Junsik Kim, Myung Hwangbo and Takeo Kanade, "Realtime Affine-Photometric KLT Feature Tracker on GPU in CUDA Framework," *Workshop on Embedded Computer Vision (held in conjunction with ICCV)*, October 2009.

Realtime Affine-photometric KLT Feature Tracker on GPU in CUDA Framework

Jun-Sik Kim, Myung Hwangbo, Takeo Kanade
Robotics Institute
Carnegie Mellon University
{kimjs,myung,tk}@cs.cmu.edu

Abstract

Feature tracking is one of fundamental steps in many computer vision algorithms and the KLT (Kanade-Lucas-Tomasi) method has been successfully used for optical flow estimation. There has been also much effort to implement KLT on GPUs to increase the speed with more features. Many implementations have chosen the translation model to describe a template motion because of its simplicity. However, a more complex model is demanded for appearance change especially in outdoor scenes or when camera undergoes roll motions.

We implement the KLT tracker using an affine-photometric model on GPUs which has not been in a popular use due to its computational complexity. With careful attention to the parallel computing architecture of GPUs, up to 1024 feature points can be tracked simultaneously at a video rate under various 3D camera motions. Practical implementation issues will be discussed in the NVIDIA CUDA framework. We design different thread types and memory access patterns according to different computation requirements at each step of the KLT. We also suggest a CPU-GPU hybrid structure to overcome GPU limitations.

1. Introduction

Feature tracking is the foundation of several high level computer vision tasks such as motion estimation, structure from motion, and image registration. Since the early works of Lucas and Kanade [8] and Shi and Tomasi [10], the Kanade-Lucas-Tomasi (KLT) feature tracker has been used as a *de facto* standard in handling point features in a sequence of images. From the original formulation a wide variety of extensions has been proposed for better performance. Baker and Matthews [4] summarized and analyzed the KLT variants in a unifying framework. Furthermore, open implementations in the public domain make this algorithm more popular in a practical use. Implementations in C language by Birchfield [5] or in the OpenCV [1] library are targeted for fast processing in either regular computers or embedded solutions.

A graphical processing unit (GPU) has been introduced in the KLT implementations to meet the demand for a higher volume of features in real-time applications. It takes benefit of acceleration in a parallel computing architecture because the tracker associated with each feature has no dependence on others. Sinha *et al*. [11] and Hedborg *et al*. [6] demonstrated their real-time GPU-accelerated KLT for more than 1000 features based on a translation model. Zach *et al*. [12] extended it to manage illumination change with one more parameter for gain adaptivity in a GPU. Ohmer and Redding [9] noticed that the main computational bottleneck of KLT lies in the selection of feature points and then proposed a Monte Carlo initialization in feature selection.

Many KLT implementations have chosen the *translational* model for template motion which allows only the position change. This model is acceptable as long as the projection of a 3D scene can be approximated by uniform shifting of the neighboring pixels. For example, the optical flow from the camera's out-of-plane rotation is similar to that from a translational camera motion. When severe projective transformation or in-plane camera rotation is involved, however, the translation can no longer handle the template deformation. The scene from a front camera of an unmanned aerial vehicle (UAV) during banking turn, for example, would fail to track features due to fast image rotation around the focal axis.

The problem above can be remedied by employing a more flexible motion model. Jin *et al*. [7] proposed a higher-order variant of the KLT. Their template motion model called *affine photometric* has 8 fitting parameters. It accounts for spatial deformation from an affine model as well as illumination change from a scale-and-offset model. A projective transformation is most general for spatial deformation but it tends to be prone to overfitting since the template size is usually small and so its deformation is well explained by other lower-order models like affine. This affine-

photometric model can successfully treat the images taken outdoors and under a camera roll motion. We will give a brief mathematical derivation of this model in Section 2.

One main drawback of the affine-photometric model is computational complexity. This prevents it from popular use in practice even though its tracking performance would be more robust. When a new feature is registered, the inverse of $n \times n$ Hessian matrix needs to be computed where n is the number of parameters. The time complexity of the Hessian computation is $O(n^3)$ and that of the update of motion parameters in tracking is also $O(n^3)$. Hence the complexity increases around 64 times when the affine-photometric model ($n = 8$) is chosen instead of the translational model ($n = 2$). To alleviate this increased computational burden, we also utilize the parallel processing ability of a GPU as previous works [11] [6] [12]. We will show that up to 1024 features can be tracked simultaneously at video rate using the affine-photometric model.

There exist two major programming tools on a GPU: Cg-script and NVIDIA CUDA framework. We choose the CUDA framework [2] because it is very similar to the ANSI C and it does not require a deep understanding about the OpenGL texture rendering process used in the Cg-script. We will discuss implementation issues and considerations in the CUDA framework in Section 3.

