# A GPU-based implementation of Motion Detection from a Moving Platform

Qian Yu, and Gérard Medioni
Institute for Robotics and Intelligent Systems
University of Southern California
{qianyu, medioni}@usc.edu

## Abstract

*We describe a GPU-based implementation of motion detection from a moving platform. Motion detection from a moving platform is inherently difficult as the moving camera induces 2D motion field in the entire image. A step compensating for camera motion is required prior to estimating of the background model. Due to inevitable registration errors, the background model is estimated according to a sliding window of frames to avoid the case where erroneous registration influences the quality of the detection for the whole sequence. However, this approach involves several characteristics that put a heavy burden on real-time CPU implementation. We exploit GPU to achieve significant acceleration over standard CPU implementations. Our GPU-based implementation can build the background model and detect motion regions at around 18 fps on 320×240 videos that are captured for a moving camera.*

## 1. Introduction

Motion detection is a fundamental issue in computer vision and has many applications in surveillance, video compression *etc*. The goal of motion detection is to segment the video frames into static background and a number of foreground regions. Motion detection from a stationary camera has been explored extensively and is regarded as a computationally efficient solution for many applications. However, to achieve the same goal from a moving platform is still computationally demanding.

The main difference between motion detection from a stationary and a moving camera, is the creation of the background model. In a stationary camera, variations in the image sequence can be modeled at the pixel level, and allow the definition of a background model for each pixel using statistic based techniques [5]. This concept can be extended to non-stationary cameras by compensating for the camera motion prior to estimating of the background model. We assume the camera motion can be approximately compensated by a 2D parametric transformation. Throughout this paper, we use a 3x3 homography to represent this 2D parametric transformation. This assumption is exact for PTZ (Pan-Tilt-Zoom)cameras and is a good approximation where the scene is far from the optic center and the depth is small compared to the distance, for example airborne image sequences. Given the transformation between any two frames, a sequence of frames can be warped to a reference frame. In the warped frames, the camera motion is stabilized relative to the reference frame. Then the background model can be obtained by collecting good statistics among the appearance of each 2D location, in the same way as motion detection from a stationary camera.

In such a process, due to the inevitable errors and outliers in the estimation of the 2D transformations, selecting a fixed frame as a reference frame, such as the first frame, accumulates errors and therefore cause erroneous background models. We adopt Kang *et al*.'s sliding window based method [9], where the center of the sliding window is selected as the reference frame and the other frames are warped to the reference frame using the pair-wise 2D homographies. This sliding window approach reduces the impact of an erroneous registration: an erroneous registration will not influence the quality of the detection for the whole sequence, but only among the frames considered in the sliding window.

This approach exhibits robustness to registration errors at the price of high computational cost. As the reference frame changes when the sliding window moves, all the warped images have to be re-computed relative to the new reference frame. This puts a heavy burden on the CPU to warp all frames in a sliding window (in most cases, we use 91 frames as a sliding window). However, this warping can be highly optimized by GPU texture coordinates generation. Moreover, instead of processing each pixel sequentially, GPU computation is designed to independently process streams of vertices and fragments in parallel. For example, NVIDIA Quadro FX 3500, the graphics card we use, contains 12 streaming processors for fragment processing. The loop accessing sequentially each pixel in the CPU implementation is replaced with pipelined processing in the

978-1-4244-2340-8/08/$25.00 ©2008 IEEE

GPU implementation.

We present here a GPU-based implementation to segment motion regions from a moving platform. The sliding window based method is implemented using the OpenGL graphics library and the Cg shading language. Our GPU implementation achieves significant speedup (10-15 times) over the CPU implementation. Note that, in this paper, we do not cover the process of registering two frames by extracting/matching 2D features and estimating the homography. This also can be efficiently achieved using GPU, as the known work in Sift-GPU, KLT-GPU [11] and OpenVidia[4]. Also, the central purpose of this method is not for video compositing or image stitching, such as in [12] and its GPU implementation in OpenVidia, but for collecting the statistics of the input sequences to build a background model at each time instant. We take input frames and the homography as input and the output of is the motion mask and/or the background image, which can be further organized for other purposes, *e.g.* moving object tracking *etc*.

