A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware

University of Virginia Technical Report CS-2003-03

Nolan Goodnight* **Gregory Lewin†** **David Luebke*** **Kevin Skadron***
*Department of Computer Science, †Department of Mechanical & Aerospace Engineering, University of Virginia

***Abstract*—We present a method for using programmable graphics hardware to solve a variety of boundary value problems. The time-evolution of such problems is frequently governed by partial differential equations, which are used to describe a wide range of dynamic phenomena including heat transfer and fluid mechanics. The need to solve these equations efficiently arises in many areas of computational science. Finite difference methods are commonly used for solving partial differential equations; we show that this approach can be mapped onto a modern graphics processor. We demonstrate an implementation of the *multigrid method*, a fast and popular approach to solving boundary value problems, on two modern graphics architectures. Our initial tests with available hardware show speedups of roughly 15x compared to traditional software implementation. This work presents a novel use of computer hardware and raises the intriguing possibility that we can make the inexpensive power of modern commodity graphics hardware accessible to and useful for the simulation community.**

***Index Terms*—Boundary value problems, partial differential equations, multigrid method, graphics hardware.**

I. Introduction

The graphics-processing unit (GPU) sold on today's commodity video cards has evolved into an extremely powerful and flexible processor in its own right. The latest graphics architectures provide tremendous memory bandwidth and computational horsepower, with fully programmable vertex and pixel processing units that support vector operations up to full IEEE single precision [1,4]. High level languages have emerged to support the new programmability of the vertex and pixel pipelines [7, 10]. In fact, Purcell *et al.* argue that the modern GPU can be thought of as general *stream processor* and can perform any computation that can be mapped to the stream-computing model [11].

We present a technique to use modern graphics hardware for general numeric computation; specifically, we present a solver for boundary value problems based on the multigrid algorithm and implemented on the latest graphics architecture. In Section II, we present background information on the multigrid algorithm, modern graphics hardware, and previous work. Section III describes our implementation; Section IV describes its extension to arbitrary boundary conditions. In Section V we present our results, focusing on the specific problem of heat transfer, and demonstrate order-of-magnitude speedups over a CPU-only implementation. Finally, we discuss some advantages and disadvantages of our approach, describe other possible applications of this solver, and conclude with some thoughts on future work.

II. Background

A. Boundary value problems and the multigrid algorithm

An enormous variety of physical problems require the solution of boundary value problems (BVPs) of the form:

$$\mathcal{L}u = f \qquad (1)$$

where $\mathcal{L}$ is some operator acting on unknown u with a non-homogeneous term f. Such problems frequently arise in scientific and engineering disciplines ranging from heat transfer and fluid mechanics to vibration theory, quantum mechanics, and plasma physics. For example, finding steady-state temperature distribution in a solid of thermal conductivity k with thermal source S requires solving a Poisson equation $k\nabla^2 u = -S$, in which $\mathcal{L}$ is the Laplacian operator ∇^2.

In practice most BVPs cannot be solved analytically and so are discretized onto a grid to produce a set of linear algebraic equations. Several means exist for solving such sets of equations including direct elimination, Gauss-Seidel iteration, conjugate-gradient techniques, and strongly implicit procedures [9]. One technique that has found wide acceptance is the *multigrid* method. Multigrid has proven quite fast for large BVPs and is fairly straightforward to implement. A full description of the multigrid method is beyond this paper; see Press *et al.* [9] for a good overview. Here we simply summarize the broad steps or *kernels* of the algorithm in order to describe how we map them to the graphics hardware.

The *smoothing* kernel approximates the solution to (1) as discretized on a particular grid. The exact smoothing algorithm will depend on the operator $\mathcal{L}$, which is the Laplacian ∇^2 in our Poisson solver example. The smoothing kernel iteratively applies a discrete approximation of $\mathcal{L}$.

The progress of the smoothing iterations is measured by calculating the *residual*. Reduction of the residual results in reduction of the error in the solution, and the solution may be considered sufficiently converged once the residual has been reduced below a (user-specified) threshold.

However, convergence on a full-resolution grid is generally too slow, due to long-wavelength errors that are slow to propagate out of the fine grid. Multigrid circumvents this problem by recursively using coarser and coarser grids to approximate corrections to the solution. The *restriction* kernel therefore takes the residual from a fine grid to a coarser grid, where the smoothing kernel is again applied for several iterations. Afterwards the coarse grid may be restricted to a still coarser grid, or the correction residual may be pushed back to a finer grid using the *interpolation* kernel. Multigrid methods typically follow a fixed pattern of smoothing, restriction, and interpolation, then test for convergence and repeat if necessary.