2. Mathematical formulation of KLT

The KLT is a local minimizer of the error function e between a template T and a new image I at frame $t+1$ given the spatial window $\mathcal{W}$ and the parameter $\mathbf{p}$ at frame t.

$$e = \sum_{\mathbf{x} \in \mathcal{W}} \left[T(\mathbf{x}) - I_{t+1}(\mathbf{w}(\mathbf{x}; \mathbf{p}_t, \delta\mathbf{p})) \right]^2 \tag{1}$$

Conventionally the Gauss-Newton method is used to search the parameter change $\delta\mathbf{p}$ in this minimization. The KLT variants differ by the tracking motion model $\mathbf{w}$ it employs and the way to update the parameter from $\delta\mathbf{p}_t$. For example, the translation model is defined with a two-dimensional translation vector $\mathbf{p}$.

$$\mathbf{w}(\mathbf{x}, \mathbf{p}) = \mathbf{x} + \mathbf{p}. \tag{2}$$

We can expect computational efficiency by switching the roles of the image I and the template T in (1).

$$e = \sum_{\mathbf{x} \in \mathbf{A}} \left[I\left(\mathbf{w}\left(\mathbf{x}, \mathbf{p}_t\right)\right) - T\left(\mathbf{w}\left(\mathbf{x}; \delta\mathbf{p}\right)\right) \right]^2 \tag{3}$$

The first-order Taylor expansion of (3) gives

$$e \approx \sum_{\mathbf{x} \in \mathbf{A}} \left[I(\mathbf{w}(\mathbf{x}, \mathbf{p}_t)) - T(\mathbf{w}(\mathbf{x}; \mathbf{0})) - \mathbf{J}(\mathbf{x})\delta\mathbf{p} \right]^2 \tag{4}$$

where the Jacobian $\mathbf{J} = \frac{\partial T}{\partial \mathbf{p}}|_{\mathbf{p}=\mathbf{0}}$. With the Hessian approximated as $\mathbf{H} = \sum \mathbf{J}^\top \mathbf{J}$, the local minimum can be found by minimizing (4) iteratively

$$\delta\mathbf{p} = \mathbf{H}^{-1} \sum_{\mathbf{x} \in \mathbf{A}} \mathbf{J}^\top \left[I(\mathbf{w}(\mathbf{x}, \mathbf{p}_t)) - T(\mathbf{x}) \right] \tag{5}$$

with the parameter update rule

$$\mathbf{w}(\mathbf{x}; \mathbf{p}_{t+1}) = \mathbf{w}(\mathbf{x}; \mathbf{p}_t) \cdot \mathbf{w}(\mathbf{x}; \delta\mathbf{p})^{-1}. \tag{6}$$

This is called *inverse compositional image alignment* and it takes advantage of a single Hessian computation only when a feature is registered. See Baker and Matthews [4] for details about various KLT methods.

2.1. Affine photometric model

The choice of the motion model needs to reflect the image distortion induced by camera motion. If a camera is in pan and tilt motions, the image change can be approximated by a translation. In case the camera is in roll motion, a more complex motion model is required. We derive the parameter update rule of the affine-photometric model [7] in the inverse compositional method.

The affine warp $(\mathbf{A}, \mathbf{b})$ and the scale-and-offset photometric parameters (α, β) describe the appearance change of a template.

$$T(\mathbf{x}; \mathbf{p}) = (\alpha + 1)\, T(\mathbf{A}\mathbf{x} + \mathbf{b}) + \beta \tag{7}$$

where $\mathbf{A} = \begin{bmatrix} 1 + a_1 & a_2 ; & a_3 & 1 + a_4 \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} a_5 & a_6 \end{bmatrix}^\top$. The Jacobian of (7) with respect to the parameter vector $\mathbf{p} = [a_1, \ldots, a_6, \alpha, \beta]$ is derived by the chain rule.

$$\mathbf{J} = \frac{\partial T}{\partial \mathbf{p}}\Big|_{\mathbf{p}=\mathbf{0}} = \begin{bmatrix} \frac{\partial T}{\partial \mathbf{a}} & \frac{\partial T}{\partial \alpha} & \frac{\partial T}{\partial \beta} \end{bmatrix} = \begin{bmatrix} \frac{\partial T}{\partial \mathbf{a}} & T & 1 \end{bmatrix} \tag{8}$$

The partial derivative w.r.t $\mathbf{a}$ is

$$\frac{\partial T}{\partial \mathbf{a}}\Big|_{\mathbf{a}=\mathbf{0}} = \frac{\partial T}{\partial \mathbf{w}} \frac{\partial \mathbf{w}}{\partial \mathbf{a}}\Big|_{\mathbf{a}=\mathbf{0}} = \frac{\partial T}{\partial \mathbf{x}} \frac{\partial \mathbf{w}}{\partial \mathbf{a}} = \nabla T \frac{\partial \mathbf{w}}{\partial \mathbf{a}} \tag{9}$$

$$= \nabla T \begin{bmatrix} x & y & 0 & 0 & 1 & 0 \\ 0 & 0 & x & y & 0 & 1 \end{bmatrix} \tag{10}$$

Finally the spatial gradient $\nabla T = (T_x, T_y)$ gives

$$\mathbf{J} = \begin{bmatrix} xT_x & yT_x & xT_y & yT_y & T_x & T_y & T & 1 \end{bmatrix}. \tag{11}$$

In (5) the Hessian $\mathbf{H}$ is approximated by $\sum \mathbf{J}^\top \mathbf{J}$ which is an 8×8 symmetric matrix and it is invariant once a new feature is registered so that we do not have to recompute it throughout the sequence. This benefit comes from the fact that the Jacobian is always computed at $\mathbf{p} = \mathbf{0}$ in the inverse compositional method. The Hessian computation is $O(n^2)$ and its inverse is $O(n^3)$. Therefore the computational complexity at the registration step increases by about 64 times compared to a 2×2 matrix in the translation-only model.

