Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid

Jeff Bolz Ian Farmer Eitan Grinspun Peter Schröder
Caltech

Abstract

Many computer graphics applications require high-intensity numerical simulation. We show that such computations can be performed efficiently on the GPU, which we regard as a full function *streaming* processor with high floating-point performance. We implemented two basic, broadly useful, computational kernels: a *sparse matrix conjugate gradient solver* and a regular-grid *multigrid solver*. Real-time applications ranging from mesh smoothing and parameterization to fluid solvers and solid mechanics can greatly benefit from these, evidence our example applications of geometric flow and fluid simulation running on NVIDIA's GeForce FX.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors; G.1.3 [Numerical Analysis]: Numerical Linear Algebra—Sparse, structured, and very large systems (direct and iterative methods); G.1.8 [Numerical Analysis]: Partial Differential Equations—Multigrid and Multilevel Methods;

Keywords: GPU Computing, Numerical Simulation, Conjugate Gradient, Multigrid, Mesh Smoothing, Fluid Simulation, Navier-Stokes

1 Introduction

High performance graphics processing units (GPUs) such as the Radeon 9700 and GeForce FX (among others) expose a flexible programming interface for their powerful floating point hardware. Due to their highly parallel nature, they are expected to outperform CPUs by an increasing margin [Semiconductor Industry Association 2002; Khailany et al. 2003], consequently they are serious contenders as high performance computational engines for floating point intensive applications. Aside from their currently utility, GPUs represent the first commercially successful examples of a class of future computing architectures key to high performance, cost effective super-computing [Khailany et al. 2001; Owens et al. 2002]. Understanding the issues in mapping a variety of fundamental algorithms to this new computing paradigm thus promises to have many longterm pay-offs.

So far the computational resources of GPUs have been applied mostly to traditional graphics problems, enhancing, for example, the shading models and effects applied at the pixel level [Olano 2002]. Current generation GPUs are capable of far more general computation, evidence the GPU-based ray tracing engines demonstrated by Purcell *et al.* [2002] and Carr *et al.* [2002].

Employing graphics hardware for purposes it was not designed for has a long tradition [Lengyel et al. 1990; Hoff et al. 1999], including early uses for numerical computing in the context of radiosity [Cohen et al. 1988; Keller 1997]. Many of these algorithms were based on rendering suitably chosen geometry with appropriate "colors" (*e.g.*, object ID tags), followed by readback from the framebuffer. With the arrival of programmable vertex and fragment units [Lindholm et al. 2001] the available repertoire increased significantly. Since vertex programs so far do not provide access to memory ("textures") most recent efforts to map numerical algorithms onto GPUs have focused on using integer based fragment processing hardware. Examples include simulation of boiling [Harris et al. 2002], fluids and steam [Li et al. 2003], non-linear diffusion [Strzodka and Rumpf 2001], and general purpose dense matrix multiplication [Larsen and McAllister 2001; Thompson et al. 2002]. Many of the severe technical limitations these authors had to deal with, such as low precision [Strzodka 2002; Harris 2002] and the need to express all operations as (fancy) texture compositing operations, have fallen away: today's fragment shaders support full floating-point arithmetic and broad access to memory.

Contributions We map two fundamental computational kernels onto the GPU: a *conjugate gradient* solver [Shewchuck 1994] for sparse, unstructured matrices and a *multigrid* solver for regular grids [Briggs et al. 2000]. Both are workhorses of physical modeling and optimization applications. We analyze their performance on NVIDIA's GeForce FX in realistic applications.

Background The need for an *unstructured sparse matrix* solver arises in many simulations involving discretization of linear and non-linear partial differential equations (PDEs) over arbitrary meshes (*e.g.*, [Müller et al. 2002; Desbrun et al. 2002; Kobbelt et al. 1998]). In these settings a *conjugate gradient* solver (or one of its variants [Barrett et al. 1994]) is often appropriate. Implementing such solvers on the GPU requires the construction of (1) *data structures* for sparse matrices, (2) *data parallel* algorithms for sparse matrix vector multiplies, and (3) *reduction operators* for inner product computations. With data laid out in texture memory, and fragment program execution having, for a given group of fragments, identical control flow, an efficient implementation furthermore requires the solution of a *texture packing* optimization problem (Section 3.2). Whenever the underlying PDE is non-linear it is desirable to compute the *matrix entries* on the GPU as well. Our algorithm accommodates this, making it well suited for both *linear* and *non-linear* PDEs on unstructured meshes.

Unstructured meshes require a non-trivial mapping of sparse data structures onto single-instruction multiple-data (SIMD) hardware [Blelloch 1990]. In contrast regular grids, used in graphics applications such as [Kass and Miller 1990; Stam 1999], lead to highly structured sparse matrices which map more directly onto the GPU since they are akin to pixel (2D) or voxel (3D) structures. For these settings we consider *multigrid* solvers [Hackbusch 1985], which lead to optimal, $O(n)$, runtime for many elliptic PDEs of interest (*e.g.*, Laplace, Bi-Laplace, Helmholtz, Poisson, *etc.*). In graphics such solvers are used, for example, for the construction of subdivision surfaces [Diewald et al. 2002]. The main challenge in performing this computation entirely on the GPU is the *construction* of the coarser-level system matrices given only a knowledge of the finest-level discretization and the *prolongation* (subdivision) operators. We evaluate the performance of our multigrid implementation on a fluid-flow problem [Stam 1999], which requires (among other steps) a Poisson solver with Neumann boundary conditions.

While this paper was under review we learned of concurrent work to make a multigrid solver onto the GPU [Goodnight et al. 2003]. The mapping of other important computational kernels onto the GPU is described in these proceedings [Krüger and Westermann 2003; Hillesland et al. 2003].

Permission to make digital/hard copy of part of all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2003 ACM 0730-0301/03/0700-0917 $5.00

2 Setup

We view the fragment shader of a modern GPU as a *stream processor* [Khailany et al. 2001]. The processor executes the same kernel (*fragment program*) to produce each element (*rasterized pixel*) of an output stream (*group of rasterized primitives*). The output stream is saved (*texture memory*) and used as input (via *texture fetches*) for downstream kernels. Our design maps the data structures and algorithms of our solvers into streams (*textures*) and kernels (*shaders*) respectively as proposed by [Purcell et al. 2002].

The design was strongly motivated by three salient features of GeForce FX hardware: (1) inexpensive gather operations, (2) lack of a scatter operation, and (3) SIMD semantics. Together, these three features characterize the *abstract streaming model*:

- **Gather Operation:** The processor provides random access memory fetch ("gather") instructions into saved streams. The latency of the random memory access is hidden so long as the bandwidth is not limited.
- **No Scatter Operation:** Each run through the kernel produces exactly one output element. There are no random access memory write operations.
- **SIMD Semantics:** The GPU has two SIMD characteristics: (1) the *same* kernel is executed over *all* elements of a stream, and (2) processor instructions operate on wide data types, *i.e.*, 4-tuples of floating point values.

