as the number of computations with a growing window is simply higher. We suspect that the authors must have introduced some advanced caching techniques, which give them the advantage here. Figure 7, in turn, presents the impact of growing number of pyramid levels on the performance of the algorithm. In case of both OpenCV implementations, especially CPU-based, the performance drop is more visible than in our implementation. Moreover, we may conclude that our method outperforms OpenCV here as it is more efficient regardless of the number of pyramid levels.

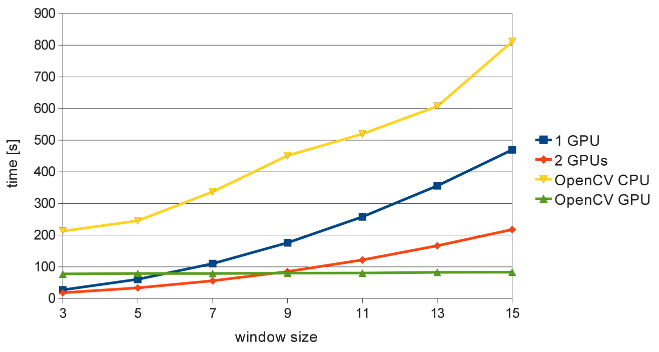

Fig. 6. Time needed to perform the optical flow algorithm depending on the window size parameter. Windows are always square, e.g. 9 corresponds to a window of size 9×9

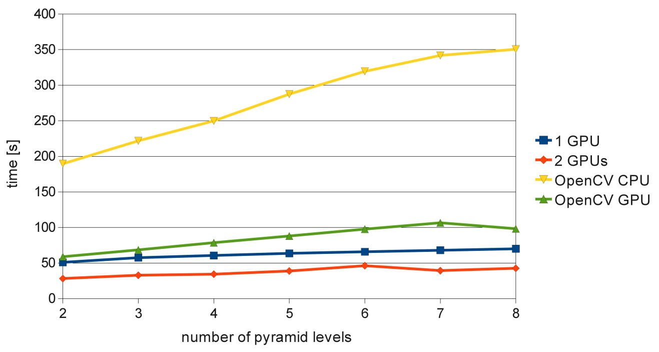

Fig. 7. Time needed to perform the optical flow algorithm depending on the number of pyramid levels

Summing up, there is no implementation that clearly outperforms all the others. While the GPU version of OpenCV's Lucas-Kanade is more robust when it comes to large windows, our implementation performs better for growing number of pyramid levels, which is especially desired when processing videos of high resolution. However, we certainly may conclude that GPU is well suited for the problem as all the GPU-based implementations were faster than the CPU one. Ultimately, the performance of the tested methods still does strongly depend on specific values of parameters, and therefore on a given real-life use case.

6.2. Multiple GPU test. One of the features of our implementation is the multiple GPU support. Therefore, to see how the algorithm may benefit from such systems we performed two tests. In the first one we measured the speedup obtained by the Harris corner detector, whereas in the second – the speedup observed in the Lucas-Kanade routine. Likewise in the previous test, computations were performed on a Full HD movie (1920×1080) consisting of 4850 frames. The presented speedup always refer to a single GPU runtime. Note that in this test we do not compare our software to any other existing implementation, since none of them supports multiple GPU configurations.

The tests were run on the following hardware:

- CPU: 2×Intel Xeon E5405, 2.00GHz,
- GPU: NVIDIA Tesla S1070 containing 4 GPUs,
- RAM: 16GB,
- OS: 64-bit Linux.

Figure 8 reveals that the speedup of the Harris corner detection routine is almost linear up to three GPUs. The use of additional fourth GPU does not improve its execution time as much. The reason is that the amount of work per single GPU is becoming too small to properly load the chip with computations. Figure 9, in turn, presents the speedup that was observed in our implementation of the Lucas-Kanade algorithm depending not only on the number of GPUs, but also on the number of pixels being tracked, which directly correlates with the amount of computations that needs to be performed. In order to carry out this kind of test we set a given number of pixels to track for each run. As we can see, the general trend is that with growing number of pixels to track the speedup obtained on the multiple GPU system improves. The maximum speedup was reached for the highest number of pixels tracked and was 3.28, 2.65 and 1.8 for 4, 3 and 2 GPUs, respectively. Hence, we may conclude that multiple GPU support indeed does influence the execution time of presented methods which are therefore perfectly suited for any system where the time is a precious resource.

