Linear Algebra Operators for GPU Implementation of Numerical Algorithms

Jens Krüger and Rüdiger Westermann
Computer Graphics and Visualization Group, Technical University Munich*

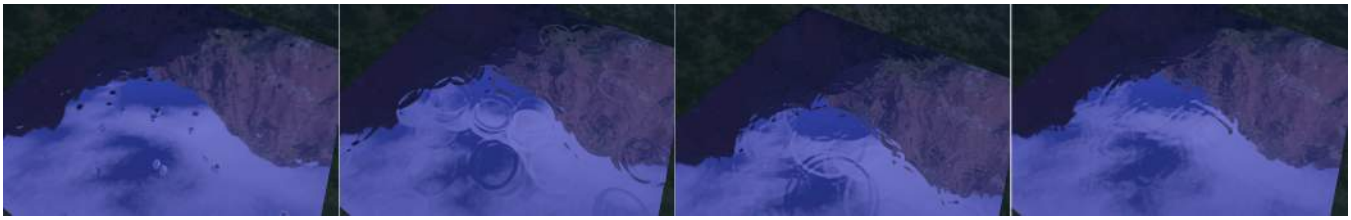

Figure 1: *We present implementations of techniques for solving sets of algebraic equations on graphics hardware. In this way, numerical simulation and rendering of real-world phenomena, like 2D water surfaces in the shown example, can be achieved at interactive rates.*

Abstract

In this work, the emphasis is on the development of strategies to realize techniques of numerical computing on the graphics chip. In particular, the focus is on the acceleration of techniques for solving sets of algebraic equations as they occur in numerical simulation. We introduce a framework for the implementation of linear algebra operators on programmable graphics processors (GPUs), thus providing the building blocks for the design of more complex numerical algorithms. In particular, we propose a stream model for arithmetic operations on vectors and matrices that exploits the intrinsic parallelism and efficient communication on modern GPUs. Besides performance gains due to improved numerical computations, graphics algorithms benefit from this model in that the transfer of computation results to the graphics processor for display is avoided. We demonstrate the effectiveness of our approach by implementing direct solvers for sparse matrices, and by applying these solvers to multi-dimensional finite difference equations, i.e. the 2D wave equation and the incompressible Navier-Stokes equations.

CR Categories: I.6.7 [Simulation and Modeling]: Simulation Support Systems—; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—

Keywords: Numerical Simulation, Graphics Hardware

1 Introduction

The development of numerical techniques for solving partial differential equations is one of the traditional subjects in applied mathematics. These techniques have a variety of applications in physics based simulation and modelling, and they have been frequently employed in computer graphics to provide realistic simulation of real-world phenomena [Kaas and Miller 1990; Chen and da Vitoria Lobo 1995; Foster and Metaxas 1996; Stam 1999; Foster and Fedkiw 2001; Fedkiw et al. 2001]. Despite their use in numerical simulation, these techniques have also been applied in a variety of computer graphics settings, e.g. the simulation of watercolor drawings [Curtis et al. 1997], the processing of polygonal meshes [Desbrun et al. 1999], or the animation of deformable models [Baraff and Witkin 1998; Debunne et al. 2001], to mention just a few.

The numerical complexity of techniques for solving sets of algebraic equations often imposes limitations on the numerical accuracy or extremely high demands on memory and computing resources. As a consequence thereof, parallelization of numerical solvers on multi-processor architectures has been an active research area for quite a long time.

An alternative direction of research is leading towards the implementation of general techniques of numerical computing on computer graphics hardware. Driven by the evolution of graphics processors from fixed function pipelines towards fully programmable, floating point pipelines, additional effort is spent on the development of numerical algorithms amenable to the intrinsic parallelism and efficient communication on modern GPUs. Recent examples include GPU implementations of matrix multiplications [Thompson et al. 2002], multi-grid simulation techniques [Bolz et al. 2003] and numerical solution techniques to least squares problems [Hillesland et al. 2003]. Particularly in computer graphics applications, the goal of such implementations of numerical techniques is twofold: to speed up computation processes, as they are often at the core of the graphics application, *and* to avoid the transfer of computation results from the CPU to the GPU for display.

Based on early prototypes of programmable graphics architectures [Olano and Lastra 1998; Lindholm et al. 2001], the design of graphics hardware as a pipeline consisting of highly optimized stages providing fixed functionality is more and more abandoned on modern graphics chips, e.g. the NVIDIA NV30 [Montrym and Moreton 2002] or the ATI R300 [Elder 2002]. Today, the user has access to parallel geometry and fragment units, which can be programmed by means of standard APIs. In particular, vertex and pixel shader programs enable direct access to the functional units, and they allow for the design of new classes of hardware supported

*jens.krueger, westermann@in.tum.de

Reprinted from "Linear Algebra Operators for GPU Implementation of Numerical Algorithms" in ACM Transactions on Graphics {Volume 22 , Issue 3 (July 2003)} © ACM, 2003.
http://doi.acm.org/10.1145/882262.882363

graphics algorithms. As a representative example for such algorithms, let us refer to [Purcell et al. 2002], where ray-tracing on programmable fragment processors was described.

In our current work, we employ the Pixel Shader 2.0 API [Microsoft 2002], a specific set of instructions and capabilities in DirectX9-level hardware, which allows us to perform hardware supported per-fragment operations. Besides basic arithmetic operations, instructions to store intermediate results in registers and to address and access multiple texture units are available. Our target architecture is the ATI Radeon 9800, which supports 32-bit floating point textures as well as a set of hardware supported pixel shader. Pixel shader provide 24-bit precision internal computations. To save rendering results and to communicate these results to consecutive passes, rendering can be directed to a 32-bit offscreen buffer. This buffer can be directly bound to a texture map, i.e. the content of the buffer is immediately available as 2D texture map.

Until today, the benefits of graphics hardware have mainly been demonstrated in rendering applications. Efforts have been focused on the creation of static and dynamic imagery including polygonal models and scalar or vector valued volumetric objects. In a few examples, strategies to realize numerical computations on graphics processors were described, usually implemented by means of low-level graphics APIs that did not yet provide programmable vertex and pixel shader.

