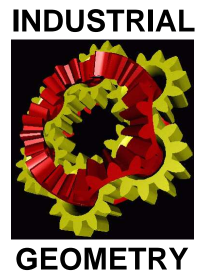
Forschungsschwerpunkt S92

Industrial Geometry

http://www.ig.jku.at



FSP Report No. 68

GPU-based Multigrid: Real-Time Performance in High Resolution Nonlinear Image Processing

H. Grossauer and P. Thoman

May 2008

FWF

Der Wissenschaftsfonds.

GPU-based Multigrid: Real-Time Performance in High Resolution Nonlinear Image Processing

Harald Grossauer and Peter Thoman*

Department of Computer Science
Universität Innsbruck
Technikerstraße 21a
A-6020 Innsbruck, Austria
harald.grossauer@uibk.ac.at
peter.thoman@uibk.ac.at
http://infmath.uibk.ac.at

Abstract. Multigrid methods provide fast solvers for a wide variety of problems encountered in computer vision. Recent graphics hardware is ideally suited for the implementation of such methods, but this potential has not yet been fully realized. Typically, work in that area focuses on linear systems only, or on implementation of numerical solvers that are not as efficient as multigrid methods. We demonstrate that nonlinear multigrid methods can be used to great effect on modern graphics hardware. Specifically, we implement two applications: a nonlinear denoising filter and a solver for variational optical flow. We show that performing these computations on graphics hardware is between one and two orders of magnitude faster than comparable CPU-based implementations.

Key words: GPGPU, multigrid methods, optical flow, partial differential equations

1 Introduction

Many important building blocks of computer vision algorithms, like denoising and dense optical flow calculation, necessitate solving complex systems of nonlinear Partial Differential Equations (PDEs). *Multigrid methods* [12, 18, 4, 5] belong to the fastest and most versatile schemes available for the numerical solution of such systems. In addition to their fast convergence, multigrid methods have another advantage: they can be composed of operations that are completely data-parallel and are thus able to benefit immensely from parallel hardware architectures.

Recent efforts in *General Purpose computation on Graphics Processing Units* (GPGPU) aim to use one ubiquitous type of such parallel hardware: modern graphics processors. However, despite the suitability of multigrid methods to

* Supported by FSP Project S9203-N12.

GPUs, there have only been a very limited number of publications attempting to combine the two [3, 11, 16]. Furthermore, multigrid methods for nonlinear systems – in the form of *Full Approximation Schemes* (FAS) [4] – have not appeared in the GPGPU literature at all.

In this paper, we demonstrate the possibilities offered by FAS on GPUs. We present in detail an implementation of variational optical flow computation on standard consumer graphics hardware which achieves rates of up to 17 dense high quality flow fields per second at a resolution of 511^2 pixels, more than 30 times as fast as comparable CPU-based implementations.

1.1 Background

GPGPU. Since the introduction of user-programmable shaders (see [13]) interest in using the vast computational capabilities offered by GPUs for tasks not directly related to graphics rendering has been on the rise. Modern graphics hardware is comprised of a large number of parallel floating point units paired with high-bandwidth memory and a cache structure optimized for 2D locality. Evidently, such a hardware architecture should be well suited to image processing.

Multigrid Methods. The mathematical basis of our work is provided by multigrid methods, in particular FAS since we deal with nonlinear problems. In this context we use the standard terminology as provided in Brandt [4] and Trottenberg et al. [18].

Nonlinear PDEs in Image Processing. To demonstrate the wide applicability of FAS schemes – and thus also of our GPGPU implementation – in computer vision we chose two common problems that lead to nonlinear PDEs. The first, relatively simple process is denoising using a nonlinear filter based on the model developed by Rudin, Osher, and Fatemi (ROF) [15]. We solve the resulting system of equations using a FAS scheme with a damped Jacobi smoother.