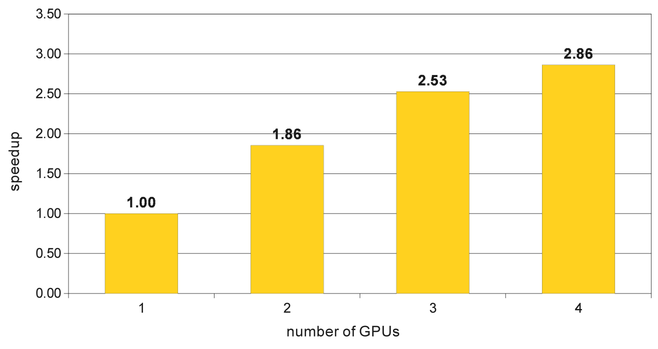

Fig. 8. The speedup of the Harris corner detector depending on the number of GPUs used. The speedup is always given as relative to a single GPU runtime

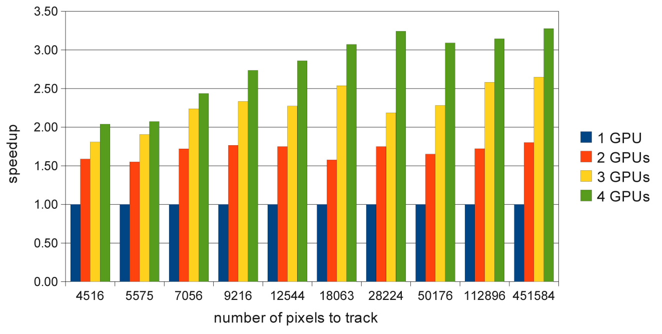

Fig. 9. The speedup of the Lucas-Kanade implementation depending on the number of GPUs used and the number of points tracked. The speedup is always given as relative to a single GPU runtime

6.3. Overall performance. To investigate which of the previously tested implementations of the optical flow algorithm is the fastest in real-life scenarios, we propose to measure their overall performance. In this test we compute the number of video frames that can be processed in a single second by each method. The measurement includes the total time spend on all the procedures, i.e.: reading the video, video decoding, conversion to grayscale, corner detection and finally application of the Lucas-Kanade method. Tests were conducted on two video sequences: one in Full HD format (4850 frames in resolution 1920×1080) and one in 4K format (1924 frames in resolution 3840×2160). It is worth noting that while in our software we used our own implementation of Harris corner detector, both CPU and GPU OpenCV runs used GPU-based corner detection from OpenCV. Therefore, the difference between the two runs of OpenCV lies only in the Lucas-Kanade method, which is executed either on CPU or GPU. Presented results are meant to compare the two complete solutions: our own and the one that may be built upon state-of-the-art OpenCV library. Obviously, in order to make the comparison fair, all the applicable parameters were set equal, in particular: the number of pyramid levels – 4, the number of iterations within each pyramid level – 3 and the window size – 7×7.

The tests were run on the following hardware:

- CPU: Intel Core 2 Quad Q8200, 2.33GHz,
- GPU: NVIDIA GeForce GTX 580 with 1.5GB of RAM,
- RAM: 8GB,
- OS: 64-bit Linux.