Hopf et al. [Hopf and Ertl 1999; Hopf and Ertl 2000] described implementations of morphological operations and wavelet transforms on the graphics processor. Numerical computations were realized by means of blending functionality, and by exploiting the functionality provided by the OpenGL imaging subset. Using similar coding mechanisms, the simulation of cellular automata and stochastic fractals was demonstrated in [nVidia 2002; Hart 2001]. Strzodka and Rumpf [Strzodka and Rumpf 2001a; Strzodka and Rumpf 2001b] proposed first concepts for the implementation of numerical solvers for partial differential equations on graphics hardware. Therefore, the intrinsic communication and computation patterns of numerical solution techniques for finite difference equations were mapped to OpenGL functionality. Non-standard OpenGL extensions were employed in [Heidrich et al. 1999; Jobard et al. 2000; Weiskopf et al. 2001] to interactively visualize 2D vector fields. At the core of these techniques, vector valued data is encoded into RGB texture maps, thus allowing for the tracing of particles by successive texture fetch and blend operations.

With the availability of programmable fragment units, the possibility to implement general numerical techniques on graphics processors was given. A number of examples have been demonstrated since then, ranging from physics based simulation of natural phenomena to real-time shading and lighting effects [nVidia 2003; ATI 2003]. Weiskopf et al. [Weiskopf et al. 2002] extended their previous work towards the interactive simulation of particle transport in flow fields. Recently, Harris et al. [Harris et al. 2002] described the implementation of an explicit scheme for the solution of a coupled map lattice model on commodity graphics cards. In both examples, numerical computations were entirely carried out in a fragment shader program. GPU implementation of matrix-multiplies based on a particular distribution strategy for 2D textures across a cube-shaped lattice was described in [Larsen and McAllister 2001].

In contrast to previous approaches, which were specifically designed with regard to the solution of particular problems, our goal is to develop a generic framework that enables the implementation of general numerical techniques for the solution of difference equations on graphics hardware. Therefore, we provide the basic building block routines that are used by standard numerical solvers. Built upon a flexible and efficient internal representation, these functional units perform arithmetic operations on vectors and matrices. In the same way as linear algebra libraries employ encapsulated basic vector and matrix operations, many techniques of numerical computing can be implemented by executing GPU implementations of these operations in a particular order. One of our goals is to replace software implementations of basic linear algebra operators as available in widespread linear algebra libraries, i.e. the BLAS (Basic Linear Algebra Subprogram) library [Dongarra et al. 1988; Dongarra et al. 1990], by GPU implementations, thus enabling more general linear algebra packages to be implemented on top of these implementations, i.e. the LAPACK (Linear Algebra Package) [Anderson et al. 1999].

In the remainder of this paper, we will first introduce the internal representation for vectors and matrices on the graphics processor, and we will describe the syntax and the semantic of the vector and matrix routines our approach is built upon. Similar to the notation used in the BLAS library, we outline specific operations on vectors and matrices. Although we use a different syntax than BLAS, and we also do not provide the many different operators contained in the BLAS function definition, it should become obvious that by means of our approach the same functionality can be achieved.

We will also address sparse matrix representation and operations on such matrices as they typically occur in numerical simulation techniques. In this way, we achieve a significant speed-up compared to software approaches. Next, GPU implementations of two methods for solving sparse linear systems as the occur in many numerical simulation techniques are described: the Conjugate Gradient (CG) method and the Gauss-Seidel solver. Finally, we demonstrate the effectiveness of our approach by solving the 2D wave equation and the incompressible Navier-Stokes equations on graphics hardware, and by directly visualizing the results on the ATI 9800.

2 Matrix Representation on GPUs

In this section, we describe the internal representation of matrices on graphics hardware. The proposed representation enables the efficient computation of basic algebraic operations used in numerical simulation techniques. The general idea is to store matrices as texture maps and to exploit pixel shader programs to implement arithmetic operations. For the sake of simplicity only column matrices, i.e. vectors and square NxN matrices are discussed. General matrices, however, can be organized in exactly the same way.

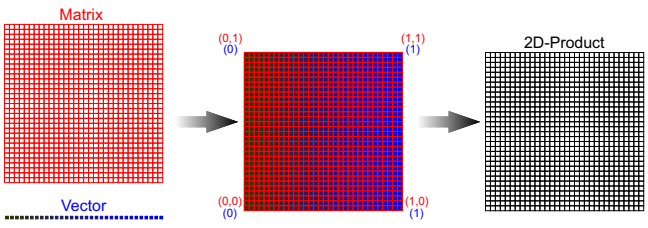

Figure 2: This illustration demonstrates the computation of a matrix-vector product using 1D and 2D textures to represent vectors and matrices, respectively. The 1D texture is continued periodically across the rendered quadrilateral.

On the graphics processor, vectors might be represented as 1D texture maps. This kind of representation, however, has certain drawbacks: First, limitations on the maximum size of 1D textures considerably reduce the number of vector elements that can be stored in one texture map. Second, rendering 1D textures is significantly slower than rendering square 2D textures with the same number of texture elements. On current graphics cards, the use of 2D textures yields a performance gain of about a factor of 2. Third, if a 1D vector contains the result of a numerical simulation on a computational grid, the data finally has to be rearranged in a 2D texture

for rendering purposes. Fourth, this representation prohibits the efficient computation of matrix-vector products. Although by means of multi-textures both the matrix and the vector can be rendered simultaneously and combined on a per-fragment basis (see Figure 2), the computed values are not in place and have to be summed along the rows of the matrix to obtain the result vector.

To circumvent the mentioned drawbacks, we represent matrices as set of diagonal vectors and we store vectors in 2D texture maps (see Figure 3). To every diagonal starting at the i-th entry of the first row in the upper right part of the matrix, its counterpart starting at the (N-i)-th entry of the first column in the lower left part of the matrix is appended. In this way, no texture memory is wasted. Each vector is converted into a square 2D texture by the application program. Vectors are padded with zero entries if they do not entirely fill the texture. This representation has several advantages:

- Much larger vectors can be stored in one single texture element.
- Arithmetic operations on vectors perform significantly faster because square 2D textures are rendered.
- Vectors that represent data on a 2D grid can directly be rendered to visualize the data.
- Matrix-vector multiplication can be mapped effectively to vector-vector multiplication.
- The result of a matrix-vector multiplication is already in place and does not need to be rearranged.

Most notable, however, the particular layout of matrices as set of diagonal vectors allows for the efficient representation and processing of banded diagonal matrices, as they typically occur in numerical simulation techniques. In a pre-process the application program inspects every diagonal, discarding those diagonals that do not carry any information. If no counterpart exists for one part of a diagonal, it is filled with zero entries.

