Realtime KLT Feature Point Tracking for High Definition Video

Hannes Fassold[1]
hannes.fassold@joanneum.at

Jakub Rosner[2]
jakub.rosner@joanneum.at

Peter Schallauer[1]
peter.schallauer@joanneum.at

Werner Bailer[1]
werner.bailer@joanneum.at

ABSTRACT

Automatic detection and tracking of feature points is an important part of many computer vision methods. A widely used method is the KLT tracker proposed by Kanade, Lucas and Tomasi. This paper reports work done on porting the KLT tracker to the GPU, using the CUDA technology by NVIDIA. For the feature point detection, we propose to do all steps of the detection process, except the final one (enforcing a minimum distance between feature points), on the GPU. The feature point tracking is done on a multi-resolution image representation to allow tracking of large motion. Each feature point is calculated in parallel on the GPU. We compare the CUDA implementation with the corresponding OpenCV (using SSE and OpenMP) routines in terms of quality and speed, noticing a significant speedup of up to factor 10. Some additional experiments are done regarding the influence of different parameterization on the runtime. Our GPU implementation achieves realtime (> 25 fps) performance for High Definition (HD) video sequences, successfully tracking several thousands of points. In summary, the GPU implementation achieves a significant speedup compared with an optimized CPU implementation and allows the analysis of high resolution video sequences in realtime.

Keywords

KLT, feature point tracking, Lucas Kanade, corner detection, optical flow, motion estimation, GPU, CUDA

1. INTRODUCTION

The automatic detection and tracking of (typically corner-like) feature points throughout an image sequence is a necessary prerequisite for many algorithms in computer vision. The gathered information about the feature points and their motion can be used subsequently for pose estimation, camera self-calibration [Koc99] and for tracking various kinds of objects like people and vehicles [Lyp07][Kan06]. One of the most popular methods for feature point tracking is the KLT algorithm which was introduced by Lucas and Kanade [Luc81] and later extended in the works of Tomasi and Kanade [Tom91] and Shi and Tomasi [Shi94]. The KLT algorithm automatically detects a sparse set of feature points which have sufficient texture to track them reliable. Afterwards, detected points are tracked by estimating for each point the translation, which minimizes the SSD dissimilarity between windows centered at the current feature point position and the translated position.

Despite being more than 20 years old, the KLT algorithm is still widely used, as it operates in a fully automatic way and its performance in terms of feature point quality and runtime is competitive compared with other methods. A problem occurs, when using the KLT algorithm in realtime applications (e.g. in surveillance), where strict runtime requirements must be fulfilled. Typically cameras deliver 25 – 30 images per second, so the runtime of the algorithm for one image may not exceed 33 - 40 milliseconds.

[1] JOANNEUM RESEARCH, Institute of Information Systems, Steyrergasse 17, 8010 Graz, Austria

[2] Silesian University of Technology, Faculty of Automatic Control and Robotics, Ulica Academicka 2, 44-100 Gliwice, Poland

Current implementations of the KLT algorithm (e.g. the OpenCV[3] routine) achieve this only when the image resolution is not higher than Standard Definition (720x576) and the number of feature points is a couple of hundreds at most. If the image resolution is higher (e.g. for HD video) and more points are to be tracked, one has to look for alternatives.

Because of its tremendous computational capability Graphic Processing Units (GPUs) gain significant importance for computer vision. In this document we describe work done on porting the KLT algorithm to the GPU using CUDA. CUDA[4] stands for Compute Unified Device Architecture and is a C-like GPU programming environment introduced by NVIDIA. We first give an introduction to GPU programming with a focus on CUDA (section 2) and discuss previous work done on implementing the KLT algorithm for the GPU (section 3). In section 4, an overview of the general KLT algorithm is given and section 5 discusses its implementation for the GPU. Finally, section 6 compares the GPU implementation with the reference CPU implementation in terms of speed and quality.

