Nonlinear Diffusion in Graphics Hardware

M. Rumpf and R. Strzodka

Universität Bonn

Abstract. Multiscale methods have proved to be successful tools in image denoising, edge enhancement and shape recovery. They are based on the numerical solution of a nonlinear diffusion problem where a noisy or damaged image which has to be smoothed or restorated is considered as initial data. Here a novel approach is presented which will soon be capable to ensure real time performance of these methods. It is based on an implementation of a corresponding finite element scheme in texture hardware of modern graphics engines. The method regards vectors as textures and represents linear algebra operations as texture processing operations. Thus, the resulting performance can profit from the superior bandwidth and the build in parallelism of the graphics hardware. Here the concept of this approach is introduced and perspectives are outlined picking up the basic Perona Malik model on 2D images.

1 Introduction

Nonlinear diffusion in multiscale image processing attracts growing interest in the last decade. Methods based on this approach are frequently used tools in image denoising, edge enhancement and shape recovery [1, 10, 12, 9]. Therein the image is considered as initial data of a suitable evolution problem. Time in the evolution represents the scale parameter which leads from noisy, fine scale to smoothed and enhanced, coarse scale representation of the data. The same kind of diffusion models can also be used for the visualization of flow fields through the construction of streamline type patterns [4].

Here our focus is on the efficient implementation of finite element schemes for the solution of the nonlinear diffusion problem. We pick up the regularized Perona and Malik model and rewrite the corresponding linear algebra operations as image processing operations supported by graphics hardware. Thus they act on vectors which are regarded as images. Before we describe the approach in detail let us argue why this unusual approach is expected to ensure superior performance over a standard implementation in software although nowadays CPU performance is superior compared to the computing performance of single operations on a graphics unit.

Memory bandwidth has become a major limiting factor in many scientific computations. Nowadays performance highly depends on the implementation's beneficial use of the hierarchy of caching levels. But automation fails here and the task of optimal use of the memory hierarchy for a given application is very complex. On the other hand PC graphics hardware has undergone a rapid development boosting its performance and functionality and thus releasing the CPU from many computations. Particularly in volume graphics, texture hardware is extensively exploited for a significant performance increase leading to interactive applications [3, 14, 13]. As an example which goes beyond basic graphics operations we cite here Hopf et al. who discussed Gaussian filtering and wavelet transformations in hardware [5, 6].

We proceed along this line and further widen the range of applications even by demonstrating that the functionality of modern graphics cards has reached a state, where the graphics processor unit may be regarded as a programmable parallel fixed-point coprocessor for certain scientific computing purposes. Observing the precision restrictions, it may be used for numerical computations where ultimate precision is not required. Then we benefit from the much higher memory bandwidth and the parallel execution of commands on large data blocks. Partial differential equations in image processing are exactly of this type. They involve large image data and our aim is not to compute exact solutions but to model numerically properties which are known for the continuous model. In case of the nonlinear diffusion these are the decreasing diffusivity in areas of large gradients and the smoothing in image regions which are expected to be apart from edges. Furthermore a maximum principle is regarded as an important property.

We will first review the nonlinear diffusion model and then concentrate on the adaption of the numerical scheme to this graphics oriented setting.

2 Nonlinear Diffusion

We briefly review the model and the discretization of the nonlinear diffusion in image processing, based on a modification of the Perona-Malik [9] model proposed by Catté, Lions, Morel, and Coll [2]. We consider the domain $\Omega := [0,1]^d$, $d = 2, 3$ and ask for solution of the following nonlinear parabolic, boundary and initial value problem: Find $u : \mathbb{R}^+ \times \Omega \to \mathbb{R}$ such that

$$\begin{aligned} \tfrac{\partial}{\partial t} u - \operatorname{div}\left(g(\nabla u_\epsilon)\nabla u\right) = 0 \quad &, \quad \text{in } \mathbb{R}^+ \times \Omega\,, \\ u(0,\cdot) = u_0\,, \quad & \text{on } \Omega\,, \\ \tfrac{\partial}{\partial \nu} u = 0 \quad &, \quad \text{on } \mathbb{R}^+ \times \partial\Omega. \end{aligned}$$

where in the basic model g is a non negative monotone decreasing function $g : \mathbb{R}_0^+ \to \mathbb{R}+$ satisfying $\lim_{s\to\infty} g(s) = 0$, e. g. $g(s) = (1+s^2)^{-1}$, and u_ϵ is a mollification of u with some smoothing kernel. The solution $u : \mathbb{R}^+ \times \Omega \to \mathbb{R}$ can be regarded as a multiscale of successively diffused images $u(t), t \in R^+$. With respect to the shape of the diffusion coefficient function g, the diffusion is of regularized "backward" type [7] in regions of high image gradients, and of linear type in homogeneous regions.

