Realtime 3D Motion Estimation on Graphics Hardware

Jens Rannacher

Heidelberg University, Germany

Abstract. 3D motion estimation in image sequences is a major task in artificial intelligence systems such as robot navigation or driver assistance systems. This involves the estimation of the optical flow between consecutive image frames and also the 3D geometry of the scene. Several solution techniques for this problem have already been presented in the literature. They can be divided into feature and energy-based methods. Feature-based methods provide sparse results and are preferred in time-critical applications because of their computational benefits. Energy-based methods usually provide more accurate and dense results but are considered to be unusable in realtime applications since they require the solution of large systems of nonlinear equations. In this work a highly ranked dense method for 3D motion estimation from stereo images is analyzed and modified for realtime realization. The derived algorithm is implemented on programmable graphic hardware via the parallel computing environment CUDA, to achieve dense and accurate velocity fields at 30 Hz on 640×480 images.

1 Introduction

1.1 Variational Methods for Motion Estimation

Estimating the 3D motion out of image sequences is a fundamental building block of various computer vision based systems. The task of motion estimation can be stated as follows: From two consecutive images of a video sequence a displacement vector field must be estimated that transforms each pixel of one image to the new position in the second image. In other words this vector field describes the projection of the apparent 3D motion of objects onto the 2D image plane caused by relative motion to the camera or by illumination changes. In the literature this motion field is also referred to as *optical flow*.

Analogously, *scene flow* is the 3D vector field that describes the motion of every visible point in the scene. Unlike optical flow, scene flow cannot be estimated from mono camera images without a priori informations (due to the projection onto the image plane and the resulting ambiguities). However, using images from a *stereo camera* resolves most ambiguities that arise by non-rigid or relative motion between the camera and the objects in the scene. With a stereo camera the scene flow can be recovered by combining the optical flow between consecutive stereo frames and the *disparity* (stereo) between the left and right

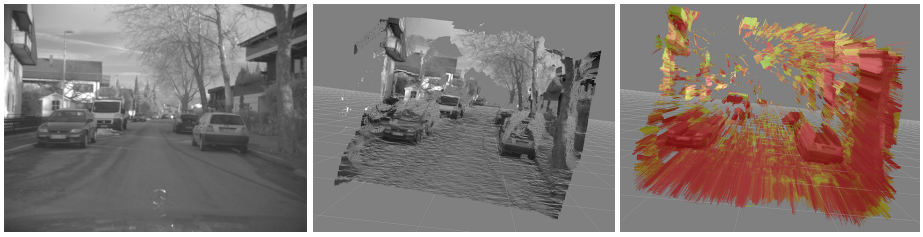

Fig. 1. Scene flow example: (1) Input image (2) 3D geometry (3) 3D motion

images. The disparity is neccessary to calculate the position of an image point in the 3D space.

Both motion estimation and stereo matching can be interpreted as a correspondence problem and can be solved using a variational framework. Compared to local methods this approach includes a global smoothness assumption and therefore allows for estimating dense motion fields. Nevertheless, variational methods are very expensive in terms of computational cost. Typically they require the solution of large systems of nonlinear equations and in general their performance is far from realtime. This prevents their use in realtime systems like driver assistance systems where computational speed is a critical aspect.

To reduce the computational complexity the scene flow estimation is split into two seperate steps (as in [1]), while the consistency of the solution ist still ensured. These two steps consist of estimating the disparity and the projection of the 3D motion (optical flow + disparity change). For the sake of convenience, in the following sections, the optical flow together with the disparity change is denoted as scene flow, since the motion of a point in 3D space can be reconstructed by these measurements.

While the two sub-problems can be solved much more efficiently, the CPU implementation of [1] has still only near-realtime capabilty on small image resolutions with reduced accuracy. But as shown in this work, in combination with further extensions the algorithm of [1] can be accelerated on modern graphics hardware to achieve realtime performance for images of size 640×480 pixels.

1.2 Graphics Hardware as Parallel Computer

Since 2003 the performance improvement of general-purpose microprocessors has slowed significantly due to power consumption issues that limit the increase of the clock frequency. At the same time *graphic processing units* (GPUs) have continuously improved their floating-point performance and memory bandwidth, so that current graphic chips deliver up to 1,000 gigaflops performance and more than 100 GB/s memory bandwith.

