Numer. Math. Theor. Meth. Appl.
doi: 10.4208/nmtma.2013.mssvm04

Vol. **6**, No. 1, pp. 72-94
February 2013

3D Anisotropic Diffusion on GPUs by Closed-Form Local Tensor Computations

Arjan Kuijper[1,2,*], Andreas Schwarzkopf[2], Thomas Kalbe[2], Chandrajit Bajaj[3], Stefan Roth[2] and Michael Goesele[2]

[1] *Fraunhofer IGD, 64283 Darmstadt, Germany.*
[2] *Department of Computer Science, TU Darmstadt, D-64289 Darmstadt, Germany.*
[3] *ICES-CVC, University of Texas at Austin, Austin, Texas 78712, USA.*

Received 6 December 2011; Accepted (in revised version) 15 May 2012

Available online 11 January 2013

Abstract. We present an efficient implementation of volumetric anisotropic image diffusion filters on modern programmable graphics processing units (GPUs), where the mathematics behind volumetric diffusion is effectively reduced to the diffusion in 2D images. We hereby avoid the computational bottleneck of a time consuming eigenvalue decomposition in $\mathbb{R}^3$. Instead, we use a projection of the Hessian matrix along the surface normal onto the tangent plane of the local isodensity surface and solve for the remaining two tangent space eigenvectors. We derive closed formulas to achieve this and prevent the GPU code from branching. We show that our most complex volumetric anisotropic diffusion filters gain a speed up of more than 600 compared to a CPU solution.

AMS subject classifications: 68U10

Key words: Image processing, enhancement, anisotropic diffusion, tensors, 3D filtering.

1. Introduction

Diffusion equations can be considered as physically motivated iterative filters applying a diffusion process on (mostly) noisy image data. They smooth out noise effectively and deliver a framework providing a scale space representation of the image, when time is considered as a natural, continuous scale space parameter [20, 21, 24, 25, 37]. They are well known in the field of image processing and have been subject to many enhancements during the last decades, see e.g. the monographs by Weickert [38, 40] for a complete and comprehensive introduction. Especially the ability to steer the direction and amount of

*Corresponding author. *Email address:* `arjan.kuijper@gris.informatik.tu-darmstadt.de` (A. Kuijper)

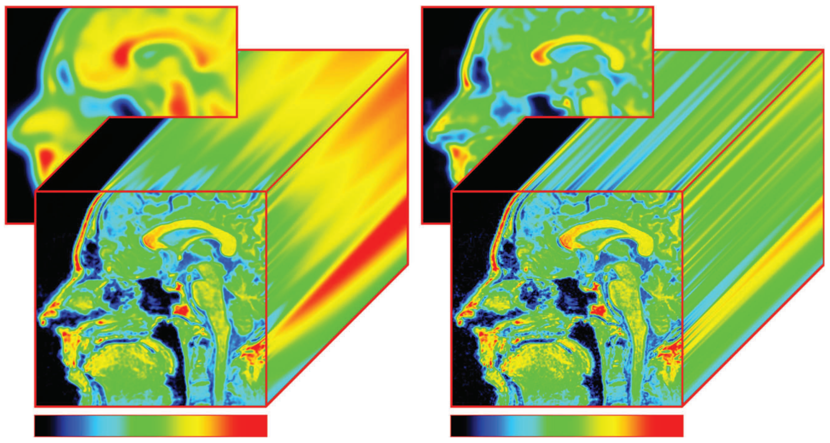


Figure 1: Scale-space representation of homogeneous (left) and inhomogeneous diffusion (right) for a 2D image. 200 iterations, $\Delta t = 0.05$.

diffusion in pre-described directions based on local image structure increased the popularity of these processes. Fig. 1 gives a visual example of the effect of two different types of diffusion on a 2D image extended with a time axes.

Recent publications in both computer graphics and computer vision apply this diffusion for smoothing of normal maps [33], fairing of surfaces and functions on surfaces and meshes [4,5,27], and image compression and inpainting [11], to mention only some possibilities. One should note the essential difference between approaches on the full voxel set and subsets – e.g. meshes – describing shapes. Anisotropic diffusion of whole volume images or general meshes [1,26] and smoothing vector valued volume images [41] are also common tasks arising in medical applications. It has been reported in 3D imaging as "the most favorable approach regarding the efficiency of noise reduction, signal preservation and computing effort" [10], but the computation time is a bottleneck on traditional CPU implementations for (close to) real-time applications. For off-line denoising and enhancing of 2D images, impressive and fast results are obtained by Weickert's (nD) AOS implementation [38]. It can be parallelized for many types of diffusion by exploiting intrinsic parallelism. Weickert et al. showed this using a regularized Perona-Malik diffusion filter on a $138 \times 208 \times 138$ 3D ultrasound data set [36]. Alternatively, fast explicit schemes can be used [13].

Anisotropic diffusion requires to solve second order partial differential equations (PDEs) numerically. As the amount of discretized data is rapidly increasing, especially in the volumetric 3D case, it is a perfect application for modern graphic cards. Current Graphics Processing Units (GPUs) can easily handle large medical data sets, for example CT or MRI images beyond the traditional 512^3 voxels boundary [2,18]. The current GPU SIMD (Single instruction, multiple data) architecture allows to solve each iteration in a few milliseconds due to massively parallel processing [7]. The big "however" is that this holds only for equations that lead to an *efficient* parallelization. This is at least difficult for most interesting, non-linear, PDEs. This is due to the local structure in each voxel that determines in which direction the diffusion (smoothing) should be performed. Anisotropic diffusion types in which locally the structure tensor and its eigensystem need to be computed are, therefore, notoriously badly suited for parallelization on GPUs. Implementing diffusion filters on GPUs thus requires to analyze the mathematics behind diffusion in order to adapt the algorithms well to the hardware.

1.1. Contribution

As main contribution we show how one can obtain the so-called local structure frame for volumetric data sets easily. This leads to nearly unconditional code without the need for in-voxel iterative loops, performing extremely well on GPUs. For this purpose, we build on a technique described by Hadwiger et al. [14]. Using a closed form solution for the computation of the diffusion tensor eigensystem, we obtain a constant time for each voxel. This is in contrast to the alternative of locally iterative computations to determine the eigensystem [12], in which convergence cycle and time differs from voxel to voxel. Anisotropic nonlinear diffusion on symmetric multiprocessor (SMP) clusters for volumetric data was discussed in [32], showing a maximum speedup of 20 on SMP clusters with up to 30 processors. In our GPU approach, we do not need to slice the volume and distribute it across a platform, as all shading processors of modern GPUs can access the same memory. We therefore achieve speedups of up to 640 on a comparably cheap GPU.

Volumetric anisotropic diffusion on GPUs using shader programs in the standard graphics pipeline has been discussed in, e.g, the works by Jeong et al. [15], Zhao [42] or Beyer et al. [6]. In contrast to this, our approach is based on NVidia's *CUDA* which is better suited for GPGPU (general purpose GPU) algorithms like volumetric diffusion. Intermediate values, such as the Hessian, are recomputed on-the-fly in each iteration and are stored temporarily in per-thread local memory. Therefore, larger data sets can be processed on the GPU. This significantly simplifies the algorithm while the results of the diffusion are very good, see Figs. 2 (right) and 9. The examples we show are iso-manifolds in the full 3D volume. The diffusion is applied to all voxels, but in order to be able to show relevant results, we choose physically motivated values - those representing the skull. To generate a surface from these voxels we need sub-voxel information, for which we implemented affine transformations (see also Section 6). Full details on implementation issues can be found in [30]

2. Diffusion on images

The linear homogeneous diffusion equation removes noise from images by solving the heat equation, a second order parabolic PDE. Initial and boundary conditions are required to find a particular solution. A general diffusion equation can then be defined as follows:

$$\frac{\partial}{\partial t}\Phi(\vec{x},t) = \operatorname{div}\left(D\nabla\Phi(\vec{x},t)\right), \qquad \text{for } \vec{x}\in\Omega, \quad t>0, \tag{2.1}$$

$$\Phi(\vec{x},0) = \Phi_0(\vec{x}), \qquad \text{for } \vec{x}\in\Omega, \tag{2.2}$$

$$\frac{\partial}{\partial \vec{n}}\Phi(\vec{x},0) = 0, \qquad \text{for } \vec{x} \in \partial\Omega. \tag{2.3}$$

Here, Φ denotes the noisy image function defined on a region Ω of the Euclidean space. D is a function, which determines the diffusion speed through the medium. D is constant (usually 1 or 1/2) in the linear case. The initial condition (2.2) initializes the function at time $t = 0$ with the original noisy image Φ_0. The boundary values are defined in (2.3) by their derivative in normal direction $\vec{n}$ to the border of the considered volume: Since the directional derivative is assumed to be 0, no flow through the boundary $\partial\Omega$ is induced. In digital image processing this is usually accomplished by restricting access outside the volume to the nearest boundary values (clamp values).