We discretize the problem with bilinear, respectively trilinear conforming finite elements on a uniform quadrilateral, respectively hexahedral grid. In time a semi-implicit first order Euler scheme is used, as purely explicit schemes pose very restrictive conditions on the timestep width. In variational formulation with respect to the FE space $\mathcal{V}^h$ we obtain:

$$\left(\frac{U^{k+1} - U^k}{\tau}, \theta\right)_h + \left(g(\nabla U^k_\epsilon)\nabla U^{k+1}, \nabla\theta\right) = 0$$

for all $\theta \in \mathcal{V}^h$. Here $(\cdot,\cdot)$ denotes the L^2 product on the domain Ω, $(\cdot,\cdot)_h$ is the lumped masses product [11], which approximates the L^2 product, and τ the current timestep width. The discrete solution U^k is expected to approximate $u(\tau k)$. Thus in the kth timestep we have to solve the linear system

$$(M_h + \tau L(U_\epsilon^k))\bar{U}^{k+1} = M_h \bar{U}^k \tag{1}$$

where $\bar{U}^k$ is the solution vector consisting of the nodal values, $M_h := \left((\Phi_\alpha, \Phi_\beta)_h\right)_{\alpha\beta}$ the lumped mass matrix, $L(U_\epsilon^k) := \left((g(\nabla U_\epsilon^k)\nabla\Phi_\alpha, \nabla\Phi_\beta)\right)_{\alpha\beta}$ the weighted stiffness matrix and Φ_α the "hat shaped" multilinear basis functions. In the concrete algorithm we replace $g(\nabla U_\epsilon^k)$ on elements by the value at the elements' center point.

As the graphics hardware offers only a fixed-point number format, it is important that we separate the small, grid specific element diameter h from the dimensionless diffusion coefficients. Thus both the coefficients and the factor $\frac{\tau}{h^2}$ are close to 1. For an equidistant grid we may rescale the above equation and get

$$\underbrace{\left(I + \frac{\tau}{h^2}\hat{L}(U_\epsilon^k)\right)}_{A^k(\bar{U}^k)} \bar{U}^{k+1} = \underbrace{\bar{U}^k}_{}$$
$$\bar{U}^{k+1} = \bar{R}^k(\bar{U}^k),$$

with the rescaled stiffness matrix $\hat{L}(U_\epsilon^k) := \left((g(\nabla U_\epsilon^k)\nabla\hat{\Phi}_\alpha, \nabla\hat{\Phi}_\beta)\right)_{\alpha\beta}$ defined by reference multilinear basis functions $\hat{\Phi}_\alpha$ with support $[-1,1]^d$.

Any implementation, also that in graphics hardware has to solve the above linear system of equations. In the following section we will therefore first consider the operations involved in solving a general linear system of equations and describe how they can be split into more basic algebraic operations, which are directly supported by graphics hardware.

3 Operations in Linear Iterative Solvers

In fact, many discretizations of partial differential equations lead to a sparse linear system of equations $A(\bar{U}^k)\bar{U}^{k+1} = \bar{R}(\bar{U}^k)$, where the matrix $A \in \mathbb{R}^{n+1,n+1}$ and the right hand side $\bar{R}$ depend on the solution vector $\bar{U}^k$ of the preceding timestep. Frequently an iterative solver is applied to approximate the solution, i.e. we consider an iteration $\bar{X}^{l+1} = F(\bar{X}^l)$ with $\bar{X}^0 = \bar{R}$. Typical smoothers are the Jacobi iteration

$$F(\bar{X}) = D^{-1}(\bar{R} - (A - D)\bar{X}), \qquad D := \text{diag}(A) \tag{2}$$

and the conjugate gradient iteration

$$F(\bar{X}^l) = \bar{X}^l + \frac{\bar{r}^l \cdot \bar{p}^l}{A\bar{p}^l \cdot \bar{p}^l}\bar{p}^l, \qquad \bar{p}^l = \bar{r}^l + \frac{\bar{r}^l \cdot \bar{r}^l}{\bar{r}^{l-1} \cdot \bar{r}^{l-1}}\bar{p}^{l-1}, \quad \bar{r}^l = \bar{R} - A\bar{X}^l \tag{3}$$

In the above formulas we can easily identify the required operations: matrix vector product, scalar product, componentwise linear combination, componentwise multiplication, application of a componentwise function, vector norm.