2. GPU PROGRAMMING & CUDA

In the last few years, GPUs have evolved from specialized devices for accelerating 3D graphics to powerful coprocessors, which can be used for general purpose GPU programming. The processing power of GPUs (measured as number of floating-point operations per second) is nearly doubling every year and exceeds modern CPUs processing capabilities by far. Moreover, while a couple of years ago GPU developers had to adapt their algorithms to fit into a special purpose computer-graphics oriented render pipeline, the advent of general purpose GPU programming languages like Brook[5], CUDA or OpenCL[6] brought much more flexibility into the field of GPU programming. In the following, we will focus on CUDA, which is currently the most mature of these and can be run on all current NVIDIA GPUs starting with the Geforce 8 series.

CUDA GPU Architecture

The following properties are characteristic for a CUDA-capable GPU:

- Manycore architecture (e.g. Geforce 280GTX has 30 multiprocessors, corresponding to 240 processing cores),
- A very fast thread management (done in hardware), which allows switching between different threads with virtually no overhead,
- Random access device memory (*global memory*) which can be accessed by all threads, but has a high latency,
- A very fast read-write cache called *shared memory* (16 KB per multiprocessor), which has to be managed by the algorithm developer,
- Other important memory types like *texture memory* (read-only, cached, offers bilinear interpolation) and *constant memory* (read-only, cached).

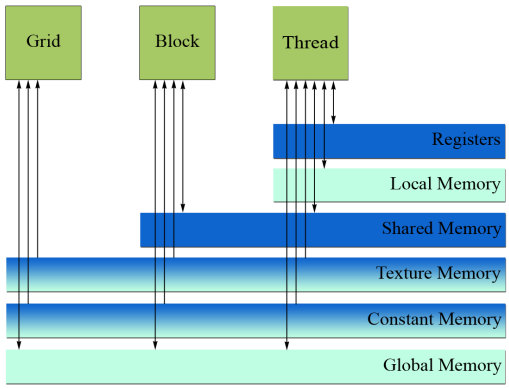

Figure 1: Different memory types of a CUDA-capable GPU. Blue = on-chip, yellow = off-chip, shaded = off-chip, but cached. The arrows indicate the allowed access type (read-only, read-write).

CUDA Programming Model

A CUDA program is typically composed of a control routine which calls a couple of CUDA *kernels*. A *kernel* can be compared to a C function, but is executed on the GPU in parallel by a large number of threads in a SIMT (single instruction, multiple threads) fashion. Each *thread* is identified by its unique *thread id*. Groups of 32 consecutive threads are organized into *warps* with *half-warps* as their first or second halves. Furthermore, sets of up to 512 consecutive threads are grouped into *thread blocks*, which then form a *grid*.

Synchronization among different thread blocks can only be achieved after the whole kernel has completed (*global synchronization*). Depending on the resources (registers, shared memory) a thread block uses, one or more of them are assigned to a multiprocessor to be executed simultaneously. After having finished, new thread blocks are assigned to the multiprocessor, until the whole grid has been completed. The order in which thread blocks are executed is not defined and depends on the number of multiprocessors of the GPU.

[3] http://sourceforge.net/projects/opencv/

[4] http://www.nvidia.com/object/cuda_home.html

[5] http://graphics.stanford.edu/projects/brookgpu/

[6] http://www.khronos.org/opencl/

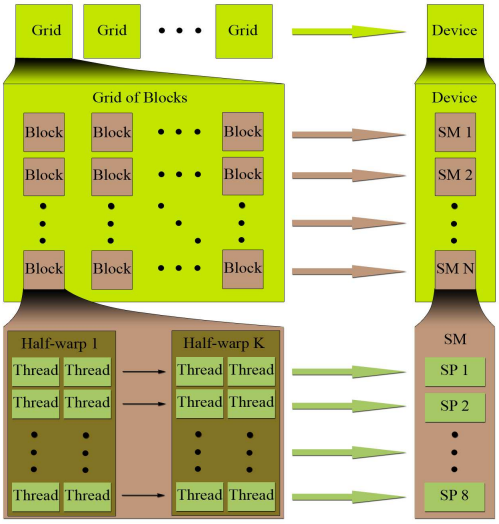

Figure 2: CUDA assigns grids to a device (GPU), thread blocks to its Streaming Multiprocessors (SM) and threads to Scalar Processors (SP).

Note that in image processing algorithms, typically one thread computes one pixel. A thread block is typically corresponding to a small tile of the image. All thread blocks corresponding to the whole image then form the grid.