Unconstrained gathers bring freedom in choosing data structures. Lack of scatter forces the stream kernels to be organized into groups. Each fragment processor can read other memory, but only write its own output: this fits well with many linear algebra operations (such as the inner product of two vectors) but not all, *e.g.*, applying the adjoint of an operator is most commonly formulated as a scatter operation. SIMD item (1) leads to a packing problem for unstructured matrices in order to minimize a *batch quantization* penalty (Section 3.2.1). SIMD item (2) motivated us to layer the four quadrants of a scalar problem into a single four channel texture for the multigrid solver.

3 Solver for Unstructured Grids

Many algorithms for physical simulation or optimization on unstructured meshes use (multi-)linear finite elements or mass-spring systems. The degrees of freedom (DOFs), such as position, velocity, temperature, *etc.*, are associated to vertices of a 2D (triangle, quadrilateral) or 3D (tetrahedral, hexahedral) mesh. The sparse linear systems that arise relate a given DOF to the DOFs on incident vertices. Our solver is designed for these sparse linear systems. For concreteness we assume a triangle mesh with linear finite elements.

Let the integers $i = 1, \ldots, n$ denote the vertices V of a 2-manifold triangle mesh (with boundary), with edges $E = \{e_{ij}\}$. Denote the set of vertices in the 1-ring neighborhood of vertex i as $N(i) = \{j | e_{ij} \in E\}$ and a (generic) DOF associated with vertex i as x_i. Example DOFs include 3D position $x_i := (x, y, z)_i$ or texture coordinates $x_i := (s, t)_i$. Our goal is to solve the $n \times n$ system, $\mathbf{A}\mathbf{x} = \mathbf{f}$,

$$\forall i = 1, \ldots, n: \quad a_{ii}x_i + \textstyle\sum_{j \in N(i)} a_{ij}x_j = f_i \qquad (1)$$

where the f_i encode the right hand side and possibly boundary conditions, depending on the original problem setup. The coefficients a_{ij} typically depend on the state of the two triangles incident on edge e_{ij}. We exploit this later for the construction of matrix entries on the GPU. If the linear system is symmetric positive definite we may use conjugate gradients to solve it. More general systems require variants of conjugate gradients [Barrett et al. 1994], which typically need explicit access to $\mathbf{A}^T$, but no additional functions otherwise. We assume that the rows of system (1) are sorted according to the number of non-zero off-diagonal coefficients a_{ij}. These can range from 0 to $n-1$, but are six on average for a single closed triangle mesh. Let i_k, $0 \le k < n$ denote the number of equations with k non-zero entries. These then occupy i_k contiguous rows starting with row index $(\sum_{l=1}^{k-1} i_l) + 1$.

Example System: Geometric Flow Many computer graphics settings give rise to linear systems with the structure of Eq. (1). We use geometric (mean curvature) flow as an example *non-linear* problem [Desbrun et al. 1999]. Vertex positions $x_i(t)$ are functions of time according to $\partial_t x_i(t) = -\lambda_i(t) H_i(t) \vec{n}_i(t)$. $H_i(t)\vec{n}_i(t)$ denotes the mean curvature normal at vertex i and time t while $\lambda_i(t)$ is the speed function. For simplicity we assume that λ_i does not depend on time. This equation is typically linearized through a semi-implicit discretization using a backward difference in time $\Delta t = t^{(k+1)} - t^{(k)}$ for the left hand side and the state of the mesh at the beginning of a time step for the approximation of $H_i(t)\vec{n}_i(t)$. Note that this requires recomputation of $\mathbf{A}^{(k)}$ at every time step. We have $a_{ij}^{(k)} = -\lambda \Delta t(\cot(\alpha_{ij}^{(k)}) + \cot(\beta_{ij}^{(k)}))$, where $\alpha_{ij}^{(k)}$ and $\beta_{ij}^{(k)}$ are the angles opposite edge e_{ij} at time $t^{(k)}$. The diagonal entry $a_{ii}^{(k)}$ is given as $4\mathsf{A}_i^{(k)} - \sum_{j \in N(i)} a_{ij}^{(k)}$ with $\mathsf{A}_i^{(k)}$ the area of the triangles at time $t^{(k)}$ incident to vertex i.

3.1 Conjugate Gradient Solver

Implementation of a conjugate gradient solver requires only a few non-trivial functions [Shewchuck 1994, p. 50]: sparse matrix-vector multiply and vector inner-product. The sparse matrix-vector multiply requires a suitable sparse matrix data structure and an associated fragment program to execute the multiply. The inner product computation requires a *sum-reduction*.

3.1.1 Sparse Matrix Vector Multiply

Principle To understand the particular layout for the unknown variables consider the implementation of the sparse matrix vector multiply. The basic computational kernel to be executed by a fragment program is the inner product between a given row and the vector of unknowns. Fragment programs must execute in SIMD style lockstep (there is no branching or early termination, as yet). Thus we "render" groups of *rows with an equal number of non-zero entries, i.e.*, each group is a rectangle associated to a fragment program specialized for a particular number of non-zero entries.

The fragment program needs to access the non-zero entries in a given row and the associated elements of the vector of unknowns. Each of these—the matrix $\mathbf{A}$ and vector $\mathbf{x}$—are stored in textures requiring appropriate indirections. Since $\mathbf{x}$ is read *and* written it must be laid out with particular care. The texture $\mathcal{X}^x$ holds $\mathbf{x}$, one element per pixel; here the superscript denotes the layout of the texture, implying that textures with the same superscript (*e.g.*, $\mathcal{X}^x, \mathcal{Y}^x, \mathcal{R}^x$) have a natural correspondence between their elements at the same location. The layout of $\mathcal{X}^x$ is discussed in Section 3.2.

Details The sparse matrix $\mathbf{A}$ is stored in two textures. The *diagonal* entries $\mathcal{A}_i^x$ are laid out the same as $\mathcal{X}^x$, *i.e.*, a_{ii} is at the same coordinate as x_i. The *off-diagonal, non-zero* entries of $\mathbf{A}$ are stored consecutively in a texture $\mathcal{A}_j^a$ as segments, with one segment per matrix row (a layout familiar from SIMD programming [Blelloch 1990]; see also Figure 1). Each segment's starting (texture-)address is stored in an *indirection* texture $\mathcal{R}^x$. Finally, we have a texture $\mathcal{C}^a$ keeping the correspondence between the addresses of a_{ij} in $\mathcal{A}_j^a$, matching x_j in $\mathcal{X}^x$. The layouts of $\mathcal{A}_j^a$ and $\mathcal{C}^a$ are identical; $\mathcal{A}_j^a$ has values a_{ij}, and $\mathcal{C}^a$ has the corresponding pointers into x_j in $\mathcal{X}^x$. Letting i now be the coordinates of a given element in $\mathcal{X}^x$, the

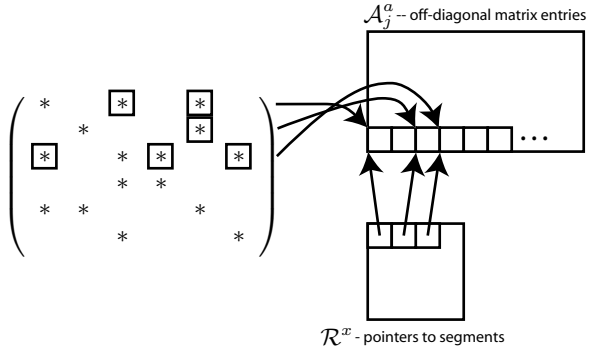

Figure 1: *Off-diagonal elements of each row are compacted into* segments *which are tightly packed into $\mathcal{A}_j^a$. A pointer to the beginning of each segment is stored in $\mathcal{R}^x$.*

inner product kernel between a sparse matrix row and the vector of unknowns becomes

$$\begin{aligned} j &= \mathcal{R}^x[i] \\ \mathcal{Y}^x[i] &= \mathcal{A}_i^x[i] * \mathcal{X}^x[i] + \sum_{c=0}^{k_i-1} \mathcal{A}_j^a[j+c] * \mathcal{X}^x[\mathcal{C}^a[j+c]], \end{aligned}$$

here $\mathcal{Y}^x$ denotes a destination texture with the same layout as $\mathcal{X}^x$. By rendering appropriate rectangles into $\mathcal{Y}^x$—each bound to a fragment program with the appropriate upper bound on the above sum—we can perform the desired sparse matrix vector product $\mathbf{y} = \mathbf{A}\mathbf{x}$ (Figure 2).