The second, far more complex and computationally intensive task is optical flow computation. The best models proposed in the literature [8] use variational formulations that lead to coupled PDEs with a large number of nonlinear terms. We will describe the mathematical foundation of such an approach in the next section, and discuss our GPU-based implementation of this algorithm after that.

Previous Work. Durkovic et al. [9] explored optical flow computation on GPUs, and demonstrated the performance advantages compared to CPUs. However, their work has two decisive drawbacks. Firstly, in terms of quality, the methods they implemented (the modified Horn-Schunck and Nagel algorithms presented in [2]) do not achieve results comparable to the most recent algorithms. Secondly, in terms of performance, they do not use a multigrid solver – these have been proven (in [7]) to give speedups of factor 200 to 2000 for optical flow calculation compared to standard iterative methods. In contrast, we implement one of the leading optical flow estimators in literature and use a multigrid method.

2 Optical Flow Model

In this section we review the variational model used in our implementation, which is essentially based on [8]. We also present some necessary adaptations of the numerical scheme to make the process more tractable on data-parallel architectures such as GPUs.

2.1 The Approach of Bruhn et al.

Let $I(\boldsymbol{x})$ be a presmoothed image sequence with $\boldsymbol{x} = (x, y, t)^T$. Here, $t \geq 0$ denotes the time while (x, y) denotes the location in the image domain Ω. Let $\boldsymbol{u} = (u, v, 1)^T$ be the unknown flow field between two frames at times t and $t+1$. Bruhn et al. [8] compute the optical flow as minimizer of the energy functional

$$E(\boldsymbol{u}) = E_{D_1}(\boldsymbol{u}) + \alpha E_{D_2}(\boldsymbol{u}) + \beta E_S(\boldsymbol{u}) \tag{1}$$

with the data and smoothness terms

$$E_{D_1}(\boldsymbol{u}) = \int_\Omega \psi(|I(\boldsymbol{x}+\boldsymbol{u}) - I(\boldsymbol{x})|^2)\, d\boldsymbol{x}\ , \tag{2}$$

$$E_{D_2}(\boldsymbol{u}) = \int_\Omega \psi(|\nabla I(\boldsymbol{x}+\boldsymbol{u}) - \nabla I(\boldsymbol{x})|^2)\, d\boldsymbol{x}\ , \tag{3}$$

$$E_S(\boldsymbol{u}) = \int_\Omega \psi(|\nabla u|^2 + |\nabla v|^2)\, d\boldsymbol{x} \tag{4}$$

and weighting parameters $\alpha, \beta \geq 0$. Thus the first data term E_{D_1} models the grey value constancy assumption, and the second data term E_{D_2} adds a constancy assumption of the spatial image gradient ∇I to improve robustness against varying illumination. The term E_S provides for spatial smoothness of the flow field. All three terms are modified by the non-quadratic penalizer

$$\psi(s^2) = \sqrt{s^2 + \epsilon^2}\ . \tag{5}$$

The Numerical Scheme. From the Euler-Lagrange equations corresponding to the energy functional outlined above, Bruhn et al. derive the finite difference approximation

$$\begin{aligned} 0 = {} & \Psi_{D_1 i}(S_{11i} du_i + S_{12i} dv_i + S_{13i}) + \alpha \Psi_{D_2 i}(T_{11i} du_i + T_{12i} dv_i + T_{13i}) \\ & - \beta \sum_{j \in \mathcal{N}(i)} \frac{\Psi_{Si} + \Psi_{Sj}}{2} \frac{u_j + du_j - u_i - du_i}{h^2}\ , \end{aligned} \tag{6}$$

$$\begin{aligned} 0 = {} & \Psi_{D_1 i}(S_{12i} du_i + S_{22i} dv_i + S_{23i}) + \alpha \Psi_{D_2 i}(T_{21i} du_i + T_{22i} dv_i + T_{23i}) \\ & - \beta \sum_{j \in \mathcal{N}(i)} \frac{\Psi_{Si} + \Psi_{Sj}}{2} \frac{v_j + dv_j - v_i - dv_i}{h^2} \end{aligned} \tag{7}$$