The first two of these operations are not directly supported by graphics hardware. Therefore we must split them into more primitive ones. The scalar product may be reformulated using the componentwise multiplication (denoted by '$\bullet$') and a vector norm $\bar{V} \cdot \bar{W} = \|\bar{V} \bullet \bar{W}\|_1$.

The matrix vector product may be expressed in terms of componentwise products with the matrix' subdiagonals $\bar{A}^\gamma := (A_{\alpha-\gamma,\alpha})_\alpha$ which are vectors, and subsequent index shift operations T_γ on vectors, defined by $T_\gamma \circ \bar{V} := (V_{\alpha+\gamma})_\alpha$:

$$(A\bar{X})_\alpha = \sum_\beta A_{\alpha,\beta} X_\beta = \sum_{|\gamma|<n} (\bar{A}^\gamma)_{\alpha+\gamma} X_{\alpha+\gamma}$$

$$A\bar{X} = \sum_{|\gamma|<n} T_\gamma \circ (\bar{A}^\gamma \bullet \bar{X}). \tag{4}$$

Above, $\alpha = 0, \ldots, n;\ \beta = 0, \ldots, n$ range over the matrix' lines or columns respectively, and $\gamma := \beta - \alpha = -n, \ldots, 0, \ldots, n$ indexes the subdiagonals. Elements of the subdiagonal vectors $\bar{A}^\gamma$ indexing matrix elements outside of the matrix A are supposed to be zero, e.g. the first element of the vector $\bar{A}^1$ is $(A^1)_0 = A_{-1,0} = 0$. If most subdiagonals of A are zero, which is always true for FE schemes, then γ ranges only over few nontrivial subdiagonals.

Thus we have successfully split the operations for the linear iterative solvers (2) and (3) into hardware supported functions. Table 1 lists these operations together with their counterparts in graphics hardware.

Table 1. Basic operations in linear iterative solvers.

operation	**formula**	**graphics operation**
linear combination	$a\bar{V} + b\bar{W}$	image blending
multiplication	$\bar{V} \bullet \bar{W}$	image blending
general function	$f \circ \bar{V}$	lookup table
index shift	$T_\gamma \circ \bar{V}$	change of coordinates
vector norms	$\|\bar{V}\|_k, k = 1, \ldots, \infty$	image histogram

4 Rewriting the FE Scheme

Now we return to the FE scheme for the nonlinear diffusion introduced in section 2. The general approach to the decomposition of the matrix vector product given in the previous section, is feasible in this case. The matrix A^k consists of only 3^d nontrivial subdiagonals.

Since in this application the vectors $\bar{U}^k = (U^k_\alpha)_\alpha$ represent images, it is appropriate to let α be a 2 or 3 dimensional multi-index, enumerating the nodes of the 2 or 3 dimensional grid respectively. Then the index offset $\gamma := \beta - \alpha$ is the spatial offset from

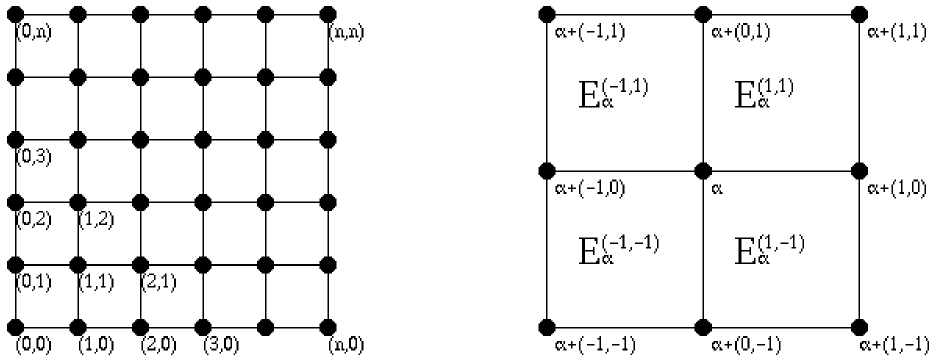

Fig. 1. On the left a 2D grid enumerated by a multi-index, on the right the neighboring elements and the local multi-index offset to neighboring nodes.

node α to node β. Figure 1 shows the enumeration for a 2D grid, and all the 3^2 offsets γ and the neighboring elements E_α^γ for a given node α.