Notes The indirection textures $\mathcal{R}^x$ and $\mathcal{C}^a$ depend only on the mesh connectivity and can be initialized at the time a mesh is first constructed. The separate storage for $\mathcal{A}_i^x$ and $\mathcal{A}_j^a$ is advantageous for diagonal preconditioning—division of the residual vector by the diagonal entries—as well as the generally different methods by which diagonal and off-diagonal entries are computed in the first place. With the setup we have given for sparse matrix vector product, transpose products require an explicit representation of $\mathbf{A}^T$. In the case that $\mathbf{A}$ and $\mathbf{A}^T$ have differing numbers of non-zero entries per row, the lesser should be padded with zeros so that they have the same size. The number of texture indirections that can be performed in a fragment program limits the number of non-zero entries per row that can be processed without additional passes. For the GeForce FX rows with up to 200 non-zero entries can be processed in a single pass.

3.1.2 Computing Matrix Entries

Principle In the case that the entries of $\mathbf{A}$ depend on $\mathbf{x}$ we require two additional kernels. One to update $\mathcal{A}_i^x$ and another for $\mathcal{A}_j^a$. In traditional FEM codes this is typically done by an iteration over all elements computing local stiffness matrices, *i.e.*, the linear operator relating all DOFs incident on the element. These local stiffness matrices are then accumulated into a global stiffness matrix. Unfortunately this requires a scatter operation which is not available on current generation GPUs. Instead we must compute the non-zero entries directly. We have two types of non-zero entries, those associated with vertices $\mathcal{A}_i^x$ and edges $\mathcal{A}_j^a$. We begin with the latter.

Details The coefficient associated with a given edge is controlled by the two incident triangles, which in turn are completely described by their incident vertices—a total of four. Consider for example the coefficients which arise in the geometric flow problem. Aside from $\mathcal{C}^a$ this requires three additional textures: $\mathcal{I}^a$, $\mathcal{N}^a$ (next), and $\mathcal{P}^a$ (previous), layout out like $\mathcal{C}^a$. Together, these four

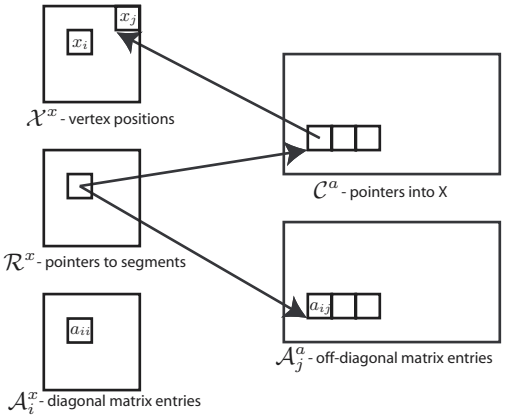

Figure 2: *When the fragment program executes on the pixel corresponding to row i, the window position is used as a texture coordinate to fetch x_i in $\mathcal{X}^x$ and a_{ii} in $\mathcal{A}_i^x$. The window position also identifies the segment pointer in $\mathcal{R}^x$, which points to the location of non-zero elements a_{ij} in $\mathcal{A}_j^a$ corresponding to row i. Finally, using the segment pointer from $\mathcal{R}^x$ we can access entries in $\mathcal{C}^a$ which reveal the addresses of x_j in $\mathcal{X}^x$ corresponding to non-zero a_{ij}.*

textures identify the vertices incident on the two triangles associated with edge e_{ij}. Computation of $\mathcal{A}_j^a$ can now be performed by the fragment program

$$\begin{aligned} \cot(a,b,c) &= \frac{(a-b)\cdot(c-b)}{\|(a-b)\times(c-b)\|} \\ x_i &= \mathcal{X}^x[\mathcal{I}^a[j]] \quad x_j = \mathcal{X}^x[\mathcal{C}^a[j]] \\ x_{j_p} &= \mathcal{X}^x[\mathcal{P}^a[j]] \quad x_{j_n} = \mathcal{X}^x[\mathcal{N}^a[j]] \\ \mathcal{A}_j^a[j] &= -\lambda\Delta t(\cot(x_j, x_{j_p}, x_i) + \cot(x_i, x_{j_n}, x_j)). \end{aligned}$$

The $\mathcal{A}_i^x$ are computed with the same overall structure as a sparse matrix vector multiply and the following fragment program

$$\mathcal{A}_i^x[i] = \sum_{c=0}^{k_i-1} 4\mathsf{A}(x_i, x_{j_p}, x_j) - \mathcal{A}_j^a[\mathcal{R}^x[i]+c].$$

Boundaries In geometric flow one can either fix vertices on the mesh boundaries (as we do in the examples), or let them flow under a length minimizing curvature flow. The latter requires its own tridiagonal linear system, which can be implemented on the GPU as well. Fixed boundaries effectively remove some vertices from the list of degrees of freedom, though they still enter into the matrix coefficient computations. So while they are stored in $\mathcal{X}^x$, they are not assigned to a rectangle with a fragment program: there are no corresponding rows in the matrix.

3.1.3 Reduction Operators

Reduction operators apply a binary associative[1] operator to all elements of a vector returning the result $r = v_1 \circ v_2 \circ \cdots \circ v_n$. The operator is not required to be commutative, *e.g.*, the vector could contain matrices and $\circ$ may be matrix multiplication. We require only sum-reduction and will take advantage of the fact that addition is commutative, *i.e.*, we will not require any particular order. This allows us to perform reduction for vectors, such as $\mathcal{X}^x$, indexed by two indices without regard to the order[2]. To compute the inner

[1] Real addition associates, floating point addition does not. We will ignore this distinction.

[2] The traditional way of dealing with higher-D is to perform reductions in each dimension in order [Blelloch 1990; The C* Team 1993].

product of two vectors $p = \mathbf{x} \cdot \mathbf{y}$, we need a sum-reduction on the pairwise products.

Let $\mathcal{X}^x$ be a vector holding elements to be sum-reduced. The reduction is achieved by rendering a quadrilateral with half the dimension along either axis, summing four elements. Applying this process repeatedly, akin to a mip map pyramid, we will finally render a single pixel quadrilateral containing the sum-reduction result.