A nice feature of this representation is, that the transpose of a matrix is generated by simply ordering the diagonals in a different way. Off-diagonals numbered i, which start at the i-th entry of the first row, now become off-diagonals numbered N-i. Entries located in the former upper right part of the matrix are swapped with those entries located in the lower left part. Swapping does not have to be performed explicitly, but it can be realized by shifting indices used to access these elements. Each index has to be shifted by the number of entries in the vector that come from the lower left part of the matrix. This can easily be accomplished in the pixel shader program, where the indexing during matrix-vector operations is performed (see below).

3 Basic Operations

Now, we describe the implementation of basic algebraic operations on vectors and matrices based on the proposed internal representation. In each operation, rendering is directed to a specific render target that can be directly bound to a 2D texture for further use. To update values in a target that is not the current target anymore, it is made the current render target again. Then, its content can be modified by consecutive rendering passes.

Vector and matrix containers are defined as classes *clVec* and *clMat*, respectively. Both containers assemble C++ arrays in that the array is decomposed into one or multiple vectors. Vectors composed of zero entries neither have to be stored nor processed.

Upon initialization, for each vector a texture is created and bound to a texture handle. Internally, each class element stores the resolution of the respective vector or matrix and of all the textures that are handled by that element. Texture handles and the size of each texture can be accessed via public functions.

3.1 Vector Arithmetic

Arithmetic operations on two vectors can be realized in a simple pixel shader program. The application issues both operands as multi-textures, which are accessed and combined in the shader program. On current graphics cards supporting the Pixel Shader 2.0 instruction set, arithmetic operations like addition, subtraction and multiplication are available. The product of a scalar times a vector is realized by issuing the scalar as a constant value in the shader program.

The function header for implementing standard arithmetic operations on vector elements looks like this:

```
void clVecOp (
 CL_enum op,
 float α, float β,
 clVec x, clVec y,
 clVec res
);
```

The enumerator *op* can be one of **CL_ADD, CL_MULT or CL_SUB**. The scalars α and β are multiplied with each element of *x* and *y*, respectively. At the beginning of each routine a consistency check is performed to verify that both vectors have the same internal representation. Then, the respective shader program is activated and vectors *x* and *y* are issued as multi-textures. Finally, a square quadrilateral is rendered, which is lined up in screen space with the 2D textures used to represent the active vectors. The result is kept as 2D texture to be used in consecutive computations.

3.2 Matrix-Vector Product

In the following, we consider the product of a matrix times a vector. A second vector might be specified to allow for the computation of *Ax op y*, where *A* is a matrix, *x* and *y* are vectors, and *op* is one of **CL_ADD, CL_MULT, CL_SUB**. To compute *Ax op y* we first render the result of *Ax* into the render target. Now, the result is bound as an additional texture, and in a final rendering pass it is combined with the vector *y* and rendered to the destination target.

The header of the function performing matrix-vector operations looks like this (if one of *A* or *x*, or *y* is NULL, only the respective component not equal to NULL considered in the operation):

```
void clMatVec (
 CL_enum op,
 clMat A,
 clVec x, clVec y,
 clVec res
);
```

Because matrices are represented as a set of diagonal vectors, matrix-vector multiplication becomes a multiplication of every diagonal with the vector. Therefore, N rendering passes are performed, in each of which one diagonal and the vector are issued as separate textures in the shader program. Then, corresponding entries in both textures are multiplied. However, to the j-th element of a diagonal that starts at the ith entry of the first row of the matrix corresponds the ((i+j) mod N)-th entry of the vector. This entry first has to be computed in the shader program before it can be used to access the vector.

Values i and N are simply issued as constant values in the shader program. Index j, however, is directly derived from the fragments

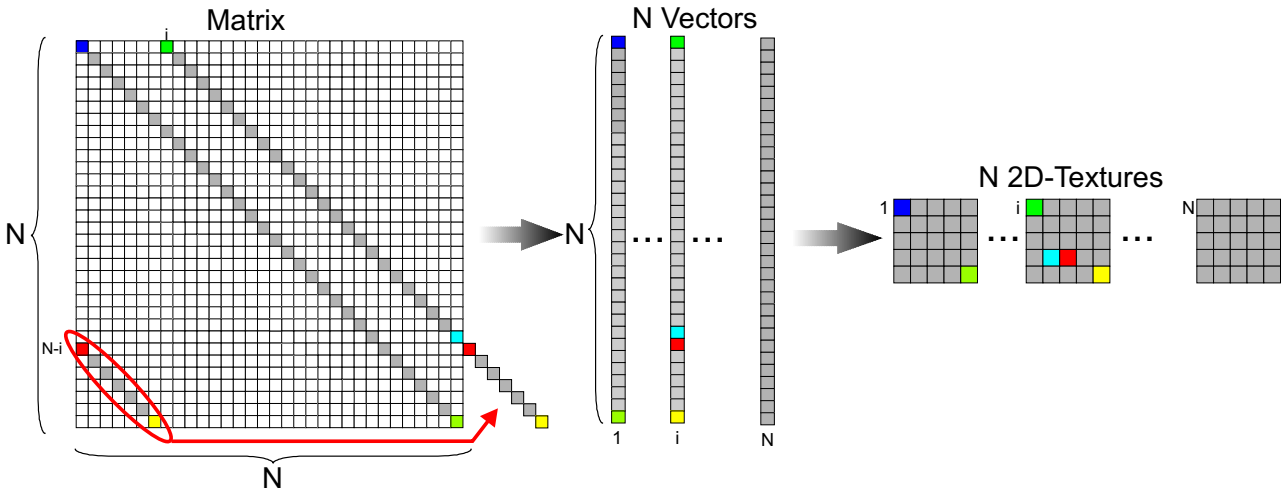

Figure 3: The representation of a 2D matrix as a set of diagonal vectors, and finally as a set of 2D textures is shown.

texture coordinates and N. Finally, the destination index ((i+j) mod N) is calculated and converted to 2D texture coordinates.

After the first diagonal has been processed as described, the current render target is simultaneously used as a texture and as a render target. Thus, fragments in consecutive passes have access to the intermediate result, and they can update this result in each iteration. After rendering the last diagonal, the result vector is already in place, and the current render target can be used to internally represent this vector.