The large performance gap between GPUs and CPUs can be explained by the differences in the design of both architecures. While CPUs are optimized for sequential code performance and therefore provide sophisticated control logic and large cache memories, GPUs are designed for compute-intensitive and highly parallel computations such as graphics rendering. Hence much more transistors

are devoted to data processing rather than data caching and flow control. As a consequence one has to consider several optimization strategies to reach the promised performance:

- The algorithms have to be structured in such a way that they expose as much data parallelism as possible. This ensures that all arithmetic units of the GPU are kept busy and no ressources are wasted.
- Fetching data from GPU main memory is about an order of magnitude slower than executing a simple floating point operation. Hence, for exploiting the computational power data transfers should be minimized. Sometimes, the best optimization might even be to recompute intermediate results and avoid any data transfer.
- To maximize the memory bandwith all memory accesses have to be organized according to optimal memory access patterns. Subject to these access patterns, for example four 32 bit float values can be read in one 128 bit access from memory, what is obviously faster than four serialized 32 bit accesses.

Therefore algorithms with high arithmetic intensity (typical in variational methods) can be implemented more efficiently on GPUs than bandwidth-intensive ones. Moreover, in connection with scene flow estimation many computational steps can be done in parallel for all image points. These are fundamental prerequisites to exploit the computational power of graphics cards.

To further improve the performance of the algorithm, the optimization strategies from [2] are applied, as described in section 3. The derived algorithm is implemented on programmable graphics hardware via the parallel computing environment CUDA. CUDA allows direct programming of GPUs using a high-level language. It is choosen for this work, since it is much more flexible than other high-level shading languages like Cg. To make this paper self-contained, the next section explains the starting point for the developed modifications of the algorithm presented in [1].

2 Scene Flow Estimation

2.1 From 2D to 3D

Consider two consecutive stereo images at times t and $t+1$. For every point $(x, y, d)^T$ scene flow provides a displacement vector $(u, v, d')^T$ in image space $\Omega \in \mathbb{R}^2$. Here, d is the disparity between the left and right image at time t and d' is the disparity change between times t and $t+1$. As in [1] the algorithm requires a pre-computed disparity map. This means only little effort, since stereo algorithms are available on dedicated hardware [3].

With known position $(x, y, d)^T$ and displacement $(u, v, d')^T$, the 3D point and its new position at time $t+1$ can be reconstructed according to

$$\begin{pmatrix} X_t \\ Y_t \\ Z_t \end{pmatrix} = \frac{b}{d} \begin{pmatrix} x - x_0 \\ y - y_0 \\ f \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} X_{t+1} \\ Y_{t+1} \\ Z_{t+1} \end{pmatrix} = \frac{b}{d + d'} \begin{pmatrix} x + u - x_0 \\ y + v - y_0 \\ f \end{pmatrix}, \qquad (1)$$

with the focal length f and the base-length b between the two camera projection centres. Those are camera intrinsic parameters and are supposed to be known. The challenge is now to estimate the projection of the scene flow onto the 2D image space, namely the triplet $(u, v, d')^T$.

2.2 Scene Flow Constraints

A common starting point for optical flow estimation is to assume that pixel intensities are translated from one image to the next, while the intensity values remain constant,

$$I_t(x, y) = I_{t+1}(x + u, y + v). \tag{2}$$

Here, $I_t(x, y)$ denotes the image intensity as a function of space $(x, y)^T$ and time t, and $(u, v)^T$ is the optical flow, i.e. the 2D velocity between consecutive images.

For scene flow calculation this simple assumption (also known as *optical flow constraint*) is augmented by further constraints that describe the geometric relationship between corresponding pixels in the stereo frames. Assuming that $I_t^l(x, y)$ and $I_t^r(x, y)$ are the intensity values of the left and right images, respectively, equation (2) can be rewritten as

$$I_t^l(x, y) = I_{t+1}^l(x + u, y + v). \tag{3}$$

Since the stereo images are rectified, two corresponding pixels in the left and right images ideally lie on the same pixel line y. The rectification step is a simple transformation of the input images and can be done before the scene flow calculation. Hence, there is the following constraint for the optical flow between the right images:

$$I_t^r(x + d, y) = I_{t+1}^r(x + d + d' + u, y + v). \tag{4}$$

This second constraint is actually redundant for solving the problem, because $I_t^l(x, y) = I_t^r(x + d, y, t)$. However, the overdetermination makes the algorithm more robust to interference such as illumination changes between both cameras.