Notes If the dimensions of $\mathcal{X}^x$ are not powers of two the reduction must deal with odd length dimensions. One could always test whether texels are out of bounds before including them in the reduction. Alternatively, separate fragment programs can be run on the boundaries. Because of less than full utilization of functional units for smaller textures this is unattractive. Texels that do not correspond to actual data elements can be reduced so long as they contain the identity of the reduction operator. Thus to initialize a sum-reduction we place zeros in unused texels.

3.2 Packing

Recall that $\mathcal{X}^x$ textures will be read and *written* by fragment programs. For improved performance of writing operations we optimize the layout of $\mathcal{X}^x$ variables. These come in groups of size i_k according to the number k of non-zero off-diagonal entries in $\mathbf{A}$. We discuss a performance model and then the optimization.

3.2.1 GPU Streaming Model

So far we have adopted an *abstract* streaming model. Now we specialize; consider the characteristics of a *GPU streaming model*:

SIMD GPUs execute fragment programs in SIMD fashion. Let p be the number of parallel pipelines; then every instruction operates on a *tuple* of p neighboring pixels. For peak performance we must make useful work of every pixel in the tuple.

Triangle Rasterization GPUs are optimized for rendering triangles. We assume that axis aligned rectangles are rendered as a pair of axis aligned right triangles. In rendering each right triangle there is wasted work along the hypotenuse since parts of some tuples will lie outside the triangle. Hence, in rendering a rectangle of tuple dimension $w \times h$, tuples along the diagonal tend to be rasterized twice (one copy is subsequently discarded, but compute power is wasted). The total number of tuples sent to the fragment stage is therefore $w \cdot h$ plus a number of wasted tuples bounded above by $\max(w, h)$. For peak performance we must minimize the wasted work along the diagonal.

Round-Robin Pipelining of Texture Memory Access A fundamental issue in streaming architectures is hiding the memory-access latency [Arvind and Iannucci 1987]. To that end, streaming processors are typically multi-threaded (one early example was the HEP computer [Leiserson et al. 1993]). In a multi-threaded architecture, q independent stream records are processed in an interleaved manner: the program with instructions $I_1, I_2, I_3, \ldots$ is executed over records $R_1 \ldots R_q$ using the sequential ordering $I_1(R_1), I_1(R_2), \ldots, I_1(R_q), I_2(R_1), \ldots, I_2(R_q), I_3(R_1) \ldots$. Note that $q - 1$ cycles elapse between potentially data-dependent instructions. The designer must choose q large enough to hide memory latency and trade this off against the required additional chip area. A key consequence is that the fragment units process tuples in batches of size q. Since a partially filled pipeline incurs the same cost as a full pipeline we can observe a *batch quantization* effect (Figure 3). For peak performance we must ensure that all q tuples provide useful work.

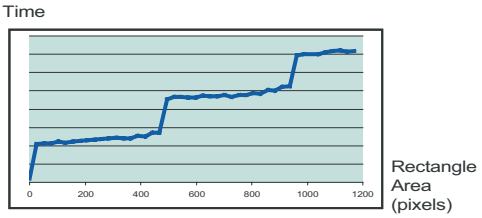

Figure 3: *Normalized plot showing time vs. rectangle area as observed on the GeForce FX, driver version 42.51. The value* $q \cdot p = 512$ *is clearly noticeable, although the area is actually just below* 500*, consistent with the assumption that fragments on the diagonal are rasterized (not rendered) twice on the GeForce FX. To isolate the effect of batch quantization, we used a long fragment program (eliminating setup, rasterization and vertex-program overhead) with no fetch operations (eliminating bandwidth overhead).*

3.2.2 Optimization

Taking into consideration our above model we pack an output stream into a sequence of *identical* rectangles, each with dimensions w by h. Optimizing for performance, we choose the dimensions w and h, partition the data stream over the rectangles, and assign a (possibly different) fragment program to every rectangle.

Dimensions The dimensions are hardware and driver dependent: we find good choices for rectangle area, $w \cdot h$, by simple experiments which reveal estimates of $p \cdot q$ (Figure 3). A rectangle of dimensions w and h produces $w \cdot h$ tuples plus some number of additional wasted tuples bounded above by $\max(w, h)$. With the fragment stage processing batches of q tuples, an optimal rectangle maximizes the amount of useful work $w \cdot h$ subject to the constraint # tuples $\leq z \cdot q$ for some integer z. This defines a family of optimal rectangles corresponding to the number $z = 1, 2, \ldots$. We choose a single solution from this family as it leads to a trivial rectangle layout algorithm. In particular, we choose $z = 1$ since this provides the finest granularity in packing the stream. We neglect per-primitive startup costs, assuming this is small for primitives containing a minimal number of tuples ($< w \cdot h$).

We assume that batch quantization is only noticeable when changing programs, *e.g.*, when a pipeline flush is *required*. In reality, pipeline flushes may or may not be performed when switching between programs, depending on the hardware and driver versions. Our model is conservative: it assumes that a change in program is *always* followed by a pipeline flush. In practice, the rectangle size chosen based on this conservative model has worked well.

The inefficiency for $z = 1$ as compared to $z \gg 1$, *e.g.*, extra diagonal waste and start-up cost, is negligible ($< 10\%$) and well worth the triviality of the resulting layout problem. On the GeForce FX we found that a rectangle of dimensions 26×18 pixels gives the lowest proportion of wasted pixels.

Rectangle Layout The uniformity of dimensions makes the packing problem trivial to precompute. The only parameter to the layout problem is the number of rectangles to lay out. We can compute off-line the optimal layout, *i.e.*, determine the best texture dimensions s and t as a function of the *number* of rectangles to lay out. At runtime a simple lookup gives a good layout.

Multiple Programs For the sparse matrix problem we have multiple streams S_k each with i_k records to be processed by a program P_k. Note that program P_k can process sparse matrix rows with k or *fewer* non-zero off-diagonals if we allow zero padding. This property is useful in meshes with few vertices of a particular valence. The underlying assumption is that a partially filled batch of q tuples costs the same as a fully filled batch due to pipeline flushes.

We adopt the following greedy solution. Lay out the streams in *decreasing* order of program cost $k = n-1, \ldots, 0$ and assign them

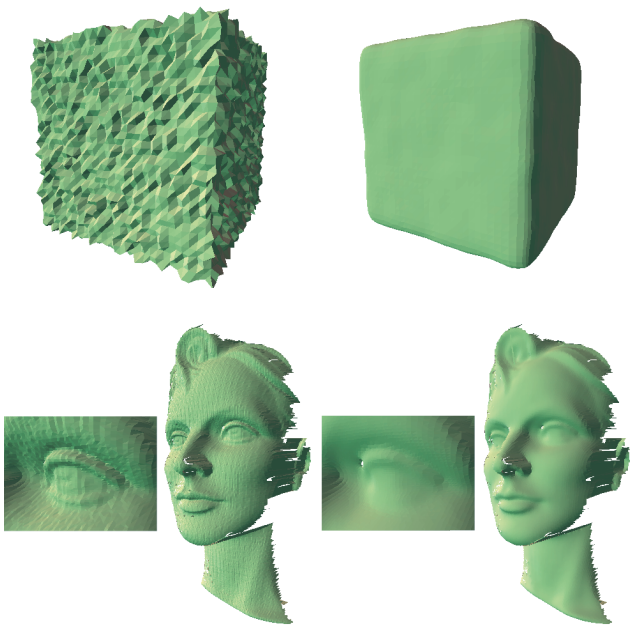

Figure 4: *On the upper left a cube with normal noise and to its right a smoothed version. On the lower left a scanned mesh contaminated with acquisition noise. It is denoised through geometric flow. Note that this mesh has complicated boundaries. For movies see http://multires.caltech.edu/pubs/GPUSim.mpg*

to rectangles $R_1, R_2, \ldots$, filling each rectangle to capacity. Use the least expensive program P_k valid for all elements of R_j.