Solving the heat equation for $D = 1$ at time $t = {}^1\!/\!{}_2\,\sigma^2$ equates to convolving the image function with a Gaussian of size σ (see [3,20,37]):

$$\Phi(\vec{x},t) = \begin{cases} \Phi_0(\vec{x}), & t = 0, \\ (G_{\sqrt{2t}} * \Phi_0)(\vec{x}), & t > 0. \end{cases}$$

Thus, this diffusion equation has exactly the same smoothing characteristics as the well known Gaussian filter. In particular boundaries blur out fast and therefore edge information gets lost quickly, see Fig. 2 (left). Although this linear diffusion equation carries a wealth in itself [20,22,23], the focus in this work is on nonlinear diffusion.

2.1. Inhomogeneous diffusion

In the context of image processing, the heat equation was modified significantly by Perona and Malik [29] by replacing D in Eq. (2.1) with an edge detector, a monotonically decreasing non-negative real function g, which attenuates the induced flow close to edges and therefore effectively prevents edges from being washed out:

$$\frac{\partial}{\partial t}\Phi = \operatorname{div}\left(g(|\nabla\Phi|)\nabla\Phi\right), \qquad \text{for } \vec{x} \in \Omega, \quad t > 0. \tag{2.4}$$

Perona and Malik proposed the following diffusivity functions

$$g(\nabla\Phi) = \exp\left[-\left(\frac{|\nabla\Phi|}{\lambda}\right)^2\right],$$
$$g(\nabla\Phi) = \frac{1}{1+\left(\frac{|\nabla\Phi|}{\lambda}\right)^2}.$$

They designated this diffusion anisotropic, but it is only locally adapting and still isotropic, as it is steered by a *scalar* diffusion coefficient. Therefore, Weickert calls this locally adapting diffusion *inhomogeneous* [37]. Inhomogeneous diffusion is able to preserve edges over a long period of time, but its smoothing capabilities close to edges are rather poor.

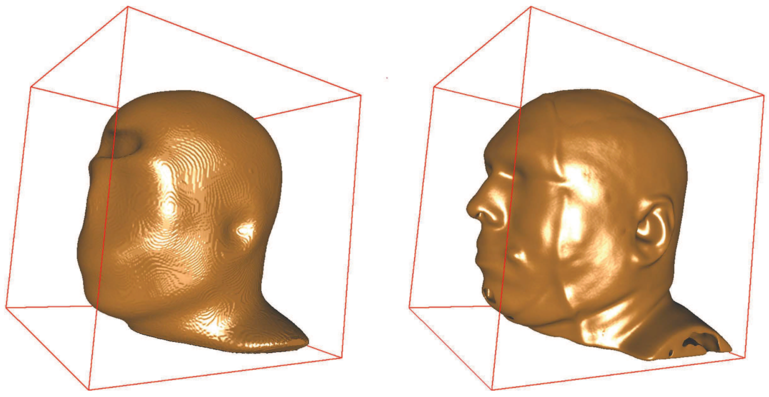


Figure 2: 100 iterations of diffusion on a volume with a time step $\Delta t = 0.05$, after which an isosurface is extracted. *Left*: Homogeneous diffusion. The frontal sinus destroys the surface structure of the forehead and small scaled details (e.g. lips, nose, ears) are lost. *Right*: Nonlinear anisotropic diffusion (here: edge enhancing diffusion) steered by a diffusion tensor based on local structure preserves fine scaled features.

2.2. Nonlinear anisotropic diffusion

Weickert introduced a new nonlinear anisotropic diffusion, which replaces the scalar D in Eq. (2.1) with a tensor, that allows for anisotropic adjustment of the diffusion flow [35, 37–39]. These works will give the interested reader a sound standing insight to the whole topic. Weickerts definition of the diffusion equation is as follows:

$$\frac{\partial}{\partial t}\Phi = \operatorname{div}(D(\mathscr{S})\nabla\Phi), \qquad \text{for } \vec{x} \in \Omega, \quad t > 0. \tag{2.5}$$

The diffusion tensor D is a function of the structure tensor $\mathscr{S} = \nabla\Phi \otimes \nabla\Phi$ and used to steer and align the diffusion flow along the local surface structure. In general the exact definition of the diffusion tensor depends on the desired results of the smoothing process. Thus, the nonlinear anisotropic diffusion is a function of the image Φ and therefore a function of space and time as well.

Edge Enhancing Diffusion (EED) attenuates diffusion flow normal to the edge or surface but promotes flow along the edge or parallel to the surface, see Fig. 2 (right). Furthermore, *Coherence Enhancing Diffusion* (CED) tries to steer diffusion along line-like structures and is able to reconnect interrupted lines [39]. In a hybrid approach, joining EED and CED to locally adapting diffusion, one is able to enhance edges, smooth out noise and to connect broken lines [10].

2.2.1. Volumetric anisotropic diffusion

In all cases the definition of a useful diffusion tensor involves the construction of a local structure frame: One needs to find a transformation which aligns the coordinate system orthogonally to the surface of the submanifold. In 2D this is rather trivial once the normal vector is given. In 3D one can proceed as follows. Let V be such a coordinate transformation, aligning the third axis normal to the surface, then we can define the diffusion tensor D in three dimensions as follows:

$$D = VD^*V^T = V \begin{pmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & \gamma \end{pmatrix} V^T. \tag{2.6}$$

Matrix D transformed to the new basis V is a diagonal matrix D^*, as the diffusion flow is aligned perfectly along the principal directions of the surface structure. For $(\alpha,\beta,\gamma) = (1,1,1)$ one gets the standard homogeneous diffusion of Eq. (2.1) and by choosing different (but constant) values one can emphasize local smoothing. This obviously becomes more interesting if local information, e.g. in terms of a weighted local gradient, is taken into account. Considering all directions equally by $\alpha = \beta = \gamma = g(|\nabla\Phi|)$, using the function g as introduced above, give the inhomogeneous Perona Malik diffusion of Eq. (2.4). The most interesting, but also complicated, types of diffusion like CED and EED are obtained by choosing different values. For CED one can set all but one eigenvalue (entry) constant, and adapt the largest - the one denoting the flow direction - using an exponential function [39]. For EED the axis normal to the surface - the normal direction - is adapted using g, i.e. $(\alpha,\beta,\gamma) = (1,1,g(|\nabla\Phi|))$, see Fig. 2 (right).

The crucial point when designing anisotropic diffusion based on local surface information (for instance when the local flow direction is used) is the *efficient construction* of this frame V. The traditional way is to obtain this basis by analyzing the structure tensor, defined as the outer product of the gradient $\nabla\Phi$ with itself. Obviously, the structure tensor is a symmetric matrix. Therefore, the existence of three real eigenvalues is assured. We can find a representation $S = V\Lambda V^T$ through eigendecomposition which gives us an orthonormal basis V and its inverse $V^{-1} = V^T$: The columns of V consist of the eigenvectors of S. In this representation Λ is a diagonal matrix whose diagonal elements are the corresponding eigenvalues. Performing an *eigendecomposition in each voxel* using standard methods is hard to parallelize, though. Most algorithms require either in-voxel iterative methods. Conditional code leads to branching and thus to divergent program execution and sequential calculations of the GPU multiprocessors. They reduce speed advantages significantly, so it is important to find a reduction of the dimension. In the next section, we present such a method and find a local frame efficiently by analyzing the Hessian, which holds structural information, as it describes the change of the surface normal.

3. Surface structure and the Hessian

When defining a diffusion tensor, it is utterly important to find a basis V whose axes are aligned exactly along the principal curvature directions of the surface, as we do not want to restrict ourselves to diffusion tensors that depend on the gradient direction only. Theoretically, this basis can be found via an eigen-decomposition of the structure tensor in 3D space. As the structure tensor is a real symmetric matrix, the eigenvalues are real and the eigenvectors are existent. However, the characteristic polynomial of a 3×3 matrix has degree 3 and therefore it is rather time consuming to solve for the roots.

On the other hand, the eigenvalues of a 2×2 matrix are computed easily by evaluating only a few closed formulas. Especially for machine code executed on modern GPUs, this is of advantage as the single execution paths are not divergent (not branching) and parallel execution on the hardware is achieved ideally. In the following, we will show how to obtain the structure frame V by evaluating closed formulas only.