A considerable speed-up is achieved by specifying multiple adjacent diagonals as multi-textures, and by processing these diagonals at once in every pass. Parameters i and N only have to be issued once in the shader program. A particular fragment has the same index j in all diagonals, and as a matter of fact it only has to be computed once. Starting with the first destination index, this index is successively incremented about one for consecutive diagonals. The number of diagonals that can be processed simultaneously depends on the number of available texture units.

Let us finally mention, that with regard to the described implementation of matrix-vector products, there is no particular need to organize matrices into sets of diagonal vectors. For instance, dense matrices might be represented as sets of column vectors, giving rise to even more efficient matrix-vector multiplication. Every column just has to be multiplied with the respective vector element, resulting in a much smaller memory footprint, yet requiring a simple shader program to which only the index of the currently processed column is input.

3.3 Vector Reduce

Quite often it is necessary to combine all entries of one vector into one single element, e.g. computing a vector norm, finding the maximum/minimum element etc. Therefore, we provide a special operation that outputs the result of such a reduce operation to the application program:

```
float clVecReduce (
 CL_enum cmb,
 clVec x, clVec y,
);
```

The enumerator *cmb* can be one of **CL_ADD, CL_MULT, CL_MAX, CL_MIN, CL_ABS**. If the second parameter **y** is not equal to NULL, the combiner operation is carried out on the product *x* times *y* rather than only on *x*.

The reduce operation combines the vector entries in multiple rendering passes by recursively combining the result of the previous pass. Starting with the initial 2D texture containing one vector and the quadrilateral lined up with the texture in screen space, in each step a quadrilateral scaled by a factor of 0.5 is rendered. In the shader program, each fragment that is covered by the shrunken quadrilateral combines the texel that is mapped to it and the three adjacent texel in positive (u,v) texture space direction. The distance between texels in texture space is issued as a constant in the shader program. The result is written into a new texture, which is now of a factor of two smaller in each dimension than the previous one. The entire process is illustrated in Figure 4. This technique is a standard approach to combine vector elements on parallel computer architectures, which in our scenario is used to keep the memory footprint for each fragment as low as possible. For a diagonal vector that is represented by a square texture with resolution 2^n, n rendering passes have to be performed until the result value *val* is obtained in one single pixel. The respective pixel value is finally returned to the application program.

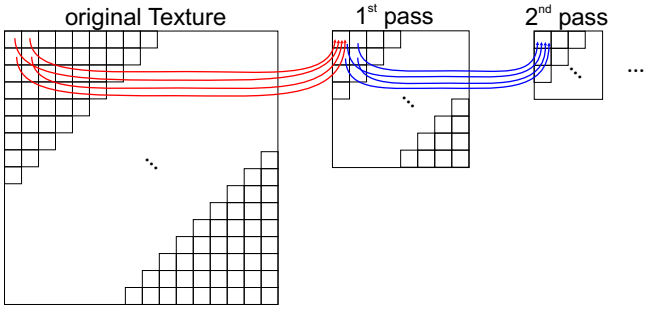

Figure 4: Schematic illustration of the reduce operation.

4 Sparse Matrices

So far, the operations we have encountered execute very efficiently on current commodity graphics hardware. On the other hand, they are not suitable to process matrices as they typically arise in numerical simulations. For instance, let us assume that the solution to the

2D wave equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

on a grid of resolution 512 x 512 has to be computed numerically (including boundary points). If first and higher order partial derivatives are approximated by finite differences, the partial difference equation writes as a set of finite difference equations for each grid point (ij):

$$\frac{u_{ij}^{t+1} - 2u_{ij}^t + u_{ij}^{t-1}}{\Delta t^2} = c^2 \left(\frac{u_{i+1j}^t + u_{i-1j}^t + u_{ij+1}^t + u_{ij-1}^t - 4u_{ij}^t}{\Delta x \Delta y} \right)$$

Using the implicit Crank-Nicholson scheme, where the average of the right-hand side is taken, i.e. for all grid points we set $u_{i,j}^t = 0.5(u_{i,j}^{t+1} + u_{i,j}^t)$, the difference equation contains more than one unknown and the system of algebraic equations has to be solved simultaneously.

If initial and boundary conditions are specified, the set of equations can be written in matrix from as $Ax = b$, where A is a 512^2 x 512^2 matrix, and both b and the solution vector x are of length 512^2. Here, x contains the unknowns u_{ij}^{t+1} to be solved for. In the particular example, A is a banded matrix (a triangular matrix with fringes) with a maximal bandwidth of six. Obviously, storing the matrix as a full matrix is quite inefficient both in terms of memory requirements and numerical operations. In order to effectively represent and process general sparse N x N matrices, in which only $O(N)$ entries are supposed to be active, an alternative representation on the GPU needs to be developed.

4.1 Banded Matrices

With regard to the internal representation of matrices as a set of diagonal vectors, we can effectively exploit the existence of a banded matrix with only a few non-zero off-diagonals. Zero off-diagonals are simply removed from the internal representation, and off-diagonals that do not have a counterpart on either side of the main diagonal are padded with zero entries.

In the above example, where the 2D wave equation has been discretized by means of finite differences, only six diagonals have to be stored internally. As a consequence, the product of this matrix times a vector costs no more than six vector-vector products.

In the general setting, however, were non-zero entries are positioned randomly in the matrix, the diagonal layout of vectors does not allow for the exploitation of the sparseness in a straight forward way.

4.2 Sparse Random Matrices

To overcome this problem, we use vertices to render the matrix values at the correct position. For each non-zero entry in a column vector one vertex is generated. The coordinate of each vertex is chosen in such a way, that it renders at exactly the same position as the respective vector element if it was rendered via the 2D texture used to represent the vector. For each column we thus store as many vertices as there are non-zero entries. Matrix values are stored as colors associated with the respective vertices.

Vertices and corresponding colors are stored in a vertex array on the GPU. As long as the matrix is not going to be modified, the internal representation does not have to be changed. Note that in case of a banded matrix, where apart from start and end conditions for each NxN block in the matrix the same band is present in every row, it is sufficient to store one representative set of vertices for inner grid points. Then, this set can be rendered using the appropriate offset with respect to the current column. Most effectively, the offset is specified in a vertex shader program, by means of which each vertex compute the exact 2D position in screen space.