B. The modern graphics processor

The modern graphics accelerator consists of tightly coupled vertex and pixel pipelines. The former performs transformations, lighting effects, and other vertex-related operations; the latter handles screen space operations and texturing and *has direct access to texture memory*, allowing the result of one computation to be used as input to a subsequent computation. This and the fact that pixel processors have enormous throughput—roughly an order of magnitude greater data throughput than vertex programs [2]—makes the pixel engine best suited for numerical algorithms.

Until recently, both pipelines were fixed-function, optimized to perform graphics-specific computations. However, the most recent generation of GPUs provide considerable programmability for these pipelines (Figure 1). They also greatly increase precision, replacing the 8-10 bits previously available with support for full IEEE single-precision floating point throughout the pipeline. Purcell *et al.* [11] argue that current programmable GPUs can be understood as parallel stream processors, the two pipelines highly optimized to run a user-specified program or *shader* on a stream of vertices or pixels, respectively. The NV30 architecture supports a fully orthogonal instruction set optimized for 4-component vector processing (e.g., RGBA color or XYZW position). This instruction set is shared by the vertex and pixel processors, with limitations—for example, the vertex processor cannot perform texture lookups and the pixel processor does not support branching. The individual processors have high resource limitations; for example, a pixel shader can have up to 1024 instructions.

Programming the GPU consists of writing vertex and pixel shaders, using either vendor-specific assembly instructions or a higher-level language, such as the Stanford Real-Time Shading Language [10] and NVIDIA's Cg [7], which can be compiled to an assembly profile. We have implemented our multigrid solver as a series of pixel shaders, using Cg coupled with an emulator for the NVIDIA NV30 chip [4], and later as an assembly pixel program on the ATI Radeon 9700 [1] for performance measurement (see Section V).

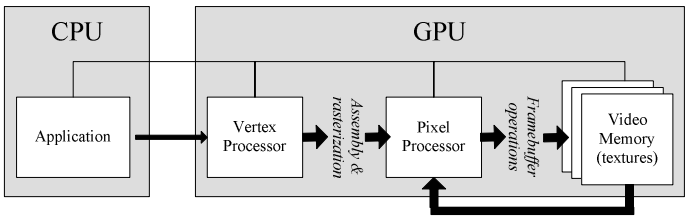

Figure 1: The modern graphics pipeline.

C. Previous work

The tremendous increase in programmability of graphics chipsets is a recent trend, and relatively little work has been done so far to exploit that programmability for computation other than variations of polygon rendering. Purcell *et al.* cast ray tracing as a series of pixel programs [10]. Their research demonstrates the flexibility of the latest graphics hardware, but still focuses on image generation. Closer in spirit to our work are approaches to GPU-accelerated physical simulation. For example, several NVIDIA demos showcase simple physical simulations modeling cloth, water, and particle system physics using vertex and pixel shaders [6]. Building on these ideas, Harris *et al.* [3] use graphics hardware for visual simulation using an extension of cellular automata known as *coupled-map lattice*. They simulate several fluid processes such as convection, diffusion, and boiling.

Thompson *et al.* apply graphics hardware to general-purpose vector processing [12]. Their programming framework compiles vector computations to streams of vertex operations using the 4-vector registers on the vertex processor. They demonstrate simple implementations of matrix multiplication and 3-SAT, with considerable speedup. Unlike their work, which uses the vertex processor, we use the faster, simpler pixel processor. This lets us feed results of one computation into the input of another, overcoming a major drawback faced by Thompson *et al.*: the need to bring results off the GPU to the CPU.

The focus of our work has been to create a hardware-accelerated framework, in the form of a multigrid solver, for solving boundary-value problems of the form discussed in Section II.A. This approach is broad, novel, and important. A fast multigrid solver has tremendously applicability compared to previous work, enabling acceleration of a whole set of real-world scientific and engineering problems. These range from modeling steady-state thermal propagation to implicit time-stepping techniques for temporal evolution of fluid mechanics. In addition, our approach involves few assumptions about the specifics of the governing equations or the structure of the solution domain.