To perform the decomposed matrix vector product (cp. (4)) we need to identify the elements of the subdiagonal vectors $\bar{A}^\gamma$, which now can themselves be regarded as images. For this task it suffices to consider the subdiagonal $\bar{L}^\gamma$ of $\hat{L}(U_\epsilon^k)$, since $\bar{A}^\gamma = \delta_\gamma \bar{1} + \frac{\tau}{h^2}\bar{L}^\gamma$, where δ_γ is the Kronecker symbol. In fact the identity indicated by $\delta_\gamma \bar{1}$ deserves no further attention. By definition we have

$$\bar{L}_\alpha^\gamma = \left(g(\nabla U_\epsilon^k) \nabla \hat{\Phi}_{\alpha-\gamma}, \nabla \hat{\Phi}_\alpha \right) .$$

Since we evaluate $g(\nabla U_\epsilon^k)$ by the midpoint rule on elements we may rewrite the matrix element for the node α as a weighted sum over contributions on the neighboring elements

$$\bar{L}_\alpha^\gamma = \sum_{E \in E(\alpha)} G_E^k C_{\gamma,E} \, ,$$

where $E(\alpha) := \{E_\alpha^\gamma \| \gamma| = d\}$ is defined as the set of elements around the node α (Fig. 1), $G_E^k := \left. g(\nabla U_\epsilon^k)\right|_E$ is the constant value of the diffusion coefficient at the element's center of mass and $C_{\gamma,E} := \left(\nabla \hat{\Phi}_{\alpha-\gamma}|_E, \nabla \hat{\Phi}_\alpha|_E \right)$ a local stiffness matrix entry. On an equidistant grid the values $C_{\gamma,E}$ depend only on γ. Since γ takes only 3^d different values, they can be precomputed.

As we have seen, we do not have to store the matrix A^k for the computation of the matrix vector product. Instead we precompute the values G_E^k - again interpreted as an image - only once for every timestep and then split the matrix vector product in the linear solver into few (3^d) products with the subdiagonals $\bar{L}^\gamma$ (cp. (4)):

$$\hat{L}(U_\epsilon^k)\bar{X} = \sum_{|\gamma| \le d} T_\gamma \circ (\bar{L}^\gamma \bullet \bar{X}).$$

In all these calculations we take care of the natural boundary condition by duplicating the borders of the image $\bar{U}^k$.

5 Implementation

Graphics cards are customized in a big variety of designs, but operationally they consist of only two main components: the graphics processor unit (GPU) and the graphics memory. (Strictly speaking, the GPU splits into a geometry and raster engine and not every graphics card has its own memory.) The GPU processes data from the graphics memory very much like the CPU does from the main memory. The most significant difference is the unified processing of data blocks by the GPU. For example, where the CPU needs to run over all nodes of a grid to perform an addition of two nodal vectors, the GPU takes only a few commands for the same task. If we stick to the analogy then the so-called framebuffers serve the same purpose in numerical computations for the GPU, as the registers do for the CPU. Usually there are at least two such framebuffers: the front buffer which is displayed on the screen and the back buffer where we can perform the calculations invisibly.

An important issue with graphics boards are the number formats supported by the GPU. Resulting from the inherent use, only the range $[0, 1]$ suitable for the representation of intensities is supported (Some GPUs offer extended formats during calculations). In any case we have to encode our numbers to cover a wider range, say $[-\rho_0, \rho_1]$. By nonlinear transformations, also unbounded intervals could be covered, but it is doubtful whether the low precision of the numbers may resolve these intervals sufficiently. Furthermore linear encoding has the advantage that there are many stages in the graphics pipeline where linear transformations can be applied, saving multiple processing of the same operand.

Table 2. Correspondence of operations in numbers and intensities.