Algorithm 1: IMU-Assisted Feature Tracking

$n_{iter}, n_{fmin}, p_{max} \leftarrow$ fixed numbers
Compute the gradient $\nabla \mathbf{I}_{t+1}$
if $n_{feature} < n_{f,min}$ **then**
  Find new features from cornerness of $\nabla \mathbf{I}_{t+1}$
  Compute $\mathbf{H}$ and $\mathbf{H}^{-1}$
  Fill in lost slots of feature table
Get camera rotation $\mathbf{R}(\mathbf{q})$ from IMU
for *pyramid level* = p_{max} **to** 0 **do**
  forall *features* **do**
    Update initial warp $\mathbf{w}_{t+1}$ from $\mathbf{w}_{imu}$ using (15)
    Warp image $I_{t+1}(\mathbf{w}(\mathbf{x}, \mathbf{p}_{t+1}))$
    Compute error $e = T - I_{t+1}(\mathbf{w}(\mathbf{x}, \mathbf{p}_{t+1}))$
    Compute update direction $\delta\mathbf{p}$ using (5)
    for $k = 1$ **to** n_{iter} **do**
      Line search for best scale s^*
    Update parameter with $s^*\delta\mathbf{p}$ using (12)-(14)
  Remove features that $e > e_{thresh}$

The update of the parameter $\mathbf{p}$ at the tracking step is

$$(\mathbf{A}, \mathbf{b})_{t+1} = (\mathbf{A}_t \delta\mathbf{A}^{-1},\ \mathbf{b}_t - \mathbf{A}_t \delta\mathbf{b}) \tag{12}$$

$$\alpha_{t+1} = (\alpha_t + 1)/(\delta\alpha + 1) \tag{13}$$

$$\beta_{t+1} = \beta_t - (\alpha + 1)\delta\beta \tag{14}$$

and this requires an inversion of the affine matrix $\mathbf{A}_t$ but it can be simply calculated from a block-wise matrix computation. Algorithm 1 shows the procedure of the inverse compositional tracking method with the affine-photometric model.

2.2. Model complexity and sensitivity

The use of a more complex motion model makes it possible to track feature motions induced by arbitrary camera motions. However it is highly likely that the cost function has more local minima in a given search region. The KLT with a high-order model is more vulnerable to failure in the nonlinear minimization process. We can conclude that there exists a trade-off between model complexity and tracking accuracy.

This trade-off can be overcome by any of three possible solutions: 1) higher tracking rate, 2) feature re-registration, and 3) the use of an external motion sensor. If the tracking frequency becomes higher, the motion parameter changes less, and the result from the previous frame can be a good initial guess. However, this approach can not be feasible for fast moving cameras. One can re-register each template frequently to have a good initial guess, while it introduces error accumulation. Once the template is re-registered, the information of the original template is lost, and the error at the point of this re-registration can not be compensated.

2.3. Fusion with IMU

Another solution is to aid feature tracking with measurements from an external motion sensor such as an Inertial Measurement Unit (IMU). When instantaneous rotation amount R_{imu} from the IMU and camera calibration matrix K are available, the original initial guess $\mathbf{p}_t$ can be refined to $\widehat{\mathbf{p}}_{t+1}$ which is expected to be closer to a true minimum.

$$\mathcal{H} = \mathbf{K}^{-1}\mathbf{R}_{imu}\,\mathbf{K}$$
$$\mathbf{A}_{imu} = \mathcal{H}_{2\times 2},\ \mathbf{b}_{imu} = \mathcal{H}\mathbf{x} - \mathbf{x},\ \alpha_{imu} = \beta_{imu} = 0$$
$$\mathbf{w}(\mathbf{x}; \widehat{\mathbf{p}}_{t+1}) = \mathbf{w}(\mathbf{x}; \mathbf{p}_{imu}) \circ \mathbf{w}(\mathbf{x}; \mathbf{p}_t) \tag{15}$$

3. Considerations in CUDA implementation

If the CUDA framework is chosen as a programming tool for a GPU, there exist several considerations to make an implementation run as efficient as possible. In addition to the performance guideline given in the CUDA reference manual [2], we will discuss four issues directly reflected on our implementation.

Various memory types Six different types of memory spaces exist in the CUDA framework; register, local memory, shared memory, constant memory, texture memory and global memory. Each memory space has its own size, access speed, and usage limitations. A constant memory of 64KB, for example, can be read but not written by GPU device instructions. It also has 8KB cache to enhance the access time. The global memory has a much larger size and is readable/writable by GPU device instructions but slower in the memory access than the constant memory. Therefore a deliberate strategy about where to locate operands of the algorithms produces significant performance increase. More details about the memory spaces and their properties can be found in the reference manual of the CUDA framework [2].