3.1. Tangent space projection of the Hessian

The following considerations are aimed at finding a basis transformation $V : \mathbb{R}^3 \mapsto \mathbb{R}^3$ with as few computations as possible and which will describe a coordinate system normal to the tangent plane of the isosurface at a given point. Despite that, the remaining two basis vectors of V spanning the tangent plane should be aligned with the orthogonal principal curvature directions.

Assuming that the inner region of a volume consists of higher density volumes, we define the *surface normal* by the gradient $\nabla\Phi = \left[\frac{\partial\Phi}{\partial x} \quad \frac{\partial\Phi}{\partial y} \quad \frac{\partial\Phi}{\partial z}\right]^T$ as $\vec{n} = -\nabla\Phi/|\nabla\Phi|$. Since $\nabla\Phi$ points towards the direction of the greatest density ascent inside the volume, $\vec{n}$ lies inside the linear span of the gradient. Therefore it points into the direction the surface moves when the iso value is increased. We can choose $\vec{n}(\vec{x})$ to be the first vector of our frame V. Note that very small gradients imply that there are no edges. So there we can simply smooth with $D = Id$.

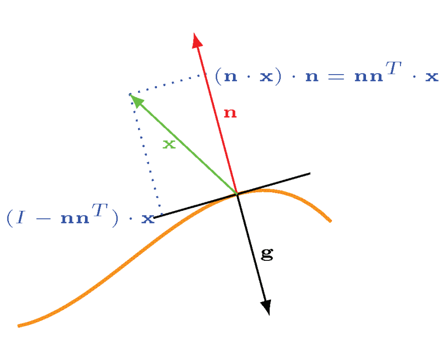
The *curvature* of a surface is defined as the ratio between change of surface normal and change of position, which is described by the gradient of $\vec{n}$: If we move in an infinitesimal close area around the point $\vec{x}$, the normal will change according to the surface. In [14] and [19] one finds methods on how to characterize the curvature of a surface based on gradient information and, moreover, how to obtain the principal curvature directions we are looking for the construction of V. The derivative of the normal field $\nabla\vec{n}^T$ at some point $\vec{x}$ contains curvature information of the surface. Note that $\nabla\vec{n}^T$ is a 3×3 matrix. According to Kindlmann et al. [19] it holds that

$$\nabla\vec{n}^T = -\frac{1}{|\nabla\Phi|}(I - \vec{n}\vec{n}^T)H. \tag{3.1}$$

Here, I denotes the 3×3 identity matrix, and $H = \nabla(\nabla\Phi)^T$ is the Hessian containing all combinations of partial second order derivatives of the image Φ.

While the gradient describes the amount of change of Φ, the Hessian describes the amount of change of the gradient, that is the amount of change of the surface normal in an infinitesimal close region to a given point $\vec{x}$. This amount of change of the gradient can be decomposed into two components, namely the changes along the gradient direction and changes in the tangent space. Only the latter is required for isosurface principal curvature directions computation.

To perform the *tangent space projection*, we proceed as follows: We may omit the scaling factor $|\nabla\Phi|^{-1}$ in Eq. (3.1) and concentrate on the remaining term. It is easy to see that



$(\mathbf{n} \cdot \mathbf{x}) \cdot \mathbf{n} = \mathbf{n}\mathbf{n}^T \cdot \mathbf{x}$
$\mathbf{n}$
$\mathbf{x}$
$(I - \mathbf{n}\mathbf{n}^T) \cdot \mathbf{x}$
$\mathbf{g}$

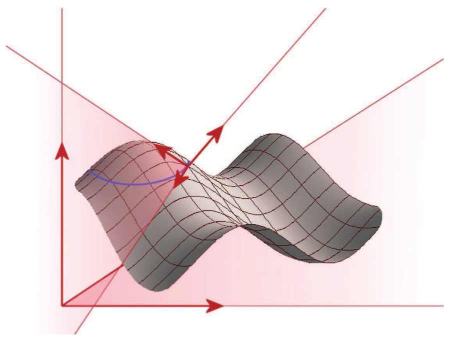


Figure 3: *Left:* Projecting a point $\vec{x}$ into the span of the normal $\vec{n}$ and its complement. *Right:* The local surface frame.

$(\vec{n}\vec{n}^T)\vec{x} = (\vec{n}\vec{x})\vec{n}$, and the operator $(\vec{n}\vec{n}^T)$ projects any point $\vec{x} \in \Omega$ onto the linear span of the normal. Therefore we are able to define a linear map

$$P = (I - \vec{n}\vec{n}^T) = \left(I - \frac{\nabla\Phi(\nabla\Phi)^T}{|\nabla\Phi|^2} \right). \tag{3.2}$$

which projects any given point $\vec{x}$ into the complement of the linear span of $\vec{n}$, which is the iso surface (see Fig. 3). The projection P extracts the gradients change of direction from the Hessian inside the tangent space. By using P we thus define the *shape operator*

$$S = P^T \frac{H}{|\nabla\Phi|} P. \tag{3.3}$$

This matrix S contains by construction exactly the eigensystem we want, with eigenvectors in the tangent space that are aligned with the principal curvatures directions. Since S is symmetric, solving the characteristic polynomial gives us three real roots and associated orthogonal eigenvectors. As long as the image is not degenerated, this yields a non-trivial decomposition. Still, the computational overhead of a full eigen-decomposition of a 3×3 matrix is rather high, especially as one eigenvector, the normal $\vec{n}$, is already known. The remaining eigenvectors in the tangent plane are the principle curvature directions with corresponding eigenvalues $\lambda_{1,2}$ which amount to the principle curvatures.

According to Hadwiger et al. [14], we can solve for the eigenvalues directly in 2D tangent space without explicitly computing S. The transformation of S into any arbitrary orthogonal basis $(\vec{u}, \vec{v})$ of the tangent space is defined as

$$S' = \begin{pmatrix} s_{11} & s_{12} \\ s_{12} & s_{22} \end{pmatrix} = (\vec{u}, \vec{v})^T \frac{H}{|\nabla\Phi|} (\vec{u}, \vec{v}). \tag{3.4}$$

For finding an arbitrary orthogonal basis $\vec{u}$, $\vec{v}$ in the tangent plane, we may proceed as follows: We choose the canonical unit vector $\vec{e}_1 = (1,0,0)^T$ assuming that $\vec{e}_1 \nparallel \vec{n}$ holds and compute the cross product $\vec{u} = \vec{e}_1 \times \vec{n}$. In case $\vec{u} = \vec{0}$ we compute the cross product again, now using the second unit vector, $\vec{e}_2 = (0,1,0)^T$. $\vec{u}$ is now normal to $\vec{n}$ and therefore it must be part of the tangent plane. We finish the new basis by adding $\vec{v} = \vec{u} \times \vec{n}$.

By using Eq. (3.4) we are now able to compute the eigenvalues $\lambda_{1,2}$ of S' by solving the characteristic polynomial

$$\det(S' - \lambda I) = \begin{vmatrix} s_{11} - \lambda & s_{12} \\ s_{12} & s_{22} - \lambda \end{vmatrix} = 0$$
$$\Rightarrow \lambda_{1,2} = \frac{\operatorname{trace}(S')}{2} \pm \sqrt{\frac{\operatorname{trace}(S')^2}{4} - \det(S')}. \tag{3.5}$$

From the eigenvalues $\lambda_{1,2}$ we compute the corresponding eigenvectors. The appropriate formula in [14] is incorrect and can be found in the correct formulation in [31, p. 96]. The eigenvectors $\vec{w}^*_{1,2}$ are computed with reference to the basis $(\vec{u}, \vec{v})$ at first, and afterwards they are transformed back into 3D space:

$$\vec{w}^*_1 = \begin{pmatrix} w^*_{1u} \\ w^*_{1v} \end{pmatrix} = \begin{cases} \begin{pmatrix} \lambda_1 - s_{22} \\ s_{12} \end{pmatrix}, & \text{for } s_{12} \neq 0, \\ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, & \text{for } s_{12} = 0, \end{cases} \tag{3.6a}$$

$$\vec{w}^*_2 = \begin{pmatrix} w^*_{2u} \\ w^*_{2v} \end{pmatrix} = \begin{cases} \begin{pmatrix} \lambda_2 - s_{22} \\ s_{12} \end{pmatrix}, & \text{for } s_{12} \neq 0, \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix}, & \text{for } s_{12} = 0. \end{cases} \tag{3.6b}$$