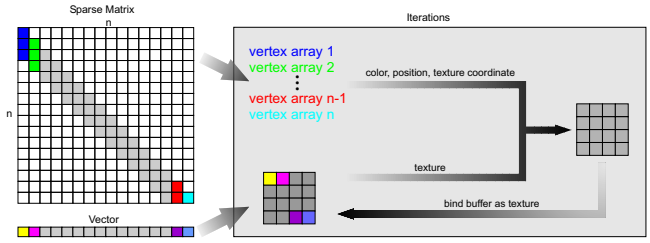

Figure 5: This image illustrates the computation of a sparse-matrix-vector product based on the internal representation of matrix columns as sets of vertices.

For the multiplication of a matrix times a vector, the color of each vertex has to be multiplied with the color of the corresponding entry in the vector. The vector, however, is not static and can thus not be coded into the vertex array. As a matter of fact, we associate with each vertex a texture coordinate, which is used to access the vector via the 2D textures used to represent it. Fortunately, these texture coordinates can also be stored on the GPU, so that only the appropriate textures have to be bound during matrix-vector computations (see Figure 5)

It is a nice feature of the described scheme, that the realization of matrix-vector operations on the GPU as it was proposed in Chapter 3 is not affected by the graphical primitives we use to internally represent and render matrices. The difference between sparse and full matrices just manifests in that we render every diagonal or column vector as a set of vertices instead of set of 2D textures. In this way, a significant amount of texture memory, rasterization operations and texture fetch operations can be saved in techniques where sparse matrices are involved. For instance, to compute the product between the sparse matrix described above and a vector only 512^2x6 textured vertices have to be rendered.

5 Examples

We will now exemplify the implementation of general techniques of numerical computing by means of the proposed basis operations for matrix-vector and vector-vector arithmetic.

5.1 Conjugate Gradient Method

The conjugate gradient (CG) method is an iterative matrix algorithm for solving large, sparse linear system of equations $Ax = b$, where $A \in R^{nxn}$. Such systems arise in many practical applications, such as computational fluid dynamics or mechanical engineering. The method proceeds by generating vector sequences of iterates (i.e. successive approximations to the solution), residuals r corresponding to the iterates, and search directions used in updating the iterates and residuals. The CG algorithm remedies the shortcomings of steepest descent by forcing the search directions $p^{(i)}$ to be A-conjugate, that is ${p^{(i)}}^T A p^{(i)} = 0$, and the residuals to be orthogonal. Particularly in numerical simulation techniques, where large but sparse finite difference equations have to be solved, the CG-algorithm is a powerful and widely used technique.

In the following, pseudo code for the unpreconditioned version of the CG algorithm is given. Lower and upper subscripts indicate the values of scalar and vector variables, respectively, in the specified iteration. For a good introduction to the CG method as well as

to other solution methods for linear system equations let us refer to [Press et al. 2002].

Unpreconditioned CG

1 $p^{(0)} = r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
2 **for** $i \leftarrow 0$ **to** *#itr*
3 $\rho_i = r^{(i)^T} r^{(i)}$
4 $q^{(i)} = Ap^{(i)}$
5 $\alpha_i = \rho_i / p^{(i)^T} q^{(i)}$
6 $x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$
7 $r^{(i+1)} = r^{(i)} - \alpha_i q^{(i)}$
8 $\beta_i = r^{(i+1)^T} r^{(i+1)} / \rho_i$
9 $p^{(i+1)} = r^{(i+1)} + \beta_i p^{(i)}$
10 convergence check

The CG method can effectively be stated in terms of the described building blocks for GPU implementation of numerical techniques (Note that using a preconditioner matrix, for instance the diagonal part of A stored in the first diagonal vector in our internal representation, only involves solving one more linear system in each iteration):

Unpreconditioned GPU-based CG

1 **clMatVec**$(CL\_SUB, A, x^{(0)}, b, r^{(0)})$ initial guess $x^{(0)}$
2 **clVecOp**$(CL\_ADD, -1, 0, r^{(0)}, NULL, r^{(0)})$
3 **clVecOp**$(CL\_ADD, 1, 0, r^{(0)}, NULL, p^{(0)})$
4 **for** $i \leftarrow 0$ **to** *#itr*
5 $\rho_i =$ **clVecReduce**$(CL\_ADD, r^{(i)}, r^{(i)})$
6 **clMatVec**$(CL\_ADD, A, p^{(i)}, NULL, q^{(i)})$
7 $\alpha_i =$ **clVecReduce**$(CL\_ADD, p^{(i)}, q^{(i)})$
8 $\alpha_i = \rho_i$ / α_i
9 **clVecOp**$(CL\_ADD, 1, \alpha_i, x^{(i)}, p^{(i)}, x^{(i+1)})$
10 **clVecOp**$(CL\_SUB, 1, \alpha_i, r^{(i)}, q^{(i)}, r^{(i+1)})$
11 $\beta_i =$ **clVecReduce**$(CL\_ADD, r^{(i+1)}, r^{(i+1)})$
12 $\beta_i = \beta_i$ / ρ_i
13 **clVecOp**$(CL\_ADD, 1, \beta_i, r^{(i+1)}, p^{(i)}, p^{(i+1)})$
14 convergence check

In the GPU implementation, the application program only needs to read single pixel values from the GPU thus minimizing bus transfer. All necessary numerical computations can be directly performed on the GPU. Moreover, the final result is already in place and can be rendered as a 2D texture map.

5.2 Gauss-Seidel Solver

Next, let us describe the GPU implementation of a Gauss-Seidel solver. Denoting with L and U the strict lower and upper triangular sub-matrices, and with D the main diagonal of the matrix A, we can rewrite A as $L+D+U$. In one iteration, the Gauss-Seidel method essentially solves for the following matrix-vector equation:

$$x^{(i)} = Lx^{(i)} + (D+U)x^{(i-1)}$$

where $x^{(k)}$ is the solution vector at the k-th iteration.

$r^{(i)} = (D+U)x^{(i-1)}$ can be derived from the previous time step by one matrix-vector product. To compute $Lx^{(i)}$, however, updates of $x^{(i)}$ have to be done in place. Based on the representation of matrices as set of column vectors, we sweep through the matrix in a column-wise order, using the result vector $x^{(i)}$ as the current render target as well as a currently bound texture. Initially, the content of $r(i)$ is copied into $x^{(i)}$. When the j-th column of L is rendered, each element is multiplied with the j-th element in $x^{(i)}$, and the result is added to $x^{(i)}$. We thus always multiply every column with the most recently updated value of $x^{(i)}$.

6 Discussion and Performance Evaluation

To verify the effectiveness of the proposed framework for the implementation of numerical simulation techniques we have implemented two meaningful examples on the graphics processor. All our experiments were run under WindowsXP on a P4 2.8 GHz processor equipped with an ATI 9800 graphics card.