Memory access pattern The CUDA framework provides a way to overcome the slow access to the global memory; fetching a bunch of ordered data at once. In other words, when consecutive threads read or write consecutive memory addresses (with some conditions satisfied [2]), a GPU can manage those data as a whole. This *coalesced* access to the global memory reduces the access time by an *order of magnitude* in practice. Therefore, reconfiguration or rearrangement of the global data in such way that guarantees the coalesced memory access makes the algorithm run faster in the CUDA framework.

Flow control One reason to take this into consideration is to ensure coalesced access to the global memory. If concurrent threads run a complicated algorithm that has branching conditions on given data, timing mismatch occurs in the next access to the memory between threads and it results in non-coalesced access. Moreover an iterative algorithm may not be possible to be implemented as a single-pass al-

gorithm, and thus, it is hard to compose from only a set of single instruction multiple data (SIMD) instructions. For example, it is better to convert an algorithm that finds a median or sorts data into an equivalent single pass algorithm that finds a minimum or maximum, if possible. If a numerical algorithm of interest fails to match the SIMD principle, its performance on a GPU would degrade severely. In that case it may be better to implement the algorithm on a CPU side in spite of additional burden to transfer data between a CPU and a GPU. This issue will be explained in the next section with more practical details in our implementation.

Data transfer Data transfer between the host (CPU) and the device (GPU) should be minimized because it could be the major bottleneck. Appropriate selection of a memory space helps avoid the unnecessary data transfer. If there are some data in a repeated use, it would be better to transfer at once and to assign them to a proper memory space in the GPU. For example, a look-up table (LUT) can be stored in either the constant or the texture memory of the GPU. However note that sometimes it is faster to compute LUT values online rather than to access to the LUT in the memory. It also depends on the access pattern to the LUT data.

4. GPU Implementation

Based on these considerations discussed in Section 3, more details on the GPU implementation of the affine-photometric tracker will be explained in this section. We focus on the memory access pattern, the usage of various memories, and the usability of the CPU-GPU hybrid implementation.

4.1. Thread types and memory access pattern

The affine-photometric feature tracker has two types of operations. One is for each pixel of an input image such as computing an image gradient or resampling to make a pyramid image. The other is for each feature track such as computing a Hessian matrix or evaluating errors from template warping. Hence we design the KLT algorithm with two kinds of threads - *pixel thread* and *feature thread*.

4.1.1 Pixel thread and texture memory

A pixel thread performs operations on its own pixel and the neighboring pixels. The routines implemented as a pixel thread are image pyramid, image gradient, and cornerness in the feature registration step. For a 320×240 image, for example, the cornerness computation takes 1.8 msec while the equivalent CPU version takes 4.2 msec. For the larger image size, we can expect more speed-up from the parallel structure of the GPU.

Accesses to input and gradient images in feature threads are fairly random since feature points are located sparsely in an image. The random access to the global memory cannot be coalesced as stated in Section 3. Therefore we register the input image, its pyramid images, and the computed gradient images on the *texture memory space* which is a cached and read-only memory for faster random access.

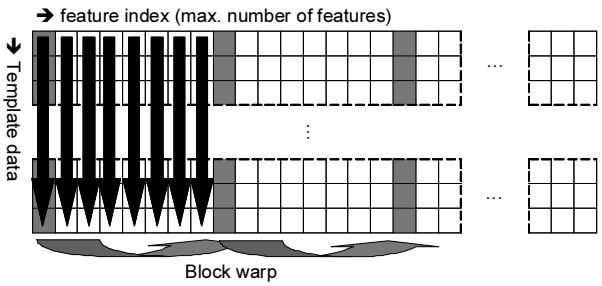


Figure 1. Memory structure of a track thread in the global memory: each thread uses each column of the table. The address increases in a row direction and n-th thread should read every n-th element in each row. This alignment enables the concurrent threads to access continuous addresses of the global memory.

4.1.2 Feature thread and global memory

A feature thread performs operations on each feature. The routines implemented as feature threads are building feature tables, computing warping errors, and updating motion parameters.

To update motion parameters of a feature, the algorithm has to access information of the feature template including template images, gradient templates, (inverse) Hessian matrices, and motion parameters for all pyramid levels. These data are stored on the global memory space aligned properly to ensure the coalesced access. In the conventional serial implementation, data for a single feature are located on continuous addresses on the memory. However, in CUDA framework, concurrent threads should read continuous addresses for the coalesced access. The associated data stored on the global memory space are aligned properly to ensure the coalesced access. For this purpose we design the feature table on the device global memory shown in Fig. 1.

The data structure is designed so that concurrent threads visit continuous memory addresses and transfer a bunch of ordered data at the same time. Note that it requires the maximum number of features be given *a priori*. In our implementation, this maximum is 512 or 1024[1] depending on the image size. Note that less memory for the given number of features is actually required, but by allocating more memory than required, execution of the algorithm becomes faster.