The transformation of the 2D eigenvectors into object space is accomplished by extending the tangent space basis with $\vec{n}$ to 3D and a retransformation into the original orientation by means of $V = \{\vec{u}, \vec{v}, \vec{n}\}$:

$$\vec{w}_i = \begin{pmatrix} u_x & v_x & n_x \\ u_y & v_y & n_y \\ u_z & v_z & n_z \end{pmatrix} \begin{pmatrix} w^*_{iu} \\ w^*_{iv} \\ 0 \end{pmatrix} = \begin{pmatrix} u_x w^*_{iu} + v_x w^*_{iv} \\ u_y w^*_{iu} + v_y w^*_{iv} \\ u_z w^*_{iu} + v_z w^*_{iv} \end{pmatrix}. \tag{3.7}$$

3.2. Summary: retrieving the diffusion tensor in closed form

Building on the results of the previous sections, we now depict a compact and easy to implement method to define the anisotropic diffusion tensor. The core of our method is a 2×2 eigen-decomposition of the Hessian projected into the tangent space of the iso surface, computed with simple, closed formulas in the second part below. The method can be outlined with the following three computational blocks:

1. Calculate the gradient $\nabla\Phi$, the normalvector $\vec{n} = -\nabla\Phi/|\nabla\Phi|$ and the Hessian $H = \nabla(\nabla\Phi)^T$. To get a 3D set of vectors, complete $\vec{n}$ with any arbitrary vectors $\vec{u}$ and $\vec{v}$ in the tangent space to an orthonormal basis.

2. Perform actual computations in 2D: Using Eq. (3.4), project H into the tangent plane and obtain the 2×2 matrix S'. Calculate its eigenvalues $\lambda_{1,2}$ using Eq. (3.5) and the

corresponding eigenvectors $\vec{w}^*_{1,2}$ with respect to the $(\vec{u}, \vec{v})$ basis using Eq. (3.6). Without loss of generality we can – if this is necessary for the definition of our diffusion tensor e.g. for CED – reorder the eigenvalues and eigenvectors: $\lambda_1 < \lambda_2$.

3. Getting back to 3D: Transform the 2D eigenvectors to the 3D eigenvectors $\vec{w}_1, \vec{w}_2$ using Eq. (3.7). Then set $V = (\vec{w}_1, \vec{w}_2, \vec{n})$ and $V^{-1} = V^T$ and define $D = V \cdot \mathrm{diag}(\alpha, \beta, \gamma) \cdot V^{-1}$.

The last step defines the desired diffusion tensor. For instance, one can choose $\alpha = \beta = 1$ and $\gamma = g(|\nabla\Phi|)$ for smoothing along the isosurface and attenuating the diffusion flow normal to the edge (EED). In that case the local re-orientation is not essential, but it is obviously further possible to define other diffusion tensors with different properties upon the frame V, like CED and variants.

4. Prefiltering

The diffusion tensor might be a direct function of the gradient of the image or, speaking more generally, a function of any interest operator I. Nevertheless, the bottom line is, that the interest operator itself is a function of $\Phi(\vec{x}, t)$. Hence, one could also write:

$$\partial_t \Phi = \mathrm{div}(D(I(\Phi))\nabla\Phi). \tag{4.1}$$

Here we can see clearly how the image function is used both for inducing the flow $(\nabla\Phi)$ and for steering it $(D(I(\Phi)))$. Overall it is important to acquire high-quality first and second order derivatives, namely for the gradient as well as the Hessian. Especially at the beginning of the diffusion process small noise components may disturb the discrete computation of the derivatives enormously.

To initialize the diffusion process optimally, one usually applies some sort of prefiltering F_{pre} to the image. In this context, Eq. (2.2) is modified to

$$\Phi(\vec{x}, 0) = F_{pre}(\Phi_0(\vec{x})). \tag{4.2}$$

Prefiltering can thus be seen as a generalization of the diffusion process: By setting F_{pre} to the identity function, we can consider diffusion filtering without prefiltering as a special case.

For prefiltering one could generally use any known filter, as for instance the box-, Gaussian- or median filter. Bajaj et al. propose to use bilateral prefiltering: “Since bilateral filtering can remove noise as Gaussian filtering can do, $\cdots$ we replace Gaussian prefiltering with bilateral prefiltering. Additionally, bilateral filtering has the advantage of preserving edge or curvature information, which is better for constructing anisotropic diffusion tensors” [3].

4.1. Bilateral filter

The bilateral filter can be seen as an expansion to Gaussian filtering by applying an additional edge term [34]. Applying a Gaussian filter, the values of close up regions are

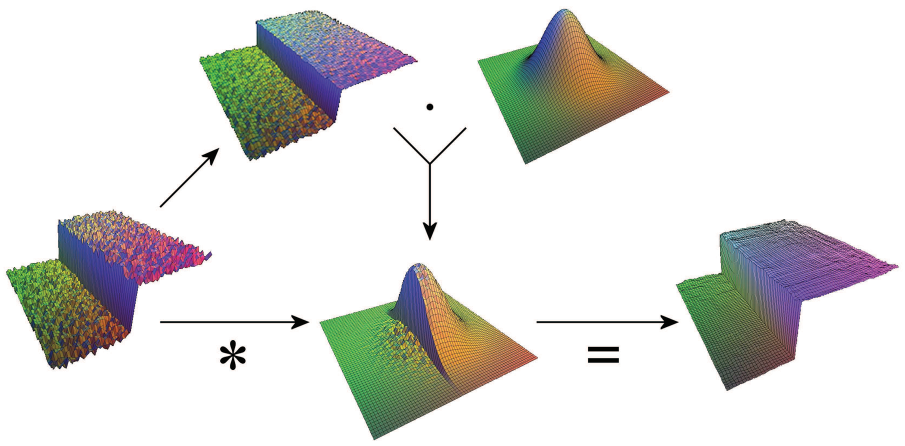


Figure 4: Structure of the bilateral filter kernel. According to [28] and [9]. ©ACM 2002.

rated higher than values of voxels that are further away. The weighting function is a normal distribution with variance σ and the weight depends on the distance solely. The bilateral filter extends the weighting function by an additional term accounting for the difference in the intensity domain. *Bilateral* means to differentiate the distance into a spatial component and an intensity component as well.

The graphical interpretation in Fig. 4 shows, how the combined weights of spatial and intensity domain result in a highly adaptive filter kernel that will keep edges despite smoothing noise efficiently.

The discretized formulation is as follow:

$$BI[\Phi]_y = \frac{1}{W_y} \sum_{x \in \Omega} \varphi_{\sigma_s}(||y-x||)\varphi_{\sigma_i}(|\Phi_y - \Phi_x|)\Phi_x. \tag{4.3}$$

Here φ_{σ_s} denotes a n-dimensional normal distribution in the spatial domain and φ_{σ_i} is a one dimensional normal distribution with variance σ_i. Choosing a large intensity range, the bilateral filter degenerates to a Gaussian: For $\sigma_i \to \infty$ we obtain $\varphi_{\sigma_i} \approx 1$. Gaussian filtering can thus be seen as a special case of bilateral filtering. The coefficient $\frac{1}{W_y}$ is used as a normalizing weight and can be obtained by summing up the weights of evaluated normal distributions.

Fig. 5 shows on real data visually the different components of the bilateral filter kernel. Considering this it is easy to see that the bilateral filter is not separable. Due to change of intensity, the kernel must be recomputed according to the actual center voxel. An implementation can profit from being executed massively parallel on a modern GPU.

5. Implementation

Our program was implemented using C for CUDA which allows for high-parallel computations on NVidia GPUs. As divergent program execution – arising from conditional code

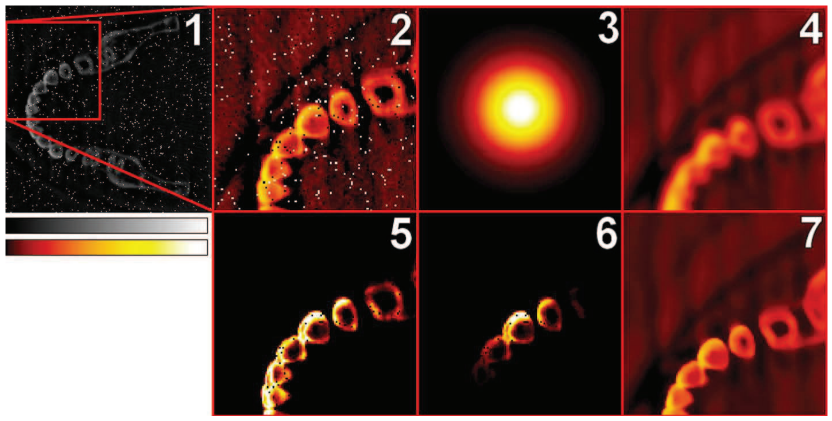


Figure 5: Visualizing the bilateral filter kernel using the data set Skull (256^3 voxels). 1) Isomanifolds of the original noisy image; 2) Enlarged section from image (1), enhanced by a transfer function; 3) Filter kernel in the spatial domain; 4) Convolution of the original image (2) with the spatial kernel (3); 5) Filter kernel in the intensity domain; 6) Bilateral filter kernel as the product of (3) and (5); 7) Convolution of the original image (2) with the bilateral filter (6).