Finally, a third constraint is introduced that enforces consistency between the left and right images at time $t + 1$ in case of illumination changes or occlusions,

$$I_{t+1}^l(x + u, y + v) = I_{t+1}^r(x + d + d' + u, y + v). \tag{5}$$

2.3 Variational Approach

Let R^l, R^r and R^d be the image gray-value residuals. Then, equations (3) to (5) can be rewritten as follows:

$$\begin{aligned} R^l &:= I_t^l(x, y) - I_{t+1}^l(x + u, y + v) = 0 \\ R^r &:= I_t^r(x + d, y) - I_{t+1}^r(x + d + d' + u, y + v) = 0 \\ R^d &:= I_{t+1}^l(x + u, y + v) - I_{t+1}^r(x + d + d' + u, y + v) = 0. \end{aligned} \tag{6}$$

Integrating the above constraints over the image domain, one obtains the following data term:

$$E_{\text{Data}} = \int_{\Omega} \Big[\lambda_1 \, \psi\big(R^l\big) + c(x,y) \, \lambda_2 \, \psi\big(R^r\big) + c(x,y) \, \lambda_3 \, \psi\big(R^d\big)\Big] \mathrm{d}x\mathrm{d}y, \quad (7)$$

where λ_1, λ_2 and λ_3 regulate the importance of the different constraints. The robust error function $\psi(x) = \sqrt{x^2 + \epsilon}$, with $0 < \epsilon \ll 1$, approximates the classical Huber norm, that penalizes small residuals quadratically and large residuals only linearly but without the need of handling two separate cases. The function $c(x, y)$ returns the value 0 if there is no disparity known at $(x, y)^T$ or the value 1 otherwise. This depends on the presents of occlusions or the usage of a sparse stereo algorithm.

The optical flow cannot be estimated in areas without structure. To achieve dense results, one needs to use some kind of regularization. The additional smoothness term penalizes local deviations in the scene flow and allows for propagation of information over large distances in the image,

$$E_{\text{Smooth}} = \int_{\Omega} \Big[\psi\big(|\nabla u|\big) + \psi\big(|\nabla v|\big) + \psi\big(|\nabla d'|\big)\Big] \mathrm{d}x\mathrm{d}y, \quad (8)$$

with $\nabla = (\partial/\partial x, \partial/\partial y)$ and the same error function ψ as above. This allows for dense scene flow estimates, even if the disparity d is not known at some points.

Both, data and smoothness terms, are combined to an energy functional,

$$E(u, v, d') = E_{\text{Data}}(u, v, d') + E_{\text{Smooth}}(u, v, d'). \quad (9)$$

To obtain a solution for (u, v, d') one minimizes energy functional (9) by firstly discretizing the integral as well as the derivatives and then seeking for a stationary point. This results in a large system of non-linear equations that may be solved through a Newton-type iteration, while the linear sub-systems are solved by fix-point methods such as successive over-relaxation (SOR) (like in [1]). Although SOR converges faster than simple iterative solvers such as the Jacobi method, it cannot be implemented very efficiently on graphics hardware, because of so-called *in-place* operations. In this work a completely different strategy is choosen to minimize the energy more efficiently. This is described in the next section.

3 Total Variation Scene Flow

3.1 Total Variation Norm

As mentioned above, the error function $\psi(x) = \sqrt{x^2 + \epsilon}$ limits the influence of constraints with larger errors. However, for small x and ϵ the derivative is still nearly singular, due to $\psi'(x) = x/\sqrt{x^2 + \epsilon}$, while for larger ϵ the properties of the model are lost. The total variation (respectively L_1) norm has proven to be a more appropriate choice, since it is not defective and large gradient features

such as edges are better preserved than by the L_2 norm. However, using the non differentiable error function $\psi(x) = |x|$ in the whole energy functional (equation (9)) would result in a minimization problem that is difficult to solve. Therefore, this function is only used in the smoothness term,

$$E_{\mathrm{Smooth}^{TV}} = \int_\Omega \Big[|\nabla u| + |\nabla v| + |\nabla d'| \Big] \mathrm{d}x\mathrm{d}y, \tag{10}$$

because there exist efficient schemes for minimizing the total variation (e.g. [4]). This approach can be adopted to the optical flow case, as described in [2], and analogously to the scene flow case considered in this work.