With regard to the realization of methods of numerical computing on graphics hardware, limited accuracy in both the internal texture stages and the shader stages is certainly the Achilles´ heel of any approach. In many cases, numerical methods have to be performed in double precision to allow for accurate and stable computations. As a matter of fact, our current target architecture does not provide sufficient accuracy in general. Other graphics cards, on the other hand, like NVIDIAs GeForceFX, already provide full IEEE floating point precision in both the shader and texture stages. Thus, it will be of particular interest to evaluate this GPU in particular as well as other near-future graphics architectures with regard to computation accuracy.

Let us now investigate the performance of our approach as well as the differences to CPU implementations of some of the described basic operations. In our experiments the resolution of vectors and matrices was chosen such as to avoid paging between texture memory and main memory and between main memory and disk. All our operations were run on vectors and matrices of size 512^2 to 2048^2. We have also not considered the constant time to initially load textures from main memory to texture memory. The reason is, that we predominantly focus on iterative techniques, where a large number of iterations have to be performed until convergence. Supposedly, in these particular applications the time required to setup the hardware is insignificant compared to the time required to perform the computations. During all iterations the data resides on the GPU and it has neither to be reloaded from main memory nor duplicated on the CPU. In other scenarios, e.g. if frequent updates of a matrix happen, this assumption may not be justifiable anymore. In this case, also the time needed to transfer data between different units has to be considered.

On vectors and full matrices the implementation of standard arithmetic operations, i.e. vector-vector arithmetic and matrix-vector multiplication, was about 12-15 times faster compared to an optimized software implementation on the same target architecture. A considerable speed-up was achieved by internally storing vectors and matrices as RGBA textures. Sets of 4 consecutive entries from the same vector are stored in one RGBA texel. Thus, up to four times as many entries can be processed simultaneously. We should note here, that operations on vectors and matrices built upon this particular internal format perform in exactly the same way as outlined. Just at the very end of the computation need the vector elements stored in separate color components to be rearranged for rendering purposes. We can easily realize this task by means of a simple shader program that for each pixel in the result image fetches the respective color component.

On average, the multiplication of two vectors of length 512^2 took 0.2 ms. Performance dropped down to 0.72 ms and 2.8 ms for vectors of length 1024^2 and 2048^2, respectively. Multiplication of

a 4096^2 full matrix times a vector was carried out in roughly 0.23 seconds. In contrast, the multiplication of a sparse banded matrix of the same size, which was composed of 10 non-zero diagonals, took 0.72 ms.

Obviously, only one vector element can be stored in a single RGBA texel if numerical operations on vector-valued data have to be performed. On the other hand, in this case also the performance of software implementations drops down due to enlarged memory footprints. Our current software implementation is highly optimized with regard to the exploitation of cache coherence on the CPU. In practical applications, a less effective internal representation might be used, so that we rather expect a relative improvement of the GPU based solution. In this respect it is important to know that also on the GPU access to higher precision textures slows down performance about a factor of 1.5-2.

The least efficient operation compared to its software counterpart is the reduce operation. It is only about a factor of three faster even though we store four elements in one RGBA texture. For instance, reducing a 1024^2 vector takes about 1.6 ms on the GPU. For a vector of length 2048^2, this time is 5.4 ms. The relative loss in performance is due to the fact, that the pixel shader program to be used for this kind of operation is a lot more complex than one that is used for vector-vector multiplication. On the other hand, even a performance gain of a factor of three seems to be worth an implementation on the GPU.

In the following, we present two examples that demonstrate the efficient solution of finite difference equations on the GPU. In both examples, a 1024 x 1024 computational grid was employed, and matrices were represented as set of diagonal vectors.

In the first example, a solution to the 2D wave equation was computed based on the implicit Crank-Nicholson scheme as described (see Figures 6 and 7 for results). Compared to explicit schemes, the implicit approach allows us to considerable increase the step size in time. To solve the system of equations we employed the GPU implementation of the conjugate gradient solver. The banded structure of the matrix was exploited by reducing the number of diagonal vectors to be rendered in one matrix-vector multiplication. The computation of one matrix-vector product ($1024^2 x 1024^2$-sparse-matrix times 1024^2-vector) took roughly 4.54 ms. Overall, one iteration of the conjugate gradient solver was finished in 15.4 ms. By performing only a limited number of iterations, five in the current example, interactive simulation at 13 fps could be achieved.

In our second example, we describe a GPU implementation of a numerical solution to the incompressible Navier-Stokes equations (NSE) in 2D:

$$\frac{\partial u}{\partial t} = \frac{1}{Re}\nabla^2 u - V\cdot\nabla u + f_x - \nabla p \quad (1)$$

$$\frac{\partial v}{\partial t} = \frac{1}{Re}\nabla^2 v - V\cdot\nabla v + f_y - \nabla p \quad (2)$$

Here, u and v correspond to the components of the velocity V in the x and y direction, respectively. Re is the Reynolds number, p the pressure, and via f_x and f_y external forces can be specified.

We first discretize partial derivative of u and v, resulting in an explicit scheme to compute new velocities at time $t+1$ from values at time t:

$$u^{t+1} = G^t + \triangle t \frac{\partial p^{t+1}}{\partial x} \quad (3)$$

$$v^{t+1} = F^t + \triangle t \frac{\partial p^{t+1}}{\partial y} \quad (4)$$

with

$$G^t = u^t + \triangle t\left(\frac{1}{Re}\nabla^2 u - V\cdot\nabla u + f_x\right) \quad (5)$$

$$F^t = v^t + \triangle t\left(\frac{1}{Re}\nabla^2 v - V\cdot\nabla v + f_y\right) \quad (6)$$

Given current values for u and v at every grid points, G^t and F^t can be directly evaluated. In the current implementation, we employ an explicit scheme to resolve for G^t and F^t. The diffusion operator is discretized by means of central differences, and, as proposed in [Stam 1999], we solve for the advection part by tracing the velocity field backward in time. Note that these operations are carried out on a 2D grid represented by a 2D texture. The involved computations are performed at each grid point in a pixel shader program. Finally, in order to compute updated velocities at time $t+1$, we have to solve for the pressure at time $t+1$. From the continuity equation for incompressible media ($div(V) = 0$), we obtain the following Poisson equation for the pressure p:

$$\frac{\partial^2 p^{t+1}}{\partial x^2} + \frac{\partial^2 p^{t+1}}{\partial y^2} = \frac{1}{\triangle t}\left(\frac{\partial F^t}{\partial x} + \frac{\partial G^t}{\partial y}\right) \quad (7)$$

The partial derivatives of F and G are solved at each grid point, represented as a 2D texture, by means of forward differences. Finally, the right hand side of equation 7 is input to the GPU implementation of the CG solver. Because vectors are internally represented as 2D matrices, the data does not have to be converted and can be directly used to feed the CG solver. Equipped with appropriate boundary conditions, the CG solver iteratively computes a solution for p at time $t+1$, which can be directly passed back to the explicit scheme to compute new velocity values by means of equations 3 and 4.