Notes The layout is resolved when the mesh is first created. However, the entries in $\mathcal{A}_j^a$, *i.e.*, the length of each row segment, must be appropriately padded. The corresponding entries in $\mathcal{C}^a$ should point to a *constant* address in $\mathcal{X}^x$ which contains the value zero (*constant* to avoid unnecessary burden on the texture cache).

3.3 Performance

We have implemented all of the components of a general conjugate gradient solver, as well as the specific matrices for geometric flow, including their recomputation for each smoothing step. The most performance critical functions are the matrix-vector multiply and the sum-reduction. We performed timing tests on a GeForce FX board using a mesh with 37k vertices (the scanner data in Figure 4).

The matrix-vector multiply takes 33 instructions on an average row of the matrix (seven non-zero elements), 21 of which are texture fetches. At 500 MHz and 37k vertices, one could theoretically perform over 500 matrix multiplies per second. In practice, we observe about 120 matrix multiplies per second. Further tests show that the discrepancy is due to the random access pattern causing cache thrashing.

A reduction, including out of bounds checking, requires ten instructions (four fetches, three adds, three compares) per destination fragment per pass. For a 200×200 layout to be reduced to 100×100, the cost of one pass of the reduction is 100k instructions. Doing this at all levels of the hierarchy increases this by a factor of $4/3$. Theoretically, the sum reduction can be performed over 15000 times per second. In practice our code executes at roughly 3400 reductions per second.

The CG inner loop does a single matrix multiply and two reductions, as well as some significantly less costly operations that can be considered free. This entire loop can be performed about 110 times per second. The CG solver typically only needs a few iterations—*e.g.*, five—for each smoothing step, so an entire smoothing step can be performed in less than $1/20^{\text{th}}$ of a second.

Notes Current drivers have a performance penalty—revalidation of the OpenGL pipeline state during pbuffer switches—which is unnecessary for our computations. At present only about 200 pbuffer switches per second are possible, severely limiting performance. This limitation will be removed in the near future. Hence all timings are given with the pbuffer overhead removed. *The movies were produced from screen dumps running on actual hardware, in effect simulating a setting with the pbuffer switch penalty removed.*

4 Solver for Regular Grids

We now turn to a solver for discretizations of elliptic PDEs over regular grids. In that case the sparse matrices have a very regular structure enabling efficient implementation of multigrid solvers.

For the generic setup we consider the Helmholtz equation with Dirichlet and/or Neumann boundary conditions on the unit square, $\Omega = [0,1]^2$

$$\begin{aligned} -\nabla^2 u(\mathbf{x}) + \sigma u(\mathbf{x}) &= g(\mathbf{x}) \quad \mathbf{x} \in \Omega \subset \mathbb{R}^2 \\ u(\mathbf{x}) &= u_D(\mathbf{x}) \quad \mathbf{x} \in \partial\Omega \quad \text{or} \\ \vec{n} \cdot \nabla u(\mathbf{x}) &= u_N(\mathbf{x}) \quad \mathbf{x} \in \partial\Omega, \end{aligned}$$

where $\vec{n}$ denotes the outward pointing normal on the boundary of the domain whenever this quantity is well defined. This PDE may be solved via discretization which leads to a matrix problem. Whether one uses finite elements, volumes, or difference, the structure of the resulting matrices is essentially identical. For concreteness we use a finite difference discretization as in [Stam 1999]

Linear System After discretization we have to solve a linear system $\mathbf{A}_h \mathbf{u}_h = \mathbf{b}_h$ in $(N+1) \times (N+1)$ variables, not all of which are free. The linear operator $\mathbf{A}_h$ acting on the 2D grid $\mathbf{u}_h$ may be described by a set of stencils, with possibly varying entries, each of size no larger than 3×3 (2×3 at the boundary, 2×2 at the corner). Care is required in the Neumann case as it has a non-trivial null space [Briggs et al. 2000, pp. 113–119].

4.1 Incompressible Navier-Stokes

Fluid simulations are a typical representative of settings which give rise to such systems. As a challenge problem we implemented Stam's solver for the incompressible viscous Navier-Stokes equations [1999]. The fluid is governed by a velocity field $\mathbf{u}$ which satisfies

$$\nabla \cdot \mathbf{u} = 0 \quad \rho \frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \rho \mathbf{b}$$

where $\mathbf{b}$ denotes external body force, ν the viscous drag, and the ρ is the fluid density (we will assume it to be unity). Using a semi-implicit time discretization as well as a projection step to ensure a divergence free vector field, the update from time $t^{(k)}$ to $t^{(k+1)} = t^{(k)} + \Delta t$ proceeds as

$$\begin{aligned} (\rho \mathbf{I} - \nu \Delta t \nabla^2)\mathbf{u}^* &= \rho \mathbf{u}^{(k)} + \Delta t(\rho \mathbf{b}^{(k)} - (\mathbf{u}^{(k)} \cdot \nabla)\mathbf{u}^{(k)}) \quad (2) \\ \nabla^2 p &= (\rho/\Delta t) \nabla \cdot \mathbf{u}^* \quad (3) \\ \mathbf{u}^{(k+1)} &= \mathbf{u}^* - (\Delta t/\rho) \nabla p\,, \end{aligned}$$

where p is a pressure field which may be discarded at the end of the time step. The first equation solves for a new velocity field using a Helmholtz solver with zero Dirichlet boundary conditions, while the second equation uses a Poisson solver with zero Neumann boundary conditions. Implementing this time stepping requires two

non-trivial functions. One for the advection step and the other for an elliptic PDE solver.

Notes To keep the multigrid solver simple we do not work with staggered grids [Fedkiw et al. 2001], but instead colocate pressure and velocity fields as was done by Stam [1999]. The domain is discretized into $(N+1) \times (N+1)$ uniformly spaced samples distance $h = 1/N$ apart with pressure and velocity variables

$$p_{i,j} \quad \mathbf{u}_{i,j} = (u_x, u_y)_{i,j} \quad i,j = 0, \ldots, N$$

In case periodic boundary conditions are used the associated indices are to be understood modulo N, *i.e.*, $0 \equiv N$.

4.1.1 Advection Step

Following Stam [1999] we trace integral curves back in time to compute the advected velocity field $\mathbf{u}^{\text{adv}} = (\mathbf{u} \cdot \nabla)\mathbf{u}$

$$\mathbf{u}^{\text{adv}}_{i,j} = C(\mathbf{u}, -\Delta t \mathbf{u})_{i,j}.$$

where we used a simple first order integrator (higher order integrators can be defined as well). The operator C returns a field which is interpolated (typically bilinear) from its first argument field using offset vectors given by the second argument field. Offsets that leave the domain are properly clamped to the domain boundary. Note that boundary conditions are automatically preserved by the interpolation operation.