CUDA Porting Guidelines

In the following section, some general guidelines are given how to port an algorithm efficiently to CUDA. First (and most important), it must be able to split the algorithm into a large number (at least hundreds) of loosely coupled threads which run in parallel on the GPU. One often has to rethink the whole algorithm or parts of it to be able to fulfill this requirement.

A very common way in writing a CUDA kernel is to divide it into three parts separated by a synchronization barrier. In the first part all the data essential for computations inside a given thread block is loaded from global memory to the very fast shared memory. It is essential to use a synchronization barrier after the loading stage to ensure that all the data has been loaded before proceeding to the next step. In the next part, the processing stage, the shared memory data is used in a way that depends on the purpose of the given algorithm - e.g. a convolution, interpolation, summation etc. The results are then stored in a temporary buffer residing also in shared memory. Another synchronization barrier has to be set to ensure completeness of the processing stage. The last part is usually the shortest one and simply writes the temporary buffer back into the proper location in global memory.

An obvious and troublesome problem for the programmer is the very high level of parallelism in kernel's execution. There can be tens of thousands of threads running concurrently on a single GPU. When two threads try to increment, compare or change in some other way the value at the same memory address simultaneously, only one of them will be likely to succeed. In that case it is strongly recommended to think of another, sometimes completely different way to implement the given algorithm to allow parallel execution. This problem can be very hard and in some cases the only way to solve it is to use atomic functions. Atomic functions will be completely serialized and for this reason can significantly decrease the overall performance.

Branches ('if-then-else', 'while') within the threads of a half-warp should be reduced to a minimum as they lead to divergent execution of threads, and are sometimes serialized. Memory transfers between CPU memory and GPU memory should also be reduced to minimum as they are very costly. The same applies (to a lesser extent) to allocations and deallocations of large memory buffers.

The available memory types as shown in Figure 1 should be understood and used properly to achieve an optimal implementation. Especially the usage of shared memory to cache a small part of the high-latency global memory is important. Furthermore for optimal performance in accessing global memory, threads within a half-warp should access memory locations of the same memory segment (*coalescing rule*). It is possible to hide a large part of memory latency in arithmetic computations by executing as many threads simultaneously as possible (the limit is 1024 per multiprocessor for the latest GPUs). Furthermore, for read-only data the usage of texture and constant memory is often helpful as they are cached.

Interactions between threads should be restricted to threads of the same thread block and done using shared memory. Shared memory is especially useful in cases when those threads use data that mutually overlaps and is generally as fast as registers, as it resides on chip, unless *bank conflicts* occur. To avoid them, threads from the same half-warp have to read memory addresses with a step size which is dividable by 4 bytes and not dividable by 8 bytes.

Register usage of a single thread block should be minimized, otherwise the number of thread blocks that can be run concurrently by a single multiprocessor may be reduced.

Shared memory, which is mainly used for communication between threads of the same block, is a very powerful ally in kernel optimization. Regarding the KLT algorithm implementation, it is used in almost every CUDA kernel and greatly improves the performance. Shared memory can also be successfully used to store small per-thread arrays,

as otherwise they would be allocated by the compiler outside the chip in the *local memory*, which has the same latency as the global memory.

For some special functions like square-root(), sine() and cosine() there exist significantly faster variants with lower precision. Although available since the NVIDIA G200 GPU series, the usage of *double* precision values should be avoided as it is significantly slower than single precision ones.

Note that in [Che08] a study has been done about the effectiveness of CUDA when porting different kind of algorithms (combinatorial logic, dynamic programming, data mining etc.) to the GPU. They report speedups ranging from moderate 2.9 times for dynamic programming (which is hard to parallelize) up to 72 times for k-mean clustering.

3. RELATED WORK

There has been done some previous work on porting the KLT tracker to the GPU. In [Sin06] OpenGL and the Cg[7] shader language are used for the GPU implementation. A fixed number of iterations is done for each feature point to avoid conditional statements. Detection of new feature points is done only every fifth frame to save computation time. In [Zac08] the KLT tracker is also implemented using Cg. Their KLT tracker compensates for varying camera gain by estimating it as a global multiplicative constant. Another Cg KLT implementation is described in [Ohm08]. They propose a modified variant of the feature detection process to circumvent some hard parallelizable parts of it. Note that to our knowledge, no CUDA KLT implementation has been reported so far in published works.

