KLT TRACKING IMPLEMENTATION ON THE GPU

Johan Hedborg, Johan Skoglund and Michael Felsberg

Computer Vision Laboratory Department of Electrical Engineering,
Linköping University, Sweden,
email: hedborg@isy.liu.se, mfe@isy.liu.se

ABSTRACT

The GPU is the main processing unit on a graphics card. A modern GPU typically provides more than ten times the computational power of an ordinary PC processor. This is a result of the high demands for speed and image quality in computer games.

This paper investigates the possibility of exploiting this computational power for tracking points in image sequences. Tracking points is used in many computer vision tasks, such as tracking moving objects, structure from motion, face tracking etc. The algorithm was successfully implemented on the GPU and a large speed up was achieved.

Index Terms— Tracking, KLT, GPGPU, GPU

1. INTRODUCTION

The algorithm that is implemented on the GPU is the Lukas-Kanade feature tracking [1]. It was first presented 1981, and together with Tomasi-Kanade [2] it is called the KLT tracker. The KLT method is over 10 years old, but is still the most commonly used method for feature tracking in computer vision. It is efficient to compute and well suited for real time applications. The method requires no knowledge of the image data before tracking. In this paper the focus is on the tracking part of the algorithm, the problem of finding good features to track is examined in [3]

2. MATHEMATICAL DERIVATION

To briefly explain the KLT it can be said that it minimizes the Euclidean distance between two image patches by a gradient search. To further understand how the search works, we start by investigating the one dimensional case. The two dimensional case is analog and only the result is presented here.

2.1. The continuous one dimensional case

Define the error between the curves I and J in the domain w as:

$$\epsilon(h) = \int_w \Big(I(x+\frac{h}{2}) - J(x-\frac{h}{2})\Big)^2 dx.$$

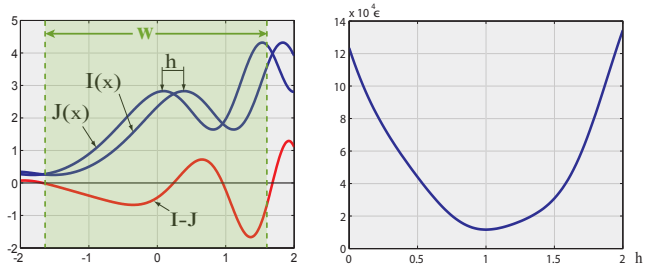

Fig. 1. Left: The curves I, J displaced with h and their error $I - J$. Right: The error ϵ for different h

Taylor expansion of I and J around x gives:

$$I(x+\frac{h}{2}) = I(x) + \frac{h}{2}I'(x) + O(h^2)$$

$$J(x-\frac{h}{2}) = J(x) - \frac{h}{2}J'(x) + O(h^2),$$

And their derivatives are Taylor expanded as:

$$I'(x+\frac{h}{2}) = I'(x) + \frac{h}{2}I''(x) + O(h^2)$$

$$J'(x-\frac{h}{2}) = J'(x) - \frac{h}{2}J''(x) + O(h^2).$$

To minimize the error $\epsilon(h)$, its derivative is set to 0:

$$\begin{aligned}\frac{d\epsilon}{dh} &= \int_w \Big(I(x) + \frac{h}{2}I'(x) - J(x) + \frac{h}{2}J'(x) + O(h^2)\Big) \\ &\times \Big(I'(x) + \frac{h}{2}I''(x) + J'(x) - \frac{h}{2}J''(x) + O(h^2)\Big)dx \\ &= 0.\end{aligned}$$

This gives us the equation system:

$$\begin{aligned}&\int_w \big(I(x) - J(x)\big)\big(I'(x) + J'(x)\big)dx \\ &= \frac{h}{2}\int_w \big(I(x) - J(x)\big)\big(I''(x) - J''(x)\big) \\ &+ \big(I'(x) + J'(x)\big)^2 dx + O(h^2). \qquad (1)\end{aligned}$$

We can simplify the expression by deleting the term $(I(x) - J(x))(I''(x) - J''(x))$. This done mainly due to three reasons:

- The term gets very small compared to $(I'(x) + J'(x))^2$ when h goes to 0.
- The term is computationally heavy.
- If h is large it is common to use scale pyramids to reduce the resolution. In the lower resolution h gets small enough again to make the term redundant.

If the term is omitted and we introduce the definition $g(x) = I'(x) + J'(x)$ and:

$$e = 2\int_w (I(x) - J(x)) \cdot g(x)dx, \qquad Z = \int_w g(x)^2 dx$$

We get from (1) the equation: $Zh = e$, and obtain our translation h by simply dividing with Z.

2.2. The two dimensional case

The procedure for two dimensions is analogue to the one above and the result is:

$$\mathbf{Z} = \int\int_w \begin{bmatrix} g_x^2 & g_x \cdot g_y \\ g_y \cdot g_x & g_y^2 \end{bmatrix} d\mathbf{p}$$
$$\mathbf{e} = 2 \cdot \int\int_w \begin{bmatrix} (I-J)g_x \\ (I-J)g_y \end{bmatrix} d\mathbf{p}$$

The translation vector is obtained by solving the equation system $\mathbf{Zh} = \mathbf{e}$.