The rest of this paper is organized as follows. In Section 2, we present the related work in motion detection and relevant GPU implementations. In Section 3, we discuss the sliding window based approach for motion detection from a moving platform. In Section 4, we present the GPU-implementation of this approach. Experiments and comparison are shown in Section 5, followed by summary and discussion of future work in Section 6.

## 2. Related work

Many background modeling methods have been proposed: some simple methods using single statistic as mean, median and mode and more sophisticated methods using a single or mixture of Gaussian model [8]. A comprehensive survey of motion detection algorithms can be found in [5]. However, most of the methods aim to detect motion from a stationary camera. To detect motion from a moving platform, we need to compensate the camera motion prior to estimating the background model. Moreover, for a moving camera, the background area may appear for a limited duration, therefore the number of samples for estimating a background model is therefore limited. A complicated background model may not be learned stably with few samples. Also, the inevitable errors in motion compensation will lead to erroneous background modeling. The method proposed in [9] addresses these issues by using a sliding window. This method considers per-pixel background model without using any temporal and spatial smoothness. This method can be combined with the methods that apply temporal and spatial smoothness [13, 14]. Mester *et al*. uses a colinearity criterion [13] to determine whether a pixel is a background and uses MRF (Markov Random Field) to represent spatial smoothness from neighbors. The method in [14] incorporates both spatial and temporal smoothness by using 3D Belief Propagation. Both of these methods require an initial background model and refine it using smoothness constraints. The assumption of using 2D parametric transformations may not be perfectly satisfied when the depth in the scene cannot be ignored compared with the object distance to the optical center. Parallax filtering techniques[15] can be applied. However, the background modeling is still required as a pre-processing [15]

Recently, GPGPU (General Purpose GPU) framework has been successfully applied in many computer vision problems to achieve high performance computation, for example, real-time stereo [10, 7], feature extraction and matching [11], foreground segmentation [6]. Some open-source GPU based libraries, such as OpenVidia [4] and MiniGPU [3], have come into the computer vision community. The video Orbits algorithm [12] has been implemented on GPU and parallel GPU in OpenVidia with real-time performance. The processes of feature extraction, feature matching and parametric transformation estimation by RANSAC, which are involved in motion compensation, have GPU-based implementations as well, such as in [11, 4].

## 3. Approach

We approximate the transformation between two images by a 2D parametric transformation. This assumption is exact for PTZ cameras and is a good approximation for telephoto lens from a long distance, where scene depth is much smaller than the distance between the object and the camera, as in UAV scenarios. We use $H_{i,i+1}$ to represent the homography between two consecutive frames, namely

$$I_{i+1} = H_{i,i+1} I_i \tag{1}$$

$H_{i(i+1)}$ can be estimated using RANSAC [16] to align feature points between frames. Registering any two frames is performed by concatenating the estimated pair-wise transforms,

$$H_{i,j} = \begin{cases} \prod_{k=i}^{j-1} H_{k,k+1} & i < j \\ I & i = j \\ (H_{j,i})^{-1} & i > j \end{cases} \tag{2}$$

In the rest of this paper, we assume $H_{i,j}$ between any two frames is known after the image registration process.

Motion compensation is achieved by warping a sequence of frames to a reference frame. The 2D image motion caused by camera movement in warped images is stabilized relative to the reference frame. To avoid accumulated errors, we adopt Kang *et al*.'s sliding window based method [9], where a number of frames are warped to a reference frame. The center frame of the sliding window is used as the reference frame. This reduces the accumulated errors by half in terms of the length of the sliding window. Also,

(a) $1^{st}$ frame (b) $25^{th}$ frame (c) $46^{th}$ frame (reference) (d) $65^{th}$ frame (e) $91^{st}$ frame

Figure 2. The warped images in the sliding window

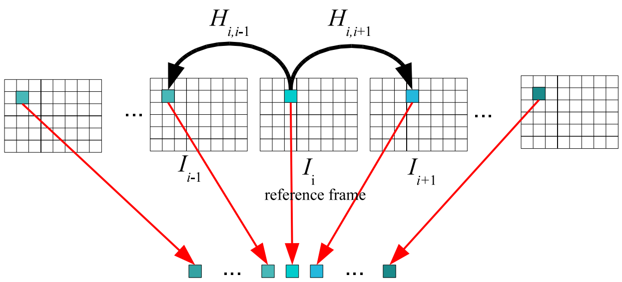



Figure 1. Illustration of the sliding window

even a very wrong registration result will not influence the motion detection in the entire sequence, but only within the number of frames considered in the sliding window. After the process of motion compensation, the decision is made according to the statistics on the corresponding pixels. Suppose the size of the sliding window is $w$, the background model for the reference frame $I_r$ is

$$I_{bg} = f(H_{r-w/2,r}I_r, \ldots, I_r, \ldots, H_{r+w/2,r}I_r) \quad (3)$$

Note that, to compute the background model of the first $w/2$ frames and the last $w/2$ in a sequence, the sliding window does not move. The statistic function $f$ in Eq.3 can be the means, the median, or the mode (the value that has the largest number of observations). Figure 1 illustrates the correspondence of pixel locations prior to the characterisation of the background model. Figure 2 shows warped images and the reference image. From Figure 2, we can see that the 2D image motion induced by the moving camera is compensated.

This method has been demonstrated to be robust to registration errors and outliers. The sliding window restricts the spread of the registration errors, but makes this method quite computationally demanding. As the reference frame changes when the sliding window moves, all the warped images have to be recomputed. This involves many float-precision interpolation operations, which puts a heavy burden for CPU computation, such as a 91-frame sliding window.

## 4. GPU based implementation

In the GPGPU framework, the fully programmable vertex and fragment processors provides powerful computational tools for general purpose computations. In order to implement an algorithm on the GPU, different computational steps are often mapped to different vertex and fragment shaders. Vertex shaders allow us to manipulate the data that describes a vertex. Fragment shaders serve to manipulate a pixel. We present a GPU implementation of the sliding window based method and separate the process into two steps, warping images and computing the background model. Also, we want to minimize memory transferring between GPU and CPU, therefore the inputs for our implementation are the sequential image data and homography transformation, and the output is the motion mask (namely the difference image) for each frame. The overview structure of the implementation is shown in Figure 3.

Our implementation needs to store all frames in the sliding window. The sequential frames are stored as two-dimension textures in the GPU memory. The most recent frame is loaded as a texture and the oldest frame is moved out of the texture pool. The warping involves changing the texture coordinates and is implemented in the vertex profile. The vertex profile takes the 3x3 homograph matrix as input and outputs the warped texture coordinates by simple matrix multiplication operations. The transformed texture coordinates are applied in the fragment profile.

To compute the background model, different statistic functions can be applied in Eq.3, such as the mode, median and the mean of the samples in the sliding window. For motion detection from a moving platform, the mode is usually preferred to the other two. This is because we have very limited samples, *i.e.*, the size of the sliding window is quite small. The mean and the median, which do not differentiate the foreground and background pixels, will lead to

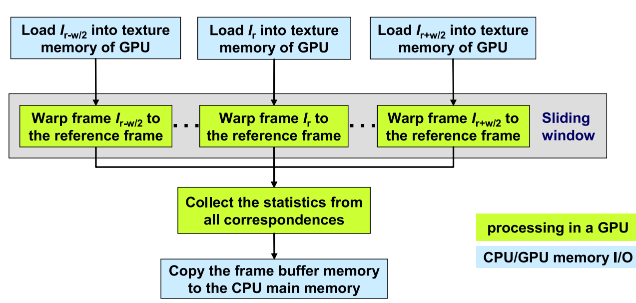


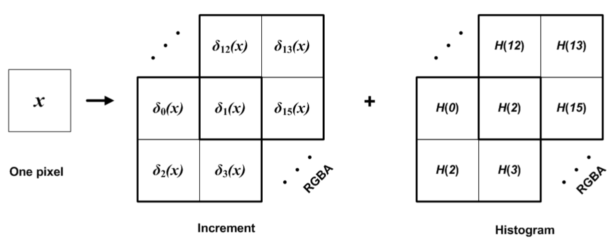

Figure 3. Overview of the GPU-based implementation

a biased background model. The mean and the median are only proper when enough samples are available. The computation of the mode requires to build a histogram and then to find the bin with the largest number of samples (there is another way to establish a dynamic histogram by constructing a binary search tree, which involves too many branching operations and dynamic data structures, thus it is not proper for GPU implementation).

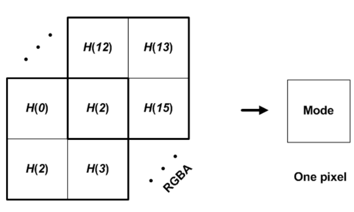
**Algorithm 1** Overview of GPU implementation

**input**: $(I_1, \ldots, I_w)$ , $(H_{1,r}, \ldots, H_{w,r})$
**output**: Difference image $I_{diff}$
**for** Each frame $I_r$ in the sliding window **do**
  1. Generate texture coordinates according to $H_{i,r}$
  2. Build a histogram for each pixel
**end for**
Compute the mode by reduction and use the average of mode bin as the background intensity.

One histogram is built for each location in the reference frame and each bin records the number of hits over the sliding window. This is different from the normal concept of a normal histogram that is built on the whole image. The overview of the GPU implementation is shown in Algorithm 1. We use a fixed number $n$ of bins to construct a histogram. By using a RGBA texture, a histogram in each




(a) construct a histogram: a 16-bin histogram is built in one RGBA texture with the doubled width and height.




(b) compute the mode: the mode is computed among all bins in the RGBA histogram texture.

Figure 4. Procedure to compute the mode

channel of $n$ bins leads to a tile size of $\sqrt{n/4} \times \sqrt{n/4}$ texture elements. For each pixel, we need to construct such a histogram, therefore the size of the RGBA texture is $W\sqrt{n/4} \times H\sqrt{n/4}$, where $W$ and $H$ are the width and the height of original images. Suppose $n = 16$ (this is enough for an eight bit dynamic range), the histogram texture memory is 4 times as large as the original one. In a RGBA texture with $n = 16$, $(H(0), H(4), H(8), H(12))$ is stored in a float4 vector in GPU with the same texture coordinates. After one frame texture is loaded, the hits in each bin are updated. A bin is indexed by an intensity value, *e.g*. between [0,15] for $n = 16$. It is not efficient to use "if-else" type statements to determine the placement of one intensity value in the histogram. For efficiency, it is preferable to use standard *Cg* functions [1] rather than using branching statements. We adopt the approach in [4], which uses the function $\delta_b(x) = \cos^2(\alpha(x - b))$ to indicate whether the value $x$ belongs to the bin $b$ and $\alpha$ is used to keep $(x - b)$ within $(-\pi/2, \pi/2)$. The $\cos(\cdot)$ function can be squared repeatedly or filtered by a floor operation to yield a more concentrated impulse. To find the mode of the histogram, we use the reduction operation as in [17]. The required number of reduction iterations is $\log_2(\sqrt{n/4})$. The procedure of computing the mode is illustrated in Figure 4, where each grid corresponds to on bin. Updating the histogram is implemented in the "ping-pong" way. After we find the mode, *i.e*. the bin with the largest number of samples, we average the samples that are located in the mode bin as the background model. This refinement is necessary to avoid the quantization error in building histograms. The difference between the background model and the reference image is the output that is transferred to CPU. For color videos, we compute the background model independently for each channel.

The approach that uses the mode as the background is independent of the result at the previous time, thus it is able avoid the spread of registration errors and outliers. However, the computation of the mode is still complicated. When the registration quality is good enough, we can use an alternative to compute the background model. It is obvious that computing the mean is much easier for GPU implementation. We first warp the background result at the previous time to the current reference frame as an estimated background model, and use the mean of the samples that are close enough to the estimated model as the new background model. If the number of the samples that are close enough to the estimated background is too small, we use the average of all samples instead. This method may inherit the errors in the previous estimate.

## 5. Experiments

The experiments are performed on a workstation with Intel Xeon Dual Core CPU 3GHz, 4G RAM and NVIDIA Quadro FX 3500. Both CPU and GPU versions take the same input, namely original frames and homographies. The CPU version uses Intel IPL 2.5 (Image Processing Library) to perform warping with linear interpolation (the interpolation method affects timing). For the GPU version, RGBA floating point textures are used for storing the frame textures on the GPU. This is supported on most GPUs profiles. For color videos, we use three RGBA textures to compute the background model in RGB channels in parallel. Most modern GPUs support four texture attachments. For both CPU and GPU version, 16 bins are used in constructing histograms. Before background modeling, we use KLT features [2] with sub-pixel SSD matching and use RANSAC to compute homographies. All time measures only focus on background modeling, exclude loading images from videos, extracting features and estimating homography.

We first evaluate the quality of the background model computed by GPU. This evaluation is in general difficult to perform as it would require ground truth background models. Hence, we compare the background model output by GPU with the one output by the CPU version. The average difference of 1000 frame GPU and CPU outputs of two methods (mean and mode) $\frac{2abs(I_{GPU}-I_{CPU})}{I_{GPU}+I_{CPU}}$ is 0.3% and the variance is 0.1%. Thus, we can regard that GPU and CPU versions are providing the same results. Some of the background model results are shown in Figure 5. We show a mosaic view of many background models over time to demonstrate the quality of background models in Figure 5(c). When we generate the mosaic, we simply overwrite the mosaic view with new frames without any blending operations and the mosaicing is implemented on a CPU. From the mosaic view, we can see the quality of the background model is consistently good over time.

We evaluation the speedup that the GPU version achieves over the CPU implementation, with different size of sliding windows and at different resolutions. We evaluate both approaches of the mode and the mean. The timing comparison between the CPU and GPU implementations is shown in Figure 6. The timing of both methods is computed by averaging of multiple runs. From Figure 6, we can see the time performance is basically proportional to the image size and the sliding window size. A larger sliding window provides more samples and generates a better background model (we usually use 91 frames in a sliding window). The GPU version using the adaptive mean as the background model with 91 frame sliding window can run at around 18 fps on 320x240 resolution videos. The mode approach can run at 10 fps with the same setting. The GPU version of the mean approach achieves around 12 speedup over its standard CPU counterpart. The GPU version of the mode approach achieves around 15 speedup.

# 6. Summary and future work

We have presented a GPU-based implementation of motion detection from a moving platform. Our GPU implementation fully exploits the parallelism of a GPU computational power. For both mean and mode approaches, a significant speedup is achieved by the GPU version over its standard CPU counterpart. We plan to test this implementation on various GPU graphics cards, for example, the newer GeForce 8800 GTX or Ultra. We expect it will run much faster on new generation graphics cards. We also expect to implement this approach using CUDA to compare with this Cg based GPU implementation. In the future, we will incorporate GPU implementations of feature extraction, matching and RANSAC with the background modeling implementation. Also, we plan to investigate the use of spatial and temporal smoothness for further foreground segmentation, such as in [6] and [14].


# Acknowledgement

This work was supported by grants from MURI-ARO W911NF-06-1-0094.


# References


[1] http://developer.nvidia.com/object/cg_toolkit.html.

[2] J. Shi and C. Tomas. Good features to track. In *CVPR*, pages 593–600, 1994.

[3] P. Babenko and M. Shah. Mingpu: A minimum gpu library for computer vision. http://server.cs.ucf.edu/ vision/MinGPU/.

[4] C. A. James Fung, Steve Mann. Openvidia: Parallel gpu computer vision. In *Proceedings of the ACM Multimedia 2005*, pages 849–852, 2005.

[5] R. J. Radke et al. Image change detection algorithms: A systematic survey. *IEEE Transactions on Image Processing*, pages 294–307, 2005.

[6] A. Griesser et al. Real-time, gpu-based foreground-background segmentation. In *Vision, Modeling, and Visualization*, pages 319–326, 2005.

[7] R. K. Patrick Labatut and J.-P. Pons. A gpu implementation of level set multiview stereo. In *International Conference on Computational Science*, pages 212–219, 2006.

[8] C. Stauffer and W. Grimson. Adaptive background mixture models for real-time tracking. In *CVPR*, pages 246–252, 1999.

[9] J. Kang, I. Cohen, and G. Medioni. Continuous tracking within and across camera streams. In *CVPR*, volume 1, pages 267–272, Jun 2003.

[10] R. Yang and M. Pollefeys. Multi-resolution real-time stereo on commodity graphics hardware. In *CVPR*, pages 211–217, 2003.

[11] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc. Gpu-based video feature tracking and matching. Technical report, Department of Computer Science, UNC Chapel Hill, 2006.

[12] S. Mann and R. W. Picard. Video orbits of the projective group a simple approach to featureless estimation of parameters. *IEEE Transactions on Image Processing*, pages 1281–1295, 1997.

[13] R. Mester, T. Aach, and L. Dumbgen. Illumination-invariant change detection using a statistical colinearity criterion. In *Proc. 23rd DAGM Symp*, pages 170–177, 2001.

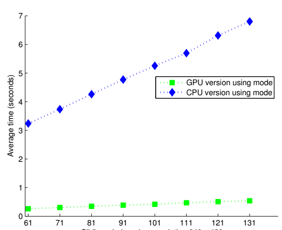
(a) Reference frame

(b) Background model at the reference frame

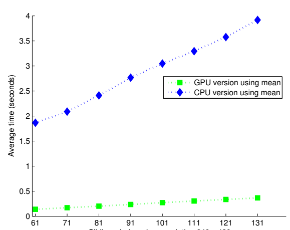
(c) Motion mask at the reference frame

(d) Mosaic view of the background models over time

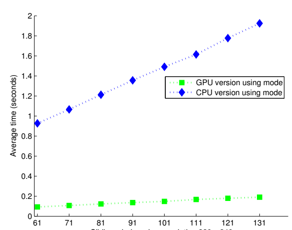
Figure 5. Visual results of background modeling using the mode approach



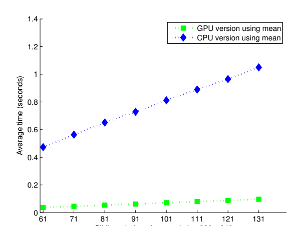

(a) mode background models mode on 640x480 videos




(b) mean background models mode on 640x480 videos




(c) mode background models mode on 320x240 videos




(d) mean background models mode on 320x240 videos

Figure 6. GPU Timings compared with its CPU counterpart for a range of sliding window size and different resolution

[14] Z. Yin and R. Collins. Belief propagation in a 3d spatio-temporal mrf for moving object detection. In *CVPR*, 2007.

[15] C. Yuan, G. Medioni, J. Kang, and I. Cohen. Detecting motion regions in presence of strong parallax from a moving camera by multi-view geometric constraints. In *PAMI*, volume 29, pages 1627–1641, 2007.

[16] M. Fisher and R. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. In *Comm. Assoc. Comp.*, volume 24(6), pages 381–39, 1981.

[17] J. Kruger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *International Conference on Computer Graphics and Interactive Techniques*, 2003.