[1]This should be a multiple of 16. Note that the number of threads in a thread block should also be a multiple of 16 for the coalesced access. Refer [2] for further explanations.

4.1.3 Shared data and constant memory

Some data may need to be shared with all the feature threads. One example is when the camera motion is predicted by an external sensor like an IMU in Algorithm 1. The rotation information is used in *every* feature thread to refine the motion parameters respectively. In the CUDA framework there are two possible solutions; pass the data as a function argument or save the data in a constant memory. We choose to use the constant memory. For IMU fusion once the prediction warp $\mathbf{w}_{imu}$ from (15) is calculated in the CPU, its nine elements are transferred to the constant memory. All the feature threads can access the constant memory simultaneously with no time penalty in execution.

4.2. Flow control and CPU-GPU hybrid approach

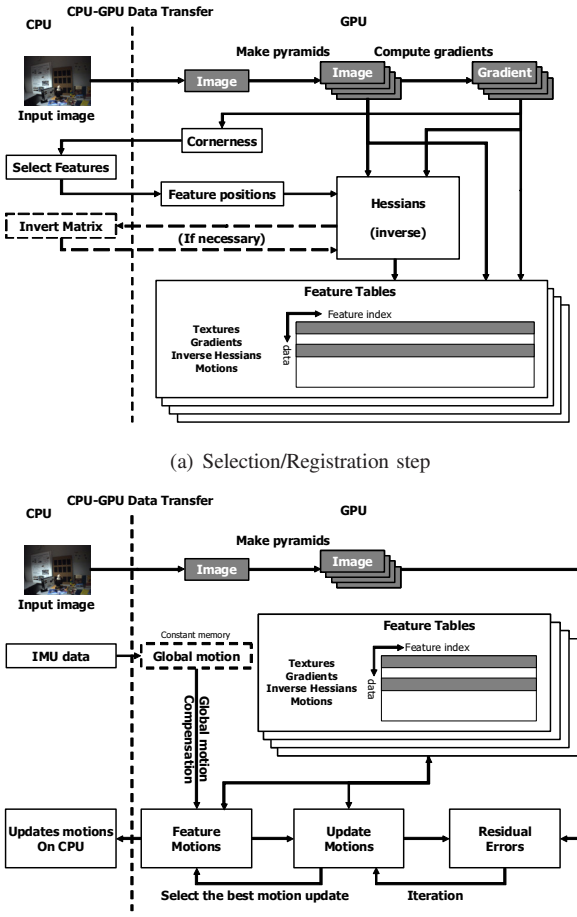
It is not true that the GPU can accelerate any algorithms. Processes that have many flow controls tend to run faster on the CPU. They branch or skip the branch depending on the data in the middle of process. This makes it hard for them to have the same instruction sets on different data values and thus it violates the SIMD principle of parallel computing. In the affine-photometric model, the sorting process of cornerness measures and the matrix inversion process of the Hessian fall in this category. They may run faster on the CPU even when accounting for additional data transfer between the CPU and the GPU.

Note that the inversion of a 8×8 Hessian matrix needs a quite sophisticated numerical algorithm while that of a 2×2 matrix in the translation model [11] is straightforward. If only a small number of Hessian matrices should be inverted, the GPU should solve more than required, and each parallel processor in the GPU is slower than the CPU. We will show the analysis on the computational complexity of the CPU and the GPU implementations in Section 5.3.

Managing the index labeled to each feature is also an issue. Some features will be lost after tracking. If more than a predetermined number of features are lost, we select and register new features. We can not know which feature will be lost in advance and rearrangement of the feature table with new features is not efficient. Therefore, we keep an index table on the CPU side which contains indices of lost features. When new features are to be selected, the index table is transferred to the GPU so that the registration threads can access which slot of the feature table is available. Figure 2 summarizes the memory allocation and data transfer flows of the implemented affine-photometric feature tracker.

5. Experiments

We have made a series of experiments to analyze the performance of the implemented affine-photometric tracker, including performance comparison with the CPU equivalent.



(a) Selection/Registration step

(b) Tracking step

Figure 2. Memory allocation and transfer flows of the affine-photometric feature tracker using GPU global memory (white box), GPU Texture memory (gray box), and constant memory (dashed box).

5.1. Performance in various motions

At first, we have tested three different motion types: translation, random shake, and roll & forward translation motions. We use the NVIDIA GeForce 8800 GTX graphics card on the Intel Xeon 3.2 GHz CPU. The image size of these "DESK" scenes in Figure 7 is 640×480 and the maximum number of features is 512. When the number of features tracked is less than 400 ($n_{f,min}$), we select and add new features to track 512 features again. We use five levels of image pyramid and each template has 15×15 pixels at every pyramid level. Each sequence has 293, 478, and 359 images respectively. Figure 7 shows optical flows generated from these camera motions.