Numbers	**Intensities**
$r : x \to \frac{1}{2\rho}(x+\rho)$	
$a \in [-\rho, \rho]$	$r(a) \in [0,1]$
$a + b$	$2\left(\frac{r(a)+r(b)}{2}\right) - \frac{1}{2}$
ab	$\frac{1+\rho}{2} - \rho\,(r(a)(1-r(b)) + r(b)(1-r(a)))$
$\alpha a + \beta$	$\alpha r(a) + (\frac{\beta}{2\rho} + \frac{1-\alpha}{2})$
$f(a)$	$(r \circ f \circ r^{-1})(r(a))$
$\sum_\alpha \alpha a_\alpha$	$\sum_\alpha \alpha r(a_\alpha) + \frac{1}{2}(1 - \sum_\alpha \alpha)$
$\rho(2y-1) \leftarrow y : r^{-1}$	

In Table 2 we list which operations on the intensities (the encoded values in the images) correspond to the desired operations on numbers (the represented values). The left column shows the operation to be performed, whereas the right column shows which operation must be performed on the encoded operands to obtain the equivalent encoded result. Obviously no other operations than those already discussed are needed to perform these transformed calculations.

By choosing ρ sufficiently large such that any intermediate computations in numbers do not transcend the range $[-\rho, \rho]$, we assure that the corresponding computations in intensities will not transcend the range $[0, 1]$ either. On the other hand a large ρ decreases the resolution of numbers, therefore it should be chosen application dependent as small as possible. The symmetric choice of the interval covers the typical number range of FE schemes and has the advantage of simpler encoded operations on intensities (Table 2).

Below we have outlined the control structures of the algorithm in pseudo code notation.

```
nonlinear diffusion $\{$
  load the original image and the parameters;
  initialize graphics hardware;
  encode the original image in graphics memory $\bar{U}^0$;
  for each timestep $k$ $\{$
    store the right hand side image $\bar{R}^k = \bar{U}^k$;
    calculate the image consisting of diffusion coefficients $\bar{G}^k = \left( g(\nabla U^k_\epsilon)\big|_E \right)_E$;
    initialize the solver $\bar{X}^0 = \bar{R}^k$;
    for each iteration $l$
      calculate a step of the iterative solver $\bar{X}^{l+1} = F(\bar{X}^l)$;
    store the solution $\bar{U}^{k+1} = \bar{X}^{l+1}$
    decode the solution and display it;
  $\}$
$\}$
```

Now, considering an implementation in OpenGL [8], the basic operations from Table 1 are more or less directly mapped onto OpenGL functionality. The addition and multiplication are achieved by selecting the proper source and destination factors for the blending function (glBlendFunc). Concerning the implementation of a general function of one variable we should keep in mind that the intensities are discretized by m bits, with $m \leq 12$. A general function can thus be represented by 2^m entries in a table. OpenGL can use such a table to automatically output the values indexed by the intensities of an image (glPixelMap), thus applying the designated function to the image. For the index-shift one simply has to change the drawing position for the image. Concerning the vector norms there is a slight difference in implementation, since the OpenGL's histogram extension does not calculate them directly in the GPU, as the other OpenGL methods do, but returns a histogram of pixel intensities (glGetHistogram) from which the CPU has to compute the norm. Let $H : \{0, \dots, 2^m - 1\} \to \mathbb{N}$ be such a histogram assigning the number of appearances to every intensity of the image $\bar{V}$, then

$$\|\bar{V}\|_k = \left(\sum_{y=0}^{2^m-1} \left(r^{-1}(y)\right)^k \cdot H(y) \right)^{\frac{1}{k}},$$

