PERFORMANCE OF OPTICAL FLOW TECHNIQUES ON GRAPHICS HARDWARE

Marko Durkovic, Michael Zwick, Florian Obermeier, Klaus Diepold

Lehrstuhl für Datenverarbeitung
Technische Universität München
Arcisstr. 21, 80333 München, Germany

ABSTRACT

Since graphics cards have become programmable the recent years, numerous computationally intensive algorithms have been implemented on the now called *General Purpose* Graphics Processing Units (GPGPUs). While the results show that GPGPUs regularly outperform CPU based implementations, the question arose how optical flow algorithms can be ported to graphics hardware. To answer the question, the optimal algorithm structure to maximize the performance gained by using graphics cards has to be found.

In this paper we compare the performance of two algorithms that are highly different in structure, implemented on both CPU and graphics hardware. Analyzing the results of the CPU and GPGPU implementation, we explore the mapping of the algorithms to the graphics hardware and thereof extract information about a preferred structure of optical flow algorithms for GPGPU based implementation.

1. INTRODUCTION

General Purpose Graphics Processing Units (GPGPUs) have undergone an enormous evolution lately. Reaching manufacturing limitations, graphics processor designers concentrated their efforts on enhancing performance by parallelizing their architecture instead of just raising clock speed. As an outcome, today's graphics processors implement up to 24 parallel pixel pipelines and this number will increase further.

Due to the fierce competition in the graphics processing market, graphics processor manufacturers were forced to outperform their competitors by fancy add-ons, which led to the implementation of programmable pixel and vertex pipelines. Actually introduced to offer highly flexible graphics processors to game developers, other fields soon discovered the programmable and highly optimized now called *General Purpose* Graphics Processing Units for their applications.

Simply comparing the number of execution units of an up-to-date GPGPU to those of a CPU demonstrates the computational power of modern GPGPUs: 8 vertex and 24 pixel pipelines with 2 execution units each, resulting in 64 GPGPU execution units versus 8 execution units of up-to-date Desktop CPUs (including SSE or AltiVec units).

Taking into account that every GPGPU execution unit operates on 4 dimensional vectors boosts the superiority of GPGPUs to a theoretical factor of $64 \cdot 4/8 = 32$, assuming that CPU and GPGPU manufacturers both squeezed out performance from gate delays equally well.

To analyze the performance gain achieved in practice, we implemented two well known gradient based optical flow algorithms, namely those of Horn and Schunck (H&S) [1] and Lucas and Kanade (L&K) [2] on both a CPU (AMD Athlon64 3500+, 512 kB Cache, 2 GB RAM) and a GPGPU (Nvidia GeForce 6800 Ultra) and tested them with some video sequences. Both algorithms were implemented with the modifications proposed by Barron et al. [3].

Barron's results and the continous popularity of the choosen algorithms justify their selection despite their age. They differ in the constraint necessary to calculate the motion component orthogonal to the direction of the gradient. More recent proposals such as [4] focus rather on enhancing this constraint than starting from ground up. To that end they tend to base their considerations on Lucas and Kanade. Insofar it is valid to compare the constraints' influence on the achievable performance gain using GPGPUs.

Results are presented in Chapter 2 and analyzed in Chapter 3.

2. RESULTS

We measured the execution times of our implementations and normalized them to a number of 100 frames. Afterwards we calculated the relative performance gain

$$P_{rel} = \frac{\textit{Execution time CPU}}{\textit{Execution time GPGPU}} \quad (1)$$

achieved by a GPGPU based implementation compared to a CPU based one (table 1).

The test sequence *blue_sky* can be downloaded from *ftp://ftp.e-technik.tu-muenchen.de/pub/testsequences*, Foreman is an MPEG-4 test sequence [5], while all other mentioned test sequences are identical to the ones used in [3]. The resolution of each test sequence is given in table 1.

Figure 1 prints out the relative performance gain with respect to the resolution of the test sequences. Since GPGPU and CPU implementations produce identical vector fields,

		$\|\|t_{L\&K}\|\|_{100}$	$\|\|t_{H\&S}\|\|_{100}$	$P_{rel,L\&K}$	$P_{rel,H\&S}$
Blue_sky	G	16.00	460.04	12.40	2.78
1920x1080	C	198.41	1276.78		
Foreman	G	0.87	23.49	11.64	2.75
352x288	C	10.13	64.65		
Nasa	G	1.57	20.86	6.11	2.85
300x300	C	9.59	59.41		
Rubic	G	1.86	15.62	3.63	2.56
256x240	C	6.76	39.93		
Taxi	G	1.85	12.39	3.05	2.65
256x190	C	5.65	32.86		
Square2	G	1.48	3.67	1.54	1.66
100x100	C	2.28	6.09		

Table 1. Execution times and relative performance gains.

correlating GPGPU execution time to CPU execution time yields an appropriate performance measure.