Figure 10 presents the overall performance of the tested implementations in terms of the number of frames processed per second. We can clearly see that the slowest solution is the CPU-based OpenCV. The GPU-based OpenCV, in turn, is only some 1.9–2.3 times faster. Therefore, we may suspect that a lot from its speedup presented in Subsec. 6.1 is actually lost by the amount of time it needs to spend on the CPU routines. Our implementation turned out to be the fastest, and at the same time the only one that achieved real-time video processing for both Full HD and 4K standards. In order to investigate this in more detail, we present Figs. 11 and 12 showing the percentage distribution of time spent on different routines for the Full HD and the 4K video, respectively. First of all, we see that the CPU Lucas-Kanade implementation is the most time-consuming routine. Therefore, it stands to reason that its parallelization is worthwhile. We can also see that both GPU-based implementations share almost identical time distribution. As a consequence, in case of GPU-based OpenCV all its three parts, namely corner detection, the Lucas-Kanade algorithm and the rest of CPU computations, were performed slower correspondingly to our solution. While the first two routines differ by definition, the time spent on CPU, mainly for video reading and decoding, should be the same. Presumably, one of the main reasons it is not so is the fact that OpenCV operates on *Mat* data type, which due to its specific behaviour requires an additional frame copy at every iteration resulting in some extra time spent on CPU. Although our solution uses OpenCV for video reading as well, it does not use *Mat* data type and therefore its overall performance is not laden with this overhead.

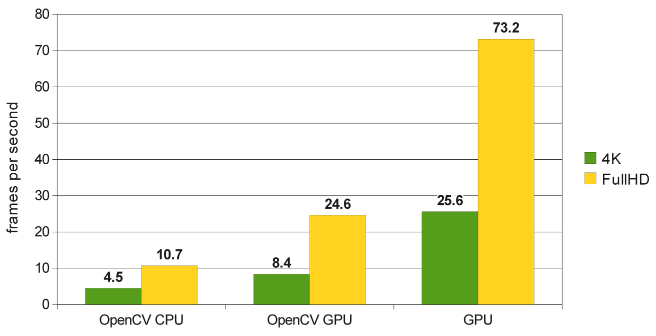

Fig. 10. The number of frames processed per second by each implementation. Two example videos were used: in Full HD and 4K resolutions

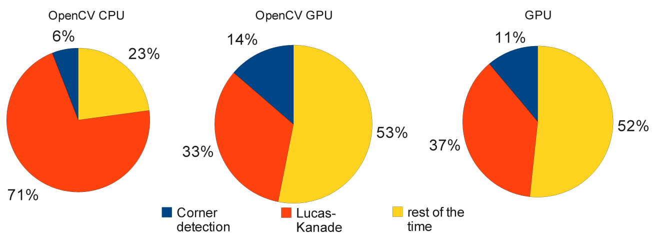

Fig. 11. The amount of time spent by each implementation on different routines. The test was carried out on a Full HD video

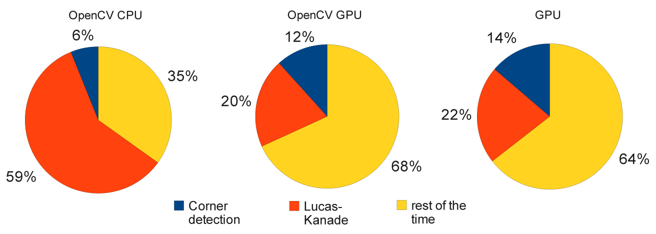

Fig. 12. The amount of time spent by each implementation on different routines. The test was carried out on a 4K video

7. Conclusions

In this work, we proposed an efficient implementation of the optical flow algorithm for the sparse motion tracking. More precisely, we developed a GPU-based software that applies the Lucas-Kanade tracking method to the previously selected video pixels. These, in turn, are selected by our implementation of the Harris corner detector. Such a combination makes it possible to track only the meaningful elements of the image, and also to apply the algorithm to high definition video sequences as the overall computational burden is much lower than in the case of dense motion tracking. Moreover, the method is optimized for the Fermi architecture, especially in the context of the memory access. Performed tests show that the proposed tool is comparable to the GPU-based OpenCV implementation, and for some values of parameters is substantially faster. Moreover, the multiple GPU support with its good scalability makes the method even faster. The software clearly outperforms the corresponding OpenCV implementation in terms of the number of video frames processed per second. It is also able to perform the motion tracking on Full HD and even 4K videos in real-time.