with the symmetric tensors

$$S := \boldsymbol{I}_\nabla {\boldsymbol{I}_\nabla}^T\ , \tag{8}$$

$$T := \boldsymbol{I}_{\nabla x} {\boldsymbol{I}_{\nabla x}}^T + \boldsymbol{I}_{\nabla y} {\boldsymbol{I}_{\nabla y}}^T \tag{9}$$

and abbreviations

$$\Psi_{D_1} := \Psi((du, dv, 1)^T S(du, dv, 1)) \,, \tag{10}$$
$$\Psi_{D_2} := \Psi((du, dv, 1)^T T(du, dv, 1)) \,, \tag{11}$$
$$\Psi_S := \Psi(|\nabla(u + du)|^2 + |\nabla(v + dv)|^2) \tag{12}$$

containing nonlinear terms. In (6) and (7) the set $\mathcal{N}(i)$ denotes the spatial neighbours of pixel i. In (8) and (9) we use the definitions $\boldsymbol{I}_\nabla := (I_x, I_y, I_z)^T$, $\boldsymbol{I}_{\nabla x} := (I_{xx}, I_{yx}, I_{zx})^T$ and $\boldsymbol{I}_{\nabla y} := (I_{xy}, I_{yy}, I_{zy})^T$, with subscripts x and y denoting spatial derivatives and $I_z := I(\boldsymbol{x} + \boldsymbol{u}) - I(\boldsymbol{x})$.

2.2 Adaptations of the Numerical Scheme

Instead of using a Gauss-Seidel method (as in [8]) to solve the nonlinear system of equations given by (6) and (7) we adopt a damped Jacobi solver, retaining the coupled point relaxation and frozen coefficients approach proposed in [10]. While offering slightly better convergence rates, a standard Gauss-Seidel solver can not be efficiently implemented on current GPUs because it accesses values computed in the current iteration step and is thus not fully data-parallel. An alternative would be a Gauss-Seidel solver with Red/Black ordering of grid points (see [18]), but the access pattern required by such a method would greatly reduce the GPU cache efficiency. As the main purpose of the iterative solver in a multigrid application is reducing high-frequency error components, using a damped Jacobi scheme does not significantly decrease the efficiency of the algorithm (see [5]).

Using a point coupled damped Jacobi method the system of equations that must be solved for each pixel i at iteration step n is given by

$$\begin{pmatrix} du^{n+1} \\ dv^{n+1} \end{pmatrix} = \frac{1}{3} \begin{pmatrix} du^n \\ dv^n \end{pmatrix} + \frac{2}{3} (M^n)^{-1} \begin{pmatrix} r_u^n \\ r_v^n \end{pmatrix} \tag{13}$$

with matrix entries

$$M_{11}^n = \Psi_{D_1 i}^n S_{11i}^n + \alpha \Psi_{D_2 i}^n T_{11i}^n + \beta \sum_{j \in \mathcal{N}(i)} \frac{\Psi_{Si}^n + \Psi_{Sj}^n}{2h^2} \,, \tag{14}$$

$$M_{22}^n = \Psi_{D_1 i}^n S_{22i}^n + \alpha \Psi_{D_2 i}^n T_{22i}^n + \beta \sum_{j \in \mathcal{N}(i)} \frac{\Psi_{Si}^n + \Psi_{Sj}^n}{2h^2} \,, \tag{15}$$

$$M_{12}^n = M_{21}^n = \Psi_{D_1 i}^n S_{12i}^n + \alpha \Psi_{D_2 i}^n T_{12i}^n \tag{16}$$

and right hand side

$$r_u^n = -\Psi_{D_1 i}^n S_{13i}^n + \alpha \Psi_{D_2 i}^n T_{13i}^n + \beta \sum_{j \in \mathcal{N}(i)} \frac{\Psi_{Si}^n + \Psi_{Sj}^n}{2} \frac{u_j + du_j^n - u_i}{h^2} \,, \tag{17}$$

$$r_v^n = -\Psi_{D_1 i}^n S_{23i}^n + \alpha \Psi_{D_2 i}^n T_{23i}^n + \beta \sum_{j \in \mathcal{N}(i)} \frac{\Psi_{Si}^n + \Psi_{Sj}^n}{2} \frac{v_j + dv_j^n - v_i}{h^2} \,. \tag{18}$$