3.2 Convex Approximation

The energy functional that has to be minimized now reads as follows:

$$E = E_{\mathrm{Data}}(\boldsymbol{u}) + E_{\mathrm{Smooth}^{TV}}(\boldsymbol{u}), \tag{11}$$

with the abbreviation $\boldsymbol{u} = (u, v, d')^T$ for the scene flow. As in [2] an auxiliary variable $\tilde{\boldsymbol{u}} = (\tilde{u}, \tilde{v}, \tilde{d'})$ is introduced yielding the following convex approximation of equation (11):

$$E_\theta = E_{\mathrm{Data}}(\boldsymbol{u}) + \frac{1}{2\theta}||\boldsymbol{u} - \tilde{\boldsymbol{u}}||^2 + E_{\mathrm{Smooth}^{TV}}(\tilde{\boldsymbol{u}}). \tag{12}$$

For $0 < \theta < 1$, $\boldsymbol{u}$ is a close approximation of $\tilde{\boldsymbol{u}}$, and for $\theta \to 0$ minimizing the above energy is equivalent to minimizing equation (11). Unlike before, the optimization problem has now become one in two variables $\boldsymbol{u}$ and $\tilde{\boldsymbol{u}}$ and can be minimized by alternatingly updating either $\boldsymbol{u}$ or $\tilde{\boldsymbol{u}}$ in every iteration. At first glance this decoupling looks like complicating things, but the two subproblems can be optimized more efficiently, especially on graphics hardware. The alternating minimization procedure can be described as follows:

1. For fixed $\tilde{\boldsymbol{u}}$, minimize in (12) with respect to $\boldsymbol{u}$,

$$\min_{\boldsymbol{u}} \int_\Omega \left[\frac{1}{2\theta}||\boldsymbol{u} - \tilde{\boldsymbol{u}}||^2 + \lambda_1 \, \psi\big(R^l\big) + c\,\lambda_2 \, \psi\big(R^r\big) + c\,\lambda_3 \, \psi\big(R^d\big) \right] \mathrm{d}x\mathrm{d}y. \tag{13}$$

   This minimization problem does not depend on spatial derivatives of $\boldsymbol{u}$ and can be solved pointwise. Since the functional in equation (13) is strictly convex, setting the first derivative to 0 is a sufficient condition for a global minimum

$$\frac{1}{\theta}\big(\boldsymbol{u} - \tilde{\boldsymbol{u}}\big) + \lambda_1 \frac{\partial R^l}{\partial \boldsymbol{u}} \frac{R^l}{\psi\big(R^l\big)} + c\,\lambda_2 \frac{\partial R^r}{\partial \boldsymbol{u}} \frac{R^r}{\psi\big(R^r\big)} + c\,\lambda_3 \frac{\partial R^d}{\partial \boldsymbol{u}} \frac{R^d}{\psi\big(R^d\big)} \;=\; 0. \tag{14}$$

   Replacing the displaced images by first-order Taylor approximations,

$$\begin{aligned} I^l_{t+1} \approx \tilde{I}^l_{t+1} + (\boldsymbol{u} - \tilde{\boldsymbol{u}})^T \, \nabla \tilde{I}^l_{t+1} \,, \quad & I^r_{t+1} = (x + u, y + v) \\ & \tilde{I}^l_{t+1} = (x + \tilde{u}, y + \tilde{v}) \end{aligned} \tag{15}$$

$$I^r_{t+1} \approx \tilde{I}^r_{t+1} + (\boldsymbol{u} - \tilde{\boldsymbol{u}})^T \nabla \tilde{I}^r_{t+1} \,, \quad \begin{aligned} I^r_{t+1} &= (x + d + d' + u, y + v) \\ \tilde{I}^r_{t+1} &= (x + d + \tilde{d}' + \tilde{u}, y + \tilde{v}) \end{aligned} \tag{16}$$

equation (14) can be rewritten as $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b}$, where $\boldsymbol{A}$ and $\boldsymbol{b}$ only depend on the fixed $\tilde{\boldsymbol{u}}$. Now the solution $\boldsymbol{u}$ can be determined directly by Gauss elimination. As $\boldsymbol{A}$ is symmetric some computation steps can be eliminated thus reducing computing time.