4.2 Multigrid

In this section we assume that $N = 2^j$ for some $j > 5$. Systems with fewer variables can be solved sufficiently fast with a diagonally preconditioned conjugate gradient algorithm. To simplify the exposition we will only describe a simple V-cycle with either Neumann or Dirichlet boundary conditions. Mixed boundaries are possible as well, but not required by our examples. The basic ingredients required for multigrid are a simple relaxation scheme—we will use weighted Jacobi—and interpolation and projection operators. The latter are used to build the system matrices at coarser levels as well as to propagate residual and correction vectors between levels of resolution. For an excellent introduction to the multigrid method we recommend the text by Briggs and co-workers [2000].

To build the coarser matrices we need an interpolation operator $\mathbf{S}$ from coarse to fine and a projection operator $\mathbf{P}$ from fine to coarse. For $\mathbf{S}$ we chose bilinear subdivision and for $\mathbf{P}$ *full weighting*, *i.e.*, $\mathbf{P} = 1/4\mathbf{S}^T$, resulting in a coarser level system matrix

$$\mathbf{A}_{2h} = \mathbf{P}\mathbf{A}_h\mathbf{S},$$

and so on recursively to some coarsest level. With these in place the V-cycle multigrid algorithm can be stated as

$\mathbf{u}_h \leftarrow$ V-Cycle$(\eta_1, \eta_2, \mathbf{v}_h, \mathbf{b}_h)$
  If (CoarsestQ (h))
    Return $\mathbf{u}_h \leftarrow$ Solve$(\mathbf{A}_h, \mathbf{v}_h, \mathbf{b}_h)$
  Else
    $\mathbf{v}_h \leftarrow$ Relax$(\eta_1, \mathbf{v}_h, \mathbf{b}_h)$ // pre-smooth
    $\mathbf{b}_{2h} \leftarrow \mathbf{P}(\mathbf{b}_h - \mathbf{A}_h\mathbf{v}_h)$ // project residual
    $\mathbf{v}_{2h} \leftarrow$ V-Cycle$(\eta_1, \eta_2, \mathbf{0}_{2h}, \mathbf{b}_{2h})$ // recurse
    $\mathbf{v}_h \leftarrow \mathbf{v}_h + \mathbf{S}\mathbf{v}_{2h}$ // interpolate & correct
    Return $\mathbf{u}_h \leftarrow$ Relax$(\eta_2, \mathbf{v}_h, \mathbf{b}_h)$ // post-smooth

Iteration towards solution at the finest level is performed by starting with an initial guess $\mathbf{v}_h$ and the right hand side $\mathbf{b}_h$ and improving it through repeated application of the V-cycle algorithm. The parameters η_1 and η_2 control the number of pre- and post-smoothing steps. Other variants such as μ-cycle and full multigrid are straightforward variations on the above code [Briggs et al. 2000]. For the smoother we chose the damped Jacobi iteration with $\omega = 2/3$

$\mathbf{u}_h \leftarrow$ Relax$(\eta, \mathbf{v}_h, \mathbf{b}_h)$
  For ($i = 0; i < \eta; i = i + 1$)
    $\mathbf{r} \leftarrow \mathbf{b}_h - \mathbf{A}_h\mathbf{v}_h$
    $\mathbf{v}_h \leftarrow \mathbf{v}_h + \omega(\mathbf{A}_h)^{-1}_{ii}\mathbf{r}$
  Return $\mathbf{u}_h \leftarrow \mathbf{v}_h$

For simplicity we will use Jacobi iteration for the coarsest level solve as well.

From the above pseudo code we see that we need the following non-trivial functions: (1) application of the interpolation operator $\mathbf{S}$ to a vector; (2) application of the projection operator $\mathbf{P}$ to a vector; (3) application of $\mathbf{A}_h$ to a vector; and (4) computation of $\mathbf{A}_{2h}$.

Texture Layout Assuming a 2D domain, we lay out all data in 2D textures. Access into textures is described through multi-index notation $i = (i_0, i_1)$, with arithmetic understood in the vector sense. To iterate over neighbors, respectively entries in a stencil, we use index sets such as $\{0,1\}^2 = \{(0,0),(0,1),(1,0),(1,1)\}$ using the standard definition of set product.

Recall that $\mathbf{A}_h$ can be seen as a map from grid points to 3×3 stencils. We store the matrix as nine 2D textures $\mathbf{A}^d_h[i]$, $d \in \{-1,0,1\}^2$ each a map from grid points i to the stencil entry indexed by d. Matrices $\mathbf{A}_{2h}, \mathbf{A}_{4h}$, *etc.* are similarly stored.

To utilize all four channels in the floating point units, the computational domain is cut into four quadrants which are layered into the (x, y, z, w) channels of a single texture with half the size in each dimension. This layering must group odd and even indexed nodes so that interpolation and projection operators can be applied correctly. Since many computations require neighbor access, this layered format needs a single pixel border all around which contains duplicates of appropriate nodes respectively zeros for border edges. These duplicates must be kept in sync during computation, issues well known from domain decomposition [Demmel 1997]. This synchronization is performed on the GPU with a set of programs that copy the appropriate data into the texture border. Layering and un-layering is performed as needed with short fragment programs which do not pose a performance problem due to their brevity. For an alternative approach to using four channels for scalar problems see [Hall et al. 2003].

Interpolation Given a vector $\mathbf{v}_{2h}$ stored in a 2D texture, interpolation to the finer level becomes

$$\mathbf{v}_h[i] = 1/4 \sum_{d \in \{0,1\}^2} \mathbf{v}_{2h}[\lfloor (i+d)/2 \rfloor].$$

For Neumann conditions the boundaries get interpolated as any other point in the domain. For Dirichlet boundary conditions the boundary should not be interpolated, but we do so anyway. Since the Dirichlet boundary conditions always vanish, no harm is done and the implementation is simplified.

Projection Since we are using full weighting, projection is performed via the adjoint of subdivision save for a division by four

$$\mathbf{v}_{2h}[i] = 1/4 \sum_{d \in \{-1,0,1\}^2} \mathbf{S}^d \mathbf{v}_h[2i + d]$$

where d indexes the stencil, $\mathbf{S} = 1/4\{1,2,1,2,4,2,1,2,1\}$ (using lexicographic order on d). This code applies at all points in the interior of the domain. On the boundary ($i_0 = 0, N/2$ and/or $i_1 = 0, N/2$) the above code is correct for Neumann boundary conditions with out of bounds accesses clamped to zero. For Dirichlet boundary conditions all boundary values of $\mathbf{v}_{2h}$ must be set to zero explicitly. This is easily achieved by initializing to zero and "rendering" a rectangle covering only indices $1 \ldots N/2 - 1$.