As future work, we are planning to apply our method for detecting and tracking more specific objects such as people or vehicles using both static and mobile cameras. An efficient motion tracking method should be of great value in many practical use-cases, e.g. camera motion estimation, event detection, motions classification etc. We are also planning to exploit the SDI capture cards[3] that allow for direct video stream capture into the GPU memory without any use of the CPU memory, which enables the software to decrease the amount of PCI-E bandwidth used for the video transmission.

Acknowledgements. The presented work was sponsored by the UCoMS project under the award number MNiSW (Polish Ministry of Science and Higher Education) Nr 469/1/N – USA/2009 in close collaboration with U.S. research institutions involved in the U.S. Department of Energy (DOE) funded grant under award number DE-FG02-04ER46136 and the Board of Regents, State of Louisiana, under the contract no. DOE/LEQSF(2004-07).

Authors would like to thank the French Community of Belgium for supporting this work under the ARC-OLIMP (AUWB-2008-12-FPMs11) research project. Authors acknowledge also the support of the European COST action IC805.

REFERENCES

[1] N. Ohnishi and A. Imiya, "Dominant plane detection from optical flow for robot navigation", *Pattern Recognition Letters* 27 (9), 1009–1021 (2006).

[2] K. Aires, A. Santana, and A. Medeiros, "Optical flow using color information", *Proc. 2008 ACM symp. on Applied Computing* 1, 1607–1611 (2008).

[3] A. Fonseca, L. Mayron, D. Socek, and O. Marques, "Design and implementation of an optical flow-based autonomous video surveillance system", *Proc. IASTED* 1, 209–214 (2008).

[4] J. Gibson, *The Perception of the Visual World*, Houghton Mifflin, Boston, 1950.

[5] B. Horn and B. Schunck, "Determining optical flow", *Artificial Intelligence* 2, 185–203 (1981).

[6] B. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision", *Proc. Imaging Understanding Workshop* 1, 121–130 (1981).

[7] M. Kulczewski, K. Kurowski, M. Kierzynka, M. Dohnalik, J. Kaczmarczyk, and A. Borujeni, "Modern hardware architectures accelerate porous media flow computations", *AIP Conference Proc.* 1, 1453, 161–166 (2012).

[8] M. Ciznicki, M. Kierzynka, K. Kurowski, B. Ludwiczak, K. Napierala, and J. Palczynski, "Efficient isosurface extraction using marching tetrahedra and histogram pyramids on multiple GPUs", *Lecture Notes in Computer Science* 7204, 343–352 (2012).

[9] S.A. Mahmoudi, F. Lecron, P. Manneback, M. Benjelloun, and S. Mahmoudi, "GPU-based segmentation of cervical vertebra in X-ray images", *HPCCE Workshop, IEEE Int. Conf. on Cluster Computing* 1, 1–8 (2010).

[10] F. Lecron, S.A. Mahmoudi, M. Benjelloun, S. Mahmoudi, and P. Manneback, "Heterogeneous computing for vertebra detection and segmentation in X-ray images", *Int. J. Biomedical Imaging: Parallel Computation in Medical Imaging Applications* 2011, 1–12 (2011).

[11] S.A. Mahmoudi, F. Lecron, P. Manneback, M. Benjelloun, and S. Mahmoudi, "Efficient exploitation of heterogeneous platforms for vertebra detection in X-ray images", *Proc. Biomedical Engineering International Conf.* 1, 1–6 (2012).

[12] J. Blazewicz, W. Frohmberg, M. Kierzynka, E. Pesch, and P. Wojciechowski, "Protein alignment algorithms with an efficient backtracking routine on multiple GPUs", *BMC Bioinformatics* 12, 181 (2011).

[13] J. Blazewicz, W. Frohmberg, M. Kierzynka, and P. Wojciechowski, "G-PAS 2.0 – an improved version of protein alignment tool with an efficient backtracking routine on multiple GPUs", *Bull. Pol. Ac.: Tech.* 60 (3), 491–494 (2012).