4. KLT ALGORITHM

The KLT algorithm can be divided into two main parts. During the detection process, salient feature points are found and added to the already existing ones. Afterwards, in the tracking process for each feature point its corresponding motion vector is calculated. In the following, we describe each part of it in more detail. The algorithm follows the standard scheme for the KLT algorithm as was proposed in the works of [Luc81][Tom91][Shi94].

Note that in the following Greek letters denote scalars, lowercase letters denote column vectors and uppercase letters denote matrices. We denote I as the current image and J as the immediately next image in the sequence. We write $\nabla I = \partial I / \partial(x, y)$ as the spatial image gradient of I, which is typically done with the Sobel or Sharr operator for robustness. Also we define as $W(p)$ a small rectangular region centered at a given point p. Typically $W(p)$ will be a 5 x 5 or 7 x 7 pixel neighborhood. As the tracking is done with sub-pixel precision, p will have non-integer coordinates. Its neighbors are then calculated using bilinear interpolation.

Feature Point Detection

The task here is to detect new feature points in a given image I and add them to the already existing feature points. In order to track feature points reliably, their pixel neighborhood should by richly structured. As a measure of 'structuredness' of the neighborhood of a pixel p, one can define the structure matrix G:

$$G = \sum_{x \in W(p)} \nabla I(x) \cdot \nabla I(x)^T$$

Its eigenvalues λ_1, λ_2 (which are guaranteed to be ≥ 0 as the matrix is positive-semidefinite) deliver useful information about the neighborhood region W. If W is completely homogenous, then $\lambda_1 = \lambda_2 = 0$. In contrast, $\lambda_1 > 0$, $\lambda_2 = 0$ indicates that W contains an edge and $\lambda_1 > 0$, $\lambda_2 > 0$ indicates a corner. The smaller eigenvalue $\lambda = \min(\lambda_1, \lambda_2)$ can now be used as a measure of the cornerness of W, where larger values means stronger corners.

The feature detection is now composed of the following steps:

1. Calculate structure matrix G and cornerness λ for each pixel in the image I.
2. Calculate the maximum cornerness λ_{max} occurring in the image.
3. Keep all pixels that have a cornerness λ larger than a certain percentage (5% - 10%) of λ_{max}.
4. Do a non-maxima suppression within the 3 x 3 pixel neighborhood of the remaining points to keep only the local maxima.
5. From the remaining points, add as many new points to the already existing points as needed, starting with the points with the highest cornerness values. To avoid points concentrated in some area of the image, newly added points must have a specific minimum distance (e.g. 5 or 10 pixels) to the already existing points as well as to other newly added points (*Minimum-Distance-Enforcement*).

Feature Point Tracking

In the tracking step, we want to calculate for each feature point p in image I its corresponding motion vector v so that its tracked position in image J is $p + v$.

As 'goodness' criterion of v we take the SSD error function $\varepsilon(v) = \sum_{x \in W(p)} (J(x+v) - I(x))^2$. It measures

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

1. Set initial motion vector $v_1 = (0,0)^T$
2. Spatial image gradient $\nabla I = \partial I / \partial (x, y)$
3. Calc. structure matrix $G = \sum_{x \in W(p)} \nabla I(x) \cdot \nabla I(x)^T$
4. for k = 1 to *maxIter*
   a) Image difference $\eta(x) = I(x) - J(x + v^k)$
   b) Calc. mismatch vector $b = \sum_{x \in W(p)} \eta(x) \cdot \nabla I(x)$
   c) Calc. updated motion $v_{k+1} = v_k + G^{-1} b$
   d) if $\| v_{k+1} - v_k \| < eps$ then stop (converged)
5. Report final motion vector v

Table 1: Pseudo-code of the calculation of the motion vector *v* for a given feature point *p*. W(*p*) is a window centered at *p*. Typically the window size is set to 5 x 5 pixel, *maxIter* to 10 and *eps* to 0.03 pixel.

the image intensity deviation between a neighborhood of the feature point position in I and its potential position in J and should be zero in the ideal case. Setting the first derivative of $\varepsilon(v)$ to zero and approximating $J(x + v)$ by its first order Taylor expansion around $v = 0$ results in a better estimate v_1. By repeating this multiple times, we obtain an iterative update scheme for v which is summarized in Table 1.