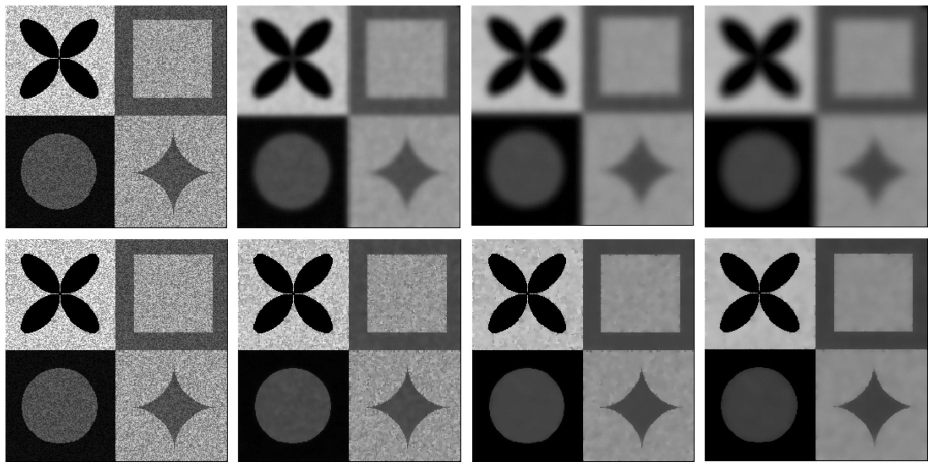

Fig. 2. Comparison of hardware implemented linear (upper row) and nonlinear diffusion (lower row).

for $k = 1, 2, \dots$, and for $k = \infty$ we simply pick up the largest $|r^{-1}(y)|$ with $H(y) > 0$, where r^{-1} is the inverse transformation from intensities to numbers. However apart from the overall control structure of the programm, this is the only computation done by the CPU while using the conjugate gradient solver. For the Jacobi solver no CPU calculation at all is required.

6 Results

The computations have been performed on a SGI Onyx2 4x195MHz R10000, with InfiniteReality2 graphics, using 12 bit per color component and the number interval $[-2, 2]$, i.e. $\rho = 2$. Convolution with a Gaussian kernel, which is equivalent to the application of a linear diffusion model is compared to the results from the nonlinear model in Fig. 2. This test strongly underlines the edge conservation of the nonlinear diffusion model.

Figure 3 shows computations with graphics hardware using the Jacobi and the cg-solver and compares them to computations in software. The precision used in the GPU obviously sufficies for the task of denoising pictures by nonlinear diffusion. Although the images produced by hardware and software differ, the visual effect is very comparable, and this is the decisive criterion in such applications.

Currently, 100 iterations of the Jacobi, cg-solver for 256^2 images take about 17 sec and 42 sec respectively, which is still slower than the software solution. The reason for this surprisingly weak performance is easily identified in the unbalanced performance of data transfer between the framebuffer and graphics memory. Before, we have already mentioned that the back buffer serves as a register, where auxiliary results are computed before they are stored in a variable in graphics memory. Because nearly all operations effect the back buffer, its access times are highly relevant for the overall performance.

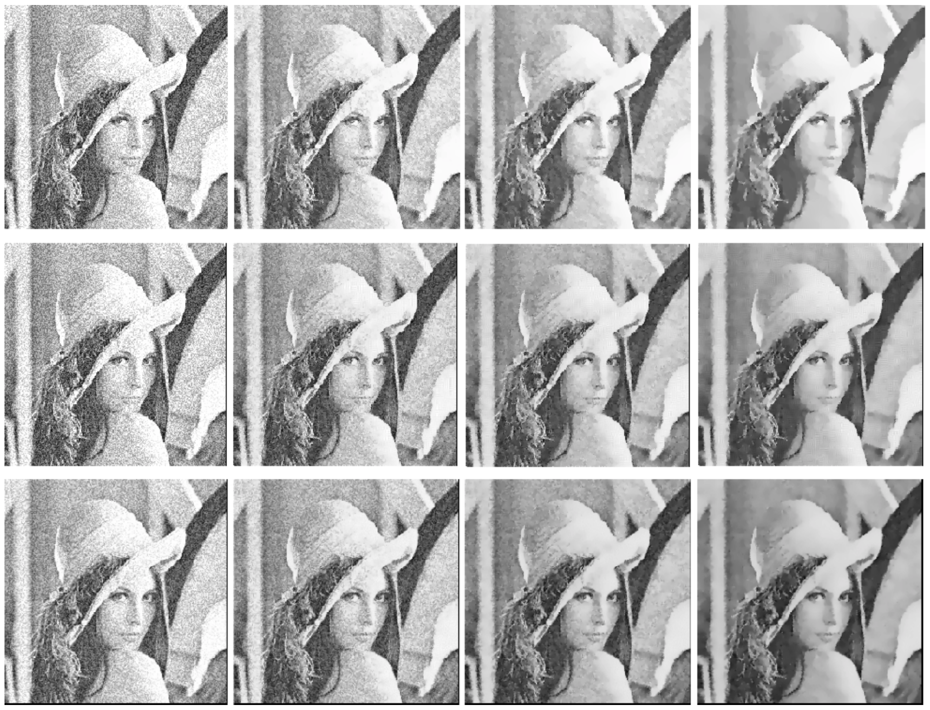

Fig. 3. Comparison of nonlinear diffusion solvers, first row: adaptive software preconditioned cg; second row: jacobi sover in graphics hardware; third row: cg-solver in graphics hardware.