Please note, that the optical flow fields we obtained matched those of [3]. For comparision in accuracy refer to that paper and [6].

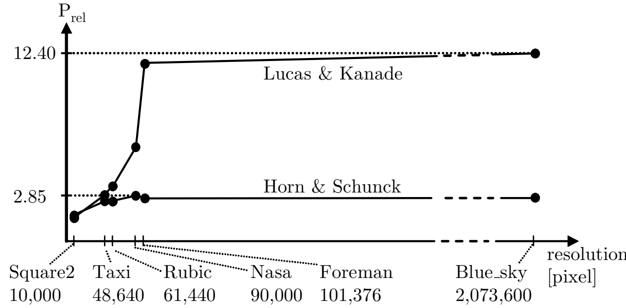

Fig. 1. Execution times and relative performance gain.

Analyzing the results given in table 1 and figure 1 we realized the following:

1. Lucas and Kanade's algorithm achieves a much higher performance gain (a factor of about 4.5 for high resolutions) than Horn and Schunck's Algorithm does.

2. The performance gain of Horn and Schuck's algorithm quickly reaches a kind of saturation and doesn't exceed 2,85.

3. The implementation of Lucas and Kanade's algorithm behaves in a quite different way: The performance gain increases exponentially with resolution, as long as the image size doesn't exceed CIF format. For higher resolutions the performance gain of Lucas and Kanade's algorithms saturates as well and does not exceed a value greater than 12.40 at 1080p (1920 x 1080).

3. ANALYZING THE RESULTS

In the following section we analyze the behavior of the performance gain.

3.1. Higher performance gain of Lucas and Kanade's algorithm compared to Horn and Schunck's algorithm

The higher performance gain of Lucas and Kanade's algorithm can be explained by the structure of the algorithm: Lucas and Kanade's algorithm allows a highly parallel processing using the GPGPU's parallel pixel shaders, while Horn and Schuck's algorithm is an iterative one and therefore not a good candidate for parallelizing.

Horn and Schunck calculated the velocity field $\mathbf{v}(\mathbf{x}, t) = [u(\mathbf{x}, t), v(\mathbf{x}, t)]^T$ by minimizing

$$\int_R (\nabla I \cdot \mathbf{v} + I_t)^2 + \lambda^2(\|\nabla u\|_2^2 + \|\nabla v\|_2^2)d\mathbf{x} \qquad (2)$$

within a region R, where $I = I(\mathbf{x}, t)$ depicts the intensity of the pixel at position (x, y) at the time t and ∇I the spatial gradient $\left[\frac{\partial I}{\partial x} \ \frac{\partial I}{\partial y}\right]^T$; λ is a smoothness factor [3]. This minimization is commonly referred to as *global smoothness constraint*.

According to [3] we determined $\mathbf{v}$ in (2) iteratively by

$$u_{n+1} = \frac{\overline{u}_n - I_x \cdot (I_x \overline{u}_n + I_y \overline{v}_n + I_t)}{\alpha^2 + I_x^2 + I_y^2} \qquad (3)$$

$$v_{n+1} = \frac{\overline{v}_n - I_y \cdot (I_x \overline{u}_n + I_y \overline{v}_n + I_t)}{\alpha^2 + I_x^2 + I_y^2}. \qquad (4)$$

In order to achieve the high performance offered by graphics hardware, equations (3) and (4) have to be partitioned into blocks that can be executed in parallel. Due to architectural constraints, parallel processes can not share temporal data or results with each other. Therefore, sharing results requires the setup of a new render cycle introducing overhead.

Since $\overline{u}_n$ and $\overline{v}_n$ are calculated from neighbouring values, every iteration step requires a separate render cycle. By writing the results of each rendering cycle into a texture, they are made available to all shaders in the following iteration.

We calculated the derivates I_x and I_y not in the way proposed by Horn and Schuck [1] but rather chose the 4-point central differences with mask coefficients $\frac{1}{12}(-1, 8, 0, -8, 1)$ as it was done by Barron, Fleet and Beauchemin [3]. We calculated the derivates for every pixel and stored them as illustrated in figure 2 in a texture referred to as *Derivatives_Texture*. Storing data in textures is the common way to supply pixel and vertex shaders with required data.