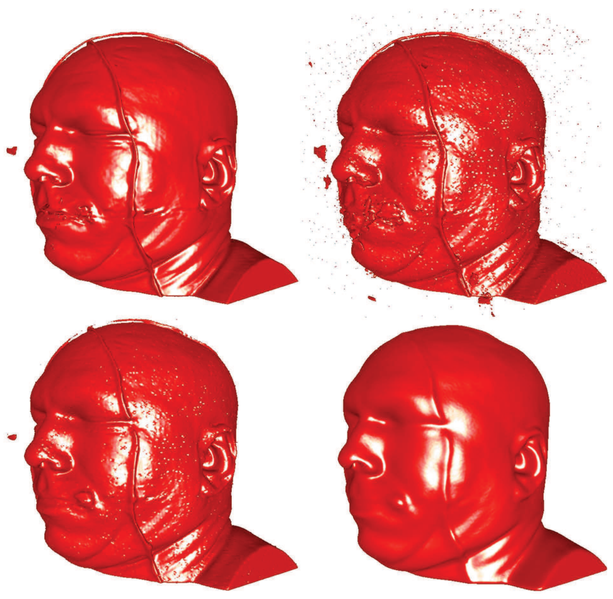


Figure 6: Edge enhancing diffusion with bilateral prefiltering. *Top left*: Iso-manifold in the original image. *Top right*: Noisy image (approx. 5% of the voxels affected, additive Gaussian noise, variance 10%). *Bottom left*: Result after bilateral prefiltering. *Bottom right*: After 50 iterations of EED ($\Delta t = 0.05$).

which leads to branching – and sequential calculations of the GPU multiprocessors could eliminate speed advantages it is important to find a reduction of the dimension: As one surface frame basis vector – the normal – is known, we can search for the remaining ones in the 2D hyper plane. This avoids, for instance, iterative methods to compute the full 3D eigensystem [12] resulting in varying execution times between voxels as their convergence is not a priori given and therefore in inefficient GPU use.

The volume data was stored as a 3D texture on the GPU. Coalesced memory access is not possible when processing volumetric data, so the best speedup was achieved by using cached texture memory. Clamping the textures automatically keeps track of all the border values: As gradients at the borders equal zero, no flow will be induced and we avoid conditional code which would slow down the program.

Besides, we can access values between the grid centers and request the hardware to do trilinear interpolation. This gives us a slight speedup for discretizing the Hessian as explained below. Finally, and most important, texture memory is cached, which compensates for the uncoalesced access and results in faster access to neighboring voxels. Performing multiple iterations can be achieved by synchronizing all threads and copying the output back to the texture memory. Copying memory within the device is a fast solution to circumvent the read only issue of texture memory. Each CUDA thread processes one voxel at a time. The code was straight forward developed from the discretized formulation of the anisotropic diffusion.

5.1. Discretization

In the following $\widehat{\Phi}$ denotes the discretized image function $\Phi : \Omega \subset \mathbb{R}^3 \mapsto \mathbb{R}$. Each grid point is associated with a value $\widehat{\Phi}_{x,y,z}$. Neighboring voxels are labeled $\widehat{\Phi}_{x_-} = \widehat{\Phi}_{x-1,y,z}$, likewise $\widehat{\Phi}_{x_+}$, $\widehat{\Phi}_{y_-}$, $\widehat{\Phi}_{y_+}$, $\widehat{\Phi}_{z_-}$ and $\widehat{\Phi}_{z_+}$.

The isotropic grid structure provides a natural spatial discretizing scheme for central differences. For temporal discretization we use forward differences. For nonlinear anisotropic diffusion we obtain the following discretization:

$$\begin{aligned}\widehat{\Phi}(t+\Delta t) &\approx \widehat{\Phi}(t) + \Delta t \cdot \operatorname{div}(D(\nabla\Phi)\nabla\Phi)\\ &= \widehat{\Phi}(t) + \Delta t \cdot \left(\frac{\partial}{\partial x}\left(d_{11}\frac{\partial \widehat{\Phi}}{\partial x} + d_{12}\frac{\partial \widehat{\Phi}}{\partial y} + d_{13}\frac{\partial \widehat{\Phi}}{\partial z}\right)\right.\\ &\qquad + \frac{\partial}{\partial y}\left(d_{12}\frac{\partial \widehat{\Phi}}{\partial x} + d_{22}\frac{\partial \widehat{\Phi}}{\partial y} + d_{23}\frac{\partial \widehat{\Phi}}{\partial z}\right)\\ &\qquad \left. + \frac{\partial}{\partial z}\left(d_{13}\frac{\partial \widehat{\Phi}}{\partial x} + d_{23}\frac{\partial \widehat{\Phi}}{\partial y} + d_{33}\frac{\partial \widehat{\Phi}}{\partial z}\right)\right). \end{aligned} \tag{5.1}$$

The entries d_{ij} represent the components of the diffusion tensor D and also depend on the spatial coordinates. We discretize Eq. (5.1) over an isotropic grid with central differences. Clearly, more sophisticated discretisation schemes exist (see, for example [8, 40]), but for our purpose – a fast and efficient GPU scheme for anisotropic diffusion – this suffices.

5.2. The diffusion tensor

The diffusion tensor as defined in step 10 of the algorithm outlined in Section 3.2 as $D = V \cdot \operatorname{diag}(\alpha, \beta, \gamma) \cdot V^{-1}$, yields a symmetric matrix D consisting of the eigenvectors $\vec{w}_1$

and $\vec{w_2}$ of the Hessian projected along $\vec{n}$ to the tangent plane of the isosurface:

$$D = \begin{pmatrix} d_{11} & d_{12} & d_{13} \\ d_{12} & d_{22} & d_{23} \\ d_{13} & d_{23} & d_{33} \end{pmatrix}, \tag{5.2}$$

with

$$\begin{aligned} d_{11} &= w_{1x}w_{1x}\alpha + w_{2x}w_{2x}\beta + n_x n_x \gamma, \\ d_{12} &= w_{1x}w_{1y}\alpha + w_{2x}w_{2y}\beta + n_x n_y \gamma, \end{aligned}$$

and accordingly for the remaining cases. The eigenvalues from Eq. (3.5) and eigenvectors from Eq. (3.6) are obtained directly in tangent space and are transformed back to object space, see Eq. (3.7).

The step size Δt needs to be small enough in order to guarantee numerical stability. Following [32], $\Delta t < 0.5/N_d$, with N_d the dimension of the problem, i.e. 3. In the following examples we used the conservative value $\Delta t = 0.05$.

6. Results

In the following two section we present the results on two different types of settings. First, we consider a relatively cheap mid-end consumer system, consisting of an NVidia GeForce 9800GT (say 100 USD) and an Intel Core 2 Duo E8500 CPU clocked at 3.16 GHz (round about 200 USD). Next, we focus on the diffusion gain when a high end state of the art system is used: an Intel Xeon E5430 CPU (2.66 GHz) and an NVidia GeForce GTX 480 with 480 shader cores. This GPU can handle a data set that is 8 times as large as the first test data set.

In the diffusion experiments we considered one time step for the following three diffusions: homogeneous (linear), inhomogeneous (Perona Malik), and anisotropic (EED). The reason to choose EED is merely that the denoising properties of EED are superb. The obtained clean images are easier to evaluate than, say, CED images that have a certain Van Gogh appearance. The main issue to show here is that the orientation of the local tangent plane is feasible at very limited computational cost on the GPU whilst gaining a significant speed-up compared to the CPU implementation.

Finally, the diffusion is performed on the full 3D data set. The images we show are iso-surfaces extracted from this data set by selecting a physically relevant value. For our experiments we used medical data sets from two freely available repositories: the Volume Library[†] and the VolVis Archiv[‡]. In order to extract the surfaces, we need sub-voxel information. This is done using affine transformations for nearest neighbor, trilinear filtering and trivariate cubic B-Splines.