[14] J. Blazewicz, W. Frohmberg, M. Kierzynka, and P. Wojciechowski, "G-MSA – a GPU-based, fast and accurate algorithm for multiple sequence alignment", *J. Parallel and Distributed Computing* 73 (1), 32–41 (2013).

[15] R. Nowotniak and J. Kucharski, "GPU-based tuning of quantum-inspired genetic algorithm for a combinatorial optimization problem", *Bull. Pol. Ac.: Tech.* 60 (2), 323–330 (2012).

[16] M. Blazewicz, S. Brandt, M. Kierzynka, K. Kurowski, B. Ludwiczak, J. Tao, and J. Weglarz, "CaKernel – a parallel application programming framework for heterogenous computing architectures", *Scientific Programming* 19 (4), 185–197 (2011).

[17] M. Blazewicz, I. Hinder, D. Koppelman, S. Brandt, M. Ciznicki, M. Kierzynka, F. Loffler, E. Schnetter, and J. Tao, "From physics model to results: An optimizing framework for cross-architecture code generation", *Scientific Programming* 21 (1–2), 1–16 (2013).

[18] M. Ciznicki, M. Kierzynka, P. Kopta, K. Kurowski, and P. Gepner, "Benchmarking data and compute intensive applications on modern CPU and GPU architectures", *Procedia Computer Science* 9, 1900–1909 (2012).

[3]NVIDIA Quadro SDI Capture: `http://www.nvidia.com/object/product_quadro_sdi_capture_us.html`

[19] S.A. Mahmoudi and P. Manneback, "Efficient exploitation of heterogeneous platforms for images features extraction", *3rd Int. Conf. on Image Processing Theory, Tools and Applications (IPTA)* 1, 91–96 (2012).

[20] P. Ricardo Possa, S.A. Mahmoudi, N. Harb, and C. Valderrama, "A new self-adapting architecture for feature detection", *22nd Int. Conf. on Field Programmable Logic and Applications* 1, 643–646 (2012).

[21] P. Ricardo Possa, S.A. Mahmoudi, N. Harb, C. Valderrama, and P. Manneback, "A multi-resolution fpga-based architecture for real-time edge and corner detection", *IEEE Trans. on Computers* 6, 130 (2013).

[22] S. Sinha, J.-M. Fram, M. Pollefeys, and Y. Genc, "GPU-based video feature tracking and matching", *EDGE, Workshop on Edge Computing Using New Commodity Architectures* 1, CD-ROM (2006).

[23] Y. Mizukami and K. Tadamura, "Optical flow computation on Compute Unified Device Architecture", *Proc. 14th International Conf. on Image Analysis and Processing* 1, 179–184 (2007).

[24] J. Huang, S. Ponce, S. Park, Y. Cao, and F. Quek, "GPU-accelerated computation for robust motion tracking using CUDA framework", *Proc. IET Int. Conf. on Visual Information Engineering* 1, CD-ROM (2008).

[25] J. Marzat, Y. Dumortier, and A. Ducrot, "Real-time dense and accurate parallel optical flow using CUDA", *Proc. WSCG* 1, 105–111 (2009).

[26] D. Douglas and T. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature", *Cartographica: Int. J. Geographic Information and Geovisualization* 10 (2), 112–122 (1973).

[27] H. Asada and M. Brady, "The curvature primal sketch", *IEEE Trans. on Pattern Analysis and Machine Intelligence* 8 (1), 2–14 (1986).

[28] F. Mokhtarian and A. Mackworth, "Scale-based description and recognition of planar curves and two-dimensional shapes", *IEEE Trans. on Pattern Analysis and Machine Intelligence* 8 (1), 34–43 (1986).

[29] R. Horaud, T. Skordas, and F. Veillon, "Finding geometric and relational structures in an image", *Proc. First Eu. Conf. Comp. Vision* 1, 374–384 (1986).