Due to the Taylor expansion around zero the given scheme is only valid for small motion vectors v. In order to allow tracking of large motions of feature points, which is quite common, we generate an image pyramid and apply the scheme for all points in each pyramid level. We are doing this from coarse pyramid level to fine one, using the result of the previous pyramid level as initial guess for the next one.

5. CUDA IMPLEMENTATION

In this section we will discuss issues which are specific for the CUDA implementation of the KLT tracker.

The very first thing that has to be done before any GPU kernel can be run is to allocate a GPU memory buffer and transfer the essential data from CPU memory to it. In order to save unnecessary allocations and deallocations of GPU memory, before processing the first image of the sequence a context object is created which holds all the necessary memory resources (for the image pyramids etc.) for the KLT algorithm. It is reused during processing of the image sequence and deleted after the processing is finished.

Feature Point Detection Implementation

The algorithm which does the feature point detection has been divided into separate steps, as explained in section 4, each being computed by one or more kernels. For optimization purposes the operations were assigned to kernels in a way that minimizes the overall number of kernels and by that read-write operations in global memory space, as those are particularly costly.

The first step of the algorithm has been divided into three different kernels, first computing the gradients for each pixel and the next two summing them up in window W in order to get the G matrix and the cornerness λ.

The second step, which determines the maximum cornerness, is a good example of an operation that seems to be conceptually very simple, but is complicated to implement efficiently for a massively parallel architecture. In this case it involves a lot of read-write hazards, when many threads want to compare and modify a single value simultaneously. A solution to this problem is to create a *reduction tree*, in which each thread determines the maximum of a couple of values and stores it at a different address. A highly optimized version of this reduction algorithm (along with some other useful algorithms for compaction, sorting etc.) can be found in the CUDA performance primitives library[8] (CUDPP) and was used by us for calculating the maximum value.

In steps 3 and 4 we mark features, that do not meet their respective conditions, as invalid, since removing them in each step separately would not only be complicated on GPU, but also inefficient.

Before doing step 5, the potential feature points have to be transferred back to the CPU memory. Considering how costly such transfers are, the feature points are compacted before that to remove the ones, which were marked as invalid. Once again, we use the CUDPP library for this purpose.

Step 5 of the feature point detection algorithm (the enforcement of a certain minimum distance between feature points) is very hard to parallelize and inefficient to run on the GPU. This fact forces us to do it on the CPU. Note that the standard algorithm as implemented in the OpenCV library is only efficient for a small number of features (less than 1000) as its computational complexity increase quadratically. For efficiently handling larger numbers of features we have implemented an alternative algorithm, whose complexity increases linearly with the number of features. It requires an additional mask image, which has the same size as the input image. The idea of this algorithm is to add a feature point to the final feature

[8] http://www.gpgpu.org/developer/cudpp

point list only when its position is not masked out in the mask image. If it is added to the list, all neighboring pixels which are within the specified minimum distance are masked out in the mask image. As the standard algorithm, it starts at the first feature with maximal cornerness and moves towards features with lower cornerness until it reaches the end of the input vector (which we got in step 4) or until enough new feature points have been added (as specified by the maximum allowed number of feature points). It is possible to choose automatically the faster minimum-distance-enforcement algorithm (standard vs. alternative), based on the minimum distance, the number of input features and the maximal number of features.

Feature Point Tracking Implementation

Unlike the feature detection, all the tracking steps have been packed inside only one complex kernel, so that a significant part of the data could be read once and then kept in the shared memory. In each pyramid level, each thread does the calculations for exactly one feature point.

Since bilinear interpolation is essential for achieving sub-pixel accuracy in the tracking algorithm, texture memory has been used to store the image pyramids for the images *I* and *J*. This allows to achieve sub-pixel reads at the cost of a normal read access.

One of the most critical optimizations was to reduce the number of necessary texture fetches, especially in the most inner loop of the algorithm, where the mismatch vector *b* is being calculated, as those are the most time-consuming operations.