2. For fixed $\boldsymbol{u}$, minimize in (12) with respect to $\tilde{\boldsymbol{u}}$,

$$\min_{\tilde{\boldsymbol{u}}_i} \int_\Omega \left[\frac{1}{2\,\theta} (\boldsymbol{u}_i - \tilde{\boldsymbol{u}}_i)^2 + |\nabla \tilde{\boldsymbol{u}}_i| \right] \mathrm{d}x\mathrm{d}y, \tag{17}$$

where $\boldsymbol{u}_i$ is the i-th component of $\boldsymbol{u}, i \in \{1, ..., 3\}$. The solution of this problem is exactly the Rudin-Osher-Fatemi model [5] and can be solved efficiently with the algorithm presented in [2, 4]. Adopted to the scene flow case, the solution of equation (17) is given by

$$\tilde{\boldsymbol{u}}_i = \boldsymbol{u}_i + \theta \, \mathbf{div}\, \boldsymbol{p}. \tag{18}$$

The dual variable $\boldsymbol{p} = (p_1, p_2)^T$ is determined iteratively by

$$\tilde{\boldsymbol{p}}_{n+1} = \boldsymbol{p}_n + \frac{\tau}{\theta} \big(\nabla (\boldsymbol{u}_i + \theta \, \mathbf{div}\, \boldsymbol{p}_n) \big), \qquad \boldsymbol{p}_{n+1} = \frac{\tilde{\boldsymbol{p}}_{n+1}}{\max \big\{ 1, |\tilde{\boldsymbol{p}}_{n+1}| \big\}}, \tag{19}$$

with $n = 1, ..., N$, $\boldsymbol{p}_0 = (0, 0)^T$, and the step size $\tau \leq 1/4$ for which the algorithm converges in practise [2].

3. Repeat step 1 - 2 until a prescribed residual accuracy or a specific number of iterations is reached.

4 Implementation

The CUDA implementation of the modified scene flow algorithm is strongly influenced by the optimization strategies as presented in the first section. This includes minimizing the data transfers between the CPU and the GPU. Therefore most programm logic is assigned to the GPU and executed in terms of so called *CUDA kernels*. These Kernels are C functions, that, when called, are executed N times in parallel by M different *CUDA threads*. To achieve high parallelism each pixel in the image is mapped to a thread. Furthermore, the entire algorithm is distributed on several kernels, since the number of available registers and the shared memory per processor are limited. The resulting algorithm is summarized in figure 2 and explained in the rest of this section:

For each frame the rectified left and right input images plus the disparity image are uploaded to the GPU, called *device*, and saved to the *device memory*. The disparity image is pre-calculated with a stereo algorithm available on a *field-programmable gate array* (FPGA) without any extra computational cost.

The whole algorithm is embedded into a coarse-to-fine warping strategy, to avoid convergence to an unwanted local minimum due to linearization of the image intensities (equations (15) and (16)). This not only allows for estimating larger displacements but also improves the overall performance of the algorithm. The pyramids for the input images are constructed using a downsampling factor of 2 combined with a 5×5 Gauss filter. Beforehand the source images are bound as readonly textures, since texture memory is cached and most memory accesses offer a good cache locality.