Figure 3 shows the computation time including data transfer between CPU and GPU in each frame. The tracking step takes about 10 msec per frame. The selection step takes about 30 msec and the overall time consumed becomes

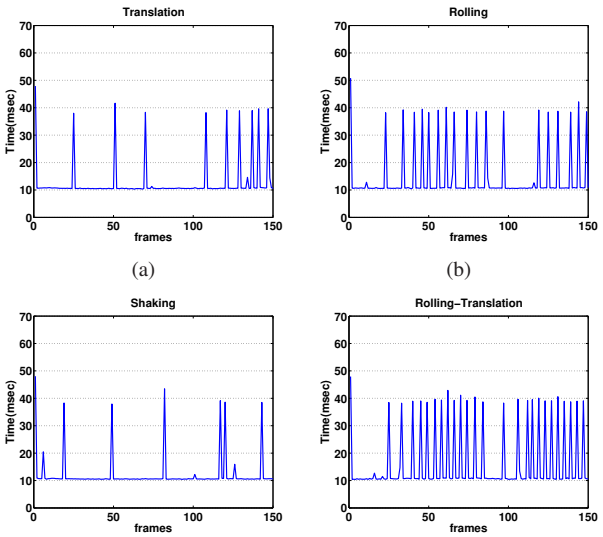


Figure 3. Computation time of the affine-photometric tracker from the scenes with (a) translation, (b) rolling, (c) shake, and (d) rolling and forward/backward camera motions for 150 frames. Each spike corresponds to additional time for new feature selection.

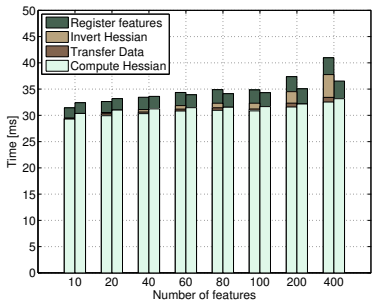


Figure 4. Performance comparison between the CPU-GPU hybrid implementation (left bars) and the GPU only method (right bars) in the registration step. If the number of new features is less than 60, the CPU-GPU hybrid implementation runs faster.

about 40 msec for the frame. One can see that more selection occurs when camera roll motion is involved because more features are lost out of the field of view of the camera. Our implementation processes 63.3, 50.3, and 53.9 of average FPS (frames per second) under translation, shake, and roll/translation motions, respectively. Figure 8 shows the tracking results when an UAV makes a bank motion. It produces a camera rolling scene that the conventional translational KLT has difficulties to handle. Furthermore these outdoor images suffer from dynamic illumination change. Even under these difficulties our affine-photometric treats them successfully in 30Hz video rate.

5.2. Time complexity of the GPU implementation

We compare the performance of our GPU implementation to that of the equivalent CPU implementation on various CPU/GPU systems. The CPU implementation uses the OpenCV library [1] with the Intel Integrated Performance Primitives (IPP).

The upper row of Figure 5 shows the comparison changing numbers of features using 25×25 templates for all 4 levels of pyramids. One can see that the tracking time has no increase on most GPUs when the number of features increases. It is because the processing time on a GPU is determined by the worst case. The only exception is the case using the NVIDIA 8600M that does not have enough number of parallel processors compared to others. Note that even the worst GPU runs faster than the best CPU. In the lower row of Figure 5, the processing time tracking 500 features increases quadratically with respect to the template size but its rate is much smaller than that of the CPU.

In Figure 6 we compare the performance on the various GPUs. All the GPUs except the NVIDIA 8600M show similar results. Feature selection/registration step tends to take more time for more features because it uses CPU for sort and selection.

5.3. CPU-GPU hybrid approach

Figure 4 compares the processing time of the CPU-GPU hybrid and the GPU-only approaches in the registration step. The left bars are for the hybrid approach and the right bars are for the GPU-only. In the CPU-GPU hybrid implementation, the LAPACK [3] library is used for inverting the Hessian matrices. One can see that the hybrid approach takes less time when the number of features is less than 60, which occurs frequently in re-selecting features. Note that this result can be different in different CPU and GPU configurations.

6. Conclusion

We have implemented the parallel version of the affine-photometric feature tracker. It is much more flexible than translation-only KLT trackers so that illumination changes and rotational template deformation can be dealt with effectively. We explain the general considerations in the CUDA framework and how to design threads and memory structures for maximum performance. In CUDA implementation we should consider 1) different memory access patterns, 2) flow controls for optimal memory access and 3) data transfer between the host and the device. For two types of operations, pixel and feature threads, we decide data memory types and design the GPU global memory map to ensure faster coalesced access.