Another important issue in the optimization process involves minimizing both shared memory and register usage, while not allowing the compiler to place often used variables in the local memory space, which has a very high access latency just like global memory.

Also, finding the best compromise between the number of registers per thread block, the amount of shared memory used and the number of threads per block is very troublesome and requires a lot of experiments. In most cases a good idea is to set the number of threads per block to 128 or 256 as those configurations allow the full utilization of multiprocessors. For the tracking kernel this number had to be reduced as each thread requires a lot of shared memory and registers on its own. Furthermore one should remember that in many cases the overall number of features to track, and therefore threads, is relatively low, like less than a few thousands, so the number of threads per block should be reduced even more. For example if there are a thousand threads in the tracking kernel, it's not efficient to set it to 256 as there would be only four blocks for four multiprocessors. In that case a GPU like the Geforce 280GTX, which has 30 multiprocessors, would use only 13% of its computing resources, as a single thread block can never be split between different multiprocessors.

Note that in contrast to the GPU KLT implementations presented in [Sin06] and [Zac08], we do not skip specific levels of the image pyramid. Furthermore, we do a convergence test after every iteration instead of employing a fixed number of iterations.

6. EXPERIMENTS AND RESULTS

This section describes experiments done with the CUDA KLT implementation. We compare the CUDA implementation with the corresponding function in the OpenCV library in terms of quality and speed. Note that the OpenCV library internally uses the Intel Performance Primitives (IPP) library and OpenMP for performance reasons. The runtime measurements were done on a 2.4 GHz Intel Xeon Quad-Core machine, equipped with a NVIDIA Geforce GX280 GPU.

Quality Tests

In Figure 3 a comparison of the detected points by both routines is given. One sees that there are neither green nor red points, indicating that the CUDA implementation of feature detection gives the same feature points as the OpenCV routine.

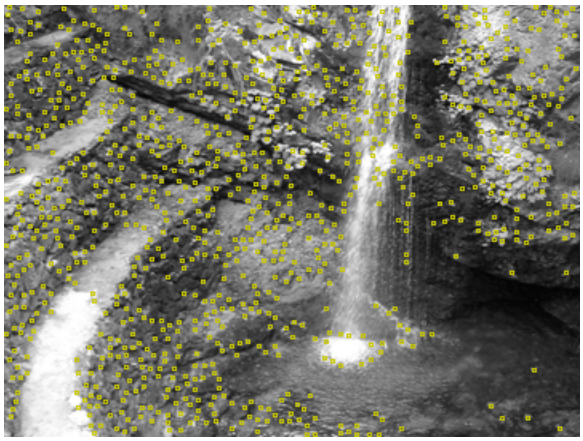

Figure 3: Quality results: Features detected by: Red = OpenCV, Green = CUDA, Yellow = Both.

The results of the tracking algorithm are shown in Figure 4. For each feature point its estimated motion vector is drawn in. Only a small percentage of the feature points has different motion vectors. These differences might arise mainly due to the usage of a more precise 5 x 5 Gaussian convolution kernel for creating the image pyramids in the CUDA implementation. For most correctly tracked feature points both implementations give similar results.

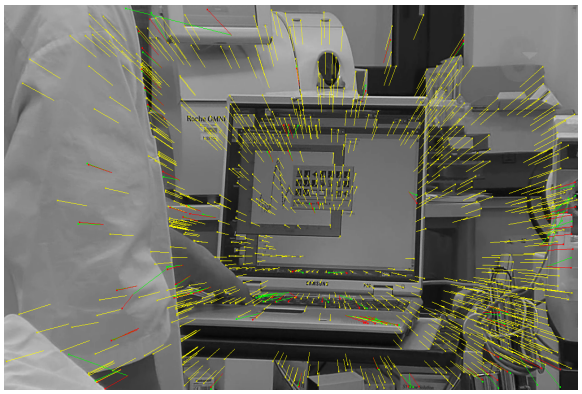

Figure 4: Quality results: Features tracked by: Red = OpenCV, Green = CUDA, Yellow = Both.