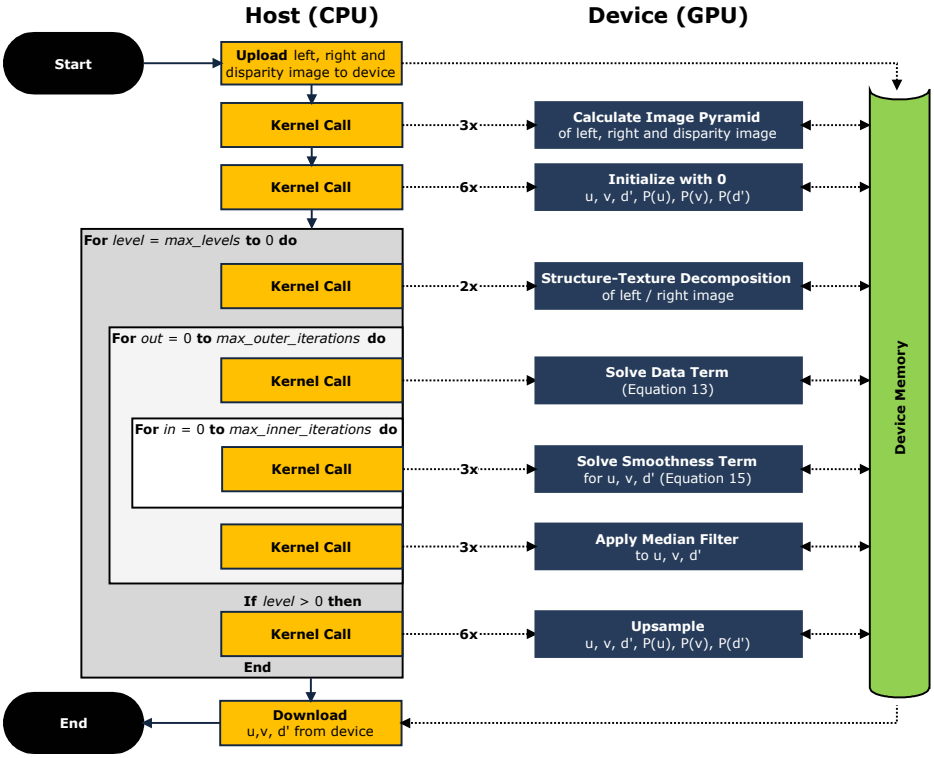

Fig. 2. Flowchart of the CUDA scene flow implementation.

To increase the robustness against illumination changes, every image in the pyramid is decomposed into a *structure* and *texture* part via total variation denoising [5], that is essentially the same algorithm as used to minimize the scene flow smoothness terms (equation (18)). The texture part of the image is generated by subtracting the original and the denoised image. Hence it contains mainly the scale-details and is less sensitive to illumination changes or shadows.

4.1 Minimization Procedure

Starting with $\tilde{\boldsymbol{u}} = (0,0,0)^T$ and $\boldsymbol{p}_i = (0,0)^T$ on the coarsest level of the pyramid, the algorithm alternatingly updates $\boldsymbol{u}$ (data term) and $\tilde{\boldsymbol{u}}$ (smoothness term) in

each *outer* iteration. For efficiency reasons, both alternating steps are implemented in two separate kernels. Despite regularization, sometimes the solution contains outliers. These can be discarded by a median filter applied to $\tilde{\boldsymbol{u}}$ without the need of additional iterations. The median filter employed is an efficient 3×3 median filter as presented in [6]. Between the pyramid levels the flow vectors $\boldsymbol{u}$ and the dual variables $\boldsymbol{p}_i$ are upsampled using a 5×5 Gauss filter.

Solving the Data Term The update step for $\boldsymbol{u}$ primarily involves the inversion of a 3×3 matrix for every pixel. Even though this results in a high number of arithmetic operations, these calculations can be used to overlap the device memory accesses with high latency. The decomposed input images are accessed via texture fetches, enabling two additional features: Texture memory provides fast bilinear interpolation that is used for the sub-pixel gradient and residual calculation. Furthermore, the image gradients are approximated by central differences, where the boundary has a zero gradient. Texture memory supports automatic handling of boundary cases, i.e. automatic clamping, that is used to avoid expensive manual border handling.