[30] C. Harris and M. Stephens, "A combined corner and edge detector", *4th Alvey Vision Conf.* 15, 147–151 (1988).

[31] J. Bouguet, "Pyramidal implementation of the Lucas Kanade feature tracker", *Intel Corporation, Microprocessor Research Labs*, 2000.

[32] H. Moravec, "Obstacle avoidance and navigation in the real world by a seeing robot rover", *Technical Report CMU-RI-TR-3*, Carnegie-Mellon University, Pittsburgh, 1980.

[33] K. Rohr, "Recognizing corners by fitting parametric models", *Int. J. Computer Vision* 9 (3), 213–230 (1992).

[34] R. Deriche and T. Blaszka, "Recovering and characterizing image features using an efficient model based approach", *Proc. Computer Vision and Pattern Recognition* 1, 530–535 (1993).

[35] L. Parida, D. Geiger, and R. Hummel, "Junctions: detection, classification, and reconstruction", *IEEE Trans. on Pattern Analysis and Machine Intelligence* 20 (7), 687–698 (1998).

[36] C. Schmid, R. Mohr, and C. Bauckhage, "Evaluation of interest point detectors", *Int. J. Coputer Vision* 37 (2), 151–172 (2000).

[37] D. Lowe, "Distinctive image features from scale-invariant keypoints", *Int. J. Computer Vision (IJCV)* 60 (2), 91–110 (2004).

[38] S. Zhu and K.-K. Ma, "A new diamond search algorithm for fast block-matching motion estimation", *IEEE Trans. on Image Processing* 9 (2), 287–290 (2000).

[39] B.D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision (DARPA)", *Proc. 1981 DARPA Image Understanding Workshop* 1, 121–130, April 1981.

[40] Y. sheng Chen, Y. ping Hung, and C. shann Fuh, "Fast block matching algorithm based on the winner-update strategy", *IEEE Trans. on Image Processing* 10, 1212–1222 (2001).

[41] B. Kitt, B. Ranft, and H. Lategahn, "Block-matching based optical flow estimation with reduced search space based on geometric constraints", *Intelligent Transportation Systems* 1, 1104–1109 (2010).

[42] S. Indu, M. Gupta, and A. Bhattacharyya, "Vehicle tracking and speed estimation using optical flow method", *Int. J. Engineering Science and Technology* 3 (1), 429–434 (2011).

[43] E.L. Andrade, S. Blunsden, and R.B. Fisher, "Hidden markov models for optical flow analysis in crowds", *Proc. 18th Int. Conf. on Pattern Recognition – Volume 01*, ICPR '06, 1, 460–463 (2006).

[44] L. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition", *Proc. IEEE* 77 (2), 257–286 (1989).

[45] Z. Kalal, K. Mikolajczyk, and J. Matas, "Face-TLD: Tracking-Learning-Detection applied to faces", *IEEE Int. Conf. on Image Processing* 1, CD-ROM (2010).

[46] F. Cupillard, A. Avanzi, F. Bremond, and M. Thonnat, "Video understanding for metro surveillance, networking, sensing and control", *IEEE Int. Conf. on Networking, Sensing and Control* 1, 186–191 (2004).

[47] N. Ihaddadene and C. Djeraba, "Real-time crowd motion analysis", *Proc. 19th Int. Conf. on Pattern Recognition (ICPR '08)* 1, CD-ROM (2008).

[48] C. Tomasi and T. Kanade, "Detection and tracking of point features", *Technical Report CMU-CS-91-132*, pp. 383–394, Carnegie Mellon University, Pittsburgh, 1991.

[49] J.M. Ready and C.N. Taylor, "GPU acceleration of real-time feature based algorithms", *Proc. IEEE Workshop on Motion and Video Computing*, WMVC '07 (1), 8–9 (2007).

[50] N. Sundaram, T. Brox, and K. Keutzer, "Dense point trajectories by GPU-accelerated large displacement optical flow", *Tech. Rep. UCB/EECS-2010-104*, University of California, Berkeley, 2010.