Runtime Tests

The various parts of the KLT algorithm depend on different parameters. All the tests were done using the parameters from Table 2, if not specified otherwise. Note that HD 1080p denotes an image resolution of 1920 x 1080 pixels, HD 720p denotes 1280 x 720 pixels and SD 720 x 576 pixels.

Video format:	HD 1080p (1920x1080)
Maximum # features:	10000
Quality level:	5 %
Minimum distance:	6 pixel
Window size (detection):	5 x 5
Enforce min. dist. algorithm:	Automatic
Pyramid levels:	6
Accuracy threshold	0.03 pixel
Maximum iterations:	10
Window size (tracking):	5 x 5

Table 2: Default parameters used for experiments.

Figure 5 shows the runtime for the first four steps of the feature detection algorithm for different image resolutions. Experiments have shown that the runtime is practically independent on any parameters apart from the image resolution.

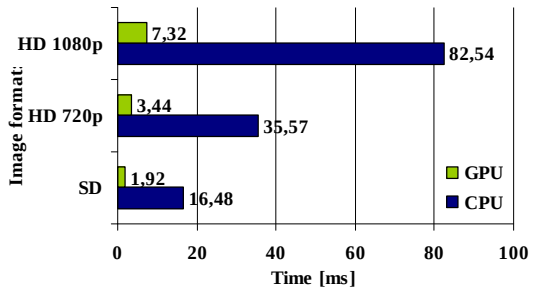

Figure 5: Runtime of the feature point detection (without minimum-distance-enforcement) for different image resolutions.

The runtime for the enforcement of the minimum distance is shown in Figure 6. It depends mainly on the number of features.

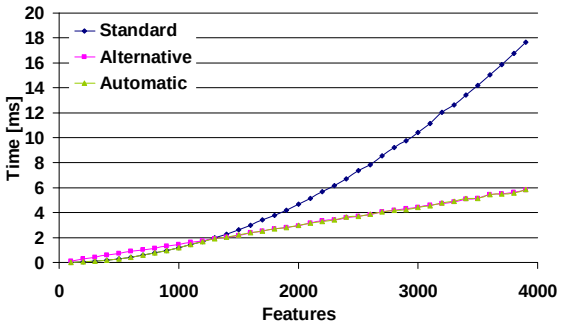

Figure 6: Runtime of the minimum distance enforcement

The runtime measurements in Figures 7 and 8 show the feature point tracking results for SD and HD 1080p material for different numbers of feature points.

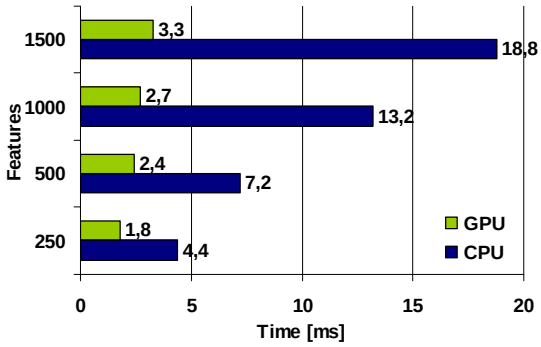

Figure 7: Runtime of the feature point tracking for different number of features for SD (720x576).

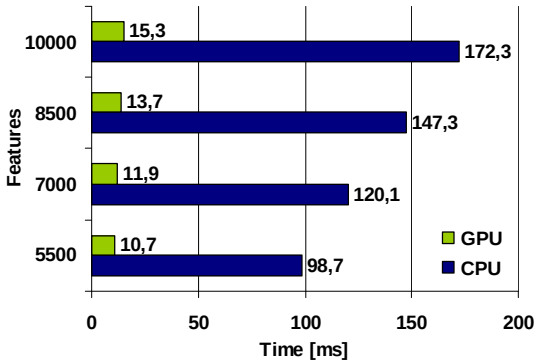

Figure 8: Runtime of the feature point tracking for different number of features for HD 1080p (1920x1080).

Some experiments were done with different window sizes for the feature point tracking and are shown in Figure 9. One can see that increasing the window size results in a significant increase in runtime, so one should use the smallest possible window size. For

most cases, a window size of 5 x 5 should suffice for feature point tracking.