Solving the Smoothness Term The fixed point iteration to update all $\boldsymbol{p}_i$ and $\tilde{\boldsymbol{u}}$ uses backward differences to approximate $\mathbf{div}\, \boldsymbol{p}$ and forward differentation for the gradient computation as in [2]. For realtime performance the fix point steps are performed several times (inner iterations) before updating $\boldsymbol{u}$, while the number of outer iterations is decreased. Unfortunately the updates of $\boldsymbol{p}_i$ and $\tilde{\boldsymbol{u}}$ must be exchanged between adjacent threads after every inner iteration, breaking down the parallelism. To improve the performance, the values are exchanged only within 16×16 *thread blocks*. This enables the usage of fast shared memory for synchronization. Nevertheless the updates between adjacent thread blocks must be exchanged after a couple of inner block iterations. In this work no visible artifacts could be observed if the updates between blocks are performed after 5 inner block iterations.

5 Results

To obtain the timing results, two different hardware setups were used for the GPU implementation: A desktop PC equipped with a NVIDIA Geforce GTX 285 card, and one equipped with a NVIDIA Geforce GTX 260 card. The results are compared to the CPU implementation executed on an Intel Core 2 Extreme processor with 4 GB DDR2 RAM. The timing results illustrated in table 1 show that more than 30 frames per second can be achieved with the approach in this work, what is 50 times faster than the CPU implementation.

To assess the quality of the algorithm, it was evaluated on synthetic stereo sequences, where the ground truth is known. Compared to the CPU implementation, it turns out that the accuracy is nearly equivalent despite the optimization steps presented above. Besides the synthetic images, the algorithm was also tested on real video sequences to demonstrate the practicality under real world conditions. Scene flow results on real images are shown in figure 3.

To allow the comparison to other methods the implementation was further evaluated with the Middlebury optical flow benchmark data base. The algorithm can be adapted to the optical flow case just by setting λ_2 and λ_3 to zero (equation (14)). On 27th July 2009 the implementation of this paper reached the 8th place in the Middlebury ranking. Please note, that most methods listed at the Middlebury benchmark require computation times in the range of seconds or hours. The presented algorithm in this paper required about 120 ms, although the parameters where tuned to quality.

Iterations per level	**GTX 285**	**GTX 260**	**Core 2 Extreme**
10 outer and 3 inner	30 ms	50 ms	1571 ms
20 outer and 5 inner	72 ms	113 ms	3558 ms
100 outer and 5 inner	286 ms	434 ms	13994 ms

Table 1. Timing results in milliseconds for different configurations at 640×480 image resolution and 5 pyramid levels.

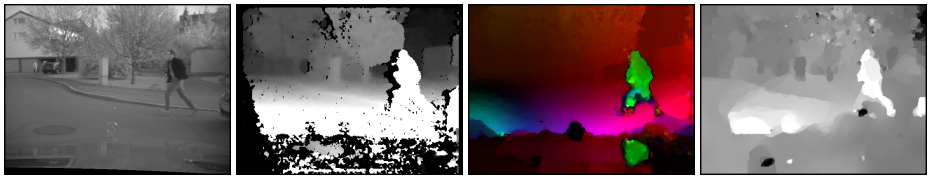

Fig. 3. Scene flow results for a real video frame: (1) Left input image (2) Given SGM disparity map (white = near, black = far) (3) Estim. optical flow (hue = direction, intensity = magnitude) (4) Estim. disparity change (black = decreasing, white = increasing).

References

1. Wedel, A., Rabe, C., Vaudrey, T., Brox, T., Franke, U., Cremers, D.: Efficient dense scene flow from sparse or dense stereo data. In: ECCV '08: Proceedings of the 10th European Conference on Computer Vision, Berlin, Heidelberg (2008) 739–751
2. Zach, C., Pock, T., Bischof, H.: A duality based approach for realtime tv-l1 optical flow. In: DAGM07. (2007) 214–223
3. Gehrig, S., Eberli, F., Meyer, T.: A real-time low-power stereo vision engine using semi-global matching. In: ICVS 2009. (2009) 134–143
4. Chambolle, A.: An algorithm for total variation minimization and applications. J. Math. Imaging Vis. **20**(1-2) (2004) 89–97
5. Rudin, L., Osher, S., Fatemi, E.: Nonlinear total variation based noise removal algorithms. Phys. D **60**(1-4) (1992) 259–268
6. McGuire, M., Whitson, K.: A fast, small-radius gpu median filter (2008)