Overall, by means of the GPU implementation of both the explicit and the implicit scheme we were able to interactively demonstrate the numerical solution of the NSE at 9 fps on a 1024^2 grid. In each time step, we use the pressure distribution from the last time step as initial guess for the CG solver. In this way, only a few iterations have to be performed, yet resulting in good accuracy. In the current implementation, four iterations were executed. Such a small number of iterations, on the other hand, yields inaccurate results once abrupt changes are applied by means of external forces. In Figure 8, we show a snapshot of our interactive tool, which allows one to interact with the velocity field, and to visualize the dynamics of injected dye into this field.

7 Conclusion

In this work, we have described a general framework for the implementation of numerical simulation techniques on graphics hardware. For this purpose, we have developed efficient internal layouts for vectors and matrices. By considering matrices as a set of diagonal or column vectors and by representing vectors as 2D texture maps, matrix-vector and vector-vector operations could be accelerated considerably compared to software based approaches.

Our emphasis was on providing the building blocks for the design of general techniques of numerical computing. This is in contrast to existing approaches, where dedicated, mainly explicit solution methods have been proposed. In this respect, for the simulation of particular phenomena some of these approaches might be superior to ours in terms of performance. On the other hand, our framework offers the flexibility to implement arbitrary explicit or implicit schemes, and it can thus be used in applications where larger step sizes and stability are of particular interest. Furthermore, because our internal matrix layout can benefit from the sparsity of columns quite effectively, we do not expect our method to be significantly slower compared to customized explicit schemes.

In order to demonstrate the effectiveness and the efficiency of our approach, we have described a GPU implementation of the conjugate gradient method to numerically solve the 2D wave equation

and the incompressible Navier-Stokes equations. In both examples, implicit schemes were employed to allow for stable computations, yet providing interactive rates. Despite precision issues, we could achieve considerably better performance compared to our software realization. On the other hand, to allow for a fair comparison we should consider timing statistics of SSE-optimized software solutions, which are supposed to perform about a factor of 2 to 3 faster.

The lack of a contiguous floating point pipeline on our target architecture still prohibits its use in numerical applications where accuracy is a predominant goal. On the other hand, with regard to the fact that full floating point pipelines are already available, the implementation of numerical techniques on commodity graphics hardware is worth an elaborate investigation. Particularly in entertainment and virtual scenarios, where precision issues might be of lesser dominant concern, such implementations can be used effectively for interactive physics based simulation.

In the future, we will implement matrix-matix operations based on the described internal layout, and we will investigate methods to efficiently update vector and matrices that are stored in texture memory. In this way, linear algebra operations like LU-decomposition or Singular Value decomposition can be implemented. In the long term, we aim at providing the functionality that is available in the BLAS library, thus allowing general linear algebra packages to be built upon GPU implementations.

8 Acknowledgements

We would like to thank ATI for providing the 9800 graphics card, and in particular Mark Segal for providing information about the technical details of this card.

Figure 6: GPU-based interactive simulation of 2D water surfaces is demonstrated. The implementation runs at 43 fps on a 512^2 grid.

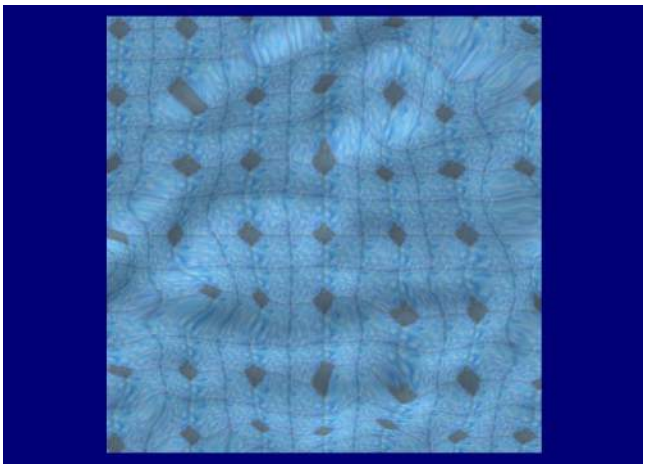

Figure 7: A GPU-based tool to interact with water surfaces in real-time is shown. By means of the mouse, the user can simulate external forces that disturb the water surface. On a 1024^2 grid the applications runs at 13 fps.

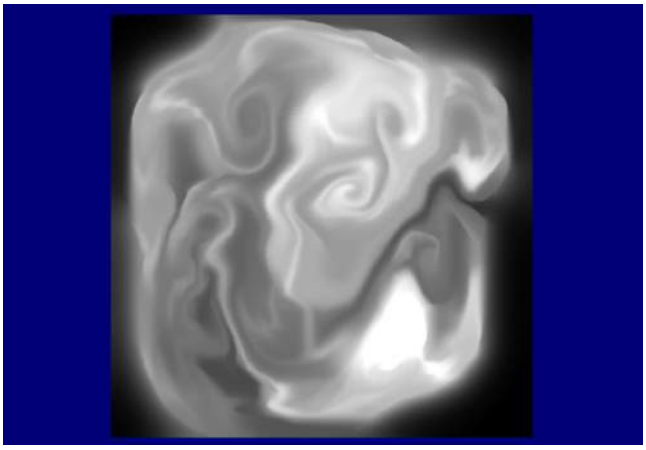

Figure 8: An interactive tool for the visualization of the solution to the 2D Navier-Stokes equations is demonstrated. The user can modify the velocity field, and dye can be injected into the field. On a 1024^2 grid the applications runs at 9 fps.

References

ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., AND SORENSEN, D. 1999. *LAPACK Users' Guide*, third ed. Society for Industrial and Applied Mathematics, Philadelphia, PA.