2.3 Our Multigrid Algorithm

Our multigrid algorithm is derived by reformulating (6) and (7) as

$$A^h(\boldsymbol{x}^h) = \boldsymbol{f}^h \tag{19}$$

with h denoting the discretization width, A^h a nonlinear operator and the vectors $\boldsymbol{x}^h = ((du^h)^T, (dv^h)^T)^T$ and $\boldsymbol{f}^h = ((f_1^h)^T, (f_2^h)^T)^T$. We use a standard FAS approach, with (13) serving as a smoother. The grid hierarchy is created by standard coarsening with full weighting as restriction operator and bilinear interpolation as prolongation method.

It is important to note that, unlike Bruhn et al. [8], we use V-cycles. While W-cycles or other variations may offer slightly better convergence rates, they require significantly more computations at very small grids where the parallelization advantages of GPUs are reduced, as shown in Table 3.

3 Optical Flow GPU Implementation

Implementing an algorithm on the GPU requires mapping the necessary data structures to textures and reformulating the computation as a series of pixel shader applications. When aiming to develop a high-performance GPU implementation, special attention must be paid to the flow and storage of data through the individual computational steps. We will now describe the data structures and the most important shader programs used in our implementation.

Data Structures. The number of texture reads and buffer writes has a significant impact on the throughput of a shader program. For this reason, it is very advantageous to pack related data values into 4-component (RGBA) textures to minimize the amount of read and write accesses to different buffers required in the shaders.

Our final implementation uses four buffers that are exclusive to the finest grid level, storing $[I; J]$, $[I_x; I_y; J_x; J_y]$, $[I_z; I_{xz}; I_{yz}]$ and $[I_{xx}; I_{xy}; I_{yy}]$ with $J := I(\cdot, \cdot, t+1)$. Furthermore, on each grid level ten buffers are required. Three of them store the symmetric tensors S and T (with six independent components each), three are general purpose buffers used for different types of data throughout the computation, and the last four store $[du; dv; un; vn]$, $[\Psi_{D_1}; \Psi_{D_2}; \Psi_S]$, $[f_1; f_2; u; v]$ and $[A(u); A(v)]$. These values correspond to the formulas provided in Section 2, except for $un := u + du$ and $vn := v + dv$, which are stored to speed up the computation and greatly reduce the number of texture reads required.

Shader Programs. The actual computation of our GPU implementation takes place in shader programs, the six most important of which we will now describe. There are a few helper programs used in our application, but those only implement basic mathematical operations or are used to move and restructure data.

– `calcIdJd` calculates the spatial derivatives of the source images.

- `calcIzIxzIyz` computes the differences between the first image relocated by the current flow field and the second image.
- `calcST` calculates the tensors S and T.
- `calcdpsi` calculates the nonlinearities Ψ_{D_1}, Ψ_{D_2} and Ψ_S.
- `calcAuAv` is used to compute the residual and right hand side for each coarser grid level. It calculates $A(u)$ and $A(v)$.
- `coupledJacobi` is the most complex shader program, performing one step of point-coupled damped Jacobi relaxation.

Computation. We shall now illustrate how the shader programs and data objects described above are used to perform the optical flow calculation. Figure 1 shows the computations required before starting each V-cycle of the multigrid solver. The spatial derivatives of the images are calculated, and the tensors S and T are computed from them. These operations are only performed at the finest grid level – the tensors are subsequently restricted to the coarser grids. In the following we will call a cycle of the algorithm that includes the recomputation of these tensors an *outer cycle*.