III. Implementation

A. Overview

We keep all grid data—the current solution, residuals, source terms, and so on—in fast on-card video memory, storing the data for each progressively coarser grid as a series of images. This allows us to use the pixel pipeline, optimized to perform image processing and texture mapping operations on billions of pixels per second, for our computations. We also eliminate the need to transfer large amounts of data from main memory to and from the graphics card (a common performance bottleneck). To keep the computation entirely on the card, we implement all operations—smoothing, residual calculation, restriction, and interpolation—using pixel shaders that read from one set of input images (or *textures*) and write to an output image.

This approach almost completely decouples the CPU from the GPU. After downloading the shader code and all data to the GPU, the CPU merely issues to the GPU the sequence of shaders to be run, and occasionally checks to see if the solution has converged yet (details in Section III.C below). This decoupling allows the GPU to proceed at maximum speed without waiting for commands from the CPU or for data from main memory.

B. Relevant graphics hardware features

Our implementation relies on certain recent advances in graphics architecture:

- Floating point throughout the pipeline. NV30 provides IEEE 32-bit floating point computation and storage.
- Multi-texturing. Modern cards provide multiple simultaneous textures, and multiple lookups from each.
- Render-to-texture. This capability enables binding the rendering output from one shader as a texture for input to another shader. This avoids copying pixel data from

framebuffer to texture memory, which Harris *et al.* found to be a significant performance bottleneck [3].

C. Mapping the multigrid algorithm to hardware

The multigrid algorithm recursively solves a boundary value problem at several grid resolutions. In our implementation all computationally intensive steps—successive kernel applications, implemented as pixel shaders—are handled by the GPU. Results from one kernel become the input to the next kernel (Figure 2). In other words, we have implemented the multigrid algorithm as a series of stream computations performed entirely in the pixel pipeline, where we use the CPU only to keep track of the recursion depth and to provide the initial stream data.

Following this stream processing abstraction, the purpose of each multigrid shader is to operate on data from multiple input streams to produce a single output stream. For example, for the smoothing kernel we discretize and store the operator $\mathcal{L}$ from the boundary value equation (1) as a five-point stencil at every grid cell (Storing a separate stencil at every cell enables non-Cartesian grids, such as cylindrical coordinates). Thus the smoothing kernel combines two data streams, one containing the discretized operator $\mathcal{L}_h$ and the other containing the current solution U_h. We use texture-mapped polygons to generate these streams as pixels streaming through the GPU pixel engine. Using the OpenGL graphics API, for each kernel the general procedure is as follows:

- Bind the texture maps that contain all necessary data for the kernel computation.
- Activate a pixel shader, programming the pixel pipeline to perform that computation on every pixel.
- Render a single quadrilateral with multi-texturing enabled, sized to cover as many pixels as the resolution of the current grid.

Using this procedure, we are able to perform all of the multigrid computations by simply switching the active pixel program and binding any combination of textures as input data to the pixel pipeline. Next we discuss the four key multigrid kernels in detail, using as an example our heat-transfer problem modeled by a Poisson equation.

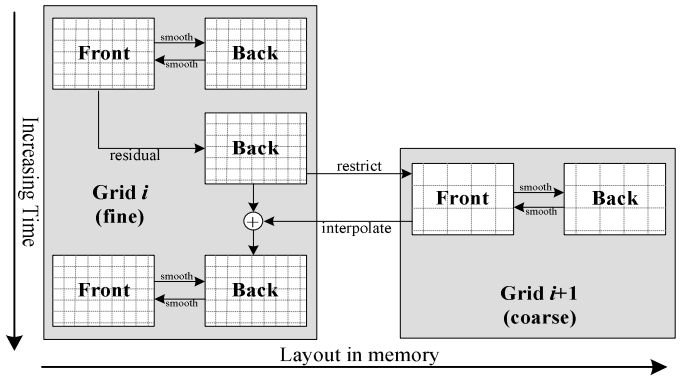

Figure 2: An illustration of two grids in the multigrid algorithm as it is implemented in graphics hardware. At grid i the smoothing pass is performed by rendering between two buffers, labeled **front** and **back**. We then restrict the residual to the front buffer for grid $i+1$ and perform the same smoothing operations on this lower-resolution grid. The approximate solution at grid $i+1$ is interpolated back to higher resolution grid i and the smoothing continues. By using two buffers at each grid level, we can bind one buffer as input and use the other buffer as a rendering target. All arrows between buffers represent render passes.