ATI, 2003. Sample effects on the ATI graphics cards. http://www.ati.com/developer/techpapers.html.

BARAFF, D., AND WITKIN, A. 1998. Large steps in cloth simulation. *Computer Graphics SIGGRAPH 98 Proceedings*, 43–54.

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHROEDER, P. 2003. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *Computer Graphics SIGGRAPH 03 Proceedings*.

CHEN, J., AND DA VITORIA LOBO, N. 1995. Towards interactive-rate simulation of fluids with moving obstacles using Navier-Stokes equations. *Graphical Models and Image Processing 57*, 2.

CURTIS, C., ANDERSON, S., SEIMS, J., FLEISCHER, F., AND SALESIN, D. 1997. Computer-generated watercolor. *Computer Graphics SIGGRAPH 97 Proceedings*.

DEBUNNE, G., DESBRUN, M., M.-P., C., AND BARR, A. 2001. Dynamic real-time deformations using space and time adaptive sampling. In *Computer Graphics SIGGRAPH 01 Proceedings*.

DESBRUN, M., MEYER, M., SCHROEDER, P., AND BARR, A. 1999. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Computer Graphics SIGGRAPH 99 Proceedings*, 317–324.

DONGARRA, J., DU CROZ, J., HAMMARLING, S., AND HANSON, R. 1988. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software 14*, 1–17.

DONGARRA, J., DU CROZ, J., HAMMARLING, S., AND HANSON, R. 1990. A set of level 3 basic linear algebra subprograms,. *ACM Transactions on Mathematical Software 16*, 1–17.

ELDER, G. 2002. Radeon 9700. In *Proceedings Eurographics/SIGGRAPH Workshop on Graphics Hardware 2002*.

FEDKIW, R., STAM, J., AND JENSEN, H. 2001. Visual simulation of smoke. *Computer Graphics SIGGRAPH 01 Proceedings*, 15–22.

FOSTER, N., AND FEDKIW, R. 2001. Practical animation of liquids. *Computer Graphics SIGGRAPH 01 Proceedings*, 23–30.

FOSTER, N., AND METAXAS, D. 1996. Realistic animation of liquids. *Graphical Models and Image Processing 58*, 5, 471–483.

HARRIS, M., COOMBE, G., SCHEUERMANN, T., AND LASTRA, A. 2002. Physically-based visual simulation on graphics hardware. In *Proceedings Eurographics/SIGGRAPH Workshop on Graphics Hardware 2002*.

HART, J. 2001. Perlin noise pixel shaders. In *Proceedings Eurographics/SIGGRAPH Workshop on Graphics Hardware 2001*.

HEIDRICH, W., WESTERMANN, R., SEIDEL, H.-P., AND ERTL, T. 1999. Applications of pixel textures in visualization and realistic image synthesis. In *ACM Symposium on Interactive 3D Graphics*, 110–119.

HILLESLAND, K., MOLINOV, S., AND GRZESZCZUK, R. 2003. Nonlinear Optimization Framework for Image-Based Modelling on Programmable Graphics Hardware. *Computer Graphics SIGGRAPH 03 Proceedings*.

HOPF, M., AND ERTL, T. 1999. Accelerating 3D convolution using graphics hardware. In *Proceedings IEEE Visualization'99*, 471–474.

HOPF, M., AND ERTL, T. 2000. Hardware accelerated wavelet transformations. In *Proceedings EG/IEEE TCVG Symposium on Visualization VisSym '00*, 93–103.

JOBARD, B., ERLEBACHER, G., AND HUSSAINI, Y. 2000. Lagrangian-Eulerian advection of noise and dye textures for unsteady flow visualization. In *Proceedings IEEE Visualization'00*, 110–118.

KAAS, M., AND MILLER, G. 1990. Rapid, stable fluid dynamics for computer graphics. *Computer Graphics SIGGRAPH 90 Proceedings*, 49–57.

LARSEN, E. S., AND MCALLISTER, D. 2001. Fast matrix multiplies using graphics hardware. In *Proceedings Supercomputing 2001*.

LINDHOLM, E., KILGARD, M., AND MORETON, H. 2001. A user-programmable vertex engine. *Computer Graphics SIGGRAPH 01 Proceedings*.

MICROSOFT, 2002. DirectX9 SDK. http://www.microsoft.com/DirectX.

MONTRYM, J., AND MORETON, H. 2002. GeForce4. In *Proceedings Eurographics/SIGGRAPH Workshop on Graphics Hardware 2002*.

NVIDIA, 2002. nvidia OpenGL game of life. http://www.nvidia.com/view.asp?IO=ogl_gameoflife.

NVIDIA, 2003. Sample effects on the nVIDIA graphics cards. http://developer.nvidia.com/view.asp?PAGE=papers.

OLANO, M., AND LASTRA, A. 1998. A shading-language on graphics hardware. *Computer Graphics SIGGRAPH 98 Proceedings*, 159–168.

PRESS, W., TEUKOLSKY, S., VETTERLING, W., AND FLANNERY, B. 2002. *Numerical Recipes in C++ : The Art of Scientific Computing*. Cambridge University Press.

PURCELL, T., BUCK, I., MARK, W., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. *Computer Graphics SIGGRAPH 98 Proceedings*, 703–712.

STAM, J. 1999. Stable fluids. *Computer Graphics SIGGRAPH 99 Proceedings*, 121–128.

STRZODKA, R., AND RUMPF, M. 2001. Nonlinear diffusion in graphics hardware. In *Proceedings EG/IEEE TCVG Symposium on Visualization 2001*, 75–84.

STRZODKA, R., AND RUMPF, M. 2001. Using graphics cards for quantized FEM computations. In *Proceedings VIIP 2001*, 98–107.

THOMPSON, C., HAHN, S., AND OSKIN, M. 2002. Using modern graphics architectures for general-purpose computing: A framework and analysis. *Proceedings of 35th International Symposium on Microarchitecture (MICRO-35)*.

WEISKOPF, D., HOPF, M., AND ERTL, T. 2001. Hardware-accelerated visualization of time-varying 2D and 3D vector fields by texture advection via programmable per-pixel operations. In *Proceedings Workshop on Vision, Modeling, and Visualization VMV'01*, 439–446.

WEISKOPF, D., HOPF, M., AND ERTL, T. 2002. Hardware-accelerated Lagrangian-Eulerian texture advection for 2D flow visualization. In *Proceedings Workshop on Vision, Modeling, and Visualization VMV '02*.