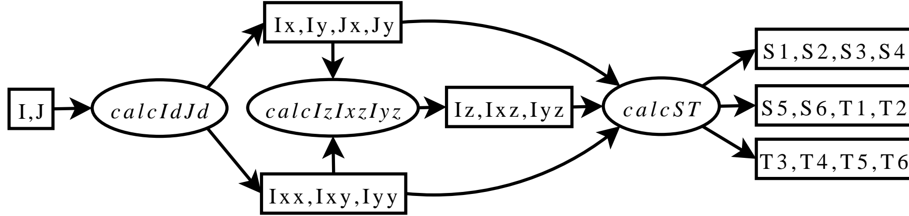


Fig. 1: Computations performed before starting each V-Cycle. Boxes represent data buffers, ellipses denote pixel shaders.

Pre- and postsmoothing operations at each level of the hierarchy follow the outline provided in Fig. 2. Note that this depiction is somewhat simplified: GPUs do not allow writing to an input buffer, therefore 2 alternating buffers are used for $[du; dv; un; vn]$.

At each grid level, after some steps of presmoothing have been applied, `calcAuAv` is used in a similar manner to `coupledJacobi` above. However, the resulting values $[A(u); A(v)]$ are required to compute the residual, which is subsequently restricted to a coarser grid level. There, f_1 and f_2 are calculated and the cycle is restarted at this coarser grid. Once the computation is complete, the correction is prolongated to the next finer grid and a number of postsmoothing steps are performed.

4 Results

The focus of our optical flow implementation is to achieve high performance, however doing so is meaningless unless the high quality of the flow field provided

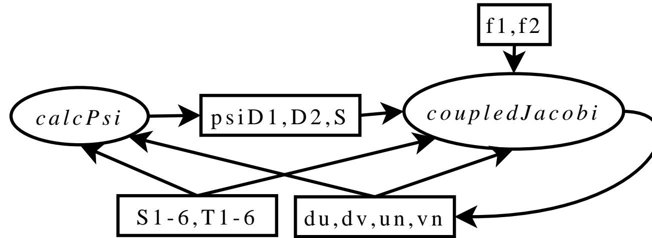


Fig. 2: Applying one smoothing iteration on the GPU. *S1-6,T1-6* is an abbreviated form which actually represents 3 separate buffer objects.

by the original algorithm of Bruhn et al. [8] is maintained. To evaluate this aspect we measure the average angular error (see [2]) achieved by our method on the well-known *Yosemite* test sequence.

Table 1 shows the qualitative performance of our algorithm and compares it to some of the best results that can be found in literature. **GPU** refers to our implementation using ten outer cycles composed of two V-cycles each and demonstrates the optimum accuracy possible using our approach, without real-time capability at high resolutions. **GPU-RT** uses five outer cycles with a single V-cycle each. This setting represents a good trade-off between accuracy and real-time performance. In both cases two pre- and postsmoothing iterations are used. The weighting parameters α and β do not change the execution time, but have a significant impact on the quality of the result and have to be chosen carefully depending on the scene and computation method. For **GPU** we use $\alpha = 0.1$ and $\beta = 0.08$, for **GPU-RT** $\alpha = 0.15$ and $\beta = 0.06$.

Table 1: Accuracy compared to results from literature. Table adapted from Bruhn et al. [8] AAE = average angular error STD = standard deviation.

Method	AAE	STD
Horn-Schunck, mod. [2]	9.78	16.19
Uras et al. [2]	8.94	15.61
Alvarez et al. [1]	5.53	7.40
Mémin-Pérez [14]	4.69	6.89
GPU-RT	**3.48**	**7.75**
GPU	**2.65**	**7.12**
Brox et al. [6]	2.46	7.31
Bruhn et al. [8]	2.42	6.70

The slight loss in accuracy compared to Bruhn et al. can be explained by a variety of factors. In terms of the numerical scheme, they use a fourth-order approximation for spatial derivatives, while we use a second-order approximation. Also, on GPUs, our computations are limited to single-precision (32 bit) float-

ing point values. Finally, the built-in hardware linear texture filtering we use to determine values of $I(\boldsymbol{x}+\boldsymbol{u})$ is not as accurate as a software implementation on the CPU.

Despite these minor drawbacks, we have established that a GPU-based solver can come very close in quality to the best results found in literature. We will now move on to examining the quantitative performance of our method.

All times in the following tables have been measured on a standard desktop PC with a Geforce 8800 GTX graphics card. Table 2 shows that our implementation achieves rates of more than 17 dense flow fields per second at a resolution of 511 pixels squared. In terms of flow vectors per second, this is a 35-fold speedup compared to the results shown by Bruhn et al. using their CPU-based implementation. Note however that this gain comes with a slight decrease in the quality of the result.

Table 2: Time required and FPS achieved for GPU-RT optical flow.

Resolution (pixels)	Time (ms)	FPS
255^2	33.192	30.128
511^2	57.409	17.419
1023^2	206.107	4.852

One surprising fact that can be seen in Table 2 is that calculating a 255^2 flow field is only about twice as fast as calculating a field with four times as many flow vectors. This means that while the computational effort increases nearly fourfold, the computation also gets twice as efficient on the larger grid. Conversely, switching over from 511^2 to 1023^2 causes the theoretically expected change of a factor of four.

To better understand this behaviour, Table 3 shows the performance of our implementation on image sequences different sizes. Evidently, at grids up to and including 255^2, *our algorithm is not completely limited by GPU performance.* Rather, external factors like driver interaction and housekeeping operations seem to dominate the computation time, as it increases at a nearly constant pace with the number of grid levels required. The time per pixel values support this notion: realizing the full potential performance of the GPU implementation is only possible at image sizes of 511^2 and beyond.

The largest drawback of this performance profile is that it renders the Full Multigrid (FMG) algorithm (see [18]) much less attractive, as that method requires a higher number of cycles at very small grid sizes to achieve its theoretical performance advantage. To weaken the impact of this overhead and parallelization problem, one interesting option is the development of a combined GPU/CPU solving strategy that performs the computations on small grids on the CPU. For simple linear problems this was shown to be effective in [17].

To demonstrate the versatility of the FAS on GPU approach we would like to add some results for the denoising filter described in Section 1.1. For this

Table 3: Performance at various resolutions with 6 pre-/postsmoothing iterations. *Outer Cycle*s contain a single V-Cycle.

Grid Levels	Resolution (pixels)	V-cycle (ms)	Outer cycle (ms)	Time/Pixel (μs)
1	1^2	0.325	0.519	519.350
2	3^2	1.509	1.916	212.867
3	7^2	2.686	3.337	68.101
4	15^2	3.957	4.478	19.902
5	31^2	5.193	6.126	6.375
6	63^2	6.193	7.566	1.906
7	127^2	7.751	9.045	0.561
8	255^2	8.979	10.520	0.162
9	511^2	20.465	22.785	0.087
10	1023^2	76.942	84.388	0.081

algorithm, we measure 14.2 ms for filtering 511^2, and 49.8 ms for 1023^2 RGBA color images. This translates to 80 megapixels per second of single-component throughput and enables the real-time filtering of high-resolution color video streams with spare performance for subsequent processing. Note that, due to the multigrid approach used, these computation times are much less dependent on the filtering strength than explicit Euler time stepping or standard iterative implicit solvers.

5 Summary and Conclusions

By implementing a state-of-the-art optical flow algorithm on graphics hardware and achieving unprecedented performance, we have demonstrated that multigrid solvers for nonlinear PDEs are well suited to data-parallel architectures like GPUs. We achieved a 35-fold speedup with only small losses in accuracy compared to the best CPU-based implementations in literature. Additionally, we implemented a fast ROF-based denoising filter as another example showing the applicability of GPU-based FAS to image processing.