1) Smoothing

In the multigrid algorithm, *smoothing* refers to the process of approximating the solution to the boundary value equation (1) at each grid level. The actual implementation will depend on the operator represented by $\mathcal{L}$; in the case of the Poisson equation, $\mathcal{L}$ is the Laplacian operator ∇^2. The first step of the smoothing kernel is to apply this operator. The inputs are simply the current solution U and the five-point discrete approximation of the Laplacian given by:

$$\nabla^2 U_{ij} \approx U_{i-1,j} + U_{i+1,j} + U_{i,j-1} + U_{i,j+1} - 4U_{i,j} \qquad (2)$$

where i and j are row and column indices into the grid. We then approximate the next time-step in the simulation using Jacobi iteration [9], and in doing so factor in the non-homogeneous term f, which for heat transfer problems is a spatially varying function of external heat source. Finally, we apply the necessary boundary conditions, as discussed later in Section IV. After performing these operations on every pixel, the output represents a closer approximation to the steady-state solution.

2) Calculating the residual

At each grid cell, the residual value is calculated by applying the operator $\mathcal{L}$ to the current solution. As the current solution converges to steady-state, the residuals approach zero. For the Poisson equation (where $\mathcal{L} = \nabla^2$), we perform the residual calculation using a single pixel shader and store the result in texture memory in preparation for the restriction pass.

We can exploit the *occlusion query* feature of recent graphics chips to determine when steady-state has been reached using the residual calculation. The occlusion query tests whether any pixels from a given rendering operation were written to the frame buffer [5]. Every N^{th} iteration—for some user-defined N—we activate a pixel shader that compares the residual at each grid to some threshold value ε, and kill the pixel (terminating the corresponding SIMD pixel processor) if the difference is less than zero. If an occlusion query for this operation returns true, we have found the solution to (1) within a tolerance ε. By varying the threshold value we can govern the accuracy, and thus the length, of the simulation.

3) Restricting the residual

If grid G_i represents the i^{th} domain resolution, then G_{i+1} is the next-coarser grid level. We restrict the residual from G_i to G_{i+1} by setting the rendering output resolution to match the dimensions of grid G_{i+1}, then activating a pixel shader that re-samples residual values from G_i using bilinear interpolation and restricts (or *injects*) the samples to the coarser grid. In other words, the *restriction* pixel shader takes as input two data streams: a pixel for every grid cell in the G_{i+1} domain and a group of pixels in G_i for every cell in G_{i+1}. The output becomes the non-homogeneous term f from (1) for the problem to be solved on the coarse grid G_{i+1}.

4) Interpolating the correction

Finding the approximate solution at grid G_{i+1} provides a *correction* we can interpolate to grid G_i. In this case we set the output rendering resolution to match the dimensions of G_i; the active pixel shader linearly interpolates solution values from one input stream (G_{i+1}) and adds these to another input stream (G_i). At each pixel of grid G_i we perform simple modular arithmetic on the texture coordinates to determine which samples from G_{i+1} to include in the interpolation.

IV. Boundary Conditions

Fundamental to the process of solving boundary-value problems for real-world situations is the ability to specify arbitrarily complex boundary conditions. In our current hardware implementation, boundary values are treated as a simple extension to the state-space of the simulation. We have chosen this approach for its generality and compliance with the stream-processing utility of the GPU. By treating boundary conditions in this way, the pixel processor is free to perform the same computation on every pixel, and we avoid the need to include boundary-related conditionals in the pixel shader.

For example, our multigrid solver accommodates general boundary conditions for second-order problems, described by:

$$\alpha_k U_k + \beta_k \frac{\partial U_k}{\partial n_k} = \gamma_k \qquad (3)$$

where $\alpha_k, \beta_k,$ and γ_k are constants evaluated at the k^{th} boundary position and U_k is the k^{th} boundary value. The second term on the left hand side is the directional derivative with respect to the normal n_k at a given boundary. (3) can be easily implemented by storing each of the constants in texture memory. For the derivative term we simply replace the five-point operator stencil—the discretized operator from (1)—with a "boundary condition" stencil. We apply all boundary conditions as part of the smoothing pass; the user can specify a single texture that contains all boundary condition information.