2.3. The discrete case

To process discrete images the equations are made discrete by changing the integration to a summation:

$$\begin{bmatrix} \sum\sum_w g_x^2 & \sum\sum_w g_x \cdot g_y \\ \sum\sum_w g_y \cdot g_x & \sum\sum_w g_y^2 \end{bmatrix} \cdot \begin{bmatrix} d_x \\ d_y \end{bmatrix}$$
$$= 2 \cdot \begin{bmatrix} \sum\sum_w (I-J)g_x \\ \sum\sum_w (I-J)g_y \end{bmatrix}$$

3. IMPLEMENTATION

Figure 2 is a schematic overview of the different parts in the GPU implementation. The six different steps form one iteration step. All these steps are done on the GPU.

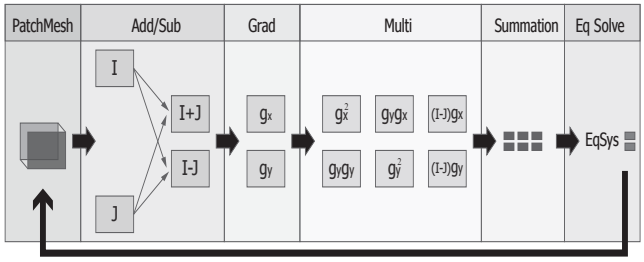

Fig. 2. The six steps in the GPU implementation (the six steps are also commented below)

1. Everything that is going to be computed on a GPU has to be in the form of a geometrical model (mesh), and the patch is modeled as a simple square mesh. The mesh has dual texture coordinate systems to handle both patch I and patch J.
2. The difference and sum of the patches are then calculated. The result is saved in two new patches $I_p + J_p$ and $I_p - J_p$.
3. In this render pass a gradient or sobel filtering of the $I_p + J_p$ is done resulting in g_x and g_y.
4. In this pass g_x , g_y and $I_p - J_p$ are multiplied in different ways to form the six shown result patches.
5. Here a summation over the six products are done, reducing them to scalars.
6. Then the 2x2 equation system is solved resulting in a translation vector which is used to update the patch position. With other words: updating the coordinate system on the square mesh. Now the algorithm is ready for another iteration.

4. PERFORMANCE ANALYSIS

When computing the KLT on the GPU, the calculations can be divided into five types of GPU computations.

- **Sampling linearly interpolated pixels** is a very suitable subproblem because it has no performance penalties what so ever. In this case the data is two dimensional and the GPU saves 3 pixel sampling calls.
- Convolution like the **sobel or gradient filtering** The GPU has a much smaller cache then the CPU but has a lot more memory bandwidth so this kind of filter operations with a small kernel and fairly big images are very well suited for the GPU.
- The **element wise operations** of the patches are very fast when the treated data is large enough. In these cases the CPU has little use of its fast but small memory cache.
- The **summation** can be done in hardware and is thereby very fast. The hardware implementation is normally used to generate scale pyramids used in 3D renderings to avoid aliasing.

4.1. Further performance issues

A key to performance in GPU applications is to be able to collect big chunks of data and process them in the same way. For the KLT it is therefore very important to gather several patches and do the computations in parallel on them, like in figure 3 where patches are packeted together to form a grid.

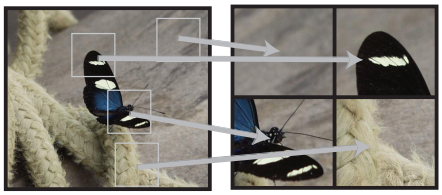

Fig. 3. A patch grid for the GPU

To get good performance, data arrays that are at least 512x512 big should be used. This can be achieved if a grid of 16x16 patches (32x32 pixels big) are used. The implementation has been further optimized by merging some of the passes.

5. RESULT

There is an open source CPU-implementation of the KLT tracker [4] made a couple of years ago by a group at the university of Standford. This implementation was used to get an estimate of the performance of the CPU implementation. What should be mentioned here is that this is not an optimal implementation in the aspect of SSE2 instruction set and other ways to speed up CPU implementations. The GPU implementation can do about 300 000 32x32 patch iterations per second if 256 patches are calculated in parallel. The number for 16x16 patches moves to 750 000 if 1024 patches are handled in parallel. The CPU implementation handles 10 000 patch iterations per second, if the patch size is 16x16.

6. DISCUSSION

The implementation has not been compared with a optimized CPU-implementation and it is hard to say how much faster tracking on the GPU becomes. A qualified guess would be that it is somewhere between 10 and 15 times faster. This is due to the many well suited algorithms for the GPU, as convolution with a smaller kernel and sampling pixels with bilinear interpolation.
Searching in scale spaces and scale pyramids are also well suited for a GPU, because it is designed to sample pixels in these kinds of image structures when mapping images to 3D models. The GPU can interpolate linearly between two scale levels in a scale pyramid almost for free. This gives the possibility to do the gradient search in between scales. Extending the translation model to include rotation, scaling or a full affine transformation is also well suited for the GPU. This is manly because the GPU is constructed to read pixels in any affine transformation from the memory. When adding more degrees of freedom to the tracker, the data to calculate grows exponentionally and the GPU:s raw power could come in even more use.

7. ACKNOWLEDGMENTS

This work has been supported by EC Grant IST-2003- 004176 COSPAL. This paper does not represent the opinion of the European Community, and the European Community is not responsible for any use which may be made of its contents.

8. REFERENCES

[1] B.D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *IJCAI81*, 1981, pp. 674–679.

[2] Carlo Tomasi and Takeo Kanade, "Shape and motion from image streams: A factorization method," Tech. Rep. CMU-CS-91-105, Carnegie Mellon University, School of Computer Science, Jan. 1991.

[3] Jianbo Shi and Carlo Tomasi, "Good features to track," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'94)*, Seattle, June 1994.

[4] Stan Birchfield, "KLT: An implementation of the kanade-lucas-tomasi feature tracker," .