We hope that our contributions enable further development in two distinct ways: Firstly, the real-time application of complex nonlinear filters to high-resolution video streams should prove beneficial in the field of computer vision. Here, using mostly the graphics hardware and keeping CPU cycles free for subsequent processing is an additional advantage. Secondly, any research in using GPGPU techniques for solving nonlinear systems of equations should also be encouraged by our findings.

References

1. Alvarez, L., Weickert, J. and Sanchéz, J.: Reliable estimation of dense optical flow fields with large displacements. International Journal of Computer Vision, 39(1):41–56, 2000.

2. Barron, J.L., Fleet, D.J. and Beauchemin, S.S.: Performance of Optical Flow Techniques. International Journal of Computer Vision, 12(1):43–77, 1994.
3. Bolz, J., Farmer, I., Grinspun, E. and Schroeder, P.: Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. ACM Transactions on Graphics, 22:917924, July 2003.
4. Brandt, A.: Multi-Level Adaptive Solutions to Boundary-Value Problems. Mathematics of Computation, Vol. 31, No. 138 (Apr., 1977), pp. 333–390
5. Briggs, W. L., Henson, V. E. and McCormick, S. F.: A Multigrid Tutorial: Second Edition. Society for Industrial and Applied Mathematics, 2000.
6. Brox, T., Bruhn, A., Papenberg, N. and Weickert, J.: High accuracy optical flow estimation based on a theory for warping. In Proc. 8th European Conference on Computer Vision, Prague, Czech Republic, 2004.
7. Bruhn, A., Weickert, J., Kohlberger, T. and Schnrr, C.: A Multigrid Platform for Real-Time Motion Computation with Discontinuity-Preserving Variational Methods. Int. J. Comput. Vision 70, 3 (Dec. 2006), 257-277.
8. Bruhn, A. and Weickert, J.: Towards Ultimate Motion Estimation: Combining Highest Accuracy with Real-Time Performance. In Proc. 10th IEEE international Conference on Computer Vision (Iccv'05) Volume 1. IEEE Computer Society, Washington, DC, 749-755.
9. Durkovic, M., Zwick, M., Obermeier, F. and Diepold, K.: Performance of Optical Flow Techniques on Graphics Hardware. 2006 IEEE International Conference on Multimedia and Expo, 9–12:241–244.
10. Frohn-Schnauf, C., Henn, S. and Witsch, K.: Nonlinear Multigrid Methods for Total Variation Denoising. Computation and Visualization in Science, 7(3–4):199–206, 2004.
11. Goodnight, N., Woolley, C., Lewin, G., Luebke, D. and Humphreys, G.: A Multigrid Solver for Boundary Value Problems using Programmable Graphics Hardware. Siggraph 2003, In Proceedings of SIGGRAPH 102-111.
12. Hackbusch, W.: Multigrid methods and applications. Springer, 1985.
13. Lindholm, E., Kilgard, M. and Moreton, H.: A user-programmable vertex engine. Siggraph 2001, In Proceedings of SIGGRAPH 149-158.
14. Mémin, E. and Pérez, P.: A multigrid approach for hierarchical motion estimation. In Proc. 6th International Conference on Computer Vision, Bombay, India, 1998.
15. Osher, S., Rudin L.I. and Fatemi, E.: Nonlinear total. variation based noise removal algorithms. Physica D,. vol. 60, pp. 259-268, 1992.
16. Rehman, T., Pryor, G. and Tannenbaum, A.: Fast Multigrid Optimal Mass Transport for Image Registration and Morphing. British Machine Vision Conference, 2007.
17. Thoman, P.: GPGPU-based Multigrid Methods. Master Thesis, University of Innsbruck, 2007.
18. Trottenberg, U., Oosterlee, C. W. and Schüller, A.: Multigrid. Academic Press, San Diego, 2001.