V. Results

We have run a series of heat transfer simulations using the multigrid Poisson solver, and compared the numeric results of our solver both to a reference software implementation on the CPU and to the analytic solution; in all cases, the GPU implementation is correct to within floating-point precision. Our principal target architecture, NVIDIA's NV30, is currently available only in software emulation, and no performance information has been released. To obtain reasonable performance estimates, we implemented the multigrid solver in pixel assembly language on the ATI Radeon 9700. This chip has somewhat less precision (24-bit floating point) and flexibility than NV30, but is available today. We achieved roughly 15X speedup for various parameter settings and grid resolutions (Table 1). All tests used an AMD Athlon 1600 with 1 GB of system memory and an ATI Radeon 9700 graphics card [1]. Both the CPU or GPU solvers are reasonably optimized, and the algorithms are comparable in memory access and number of computations.

Grid Dimensions	CPU iters/sec	GPU iters/sec	Speedup
256 X 256 pixels	4.7	65.8	14.00
512 X 512 pixels	1.1	17.4	15.82
1024 X 1024 pixels	0.3	4.1	13.67

Table 1: Speedup measurements for a multigrid heat transfer simulation using three grids and five smoothing iterations per pass.

VI. Discussion and Future Work

We have implemented a general multigrid solver on the NV30 architecture, demonstrating a specific and broadly useful application of stream computing using graphics hardware. We achieve high performance by keeping all data—current solution, residuals, source terms, operators, and boundary information—on the graphics card stored as textures, and by performing calculations entirely in the pixel pipeline, using pixel shaders to implement the multigrid kernels: smoothing, residual, restriction, and interpolation. In general we can use our framework to solve a variety of boundary value problems; as a concrete example, we solve the Poisson equation, using our solver to model classic heat transfer. Our solver outperforms a comparable CPU implementation by a factor of about 15, demonstrating the computational power that can be harnessed by efficient use of graphics hardware.

A. Limitations

While the advent of 32-bit floating point throughout the modern GPU pipeline is a leap forward, many physical simulations require even greater precision. We still need to characterize whether workarounds could be developed for higher precision. Another limitation is the size of video memory, limited to 256 MB on current boards. Texture compression may help mitigate this problem.

B. Avenues for future work

We plan to implement more complex (non-Cartesian) grid structures, to extend the current multigrid implementation to support 3D grids, and to simulate more complicated physical phenomena. We are confident that we can accelerate a wide range of simulations that require fast and efficient solutions to boundary-value problems, such as fluid mechanics models and computer graphics tone-mapping operators. Our preliminary work raises the tantalizing possibility that scientists may be able to accelerate their simulation by an order of magnitude by investing $300 in a commodity graphics card. We are also interested in parallelizing the multigrid computation, perhaps augmenting existing computational *clusters* with inexpensive graphics cards to provide huge speedups on some problems. Finally, we are exploring an even more general framework for expressing numeric methods (not just multigrid) as streaming computations for execution using graphics hardware, especially the massive parallelism of the pixel engine.

References

[1] ATI Corporation. 2002. Radeon 9700 Pro. http://mirror.ati.com/products/pc/radeon9700pro/

[2] Carr, N. A., Hall, J. D., Hart, J. C. The Ray Engine. In *Proc. of 2002 SIGGRAPH / Eurographics Workshop on Graphics Hardware,* 2002.

[3] Harris, M. J., Coombe, G., Scheuermann, T., Lastra, A. Physically-Based Visual Simulation on Graphics Hardware, In *Proc. of 2002 SIGGRAPH / Eurographics Workshop on Graphics Hardware,* 2002.

[4] NVIDIA Corporation. 2002. GeForce FX. http://www.nvidia.com/view.asp?PAGE=geforcefx

[5] NVIDIA Corporation. OpenGL Extension Specifications. 2002

[6] NVIDIA Corporation. Demos available at http://developer.nvidia.com

[7] NVIDIA Corporation. Cg Language Specification. 2002.

[8] OpenGL Extension Specifications http://www.opengl.org/developers/documentation/specs.html

[9] Press, W. H., Tuekolsky, S. A., Vetterling, W. T., Flannery, B. P. *Numerical Recipes in C: The Art of Scientific Computing,* 2nd ed. Cambridge University Press, 1992.

[10] Proudfoot, K., Mark, W., Tzvetkov, S., Hanrahan, P. A Real-time Procedural Shading Language for Programmable Graphics Hardware. In *Proc. of SIGGRAPH 2001*.

[11] Purcell, T. J., Buck, I., Mark, W. R. and Hanrahan, P. Ray Tracing on Programmable Graphics Hardware. In *Proc. of SIGGRAPH 2002*.

[12] Thompson, C. J., