Machine Vision and Applications
DOI 10.1007/s00138-007-0105-z

ORIGINAL PAPER

Feature tracking and matching in video using programmable graphics hardware

Sudipta N. Sinha · Jan-Michael Frahm · Marc Pollefeys · Yakup Genc

Received: 16 July 2006 / Accepted: 28 March 2007

Abstract This paper describes novel implementations of the KLT feature tracking and SIFT feature extraction algorithms that run on the graphics processing unit (GPU) and is suitable for video analysis in real-time vision systems. While significant acceleration over standard CPU implementations is obtained by exploiting parallelism provided by modern programmable graphics hardware, the CPU is freed up to run other computations in parallel. Our GPU-based KLT implementation tracks about a thousand features in real-time at 30 Hz on 1,024 × 768 resolution video which is a 20 times improvement over the CPU. The GPU-based SIFT implementation extracts about 800 features from 640 × 480 video at 10 Hz which is approximately 10 times faster than an optimized CPU implementation.

Keywords Visual tracking · Vehicle tracking · Video surveillance · Visual inspection · Vision system · Robot navigation

1 Introduction

Extraction and matching of salient 2D feature points in video is important in many computer vision tasks like object detection, recognition, structure from motion and marker-less augmented reality. While certain sequential tasks like structure from motion for video [18] require online feature point tracking, others need features to be extracted and matched across frames separated in time (eg. wide-baseline stereo). The increasing programmability and computational power of the graphics processing unit (GPU) present in modern graphics hardware provides great scope for acceleration of computer vision algorithms which can be parallelized [3,11,12,14–17]. GPUs have been evolving faster than CPUs (transistor count doubling every few months, a rate much higher than predicted by Moore's Law), a trend that is expected to continue in the near future. While dedicated special-purpose hardware or reconfigurable hardware can be used for speeding up vision algorithms [1,2], GPUs provide a much more attractive alternative since they are affordable and easily available within most modern computers. Moreover with every new generation of graphics cards, a GPU implementation just gets faster.

In this paper we present GPU-KLT, a GPU-based implementation for the popular KLT feature tracker [6,7] and GPU-SIFT, a GPU-based implementation for the SIFT feature extraction algorithm [10]. Our implementations are 10–20 times faster than the corresponding optimized CPU counterparts and enable real-time processing of high resolution video. Both GPU-KLT and GPU-SIFT have been implemented using the OpenGL graphics library and the Cg shading language and tested on modern graphics hardware platforms. As an application, the GPU-KLT tracker has been used to track 2D feature points in high-resolution video streams within a vision based large-scale urban 3D modeling system described in [19].

Our work is of broad interest to the computer vision, image processing and medical imaging community since many of the key steps in KLT and SIFT are shared by other algorithms, which can also be accelerated on the GPU. Some of these are (a) image filtering and separable convolution, (b) Gaussian

S. N. Sinha (✉) · J.-M. Frahm · M. Pollefeys
Department of Computer Science, CB# 3175 Sitterson Hall,
University of North Carolina at Chapel Hill,
Chapel Hill, NC 27599, USA
e-mail: ssinha@cs.unc.edu

Y. Genc
Real-time Vision and Modeling Department,
Siemens Corporate Research, 755 College Road East,
Princeton, NJ 08540, USA

scale-space construction, (c) non-maximal suppression, (d) structure tensor computation, (e) thresholding a scalar field and (f) re-sampling discrete 2D and 3D scalar volumes. This paper is organized as follows. Section 2 describes the basic computational model for general purpose computations on GPUs (GPGPU). Section 3 presents the basic KLT algorithm followed by its GPU-based implementation and experiments on real video and an analysis of the results obtained. Section 4 describes similar aspects of GPU-SIFT. Finally we present our conclusions in Sect. 5.

2 GPGPU concepts

Modern programmable graphics hardware contains powerful coprocessors (GPUs) with a peak performance of hundreds of GFLOPS which is an order of magnitude higher than that of CPUs [21]. They are designed to independently process streams of vertices and fragments (pixels) in parallel. However, their data parallel single instruction multiple data (SIMD) architecture also provides an abstraction for performing general purpose computations on GPUs (GPGPU) and for treating the GPU as a stream processor.

In the GPGPU framework, the fully programmable vertex and fragment processors perform the role of the computational kernels while video memory (frame-buffers, textures, etc.) provides it with a memory model (see Fig. 1 for an overview of the graphics pipeline implemented in hardware). Texture mapping on the GPU is analogous to the CPU's random read-only memory interface while the ability to render directly into texture (off-screen rendering) provides a memory-write mechanism. However, by virtue of its specialized design, the GPU has a more restricted memory model when compared to a CPU (scatter operations i.e. random memory writes are not allowed). Texture memory caches are designed for speed and prevent concurrent read and write into the same memory address. Thus distinct read and write textures must be used. They can be swapped after each render pass making the write texture available as input and vice versa (ping-pong rendering).

In order to implement an algorithm on the GPU, different computational steps are often mapped to different fragment programs. For each computational step, the appropriate fragment program is bound to the fragment processor and a render operation is invoked. The rasterization engine generates a stream of fragments and also provides a fast way of interpolating numbers in graphics hardware. Most GPGPU applications execute multiple fragment programs in a series of successive off-screen rendering passes. While pixel-buffers (pBuffers) exist on older graphics cards, recently frame-buffer objects (FBOs) were introduced, providing a simple and efficient off-screen rendering mechanism in OpenGL. Details about GPGPU programming are available in [20,22].

Many computer vision algorithms map well into this parallel stream processing model. Image processing tasks which can process multiple pixels independently (eg. convolution) can be performed very fast by fragment programs (computation kernels) exploiting the high parallelism provided by multiple fragment pipes (upto 24 in modern cards). A large fraction of the GFLOPS dedicated to texture mapping in GPUs is non-programmable. While image processing applications can sometimes leverage this by using the bilinear interpolation of texture mapping, they also benefit from the 2D texture cache layouts designed for fast texture mapping.

Recently there has been a growing interest in the computer vision community to solve important computationally expensive problems like image registration [16], stereo and segmentation using graphics hardware. A correlation-based real-time stereo algorithm for the GPU was first proposed by [11] while more complex formulation of stereo [12–14] were implemented more recently. GPUs have been successfully used by [15,17] to accelerate background segmentation in video, often used as a first step in many vision applications. A versatile framework for programming GPU-based computer vision tasks (radial undistortion, image stitching, corner detection, etc.) was recently introduced by [3,4] and real-time GPU-based image processing was evaluated by [5] under various conditions.

3 KLT tracking on GPU

3.1 The algorithm

The KLT tracking algorithm [6,7] computes displacement of features or interest points between consecutive video frames when the image brightness constancy constraint is satisfied and image motion is fairly small. Assuming a local translational model between subsequent video frames, the displacement of a feature is computed using Newton's method to minimize the sum of squared distances (SSD) within a tracking window around the feature position in the two images.

Let $I(*, *, t)$ represent the video frame at time t. If the displacement of an image point (x, y) between time t and $t + \Delta t$, denoted by $(\Delta x, \Delta y)$ is small, then according to the brightness constancy constraint,

$$I(x, y, t + \Delta t) = I(x + \Delta x, y + \Delta y, t)$$

Let $\mathbf{x} = (x, y)^T$ and $\mathbf{v} = (\Delta x, \Delta y)^T$. In the presence of image noise r,

$$I(\mathbf{x}, t + \Delta t) = I(\mathbf{x} + \mathbf{d}, t) + r$$

KLT will compute the displacement vector $\mathbf{d}$ that minimizes the following error

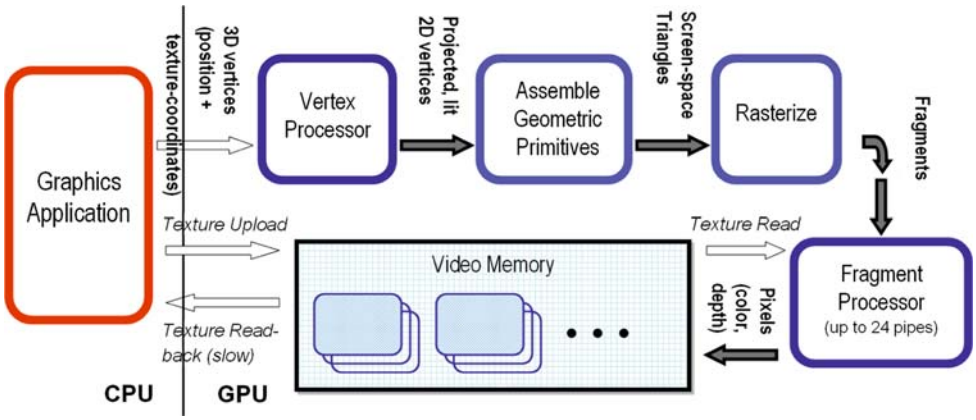

Fig. 1 Overview of the 3D Graphics Pipeline. The fragment processor and direct off-screen rendering capability is frequently used in GPGPU applications

KLT Tracking (*ft_list*, F_0, F_1) {

(1) Build-Pyramid: builds multi-resolution intensity and gradient pyramid from images F_0, F_1

(2) Track:

For all pyramid levels from coarse to fine

For multiple iterations

For each feature *f* in *ft_list*

compute coefficients of **A** and **b** as shown in Equation 1.

solve **A d = b**

evaluate **d** and update track of feature

}

Re-select-Features (*ft_list*) {

mask = mask_out_region (*ft_list*)

c_map = evaluate corner-ness measure *c* over whole image

// Perform non-maximal suppression

pts = find_features (*#max_feats*, *mask*, sort (*c_map*))

add_new_features (*ft_list*, *pts*)

}

Fig. 2 Pseudo-code for the two fundamental routines in the KLT Tracking algorithm

$$r = \sum_W (I(\mathbf{x}+\mathbf{d}, t) - I(\mathbf{x}, t+\Delta t))^2$$

over a small image patch W. Approximating $I(\mathbf{x}+\mathbf{d}, t)$ by its Taylor expansion, one obtains the following linear system to estimate the unknown $\mathbf{d}$ where $\mathbf{G} = [\frac{\partial I}{\partial x} \ \frac{\partial I}{\partial y}]$ is the image gradient vector at position $\mathbf{x}$.

$$\left(\underbrace{\sum_W \mathbf{G}^T\mathbf{G}}_{\mathbf{A}}\right)(\mathbf{d}) = \underbrace{\sum_W \mathbf{G}^T \, \Delta \, \mathbf{I}(\mathbf{x}, \Delta t)}_{\mathbf{b}} \qquad (1)$$

Tomasi later proposed a variation of the KLT equation which uses both images symmetrically. This equation, derived in [8] is identical to Eq. 1 except that here

$$\mathbf{G} = \left[\frac{\partial(I(*,t) + I(*,t+\Delta t))}{\partial x} \quad \frac{\partial(I(*,t) + I(*,t+\Delta t))}{\partial y}\right]$$

This symmetric version is used in our GPU implementation.

Feature to track are selected by finding image points where a saliency or corner-ness measure

$$\mathbf{c} = \min\left(\mathrm{eig}\left(\sum_W \left[\frac{\partial I}{\partial x} \ \frac{\partial I}{\partial y}\right]^T \left[\frac{\partial I}{\partial x} \ \frac{\partial I}{\partial y}\right]\right)\right)$$

(the minimum eigen-value of the 2×2 structure tensor matrix obtained from gradient vectors) is a local maximum. It is evaluated over the complete image [6,7] and a subsequent non-maximal suppression is performed. The KLT algorithm is described in Fig. 2.

Since the linearity assumption is only valid for a small displacement $\mathbf{d}$, a multi-resolution KLT tracker is often used in practice for handling larger image motion. It first tracks at coarse resolutions and then refines the result in finer resolutions. Multiple iterations are performed at each resolution for better accuracy. Due to camera motion and occlusion, features tracks are eventually lost; hence new features must be re-selected from time to time to maintain a roughly fixed number of features in the tracker.

3.2 GPU implementation details

Graphics processing unit-KLT maps various steps of the tracking algorithm to different fragment programs. Every video frame is uploaded to video memory where it is smoothed and its multi-resolution pyramid of image intensity and gradients is constructed. The tracking is done on every frame using the image pyramids corresponding to the current and previous frames. Feature re-selection is performed once in every k frames to keep a roughly constant feature count in the tracker. The value of k was set to 5 for all our experiments but this generally depends on camera motion and the number of lost features.

3.2.1 Implementation strategies

RGBA floating point textures were used for storage on the GPU. This is supported on most modern GPUs. Section 3.3 discusses the precision that was required by our implementation on different hardwares. The multi-resolution image pyramid and the associated gradient vectors are represented by a set of RGBA textures where different channels are used for the intensity and gradient magnitudes. A second set of identical image pyramid textures is needed during the construction of the image pyramid on the GPU (as explained below). The corner-ness map is represented by a pair of textures; one for partial sums and the second for the final values. The feature list table is represented by a $m \times n$ texture where m stands for the maximum feature count while n stands for (#tracking iterations) $\times$ (# pyramid levels). Three other texture units are used for computing and storing intermediate values computed during tracking and computing the elements of matrix **A** and vector **b** (refer Eq. 1).

3.2.2 Build-pyramid

The multi-resolution pyramid of the image intensity and its gradients are computed by a series of two-pass separable Gaussian convolutions performed in fragment programs. The fragment program uses OpenGL's multiple texture coordinates (TEXCOORD0 …TEXCOORD7) to read a row or column of pixels. The 1D convolution filter kernel size limited to 7, accounts for most practical values of σ. Since fragment programs support vector operations, the blurred pixel and the gradient magnitudes are computed simultaneously. The second set of textures are used to store the results of the row convolution pass and are subsequently read by the column convolution pass. Since the tracker requires information for only two video frames, textures for two image pyramids are allocated and a pointer indicating the current frame alternates between the two.

3.2.3 Track

KLT tracking performs a fixed number of tracking iterations at each image resolution starting with the coarsest pyramid level. Each tracking iteration constructs a linear system of equations in two unknowns for each interest point (see Eq. 1), $\mathbf{A}\,\mathbf{d} = \mathbf{b}$ and directly solves them to update the estimated displacement. This is done in four steps by four fragment programs on the GPU. First a fragment program bilinearly interpolates intensity and gradient magnitudes in 7×7 patches around each KLT feature in the two images and stores them in a temporary texture. While NVIDIA provides hardware support for bilinear interpolation of floating point textures, a fragment program is required to do this on ATI. Various quantities evaluated at 7×7 image blocks are added in two passes; first computing partial row sums followed by a single column sum. The second and third fragment program evaluates all the six elements of the matrix **A** and the vector **b** and writes them into a different texture for the next fragment program to use. Finally Eq. 1 is solved in closed form by the fourth fragment program which writes the currently tracked position into the next row in the feature table texture. The invocation of these four fragment programs corresponds to a single tracking iteration in the original algorithm (see Fig. 2).

At the end of (*#max-iterations*) $\times$ (*#pyramid-levels*) tracking iterations, the final feature positions (the last row in the feature table texture) are read back to the CPU along with two other values per feature - $\Delta\,\mathbf{d}$, the final tracking update of the iterative tracker and **res**, the SSD residual between each initial and tracked image patch. An inaccurate feature track is rejected when its $\Delta\,\mathbf{d}$ and **res** exceeds the respective thresholds. While KLT originally performs these tests after every tracking iteration, GPU-KLT skips them to avoid conditional statements in fragment programs for speed. This however, forces it to track all **N** (=*#max-features*) features for all the iterations. Hence GPU-KLT's running time depends on **N** and not the number of valid features being tracked.

GPU-KLT performs tracking completely on the GPU contrary to [3] who builds the matrices (Eq. 1) on the GPU using fragment programs, performs a readback and then solves a stacked linear system on the CPU. Multi-resolution, iterative tracking is ruled out in their case due to the CPU-GPU tranfer bottleneck. [3] also does not compare CPU and GPU implementations for accuracy and timings. Our multi-resolution, iterative tracker handles larger image motions than [3] and performs accurate tracking in real-time on high-resolution video (Fig. 3).

3.2.4 Re-select-features

The KLT corner-ness map is computed in two passes. The first pass computes the 2×2 structure tensor matrix at each pixel. The values in a 7×7 window centered at every pixel are

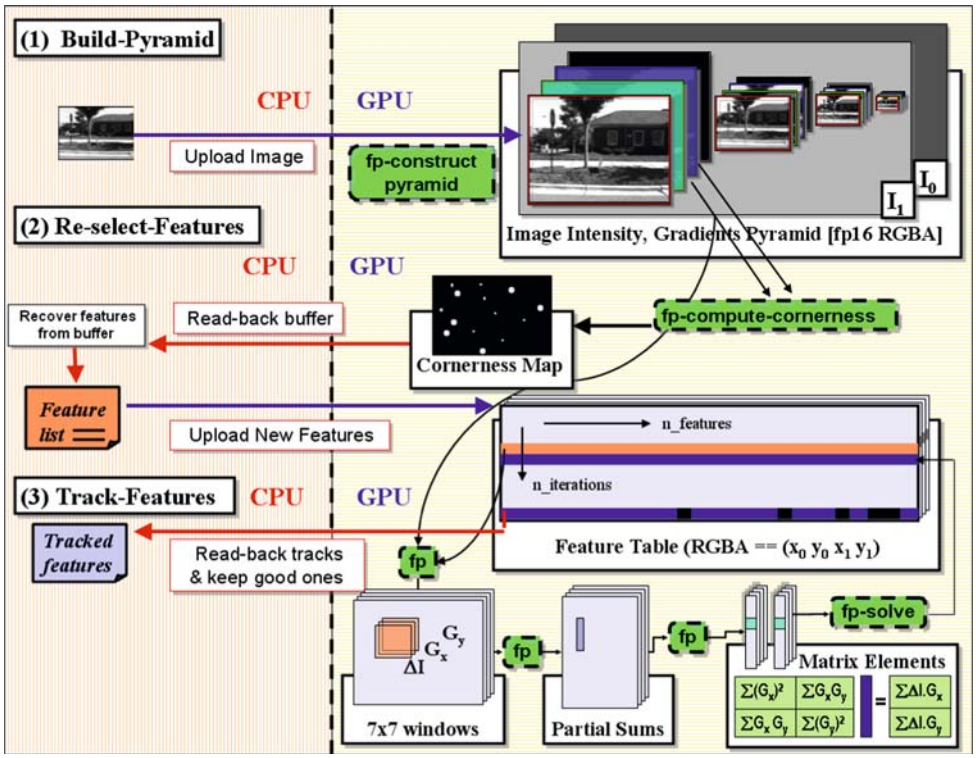

Fig. 3 Overview of the steps in the GPU-KLT implementation

added using partial row sums followed by a column sum. The minimum eigen value of the resulting matrix is stored in the corner-ness map. During feature re-selection, the neighborhood of existing features is invalidated and early Z-culling is used to avoid computations in these image regions. Early Z-culling works as follows. In the first pass, a binary mask is created by rendering $t \times t$ quads (where t is the desired minimum distance between features) for every existing valid feature. The depth test in the graphics pipeline is disabled while depth writing is enabled and this binary mask is loaded into the graphics hardware's depth buffer. In the next pass, depth writing is disabled while the depth test is enabled. With early Z-culling hardware support, fragments corresponding to invalidated pixels are not even generated when the corner-ness map is being computed. Finally a corner-ness map with sparse entries is read back to the CPU. Non-maximal suppression is done on it to find new additional features to track. Using the GPU for invalidating image regions before computing the corner-ness map makes this final step on the CPU much faster.

3.3 Results

To evaluate the performance of GPU-KLT, tests were performed on various ATI (850XT, 1,800XT, 1,900XT) and NVIDIA (7,800GTX, 7,900GTX) graphics cards. These tests showed an improvement of one order of magnitude in speed over a standard KLT implementation [9]. A 20× speedup over the CPU version was observed on a ATI 1900XT, where GPU-KLT tracks 1,000 features in 1,024 × 768 resolution video at 30 Hz. The performance measurements are shown in Fig. 4. The evaluation shows that currently all ATI graphic cards outperform the tested NVIDIA graphics cards. This is due to the precision required for solving Equation 1 within a fragment program. The required 32 bit floating point precision is always provided by ATI cards even when the storage textures have only 16 bit floating point precision. In contrast to ATI, NVIDIA cards could only provide 32 bit precision computations in the fragment programs when the allocated textures too had 32 bit precision. This increased the memory bandwidth during processing on NVIDIA cards and explains their lower speeds. Furthermore, the measurements in Fig. 4 show that GPU-KLT is bandwidth limited on all tested graphics cards and its computational complexity linearly depend on the number of features as well as on the number of pixels in the images.

Graphics processing unit-KLT was also tested qualitatively for tracking accuracy. This evaluation is in general difficult to perform as it would require ground truth tracks. To our knowledge there is no standard data set for such an evaluation. Hence we compared GPU-KLT and the standard KLT to each other. Due to the different orders of operations and the different ALU's in the GPU and the CPU the results are in general not equal. We tested the tracking inside an application for camera pose estimation [19] using the quality of the estimated camera poses as the criteria for tracking accuracy.

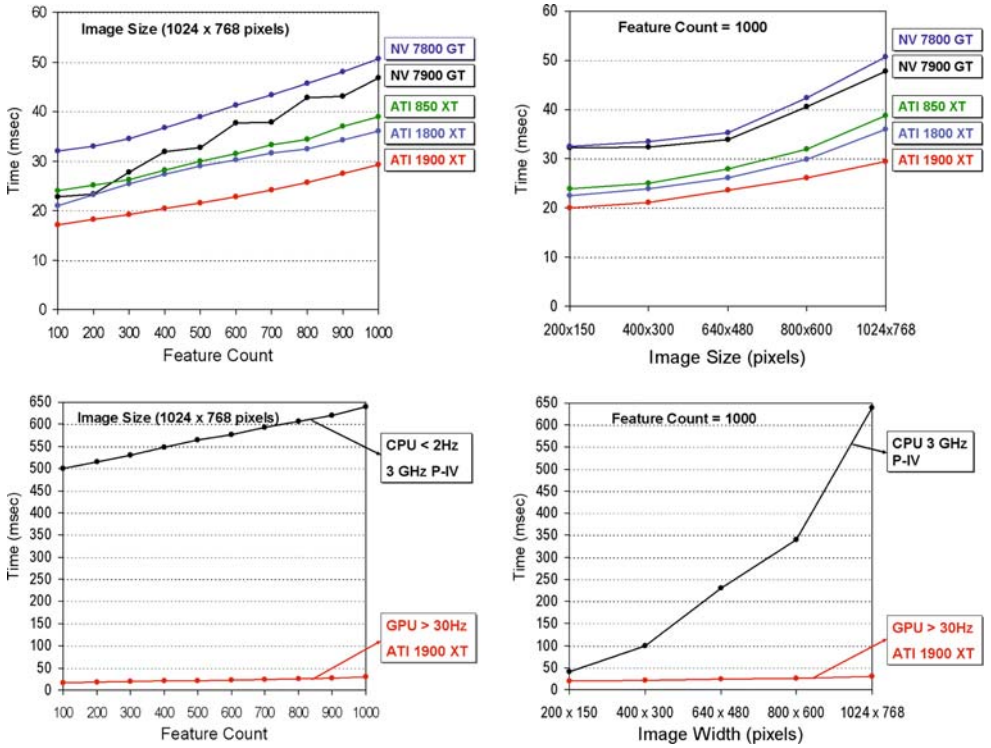

Fig. 4 GPU-KLT timings on various graphics cards (*Top*). A timing comparison of GPU-KLT with CPU implementations of KLT—(1) The OpenCV tracker (http://www.intel.com/technology/computing/opencv/) and (2) Birchfield's KLT library [9] ran at comparable speeds (CPU specs - 3.4 GHz PentiumD processor, 1 GB RAM) (*Below*)

It showed that both trackers provide in general the same quality of tracks. Thus we conclude that GPU-KLT provides an order of magnitude of speedup over CPU implementations while maintaining the same tracking quality. Tests were performed on a wide range of video (see Figs. 5, 6). Our open source implementation is available at http://www.cs.unc.edu/~ssinha/Research/GPU_KLT.

4 SIFT feature extraction on GPU

4.1 The algorithm

The scale invariant feature transform (SIFT) [10] algorithm is a popular candidate for extraction of interest points invariant to translation, rotation, scaling and illumination changes in images. It first constructs a Gaussian scale-space pyramid from the input image while also calculating the gradients and difference-of-gaussian (DOG) images at these scales. Interest points are detected at the local extremas within the DOG scale space. Once multiple keypoints have been detected at different scales, the image gradients in the local region around each feature point are encoded using orientation histograms and represented in the form of a rotationally invariant feature descriptor. The details are described in [10] (Fig. 7).

4.2 GPU implementation details

The construction of the Gaussian scale space pyramid is accelerated on the GPU using fragment programs for separable Gaussian convolution. The intensity image, gradients and the DOG values are stored in a RGBA texture and computed in the same pass using vector operations in fragment programs. Blending operations in graphics hardware are used to find local extremas in the DOG pyramid in parallel at all pixel locations. The Depth test and the Alpha test is used to threshold these keypoints; The local principal curvatures of the image intensity around the keypoint is inspected; this involves computing the ratio of eigenvalues of the 2×2 structure tensor matrix of the image intensity at that point. The keypoint locations are implicitly computed in image-sized, binary buffers, one for each scale in the pyramid. A fragment program compresses (a factor of 32) the binary bitmap into RGBA data, which is readback to the CPU and decoded there.

At this stage, a list of keypoints and their scales have been retrieved. Since reading back the gradient pyramid (stored in texture memory) to the CPU is expensive, the subsequent

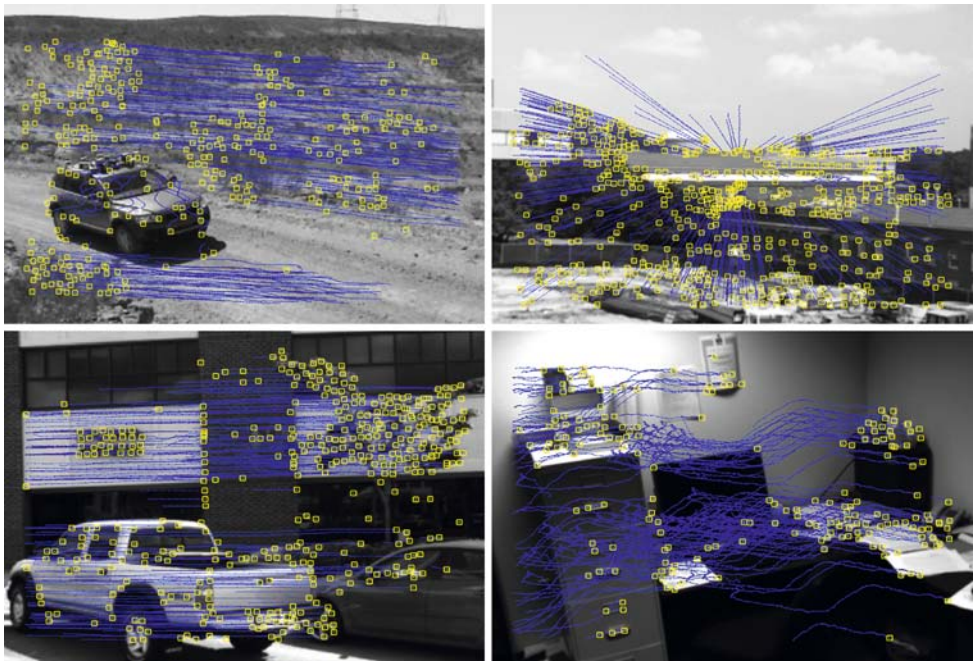

Fig. 5 Real-time tracking using GPU-KLT on (*Upper Left*) hand-held video, (*Upper Right*) a zoom sequence from a surveillance camera, (*Lower Left*) video from a camera mounted on a driving vehicle, (*Lower Right*) streaming video from a hand-held Firewire camera

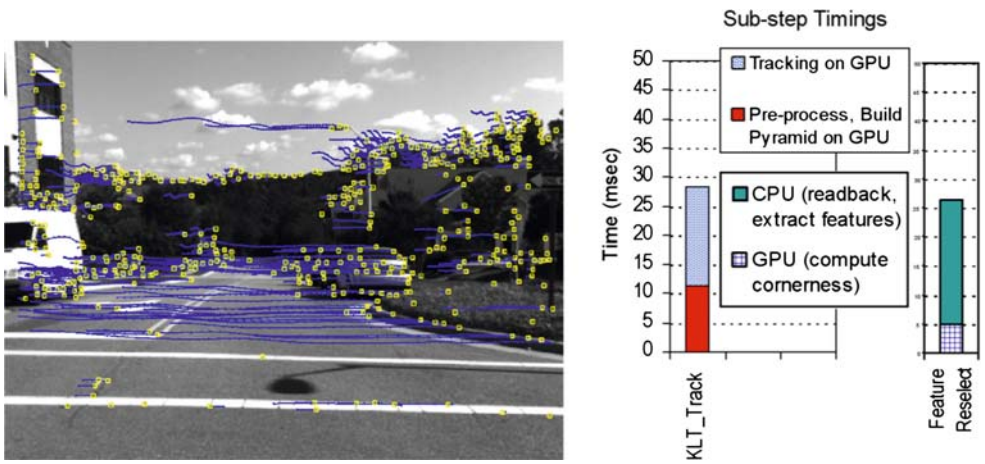

Fig. 6 GPU-KLT tracks at 30 Hz, upto 1,000 features in hi-res video captured by a camera on a driving vehicle (*Left*). Relative timings of the steps of GPU-KLT (*Right*)

steps in SIFT are also performed on the GPU. Gradient vectors near the keypoint location are Gaussian weighted and accumulated inside an orientation histogram by another fragment program. The orientation histogram is read back to the CPU, where its peaks are detected. Computing histograms on the GPU was found to be more expensive [3] than doing it on the CPU following a small readback. The final step involves computing 128 element SIFT descriptors. These consist of a set of orientation histograms built from 16×16 image patches in invariant local coordinates determined by the associated keypoint scale, location and orientation. SIFT descriptors cannot be efficiently computed completely on the GPU, as histogram bins must be blended to remove quantization noise. Hence we partition this step between the CPU and the GPU. We resample each feature's gradient vector patch, weight them using a Gaussian mask using blending support on the GPU. The resampled and weighted gradient vectors are collected into a tiled texture block which is subsequently transferred back to the CPU and then used to compute the descriptors. This CPU-GPU partition was done to minimize data readback from the GPU since transferring the whole gradient pyramid back to the CPU is impractical. Moreover texture re-sampling and blending are efficient operations on the GPU; hence we perform those steps there. This also produces a compact tiled texture block which can be transferred to the CPU in a single readback.

GPU-SIFT gains a large speed-up in the Gaussian scale-space pyramid construction and keypoint localization steps.

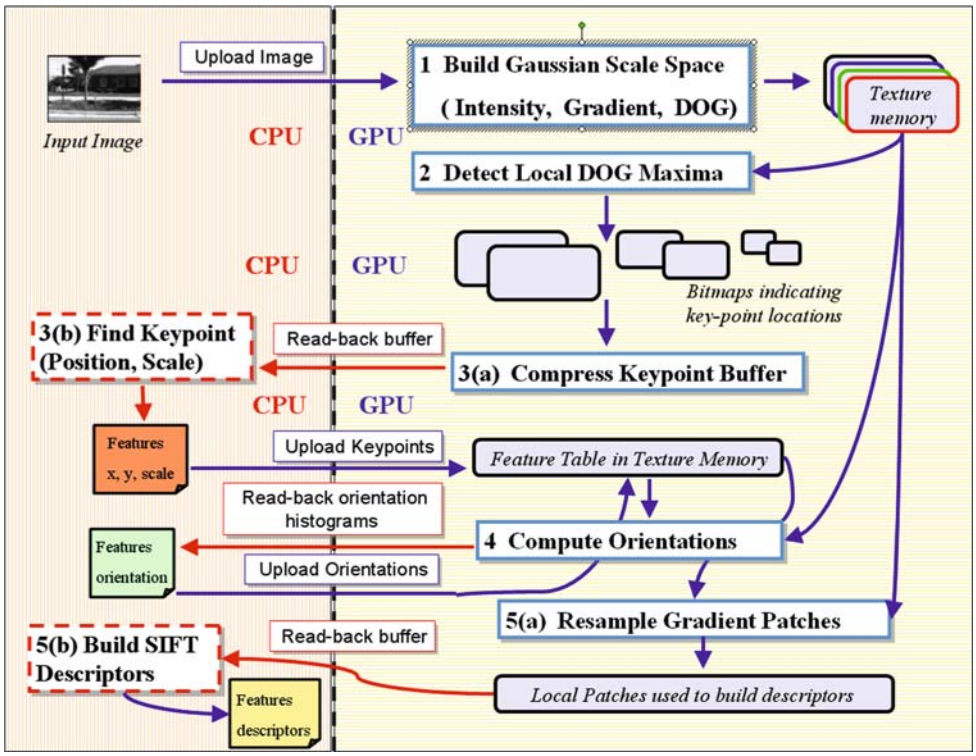

Fig. 7 Overview of the steps in the GPU-SIFT implementation

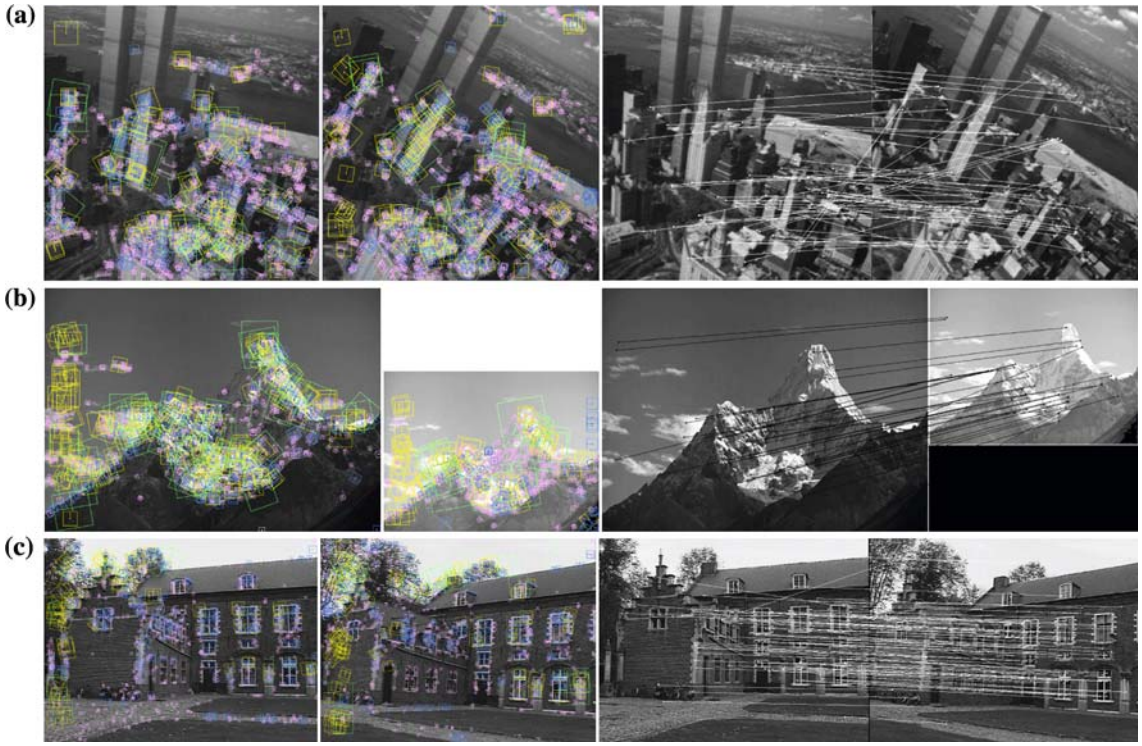

Fig. 8 GPU-SIFT Examples: About 1,000 interest points were extracted in each of the above three image pairs. Some of the initial matches (containing outliers) is shown. Features were matched despite scale, illumination and viewpoint change and image rotation

The compressed readback of binary images containing feature positions reduces the readback data-size by a factor of 32. The feature orientation and descriptors computation is partitioned between the CPU and GPU in a way that minimizes data transfer from GPU to CPU. Overall a 8–10× speedup is observed compared to CPU versions of SIFT.

4.3 Results

GPU-SIFT was implemented in OpenGL/Cg using Pbuffers for off-screen rendering. A texture manager allocates and manages all the data in GPU memory within a few double-buffered pixel buffers (PBuffers). In future we hope to replace PBuffers with FBOs which are increasingly being supported by the latest hardwares and drivers. Our current GPU-SIFT implementation has been tested on NVIDIA hardware (Geforce 7,800GTX and 7,900GTX cards). Figure 8 shows GPU-SIFT features extracted from image pairs containing changese in scale, orientation and illumination. Figure 9 compares timings between the CPU and GPU implementations for a range of image resolution and feature-count. The NVIDIA 7,900GTX gave a 10× speedup over an optimized CPU implementation. GPU-SIFT running on the NVIDIA 7,900GTX could extract about 1,000 SIFT features from streaming 640 × 480 resolution video at an average framerate of 10 Hz.

Nothing prevents GPU-SIFT from running on modern ATI cards with the latest drivers. At the time of implementation, we chose the NVidia / OpenGL platform which led to certain design choices that made our software prototype incompatible with ATI cards. Specifically render to rectangular texture targets OpenGL extensions (available only on NVidia) and integer texture coordinates were used extensively. With recent drivers supporting OpenGL 2+, GL_TEXTURE_2D targets and identical texture coordinate arithmetic can now be used in OpenGL on both ATI and NVidia platforms. Once this is changed, GPU-SIFT will run on ATI cards too.

GPU-SIFT was compared to Lowe's SIFT [10] (see Fig. 10) by using both for robustly estimating motion models parameters (2D homography, epipolar geometry) from multiple image-pairs. RANSAC-based robust algorithms were used to remove outliers present in the initial feature matches. The percentage of correct matches with GPU-SIFT was found to be about 70–98% of that obtained with SIFT on a set of 20 representative image-pairs. The slightly lower stability of GPU-SIFT features can be attributed to the following reasons—(a) we build the Gaussian scale space using approximate convolution kernels; (b) we do not double the image size if this causes the Pbuffer size to exceed the maximum texture size allowed on a particular graphics card (typically 4K × 4K). Morever GPU-SIFT skips two refinement steps that improve keypoint localization but are difficult to perform on the GPU—(a) sub-pixel localization of maximas of DOG

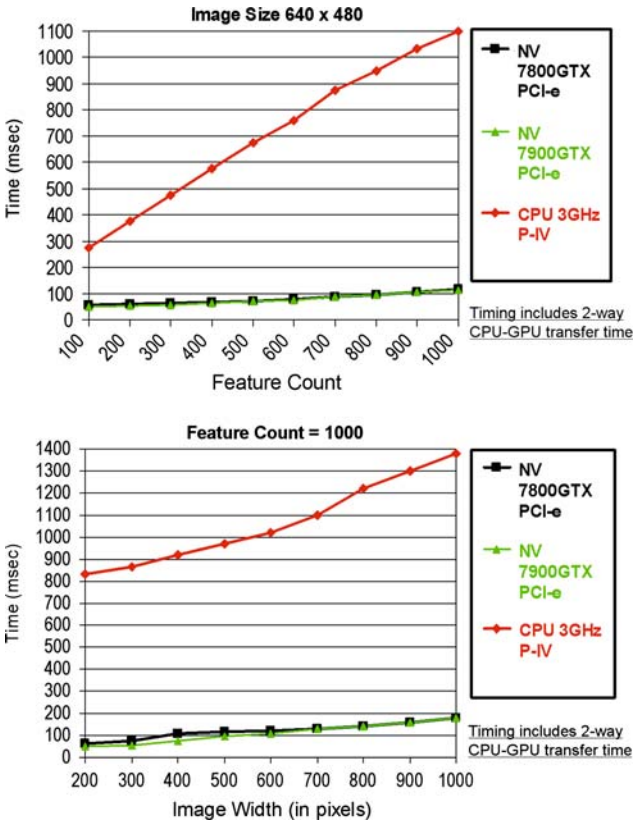

Fig. 9 GPU-SIFT timings compared with an optimized CPU implementation for a range of image-sizes and feature-counts. GPU-SIFT has a 10–12× speed-up

scale-space by fitting a local quadratic fit to the initial interest point locations; (b) refining the orientation histogram peaks through a quadratic fit of the three closest discrete samples. For higher accuracy, these steps could be included but they must be performed on the CPU. Figure 10 shows how different steps in the algorithm scale with varying input (image size, feature count). As the image resolution increases, scale-space construction and keypoint localization performed on the GPU dominates running time while as the feature count increases, more time is spent in computing SIFT descriptors on the CPU. As more descriptors need to be computed, the speedup due to bilinear interpolation in hardware is outweighted by the subsequent CPU computations. In future, efficient strategies for computing histograms on the GPU will be explored as this would further improve running times.

5 Conclusions

Both SIFT and KLT have been used for a wide range of computer vision tasks ranging from structure from motion, robot navigation, augmented reality to face recognition, object detection and video data-mining with quite promising results. We have successfully ported these popular algo-

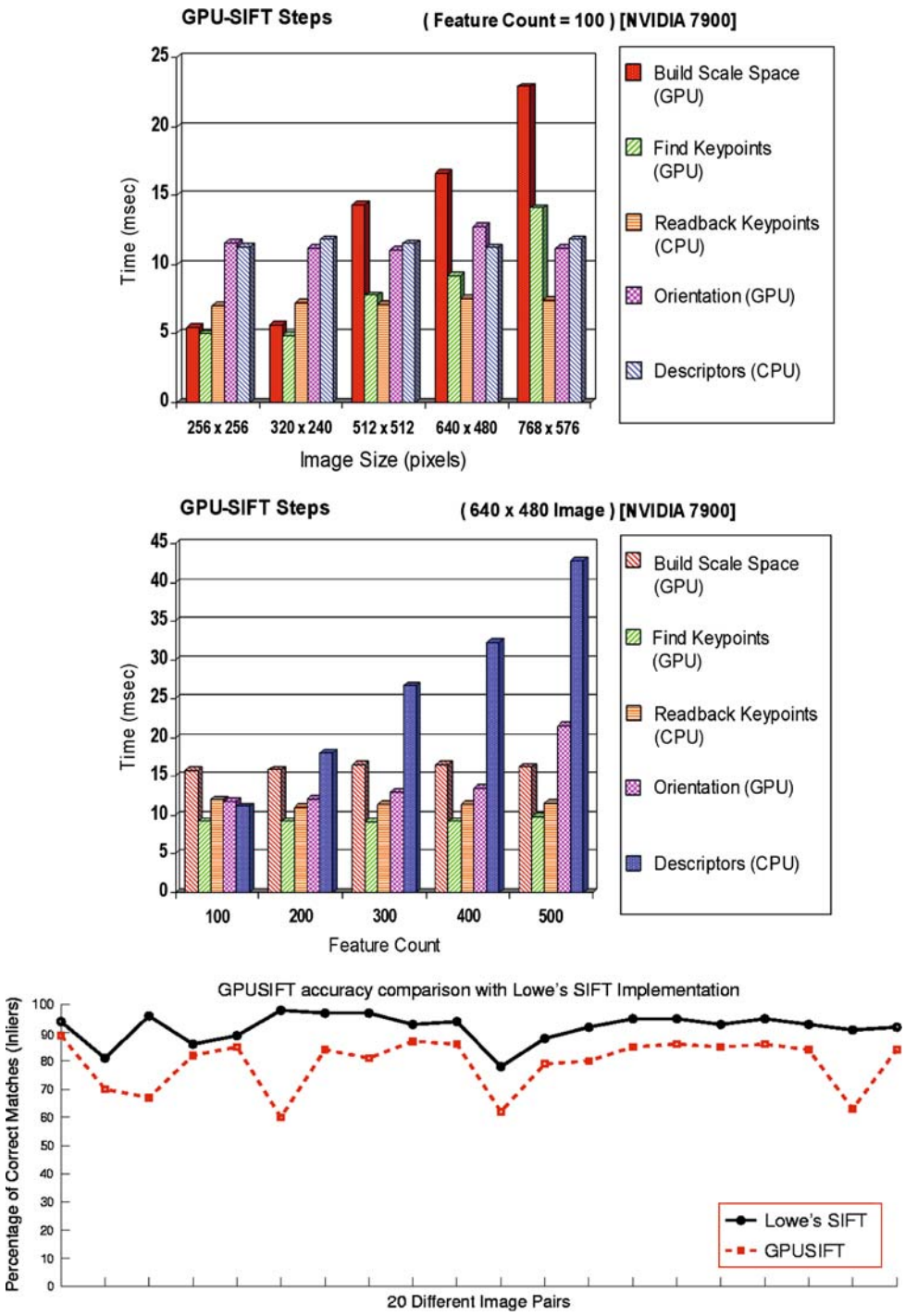

Fig. 10 Timings of various steps of the GPU-SIFT algorithm are shown. GPU computations scale linearly with image size while CPU computations scale linearly with an increasing feature count (*Top and middle row*). The percentage of correct matches (satisfying 2D homography, epipolar geometry) obtained for features extracted by GPUSIFT and Lowe's SIFT detectors for 20 different image pairs (*Bottom row*)

rithms to the GPU. In both cases, strategies were developed for dividing computation between the CPU and GPU in the best possible way under the restrictions of the GPU's computational model. Our GPU implementations which exploited the parallelism and incredible raw processing power provided by today's commodity graphics hardware are considerably faster than optimized CPU versions. As new generation graphics cards evolve (faster than predicted by Moore's law), our implementations would run even faster. This now makes it possible to perform high quality feature tracking, interest point detection and matching on high resolution video in real-time on most modern computers without resorting to the need for special-purpose hardware solutions.

Acknowledgements We gratefully acknowledge the support of the Real-Time Vision group at Siemens Corporate Research, the NSF Career award IIS 0237533 and the Packard Foundation.

References

1. Bramberger, M., Rinner, B., Schwabach, H.: An embedded smart camera on a scalable heterogeneous multi-DSP system. In: Proceedings of the European DSP Education and Research Symposium (EDERS 2004) (2004)
2. Klupsch, S., Ernst, M., Huss, S.A., Rumpf, M., Strzodka, R.: Real time image processing based on reconfigurable hardware acceleration. In: Proceedings of IEEE Workshop Heterogeneous Reconfigurable Systems on Chip (2002)
3. Fung, J., Mann, S.: OpenVIDIA: parallel GPU computer vision. ACM MULTIMEDIA 2005, pp. 849–852 (2005)
4. Fung, J., Mann, S.: Computer vision signal processing on graphics processing units. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2004), Montreal pp. V-93–V-96 (2004)
5. Gong, M., Langille, A., Gong, M.: Real-time image processing using graphics hardware: a performance study. In: International Conference on Image Analysis and Recognition, pp. 1217–1225 (2005)
6. Tomasi, C., Kanade, T.: Detection and Tracking of Point Features. Tech. Rept. CMU-CS-91132, Carnegie Mellon University (1991)
7. Lukas, B.D., Kanade, T.: An iterative image registration technique with an application to stereo vision. In: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 674–679 (1981)
8. Birchfield, S.: Derivation of Kanade-Lucas-Tomasi tracking equation. unpublished notes (1997)
9. Birchfield, S.: KLT: An Implementation of the Kanade-Lucas-Tomasi Feature Tracker. http://www.ces.clemson.edu/~stb/klt (2005)
10. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. IJCV **60**(2), 91–110 (2004)
11. Yang, R., Pollefeys, M.: Multi-resolution real-time stereo on commodity graphics hardware. In: Conference on Computer Vision and Pattern Recognition (CVPR) pp. 211–217 (2003)
12. Zach, C., Bischof, H., Karner, K.: Hierarchical disparity estimation with programmable 3D hardware. In: WSCG (International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision). Short Communications, pp. 275–282, Plzen, Slowakei (2004)
13. Woetzel, J., Koch, R.: Real-time multi-stereo depth estimation on GPU with approximative discontinuity handling. In: European Conf. on Visual Media Production (2004)
14. Labatut, P., Keriven, R., Pons, J.-P.: A GPU implementation of level set multiview stereo. Int. Conf. Comput. Sci. **4**, 212–219 (2006)
15. Yang, R., Welch, G.: Fast image segmentation and smoothing using commodity graphics hardware. J. Graph. Tools **7**(4), 91–100 (2002)
16. Strzodka, R., Droske, M., Rumpf, M.: Image registration by a regularized gradient flow—a streaming implementation in DX9 graphics hardware. Computing **73**(4), 373–389 (2004)
17. Griesser, A., Roeck, S.D., Neubeck, A., Gool, L.J.V.: GPU-based foreground-background segmentation using an extended colinearity criterion. In: Vision, Modeling, and Visualization (VMV) (2005)
18. Pollefeys, M., Gool, L.J.V., Vergauwen, M., Verbiest, F., Cornelis, K., Tops, J., Koch, R.: Visual Modeling with a Hand-Held Camera. IJCV **59**(3), 207–232 (2004)
19. Akbarzadeh, A., Frahm, J.-M., Mordohai, P., Clipp, B., Engels, C., Gallup, D., Merrell, P., Phelps, M., Sinha, S., Talton, B., Wang, L., Yang, Q., Stewenius, H., Yang, R., Welch, G., Towles, H., Nistr, D., Pollefeys, M.: Towards urban 3D reconstruction from video, Invited paper. In: 3rd International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT) (2006)
20. GPGPU: General-Purpose Computation on GPUs. http://www.gpgpu.org (2004)
21. Bjorke, K.A.: NVIDIA Corporation. Image processing using parallel GPU units. Proceedings of SPIE, vol. 6065 (2006)
22. Pharr, M., Fernando, R.: GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley Prof, Reading (2005)