The experiments show the video rate is achievable even with the high computational complexity of the affine-

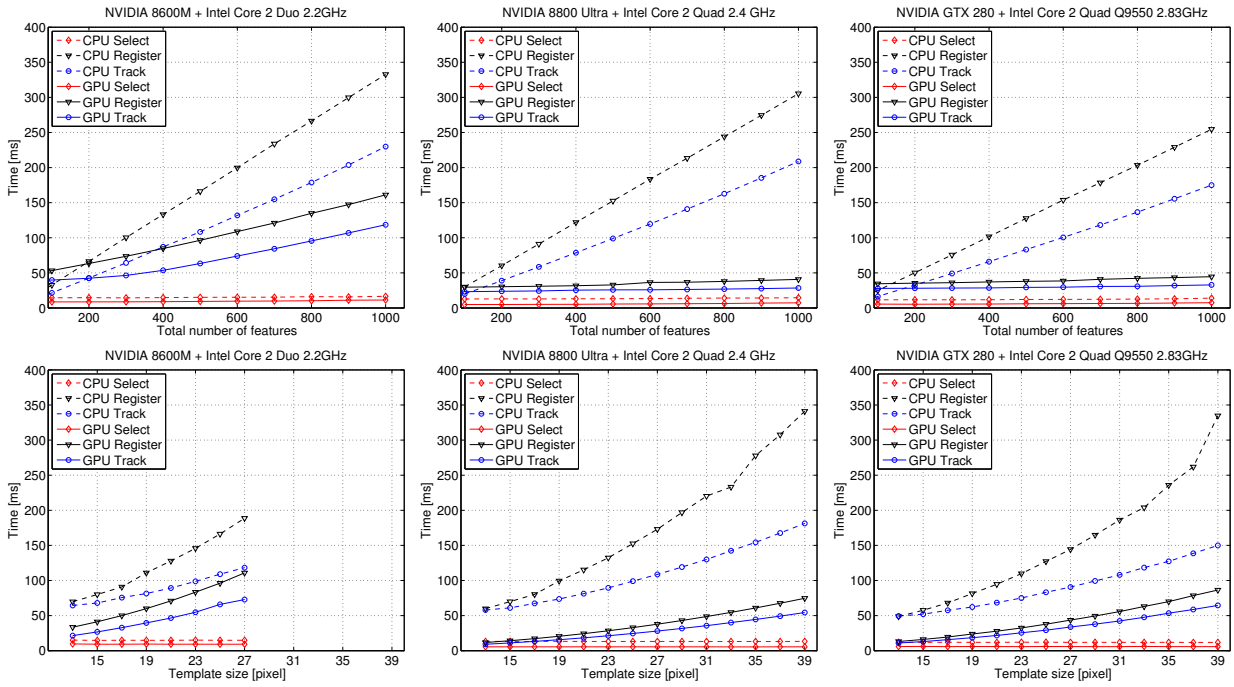


Figure 5. Performance comparison between the CPU and GPU implementations on various platforms with respect to the number of features using 25×25 templates (upper row), and the template size with 500 features (lower row). Dashed lines are for the CPU and solid lines for the GPU. The GPU has no performance degradation when the number of features increases. Due to the 128 MB memory limit of the NVIDIA 8600M, the tracker runs only up to 27×27 template size.

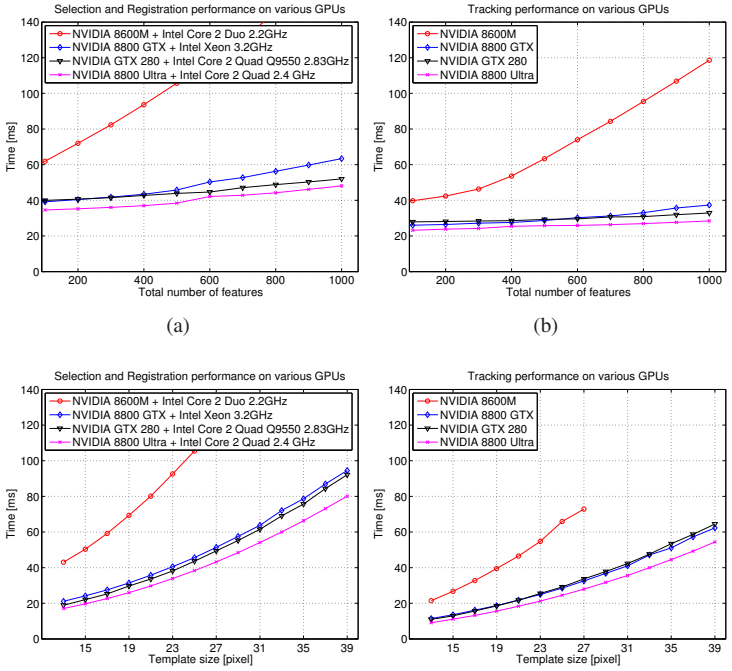


Figure 6. Performance comparison in computation time on various GPUs: (a) selection/registration step and (b) tracking step with different total numbers of features using 25×25 templates, and (c) selection/registration step and (d) tracking step with different template sizes for 500 features.