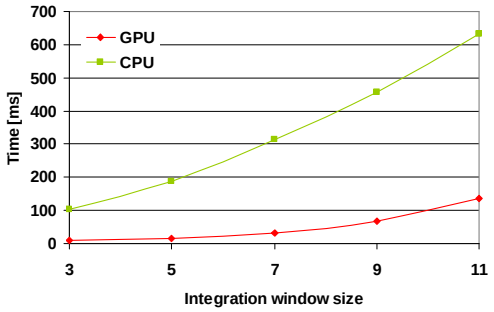

Figure 9: Runtime of the feature point tracking for different window sizes for HD 1080p (1920x1080).

Finally, Figure 10 shows the overall runtime of the KLT algorithm (feature point detection & tracking).

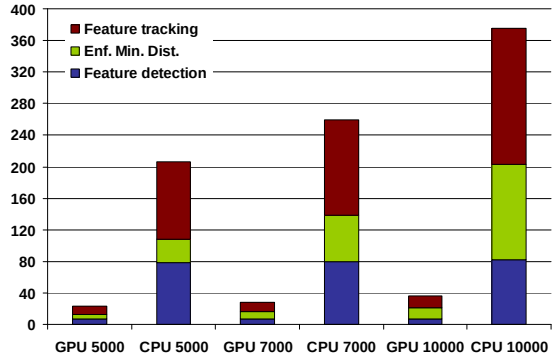

Figure 10: Overall runtime for different numbers of feature points for HD 1080p (1920x1080).

Overall, the CUDA implementation achieves a speedup of approximately 5 – 10. The speedup is higher for larger images and more feature points. This might be due to better utilization of the GPU's processing capabilities.

7. CONCLUSION

The well known KLT algorithm was ported to the GPU using CUDA. Experiments were done which show that the GPU implementation has the same quality as the corresponding CPU (OpenCV) routine, but runs significantly faster (approximately 5 to 10 times). The usage of the GPU makes it possible to track several thousands of feature points on Full-HD material in realtime (>25fps).

8. ACKNOWLEDGMENTS

This work has been funded partially under the 7th Framework program of the European Union within the project "2020 3D Media – Spatial Sound and Vision" (ICT-FP7-215475) and by the Austrian Federal Ministry for Transport, Innovation and Technology within the project "VAN-DAL".

9. REFERENCES

[Che08] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, K. Skadron, A performance study of general-purpose applications on graphics processors using CUDA, Journal of Parallel and Distributed Computing, 2008

[Kan06] N.K. Kanhere, S.T. Birchfield, W. A. Sarasua, Vehicle Segmentation and Tracking in the Presence of Occlusions, Transportation Research Board Annual Meeting, 2006

[Koc99] R. Koch, B. Heigl, M. Pollefeys, L. V. Gool, H. Niemann, Calibration of Hand-held camera sequences for plenoptic modeling, Proceedings of the International Conference on Computer Vision, pp. 585-591, 1999

[Luc81] B.D. Lucas, T. Kanade, An Iterative Image Registration Technique with an Application To Stereo Vision, Joint Conference on Artificial Intelligence, pp. 674-679, 1981

[Lyp07] Y. Lypetskyy, Robust pedestrian detection and tracking in crowded scenes, Proceedings of the SPIE, Volume 6764, 2007

[Ohm08] J. Ohmer, N. Redding, GPU-Accelerated KLT Tracking with Monte-Carlo-Based Feature Reselection, Digital Image Computing: Techniques and Applications, 2008

[Tom91] C. Tomasi, T. Kanade, Detection and Tracking of Point Features, Technical Report CMU-CS-91-132, Carnegie Mellon University, 1991

[Sin08] S. Sinha, J. Frahm, M. Pollefeys, Y. Genc, GPU-Based Video Feature Tracking and Matching, Technical Report 06-012, Department of Computer Science, UNC Chapel Hill, 2006

[Shi94] J. Shi, C. Tomasi, Good Features to Track, IEEE Conference on Computer Vision and Pattern Recognition, pp. 593-600, 1994

[Zac08] C. Zach, D. Gallup, J.M. Frahm, Fast gain-adaptive KLT tracking on the GPU, CVPR Workshop on Visual Computer Vision on GPUs, 2008