[†]`http://www9.informatik.uni-erlangen.de/External/vollib/`

[‡]`http://www.volvis.org/`

6.1. Mid-end consumer system

Tables 1, 2, and 3 show the measured results of our implementation. The runtimes of single iterations are also visualized in Fig. 7. The results presented in this section refer to a volume with 256 × 256 × 256 voxels. We evaluated several volume sizes, but as the filters are independent of the data, we found, that the execution times are proportional and particularly the fraction of CPU to GPU times is constant, so one can easily extrapolate to other data sizes. As modern GPUs provide up to 4 GB memory (i.e. NVidia Quadro FX5800), it was easily possible to process datasets with up to 800^3 voxels on the graphic card. The timings were taken on a system consisting of an Intel E8500 CPU (*Intel(R) Core(TM)2 Duo CPU, 3.16GHz*) and a NVidia G92 GPU (GeForce 9800 GT).

Each record of the tables contains the following information:

Code: We measured the execution time for CPU code (*cpu*), GPU code (*gpu*) and GPU code with texture memory caching (*tex*) for one iteration of the filter.

MemCpy: For GPU based code we measured up- and download time of the volume data.

Table 1: Measurements for the *Prefilter*, volume size: 256^3.

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Gross	Net
Boxfilter (5 × 5 × 5 Kernel):					
cpu	0.00	6 222.91	6 222.91	1.0	1.0
gpu	52.60	1 189.98	1 242.58	5.2	5.0
tex	51.42	271.73	323.16	22.9	19.3
Boxfilter (separated, 5 × 5 × 5 Kernel):					
cpu	0.00	1 115.91	1 115.91	1.0	1.0
gpu	50.82	96.16	146.97	11.6	7.6
tex	51.57	78.90	130.47	14.1	8.6
Gaussfilter (3 × 3 × 3 Kernel):					
cpu	0.00	91 396.26	91 396.26	1.0	1.0
gpu	50.87	1 272.99	1 323.86	71.8	69.0
tex	51.33	710.52	761.85	128.6	120.0
Gaussfilter (separated, 3 × 3 × 3 Kernel):					
cpu	0.00	10 678.07	10 678.07	1.0	1.0
gpu	50.81	121.95	172.76	87.6	61.8
tex	51.45	114.58	166.03	93.2	64.3
Medianfilter (3 × 3 × 3 neighborship):					
cpu	0.00	14 754.21	14 754.21	1.0	1.0
gpu	50.82	1 832.01	1 882.83	8.1	7.8
tex	52.36	1 623.84	1 676.20	9.1	8.8
Bilateraler Filter (3 × 3 × 3 Kernel):					
cpu	0.00	301 215.50	301 215.50	1.0	1.0
gpu	50.93	1 452.63	1 503.56	207.4	200.3
tex	52.69	1 371.56	1 424.25	219.6	211.5

Table 2: Measurements for the *Diffusion Filters*, volume size: 256^3.

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Gross	Net
Linear homogenous diffusion:					
cpu	0.00	8 107.83	8 107.83	1.0	1.0
gpu	50.85	247.04	297.89	32.8	27.2
tex	51.37	147.25	198.61	55.1	40.8
Non-linear inhomogenous diffusion:					
cpu	0.00	1 916.54	1 916.54	1.0	1.0
gpu	50.95	35.26	86.21	54.4	22.2
tex	51.47	41.30	92.78	46.4	20.7
Nonlinear anisotropic diffusion (EED):					
cpu	0.00	5 954.42	5 954.42	1.0	1.0
gpu	54.15	184.28	238.43	32.3	25.0
tex	51.34	98.75	150.09	60.3	39.7

Table 3: Measurements for the *Affine Transformations*, volume size: 256^3.

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Gross	Net
Affine Transformation (Nearest Neighbor):					
cpu	0.00	1 173.75	1 173.75	1.0	1.0
gpu	50.75	33.53	84.28	35.0	13.9
tex	51.34	17.59	68.93	66.7	17.0
Affine Transformation (Trilinear):					
cpu	0.00	1 664.23	1 664.23	1.0	1.0
gpu	50.77	246.15	296.93	6.8	5.6
tex	51.86	31.46	83.31	52.9	20.0
Affine Transformation (Cubic B-Splines):					
cpu	0.00	4 151.91	4 151.91	1.0	1.0
gpu	51.75	1 846.91	1 898.66	2.2	2.2
tex	51.39	341.39	392.79	12.2	10.6

Filter: Execution time of the filter code.

Total: Total runtime including possible up- and download times.

Gross: [§] The gross value is a speedup factor based on the CPU time of the given filter (excluding any MemCpy timings).

Net: The net value is a speedup factor based on the CPU time of the given filter (including any MemCpy timings).

[§]Example: In case the CPU uses 80 ms and the GPU 20 ms for the filtering itself, the gross speedup is 80ms/20ms = 4. If the GPU code additionally needs up- and download times of 10 ms for the volume data this is a net speedup of $80\text{ms}/(20\text{ms}+10\text{ms}+10\text{ms}) = 2$.

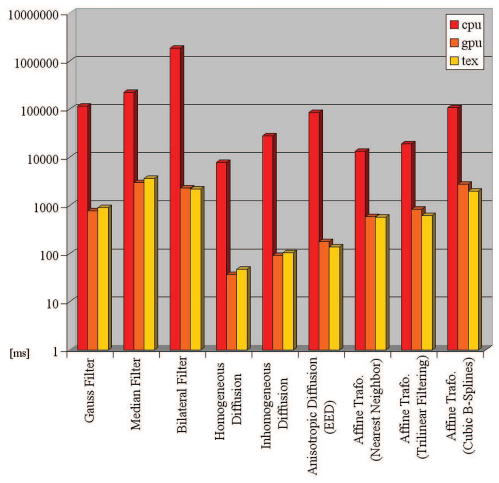


Figure 7: Runtimes for one iteration of the presented kernels from Tables 1-3 on a logarithmic scale (CPU: Intel E8500; GPU: NVidia G92).

Our program is capable of affine transformations above the before mentioned prefilters. Affine transformations are realized by applying 4×4 matrices on the data, containing rotations and translations in homogeneous coordinates. Accessing the volumes values between the existing grid points was done with nearest neighbor, trilinear filtering and with trivariate cubic B-Splines [16]. The measured results are contained in the following tables as well.

Bilateral prefiltering on the GPU achieves more than 210-times the execution speed than its corresponding CPU implementation. Although bilateral filtering seems to be a good choice for prefiltering, it is very time consuming, as it is not separable like the box or Gauss filters. Arithmetic intensity is slightly higher than for Gauss filtering – four weights have to be calculated – and because it is not separable, the GPU implementation gains a huge benefit from being executed in parallel. Bilateral prefiltering obviously is a perfect candidate for GPU implementations.

The GPU net speedup of affine transformations via *nearest neighbor access* is roughly 15-20 times faster than the CPU code. Trilinear filtering can be accomplished with just one line of code: When working with 3D texture memory, the hardware will use built-in trilinear filtering. As affine transformations lack a high arithmetic intensity – the transformation matrix is precomputed for all voxels – these variants would not exceed net speedups higher than 20 times the CPU speed. As affine transformations and prefilters might be used before the diffusion process starts, one might also take the gross speedup as a basis, which is in the range from 60 up to 75 times the execution speed of the CPU. Affine transformations using B-Splines might be a promising candidate at first glance, but two facts count against it: On the one hand 64 samples must be fetched from memory, on the other hand one can use lookup tables to speed up CPU implementations. By using texture memory we can attenuate at least the ramifications of the many memory fetches and achieve up to 10 times faster execution times than on the CPU.

As the diffusion filters will typically need many iterations, we may neglect the net speedup and only take into account, what *gross* timings got measured for the diffusion filter itself. We can achieve about 55 times the execution speed for linear homogeneous diffusion filtering, 45 times the execution speed for nonlinear inhomogeneous diffusion filtering and 60 times the execution speed for nonlinear anisotropic diffusion filtering (EED). In our program we were able to hide the latency induced by memory access by optimizing and reordering of the source code. As other anisotropic diffusion filters have even higher arithmetic intensity one can expect them to be even faster compared to their CPU variants.

Comparing CPU and GPU speed is a rather complicated issue. One must carefully choose what to oppose and how to draw reasonable conclusions. Let us consider a *real-world application* on a mid-end consumer system. The program was measured on a NVidia GeForce 9800GT being available for 100 USD. The CPU used in that system was an Intel Core 2 Duo E8500 CPU clocked at 3.16 GHz having a market value of round about 240 USD. As the reader has to judge for himself how to weight the presented facts and results, we would like to mention a few more aspects, which must be seen in this context: We used a rather strong CPU, GPUs are evolving much faster, we have optimized GPU and CPU code independently, as both architectures have their own advantages and constraints and we did not make use of the CPUs concurrency which might be used for different calculations as long as the GPU is busy.