Fig. 2. Calculation of the derivates.

For the same reason we generated a second texture referred to as *Vels_Texture* to store the computed velocities.

Starting with an estimation of $u_0 = v_0 = 0.0$ we iteratively (100 times) first smoothed the actual *Vels_Texture* by applying the filter shown in figure 3 to achieve $\overline{u}_k$ and $\overline{v}_k$.

1/12	1/6	1/12
1/6	0	1/6
1/12	1/6	1/12

Fig. 3. Filter for smoothing the calculated velocities.

Afterwards we calculated u_{k+1} and v_{k+1} by

$$\begin{bmatrix}u_{k+1}\\ v_{k+1}\\ \#\\ 0\end{bmatrix} = \begin{bmatrix}\overline{u}_k\\ \overline{v}_k\\ 1.0\\ 0\end{bmatrix} - \frac{[\overline{u}_k,\overline{v}_k,1.0,0.0]\begin{bmatrix}I_x\\ I_y\\ I_t\\ 0.0\end{bmatrix}}{[I_x,I_y,\alpha,0.0]\begin{bmatrix}I_x\\ I_y\\ \alpha\\ 0.0\end{bmatrix}}\begin{bmatrix}I_x\\ I_y\\ I_t\\ 0.0\end{bmatrix}, \quad (5)$$

whereas $\#$ denotes an irrelevant value.

Although (5) is a nice vector equation and therefore might be ideally suited for GPGPU calculation (and executes thereon indeed up to 2.85 times faster compared to a CPU), the calculation has a severe disadvantage compared to Lucas and Kanade's algorithm: The high number of 100 iterations makes the performance suffer from overhead. Every time the loop iterates to calculate $\mathbf{v}_{k+1}$ from $\overline{\mathbf{v}}_k$ the GPGPU's internal control registers have to be setup. This is due to the fact that the source and the target textures are switched before another iteration as demonstrated by figure 4.

Unlike Horn and Schuck's algorithm Lucas and Kanade's solution is not an iterative one and therefore does not suffer from the afore mentioned expensive overhead. The solution to Lucas and Kanade's minimization problem ([2], [3])

$$\sum_{\mathbf{x}\in\Omega} W^2(\mathbf{x})[\nabla I(\mathbf{x},t)\cdot\mathbf{v} + I_t(\mathbf{x},t)]^2, \quad (6)$$

is simply obtained by

$$\mathbf{v} = \mathbf{M}^{-1}\mathbf{B}, \quad (7)$$

where $\mathbf{M}$ depicts the 2×2-matrix

$$\mathbf{M} = \begin{bmatrix}\Sigma W^2(\mathbf{x})I_xI_x & \Sigma W^2(\mathbf{x})I_xI_y\\ \Sigma W^2(\mathbf{x})I_yI_x & \Sigma W^2(\mathbf{x})I_yI_y\end{bmatrix} \quad (8)$$

and $\mathbf{B}$ the twodimensional vector

$$\mathbf{B} = \begin{bmatrix}\nabla I(\mathbf{x}_1)\\ \vdots\\ \nabla I(\mathbf{x}_n)\end{bmatrix}^T \begin{bmatrix}W(\mathbf{x}_1) & & \\ & \ddots & \\ & & W(\mathbf{x}_n)\end{bmatrix} \begin{bmatrix}-I_t(\mathbf{x}_1)\\ \vdots\\ -I_t(\mathbf{x}_n)\end{bmatrix}. \quad (9)$$

Equation (7) is also known as *local smoothness constraint*.

Determining I_x, I_y and I_t in the same way as depicted in figure 2, we created two textures $Texture_I_{xy}$ and $Texture_I_{xyt}$. In every pixel of $Texture_I_{xy}$ we stored the associated 4 element vectors $[I_xI_x, I_xI_y, I_yI_x, I_yI_y]^T$ and in $Texture_I_{xyt}$ the associated vectors $[I_xI_t, I_yI_t, I_xI_t, I_yI_t]^T$.

According to [7] we applied a twodimensional weighting function with the kernel elements $\frac{1}{16}(1,4,6,4,1)$ on these textures. This was done in the same manner as the spatial (not temporal) filtering of *frame i* in figure 2.

Fig. 4. Switch between source and target texture.

After the filtering process every pixel of $Texture_I_{xy}$ contains in its 4 element vector the matrix elements

$\mathbf{M}[0,0], \mathbf{M}[0,1], \mathbf{M}[1,0]$ and $\mathbf{M}[1,1]$, i.e. the complete matrix $\mathbf{M}$. Accordingly, $Texture_I_{xyt}$ contains the matrix $\mathbf{B}$.