But compared to a computation in software where the reading and writing of a register in the CPU takes the same time, because the operations are supposed to be needed just as often, the graphics of the Onyx2, in contrary, is writing an image from the graphics memory to the framebuffer about 60 times faster than reading it back, because the reading back from the framebuffer to graphics memory is not a very common operation in graphics applications. The histogram extension used for the computation of the scalar products in the cg-solver is even less common, and being even slower than the reading, it further reduces performance. However, the growing use of such extensions in different areas of visualization and image processing will certainly lead to an optimization. There are already graphics cards with less discrepant read/write operations between the framebuffer and the graphics memory and we are working on a respective porting which, however, incorporates some additional difficulties.

7 Conclusions

We have introduced a framework which facilitates the use of modern graphics boards as fixed-point coprocessors for image processing. By showing how common PDE solvers can be split into basic operations, which are directly supported by graphics hardware, we have demonstrated that a wide range of applications could benefit from the large memory bandwidth, which usually is the bottleneck in many scientific calculations. The implementation of nonlinear diffusion has underlined how existing algorithms can quickly be adapted to this graphics oriented setting and that the low precision of numbers does not do any harm to many applications aiming at visual results. The visualization of flow fields based on this approach, for example, is one of our future goals. Finally we have discussed the issue of performance which could not fully unfold itsself yet. But we are very confident that in the near future the application of new graphics cards and drivers will overcome this difficulty, raising the speed of such implementations far beyond pure software solutions.

Acknowledgments

We would like to thank Matthias Hopf from Stuttgart for a lot of valuable information on the graphics hardware programming and Michael Spielberg from Bonn for assistance in the adaption of the finite element code to the graphics hardware.

References

1. L. Alvarez, F. Guichard, P. L. Lions, and J. M. Morel. Axioms and fundamental equations of image processing. *Arch. Ration. Mech. Anal.*, 123(3):199–257, 1993.
2. F. Catté, P.-L. Lions, J.-M. Morel, and T. Coll. Image selective smoothing and edge detection by nonlinear diffusion. *SIAM J. Numer. Anal.*, 29(1):182–193, 1992.
3. T.J. Cullip and U. Neumann. Accelerating volume reconstruction with 3d texture hardware. Technical Report TR93-027, University of North Carolina, Chapel Hill N.C., 1993.
4. U. Diewald, T. Preußer, and M. Rumpf. Anisotropic diffusion in vector field visualization on euclidean domains and surfaces. *Trans. Vis. and Comp. Graphics*, 6(2):139–149, 2000.
5. M. Hopf and T. Ertl. Accelerating 3d convolution using graphics hardware. In *Visualization '99*, pages 471–474, 1999.
6. M. Hopf and T. Ertl. Hardware accelerated wavelet transformations. In *Symposium on Visualization VisSym '00*, 2000.
7. B. Kawohl and N. Kutev. Maximum and comparison principle for one-dimensional anisotropic diffusion. *Math. Ann.*, 311 (1):107–123, 1998.
8. OpenGL Architectural Review Board (ARB), http://www.opengl.org/. *OpenGL: graphics application programming interface (API)*, 1992.
9. P. Perona and J. Malik. Scale space and edge detection using anisotropic diffusion. In *IEEE Computer Society Workshop on Computer Vision*, 1987.
10. J. A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1999.
11. V. Thomee. *Galerkin - Finite Element Methods for Parabolic Problems*. Springer, 1984.
12. J. Weickert. *Anisotropic diffusion in image processing*. Teubner, 1998.
13. R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. *Computer Graphics (SIGGRAPH '98)*, 32(4):169–179, 1998.
14. O. Wilson, A. van Gelder, and J. Wilhelms. Direct volume rendering via 3d textures. Technical Report UCSC CRL 94-19, University of California, Santa Cruz, 1994.