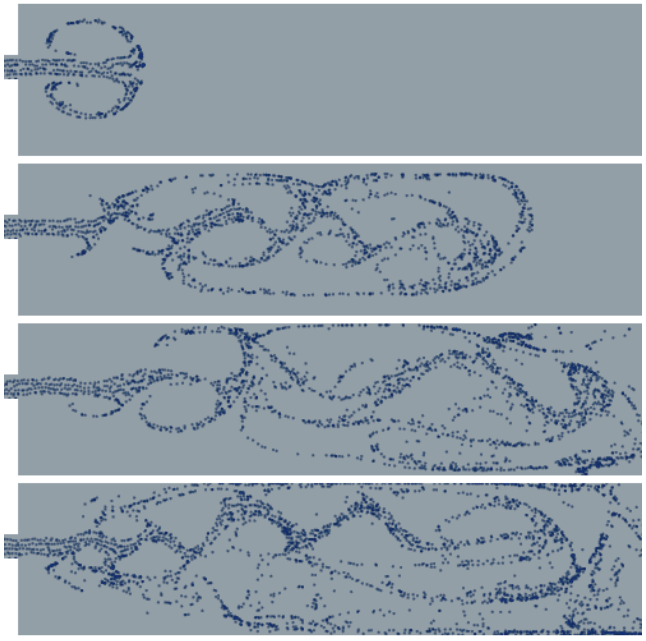

Figure 5: *An example of a few time steps during a simulation run with particle advection used for visualization of the velocity field (For movies see http://multires.caltech.edu/pubs/GPUSim.mpg). The inlet on the left has a high inward velocity (two grid spaces per timestep). This particular simulation was run with a domain size of* 513×129*. When fluid is pushed into the larger area, which is initially at rest, vortices are shed, a phenomenon clearly visible in the pattern of the particles.*

Matrix Vector Multiply Here stencil accesses to the matrix $\mathbf{A}_h$ must be mapped appropriately to index offsets in the vector $\mathbf{u}_h$

$$\mathbf{v}_h[i] = \sum_{d \in \{-1,0,1\}^2} \mathbf{A}_h^d[i]\mathbf{u}_h[i+d].$$

For Dirichlet boundaries this code is executed only in the interior. For Neumann boundaries it is also executed on the boundary with out of bounds accesses clamped to zero.

Composition of Stencils The most involved operation is the computation of coarser level stencils as the composition of operators $\mathbf{S}$, $\mathbf{A}_h$, and $\mathbf{P}$ each of which is given as a stencil. Formally we can express this as the triple operator composition $\mathbf{A}_{2h} = \mathbf{P}\mathbf{A}_h\mathbf{S}$, which may be expanded in index notation as

$$\mathbf{A}_{2h}^d[i] = 1/4 \sum_{e,g \in \{-1,0,1\}^2} \mathbf{S}^e\mathbf{S}^{e+g-2d}\mathbf{A}_h^g[2i+e]$$

Stencils on the boundary only need to be computed for Neumann boundaries, in which case out of bounds accesses are clamped to zero.

Performing the stencil composition correctly was one of the most subtle implementation issues in our algorithm. In the above expression we assume that out of bounds accesses on $\mathbf{S}$ are clamped to zero, easily achieved by creating a very small, suitably padded texture map for $\mathbf{S}$. The sum could also be teased apart with careful analysis of which terms actually contribute to the sum. This not being a time critical part of the algorithm we refrained from that optimization.

4.3 Performance

We have implemented the fluid solver of Stam [1999] as a realistic application context for a multigrid solver. Following [Fedkiw et al. 2001] we did not perform diffusion ($\nu = 0$) on the velocity variables, as visually the numerical diffusion is sufficient. Consequently we used the multigrid solver only for the Neumman problem involved in the pressure computation (Equation 3).

The most expensive operation in the multigrid V-cycle is the matrix-vector multiply, *i.e.*, application of the stencil. The program has 27 instructions, 18 of which are texture fetches. On a 257×257 grid (stored as a 129×129 4-deep texture), this operation can be performed over 1370 times per second at 500 MHz (the theoretical peak is 4500 per second). The operation is proportionally cheaper on smaller grids, except that on very coarse grids there is a penalty due to the batch quantization. On a 17×17 fragment grid this penalty is almost 50%, at 33×33 it is 29%, and decreases as the grid gets larger.

The interpolation and projection steps are applied once per level in the V-cycle algorithm. The interpolation kernel is ten instructions long while projection takes 19. These kernels can execute 4800 respectively 6000 times per second to move between 257×257 and 129×129. We have found that a good choice of parameters is to have several Jacobi iterations per level, which decreases the performance impact of interpolation and projection. In general, they do not limit performance.

The advection step is not performed directly on the layered velocity field because it is impossible to fetch each of four channels from different advected locations (texels) in a single instruction. Similarly the bilinear interpolation needed for the advection step must be coded explicitly since floating point buffers do not provide this functionality. Consequently the advection step first un-layers the velocity field, performs the computations for advection, and then layers the velocity variables again. Since this must only be performed once per timestep the cost relative to the solver step is negligible.

The number of pre- and post-smoothing iterations can be tweaked as necessary depending on the particular problem being solved. The system in Figure 5, with at times large divergence only required four pre-smoothings, two post-smoothing steps, and two V-cycles.

Neglecting pbuffer overhead, a single timestep with two multigrid V-cycles, using four pre-smoothing and two post-smoothing steps could be applied to a 513×513 grid in approximately $1/20^{\text{th}}$ of a second. For our simulations we used a grid of 513×129.

5 Conclusion

We have demonstrated a mapping of two widely applicable solvers to the GPU, and have provided solutions to many of the peculiarities that arise on this hardware. These design choices can be applied to other algorithms as well. Neglecting the unnecessary overhead of pbuffer switching, our implementation performs well. Both applications would run in realtime for the given problem instances. We implemented CPU versions of the matrix multiply kernels using SSE, and tested them on a 3GHz Pentium 4. The GPU implementation achieves 120 unstructured matrix multiplies per second whereas the CPU implementation can only do 75 per second on the stated problem instance. For the structured matrix multiply, the GPU can do 1370 matrix multiplies per second whereas the CPU can do 750 per second. Our tests have shown that both the CPU and GPU implementations are bandwidth limited.

The multigrid solver has enormous performance potential, and would be even more useful if it were applied to irregular grids. The issues involved in this deserve further study. A more straightforward extension would be application of the proposed sparse matrix conjugate gradient algorithm to simulations involving tetrahedral meshes.

The Cg language could provide an alternative implementation path, but it is not as well suited to scientific computing applications.

For example, the concept of a pointer is meaningful for numerical applications, but not necessarily for graphics. The ability to "allocate" parts of a texture—similar to "processor allocation" [Blelloch 1990] in SIMD programming—would alleviate some of the tedium of packing. A specialized compiler that hides such distinctions would make this hardware more accessible to those less familiar with computer graphics.

Performance could be boosted further if texture fetch instructions allowed additional offsets to be added to input coordinates. This could decrease the instruction count of the CG matrix multiply by as much as 20%. Finally, reduction operators would benefit greatly from a few globally writable registers. For example, a single accumulation register would allow a sum-reduction to be performed in a single pass. Limiting such registers to commutative operators would avoid troublesome order dependencies. Reductions would also be simplified by allowing borders on floating point textures.