Calculating $\mathbf{M}^{-1}$ by

$$\mathbf{M}^{-1} = \frac{1}{det\mathbf{M}} \begin{bmatrix} \mathbf{M}[1][1] & -\mathbf{M}[1][0] \\ -\mathbf{M}[0][1] & \mathbf{M}[0][0] \end{bmatrix} \quad (10)$$

makes it possible to calculate $\mathbf{v}$ as a sequence of matrix-vector-operations that have to be executed for every pixel only *once* and not iteratively as it was done by the algorithm of Horn and Schunck. Therefore there is no overhead from iteratively setting up the GPGPU's control registers, which makes Lucas and Kanade's algorithm by far the better choice for GPGPU based implementations.

3.2. Performance gain saturation and cache influences

For small resolutions, the achievable performance gain of a GPGPU implementation is comparatively small for both algorithms. This is due to the relatively high overhead, which is caused by setting up new render cycles and the data transfers from RAM to GPGPU. For increasing resolution, the fraction of time to handle the overhead diminishes and the performance gain becomes more significant.

Above that, the CPU performance is closely related to its cache hit rate. While the resolution stays small, the hit rate is high. For higher resolutions the amount of memory accesses gets boosted resulting in a disproportionate set back of CPU performance. Once the CPU is in saturation, it can not benefit from its cache any further. In contrast, the GPGPU could not profit from its very small texture cache in the first place, so that it is not affected by such a kind of performance breakdown.

Once the resolution exceeds a specific limit, the execution times the CPU, respectively the GPGPU require for optical flow computation, nearly scale linearly. From there, the relative performance gain of the GPGPU implementation almost remains the same for further increasing resolutions (figure 1).

Our implementation calculates four optical flow fields per second for 1080p on the described hardware. Most recent GPGPUs are equipped with 48 pixel pipelines and up to four of these units can be employed in one computer simultanously. The resulting amount of 192 pixel pipelines is 12 fold what we used, each running at a higher clocking frequency. Further optimization and use of state of the art hardware will allow real-time computation of optical flow for 1080p image resolution.

4. CONCLUSION

This report presents a way to compute optical flow on General Purpose Graphics Processing Units. Our investigations proved that optical flow can be computed on GPGPUs. Without loosing accuracy, the optical flow is obtained in a fraction of the execution time of a CPU implementation.

Because of its non-iterative structure, the algorithm of Lucas and Kanade is particularly suitable. Compared to the local smoothness constraint of Lucas and Kanade the performance penalty imposed on Horn and Schunck's algorithm[1] by the global smoothness constraint is already known to be significant in straight-forward CPU implementations. The situation gets worse, when both algorithms are ported to a GPGPU: the execution time of Lucas and Kanade's algorithm could be reduced to less than 1/12, compared to more than a 1/3 for the algorithm of Horn and Schunck.

Making use of the tremendous processing power of modern parallel hardware allows for more sophisticated constraints as well as additional steps, such as pre- and postprocessing, formerly considered too expensive.

5. ACKNOWLEDGMENT

The authors would like to thank NVIDIA Germany for their contribution of a NVIDIA GeForce 5900 for our first steps in GPGPU programming.

6. REFERENCES

[1] B. K. P. Horn and B. G. Schunck, "Determining optical flow," Tech. Rep. A.I. Memo No. 572, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, April 1980.

[2] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proceedings of Image Understanding Workshop*, 1981, pp. 121–130.

[3] J.L. Barron, D.J. Fleet, and S.S. Beauchemin, "Performance of optical flow techniques," *International Journal of Computer Vision*, vol. 12, no. 3, pp. 43–77, 1994.

[4] M. Irani, "Multi-frame correspondence estimation using subspace constraints," *International Journal of Computer Vision*, vol. 48, pp. 173–194, 2002.

[5] T. Alpert, V. Baroncini, L. Choi, D. Contin, R. Koenen, F. Pereira, and H. Peterson, "Subjective evaluation of mpeg-4 video codec proposals: Methodological approach and test procedures," in *Signal Processing: Image Communication 9*, 1997, pp. 305–325.

[6] B. Galvin, B. McCane, K. Novins, D. Mason, and S. Mills, "Recovering motion fields: An evaluation of eight optical flow algorithms," 1998.

[7] Adelson E. H. Simoncelli E. P. and Heeger D. J., "Probability distributions of optical flow," *IEEE Proc. of CVPR*, pp. 310–315, 1991.

[1] severely influenced by limited memory bandwidth