A typical example for a complete chain of all implemented filters might be the following: A data set consisting of 256^3 values is translated, rotated and scaled by an affine transformation with trilinear filtering. The resulting data set is then prefiltered with a bilateral filter and afterwards one hundred iterations of EED are applied. Based on the results of our test system, we could expect the following timings:

$$\begin{aligned} t_{\text{cpu}} &= t_{\text{afftri\_cpu}} + t_{\text{bilat\_cpu}} + 100 \cdot t_{\text{eed\_cpu}} \\ &= 1664 \text{ ms} + 301216 \text{ ms} + 100 \cdot 5954 \text{ ms} = 898\ 280 \text{ ms}, \\ t_{\text{gpu}} &= t_{\text{mcpy\_gpu}} + t_{\text{afftri\_gpu}} + t_{\text{bilat\_gpu}} + 100 \cdot t_{\text{eed\_gpu}} \\ &= 51 \text{ ms} + 31 \text{ ms} + 1372 \text{ ms} + 100 \cdot 99 \text{ ms} = 11\ 354 \text{ ms}. \end{aligned}$$

So the GPU code is $\frac{t_{cpu}}{t_{gpu}} \approx 80$ times faster than the CPU solution. The result of such a calculation was available in 11 seconds rather than approximately 15 minutes – without using high end GPU hardware or the usage of idle CPU Threads.

6.2. State of the art GPU

The single step diffusion results of our GPU implementation as well as a CPU implementation for a volume with size 512^3 voxels are given in Table 4. Unless specified otherwise, the timings were taken on a system consisting of an Intel Xeon E5430 CPU (2.66 GHz) and a NVidia GeForce GTX 480 with 480 shader cores. The CPU variants were ported from the GPU code in a straightforward way without any further optimizations.

The table shows the timings, separated into transfer times of the volume to the GPU (obviously not applicable to CPU) and the times needed for one iteration of the code. The

Table 4: Timings and speedups of one iteration step of the discretized PDE, Eq. (5.1), with a data set of size 512^3 voxels (CPU: Intel Xeon E5430; GPU: NVIDIA GTX 480).

		Time ([ms])		Speedup
		MemCpy	Iteration	Iteration
Linear	cpu	0	1 917	1
homogeneous	gpu	256	11	173
Nonlinear inhomo-	cpu	0	33 499	1
geneous (Perona-Malik)	gpu	281	105	320
Nonlinear	cpu	0	118 717	1
anisotropic (EED)	gpu	276	185	641

speedups are given for one iteration alone. Since diffusion equations typically need many iterations we neglect the transfer times and only take the iteration timings into account. We achieve a speedup of about 170 to 320 for the homogeneous and inhomogeneous diffusion. For the nonlinear anisotropic diffusion (EED) the results are even better: Five iterations require 10 minutes CPU time compared to 1 second on the GPU. As other anisotropic diffusion equations have even higher arithmetic intensity one can expect them to be even faster compared to their CPU variants.

The runtimes of single iterations are also visualized in Fig. 8. As the filters are independent of the data, execution times are proportional to volume sizes and particularly the fraction of CPU to GPU times is constant, so one can easily extrapolate to other data sizes. Modern GPUs provide up to 4 GB memory, so it is possible to process data sets with up to 800^3 voxels on the GPU.

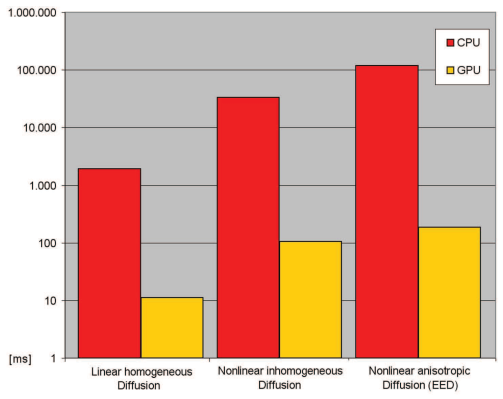


Figure 8: Runtimes for one iteration of the presented kernels from Table 2 on a logarithmic scale (CPU: Intel Xeon E5430; GPU: NVIDIA GTX 480).

As mentioned before, it is important to acquire high-quality first and second order derivatives for the gradient as well as the Hessian. Especially at the beginning of the diffusion process significant noise components may disturb the discrete computation of the derivatives enormously. Also in these experiments we applied several sorts of (pre-

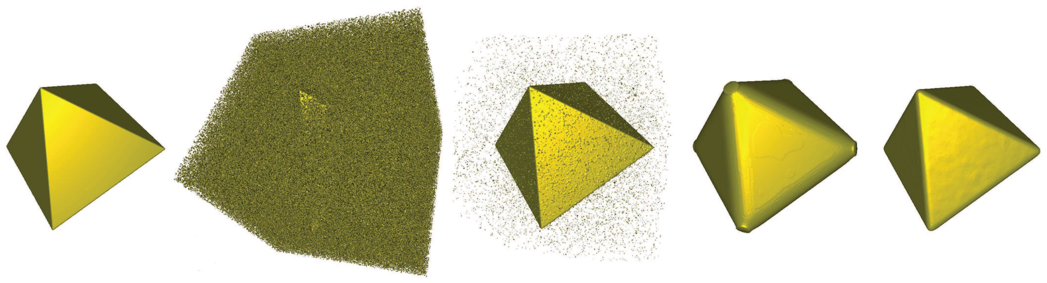


Figure 9: Restoring the functional data $\Phi(x,y,z) = |x| + |y| + |z|$ (256^3 voxels). *From Left to Right*: Isosurface of the ground truth, Gaussian noise added ($\approx$ 20% voxels affected, zero mean and variance 30%), after bilateral prefiltering, after 100 iterations ($\Delta t = 0.05$) of homogeneous and edge enhancing diffusion, respectively, of the prefiltered data set.

)filters and tested their GPU-CPU performance. In our GPU implementation we achieve a speedup of 800 for the bilateral filter. The simpler filters also benefit from massively parallelization, albeit less due to their simplicity. The box filter has a speed up factor of 43, the Gaussian filter one of 151, and the median filter is 76 times faster. All filters took around 1 to 2 seconds on the 512^3 data set, showing the benefit of the newer generation of GPUs enabling larger images without serious extra timing costs. In Fig. 9 an example of a 3D model endowed with a significant amount of noise on the ca. 20% of the voxel values is given. Here applying the bilateral filter before the diffusion definitely makes sense.

7. Conclusions

In this work we have shown that volumetric nonlinear anisotropic diffusion can be mapped *efficiently* onto the GPU. As efficient GPU code should avoid branching if possible, we derived *closed formulas for the 3D eigenvalue analysis* of the shape operator that allows for reducing the problem from 3D object space onto 2D tangent space: We have presented closed formulas for creating a structure frame along the three principal curvature directions, which is especially attractive when local shape structure needs to be considered. Building on that, we defined the diffusion tensor for nonlinear anisotropic diffusion and achieved over 600 times the speed compared to a conventional CPU solution. Among the different possible pre-filters for very noisy images we have seen that the bilateral filter is a promising candidate for being processed on the GPU, achieving 830 times the speed of our CPU solution.

As the technical development of GPUs is rapidly progressing and available memory expands, increasingly larger data sets can be processed directly on the GPU. Apart from scalar volume data, one could also process vector data sets on the GPU, as they arise, for example in DW-MRI (diffusion-weighted magnetic resonance imaging). A starting point for this could be 3D-RGBA-textures, representing 4D vectors. Another question concerns automatic parameter detection. Presumably it was necessary to construct and analyze complete or statistically representative image and gradient histograms, which could be done directly on the GPU. Also, one could examine how CED or hybrid diffusion performs on GPUs, as the arithmetic intensity is higher. Building on successful (pre-)filtering and the diffusion process one could try to deal with segmentation as well, in order to present a seamless GPU solution.

Acknowledgments The authors would like to thank the participants of the Scale Space and Variational Methods conference and Dagstuhl seminar 11471 on Efficient Algorithms for Global Optimisation Methods in Computer Vision for the helpful suggestions. Especially we would like to thank Joachim Weickert for his comments on this research. This work is based on the thesis of AS [17,30] and the resulting SSVM paper [17]. The research of CB was supported in part by the NIH contracts R01-EB00487 and R01-GM074258, and a grant from the UT-Portugal colab project. The research of MG was supported in part by an Emmy Noether grant from the German Research Foundation DFG.

References

[1] L. Agelas and R. Masson. Convergence of the finite volume MPFA O scheme for heterogeneous anisotropic diffusion problems on general meshes. *Comptes Rendus Mathematique*, 346(17-18):1007–1012, 2008.