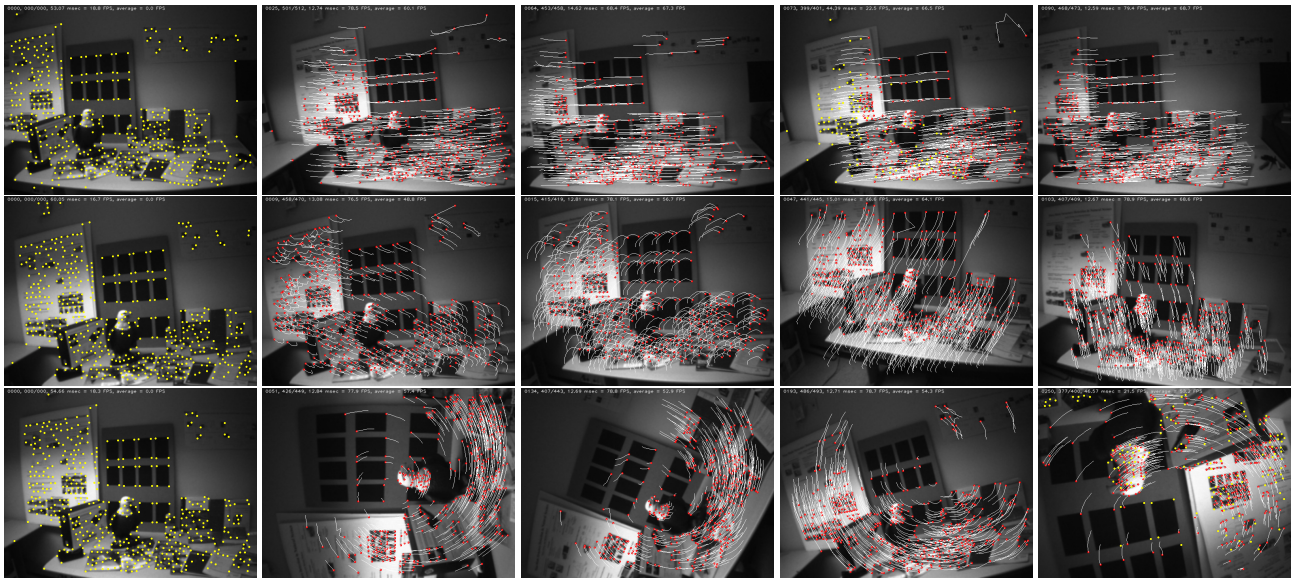


Figure 7. Tracking samples of the DESK scenes: Each row from the top is translation, shake, and rolling with forward/backward motion respectively. White tails show the tracks over last five frames. New feature points are in yellow and tracking ones are in red.

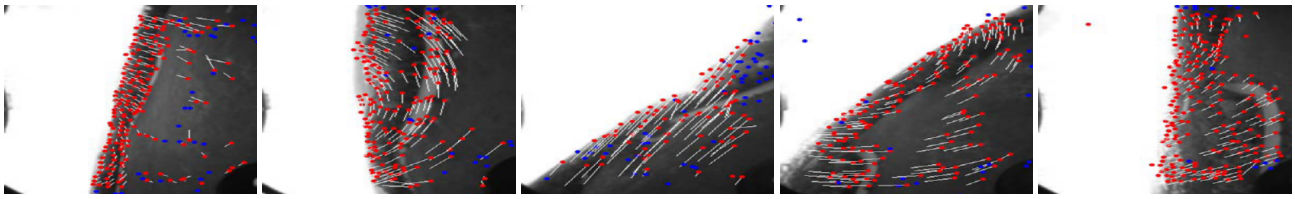


Figure 8. Optical flows from the KLT for the UAV image sequence. The input images suffer from severe illumination changes as well as camera rolling motion.

photometric tracker. The registration step is still the major bottleneck but the tracking step runs 50 FPS under any camera motion. Performance comparison on various CPU/GPU configurations is also presented.

References

[1] http://sourcefouge.net/projects/opencvlibrary.

[2] http://www.nvidia.com/cuda/.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[4] S. Baker and I. Matthews. Lucas-Kanade 20 Years On: A Unifying Framework. *International Journal of Computer Vision*, 56(3):221 – 255, March 2004.

[5] S. Birchfield. KLT: An implementation of the Kanade-Lucas-Tomasi feature tracker. Available online, http://www.ces.clemson.edu/∼stb/klt/, 1997.

[6] J. Hedborg, J. Skoglund, and M. Felsberg. KLT tracking implementation on the GPU. In *Proceedings SSBA 2007*, Linkoping, Sweden, March 2007.

[7] H. Jin, P. Favaro, and S. Soatto. Real-time feature tracking and outlier rejection with changes in illumination. In *International Conference on Computer Vision*, pages 684–689, 2001.

[8] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 1981 DARPA Image Understanding Workshop*, pages 121–130, April 1981.

[9] J. Ohmer and N. Redding. GPU-accelerated KLT tracking with monte-carlo-based feature reselection. In *Computing: Techniques and Applications, 2008. DICTA '08. Digital Image*, pages 234–241, Dec. 2008.

[10] J. Shi and C. Tomasi. Good features to track. In *Computer Vision and Pattern Recognition, 1994 IEEE Computer Society Conference on*, pages 593–600, 1994.

[11] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc. Feature tracking and matching in video using programmable graphics hardware. *Machine Vision and Applications*, November 2007.

[12] C. Zach, D. Gallup, and J.-M. Frahm. Fast gain-adaptive KLT tracking on the GPU. In *Computer Vision and Pattern Recognition Workshop on Visual Computer Vision on GPU's (CVGPU), 2008. IEEE Computer Society Conference on*, pages 1–7, June 2008.