Acknowledgment This work was supported in part by NSF (DMS-0220905, DMS-0138458, ACI-0219979), the DOE (W-7405-ENG-48/B341492), NVIDIA, the Center for Integrated Multiscale Modeling and Simulation, Alias|Wavefront, Pixar, and the Packard Foundation. Special thanks to Matt Papakipos, Nick Triantos, David Kirk, Paul Keller, Mark Meyer, Mika Nyström, Niles Pierce, Burak Aksoylu, Michael Holst, Jason Hickey, André DeHon, Ian Buck, Mark Harris and all the speakers and students in the "Hacking the GPU" class (Caltech, Fall 2002).

References

ARVIND, AND IANNUCCI, R. A. 1987. Two Fundamental Issues in Multiprocessing. In *Proceedings of DFVLR - Conference 1987, Parallel Processing in Science and Engineering*.

BARRETT, R., BERRY, M., CHAN, T. F., DEMMEL, J., DONATO, J., DONGARRA, J., EIJKHOUT, V., POZO, R., ROMINE, C., AND DER VORST, H. V. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd ed. SIAM.

BLELLOCH, G. E. 1990. *Vector Models for Data-Parallel Computing*. MIT Press.

BRIGGS, W. L., HENSON, V. E., AND MCCORMACK, S. F. 2000. *A Multigrid Tutorial*, 2nd ed. SIAM.

CARR, N. A., HALL, J. D., AND HART, J. C. 2002. The Ray Engine. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*.

COHEN, M. F., CHEN, S. E., WALLACE, J. R., AND GREENBERG, D. P. 1988. A Progressive Refinement Approach to Fast Radiosity Image Generation. *Computer Graphics (Proceedings of SIGGRAPH) 22*, 75–84.

DEMMEL, J. W. 1997. *Applied Numerical Linear Algebra.* SIAM, Philadelphia, PA.

DESBRUN, M., MEYER, M., SCHRÖDER, P., AND BARR, A. H. 1999. Implicit Fairing of Irregular Meshes Using Diffusion and Curvature Flow. In *Proceedings of SIGGRAPH*, 317–324.

DESBRUN, M., MEYER, M., AND ALLIEZ, P. 2002. Intrinsic Parameterizations of Surface Meshes. *Computer Graphics Forum (Proceedings of Eurographics) 21*, 3, 209–218.

DIEWALD, U., MORIGI, S., AND RUMPF, M. 2002. A Cascadic Geometric Filtering Approach to Subdivision. *Comput. Aided Geom. Des. 19*, 9, 675–694.

FEDKIW, R., STAM, J., AND JENSEN, H. W. 2001. Visual Simulation of Smoke. In *Proceedings of SIGGRAPH*, 15–22.

GOODNIGHT, N., LEWIN, G., LUEBKE, D., AND SKADRON, K. 2003. A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware. Tech. Rep. CS-2003-03, University of Virginia.

HACKBUSCH, W. 1985. *Multi-Grid Methods and Applications*. Springer Verlag, Berlin.

HALL, J. D., CARR, N. A., AND HART, J. C. 2003. Cache and Bandwidth Aware Matrix Multiplication on the GPU. Tech. rep., University of Illinois.

HARRIS, M. J., COOMBE, G., SCHEUERMANN, T., AND LASTRA, A. 2002. Physically-Based Visual Simulation on Graphics Hardware. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*.

HARRIS, M. J. 2002. Analysis of Error in a CML Diffusion Operation. Tech. Rep. TR02-015, UNC Chapel Hill.

HILLESLAND, K. E., MOLINOV, S., AND GRZESZCZUK, R. 2003. Nonlinear Optimization Framework for Image-Based Modeling on Programmable Graphics Hardware. *ACM Transactions on Graphics*.

HOFF, K. E., KEYSER, J., LIN, M., MANOCHA, D., AND CULVER, T. 1999. Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware. In *Proceedings of SIGGRAPH*, 277–286.

KASS, M., AND MILLER, G. 1990. Rapid, Stable Fluid Dynamics for Computer Graphics. *Computer Graphics (Proceedings of SIGGRAPH) 24*, 4, 49–57.

KELLER, A. 1997. Instant Radiosity. In *Proceedings of SIGGRAPH*, 49–56.

KHAILANY, B., DALLY, W. J., RIXNER, S., KAPASI, U. J., MATTSON, P., NAMKOONG, J., OWENS, J. D., TOWLES, B., AND CHANG, A. 2001. Imagine: Media Processing with Streams. *IEEE Micro 21*, 2, 35–46.

KHAILANY, B., DALLY, W. J., RIXNER, S., KAPASI, U. J., OWENS, J. D., AND TOWLES, B. 2003. Exploring the VLSI Scalability of Stream Processors. In *Proceedings of the Ninth Symposium on High Performance Computer Architecture*.

KOBBELT, L., CAMPAGNA, S., VORSATZ, J., AND SEIDEL, H.-P. 1998. Interactive Multi-Resolution Modeling on Arbitrary Meshes. In *Proceedings of SIGGRAPH*, 105–114.

KRÜGER, J., AND WESTERMANN, R. 2003. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics*.

LARSEN, E. S., AND MCALLISTER, D. K. 2001. Fast Matrix Multiplies using Graphics Hardware. In *Supercomputing*.

LEISERSON, C., ROSE, F., AND SAXE, J. 1993. Optimizing synchronous circuitry by retiming. In *Third Caltech Conference On VLSI*.

LENGYEL, J., REICHERT, M., DONALD, B. R., AND GREENBERG, D. P. 1990. Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware. *Computer Graphics (Proceedings of SIGGRAPH) 24*, 327–335.

LI, W., WEI, X., AND KAUFMAN, A. 2003. Implementing Lattice Boltzmann Comptuation on Graphics Hardware. *The Visual Computer*. To appear.

LINDHOLM, E., KILGARD, M. J., AND MORETON, H. 2001. A User-Programmable Vertex Engine. In *Proceedings of SIGGRAPH*, 149–158.

MÜLLER, M., DORSEY, J., MCMILLAN, L., JAGNOW, R., AND CUTLER, B. 2002. Stable Real-Time Deformations. In *ACM SIGGRAPH Symposium on Computer Animation*, 49–54.

OLANO, M., Ed. 2002. *Real-Time Shading Languages*. Course Notes. ACM SIGGRAPH.

OWENS, J. D., KHAILANY, B., TOWLES, B., AND DALLY, W. J. 2002. Comparing Reyes and OpenGL on a Stream Architecture. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 47–56.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics 21*, 3, 703–712.

SEMICONDUCTOR INDUSTRY ASSOCIATION, 2002. International Technology Roadmap for Semiconductors. http://public.itrs.net/, December.

SHEWCHUCK, J. R. 1994. An Introduction to the Conjugate Gradient Method without the Agonizing Pain. http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.ps., August.

STAM, J. 1999. Stable Fluids. In *Proceedings of SIGGRAPH*, 121–128.

STRZODKA, R., AND RUMPF, M. 2001. Nonlinear Diffusion in Graphics Hardware. In *Visualization*, 75–84.

STRZODKA, R. 2002. Virtual 16 Bit Precise Operations on RGBA8 Textures. In *Vision, Modeling and Visualization*.

THE C* TEAM. 1993. *C* Manual*, 2nd ed. Thinking Machines Corporation.

THOMPSON, C. J., HAHN, S., AND OSKIN, M. 2002. Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis. In *International Symposium on Microarchitecture*.