[2] A. Al-Amoudi, D. Castaño-Diez, D.P. Devos, R.B. Russell, G.T. Johnson, and A.S. Frangakis. The three-dimensional molecular structure of the desmosomal plaque. *Proc. Natl. Acad. Sci. USA*, 108(16):6480–5, 2011.

[3] C.L. Bajaj, Q. Wu, and G. Xu. Level set based volumetric anisotropic diffusion for 3D image denoising. ICES TR03-10, UTexas, Austin USA, 2003.

[4] C.L. Bajaj and G. Xu. Adaptive surfaces fairing by geometric diffusion. In *Symp. CAGD*, pages 731–737, 2001.

[5] C.L. Bajaj and G. Xu. Anisotropic diffusion of surfaces and functions on surfaces. *ACM Trans. Graph.*, 22(1):4–32, 2003.

[6] J. Beyer, C. Langer, L. Fritz, M. Hadwiger, S. Wolfsberger, and K. Bühler. Interactive diffusion-based smoothing and segmentation of volumetric datasets on graphics hardware. *Methods Inf. Med.*, 46(3):270–274, 2007.

[7] A. Binotto, D. Weber, C. Daniel, A. Stork, C.E. Pereira, A. Kuijper, and D. Fellner. Iterative sle solvers over a cpu-gpu platform. In *12th IEEE International Conference on High Performance Computing and Communications, IEEE HPCC-10 (September 1-3, 2010, Melbourne, Australia)*, pages 305–313. IEEE, 2010.

[8] Y. Duan, Y. Wang, X.-C. Tai, and J. Hahn. A fast augmented lagrangian method for euler's elastica model. In *Third International Conference on Scale Space Methods and Variational Methods in Computer Vision (SSVM) (May 29th - June 2nd, 2011, Ein Gedi, Israel), LNCS 6667*, pages 144–156, 2011.

[9] F. Durand and J. Dorsey. Fast bilateral filtering for the display of high-dynamic-range images. *ACM Trans. Graph.*, 21(3):257–266, 2002.

[10] A.S. Frangakis and R. Hegerl. Nonlinear anisotropic diffusion in three-dimensional electron microscopy. In *Scale-Space Theories in Computer Vision, Second International Conference, Scale-Space'99, LNCS 1682*, pages 386–397, 1999.

[11] I. Galic, J. Weickert, M. Welk, A. Bruhn, A. Belyaev, and H.-P. Seidel. The structure of images. *Journal of Mathematical Imaging and Vision*, 31:255–269, 2008.

[12] G.H. Golub and C.F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[13] S. Grewenig, J. Weickert, and A. Bruhn. Parallel implementations of AOS schemes: A fast way of nonlinear diffusion filtering. In *M. Goesele, S. Roth, A. Kuijper, B. Schiele, K. Schindler (Eds.): Pattern Recognition. LNCS 6376*, pages 533–542, 2010.

[14] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M.H. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. *Comp. Graph. Forum*, 24(3):303–312, 2005.

[15] W. Jeong, R. Whitaker, and M. Dobin. Interactive 3D seismic fault detection on the graphics hardware. In *Proc. Volume Graphics*, pages 111–118, 2006.

[16] T. Kalbe, T. Koch, and M. Goesele. High-quality rendering of varying isosurfaces with cubic trivariate c^1-continuous splines. In *Advances in Visual Computing, 5th International Symposium, ISVC 2009 (1), LNCS 5875*, pages 596–607, 2009.

[17] T. Kalbe, A. Schwarzkopf, M. Goesele, A. Kuijper, and C. Bajaj. Volumetric nonlinear anisotropic diffusion on GPUs. In *Third International Conference on Scale Space Methods and Variational Methods in Computer Vision (SSVM) (May 29th - June 2nd, 2011, Ein Gedi, Israel), LNCS 6667*, pages 62–73, 2011.

[18] P.J. Keller, A.D. Schmidt, A. Santella, K. Khairy, Z. Bao, J. Wittbrodt, and E.H.K. Stelzer. Fast, high-contrast imaging of animal development with scanned light sheet-based structured-illumination microscopy. *Nat. Methods*, 7(8):637–42, 2010.

[19] G. Kindlmann, R. Whitaker, T. Tasdizen, and T. Möller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *Proc. IEEE Vis.*, pages 513–520, 2003.

[20] J.J. Koenderink. The structure of images. *Biological Cybernetics*, 50:363–370, 1984.

[21] A. Kuijper. Geometrical PDEs based on second order derivatives of gauge coordinates in image processing. *Image and Vision Computing*, 27(8):1023–1034, 2009.

[22] A. Kuijper, L.M.J. Florack, and M.A. Viergever. Scale space hierarchy. *Journal of Mathematical Imaging and Vision*, 18(2):169–189, April 2003.

[23] A. Kuijper and L.M.J. Florack. The hierarchical structure of images. *IEEE Transactions on Image Processing*, 12(9):1067–1079, 2003.

[24] T. Lindeberg. *Scale-Space Theory in Computer Vision*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1994.

[25] T. Lindeberg. Generalized Gaussian scale-space axiomatics comprising linear scale-space, affine scale-space and spatio-temporal scale-space. *Journal of Mathematical Imaging and Vision*, pages 1–46, 2010.

[26] K. Lipnikov, M. Shashkov, D. Svyatskiy, and Y. Vassilevski. Monotone finite volume schemes for diffusion equations on unstructured triangular and shape-regular polygonal meshes. *Journal of Computational Physics*, 227(1):492–512, 2007.

[27] S. Morigi, M. Rucci, and F. Sgallari. Nonlocal surface fairing. In *Third International Conference on Scale Space Methods and Variational Methods in Computer Vision (SSVM) (May 29th - June 2nd, 2011, Ein Gedi, Israel), LNCS 6667*, pages 38–49, 2011.

[28] S. Paris, P. Kornprobst, J. Tumblin, and F. Durand. A gentle introduction to bilateral filtering and its applications. SIGGRAPH course, 2007.

[29] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 12:629–639, 1990.

[30] A. Schwarzkopf. Volumetrische anisotrope diffusion auf der GPU. Technische Universität Darmstadt, 2010.

[31] C. Sigg. *Representation and Rendering of Implicit Surfaces*. PhD thesis, ETH Zurich, 2006.

[32] S. Tabik, E.M. Garzon, I. Garcia, and J.J. Fernandez. Implementation of anisotropic nonlinear diffusion for filtering 3D images in structural biology on SMP clusters. In *Parallel Computing:*

Current & Future Issues of High-End Computing,, volume 33 of *Proc. Int. Conf. ParCo*, pages 727–734, 2005.

[33] T. Tasdizen, R. Whitaker, P. Burchard, and S. Osher. Geometric surface smoothing via anisotropic diffusion of normals. In *Proc. VIS '02*, pages 125–132, 2002.

[34] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Proc. IEEE Int. Conf. on Computer Vision '98*, pages 839–846, 1998.

[35] J. Weickert. Scale-space properties of nonlinear diffusion filtering with a diffusion tensor. Technical report, No. 110, Laboratory of Technomathematics, University of Kaiserslautern, Germany, 1994.

[36] J. Weickert, K.J. Zuiderveld, B.M. ter Haar Romeny, and W.J. Niessen. Parallel implementations of AOS schemes: A fast way of nonlinear diffusion filtering. In *Proc. 1997 IEEE International Conference on Image Processing (ICIP-97, Santa Barbara, Oct. 26-29, 1997), Vol. 3*, pages 396–399, 1997.

[37] J. Weickert. A review of nonlinear diffusion filtering. In *Scale-Space Theory in Computer Vision*, volume 1252 of *Lecture Notes in Computer Science*, pages 1–28. Springer Berlin/Heidelberg, 1997.

[38] J. Weickert. *Anisotropic Diffusion in Image Processing*. B.G. Teubner Stuttgart, 1998.

[39] J. Weickert. Coherence enhancing diffusion filtering. *International Journal of Computer Vision*, 31:111–127, 1999.

[40] J. Weickert. Partial differential equations in image processing and computer vision. Habilitation thesis, University of Mannheim, 2001. `http://www.mia.uni-saarland.de/weickert/Papers/habil.pdf`.

[41] X.F. Zhang, W.F. Chen, L. Qian, and H. Ye. Affine invariant non-linear anisotropic diffusion smoothing strategy for vector-valued images. *Imaging Science Journal*, 58(3):119–124, 2010.

[42] Y. Zhao. Lattice Boltzman based PDE solver on the GPU. *The Visual Computer*, 24(5):323–333, 2008.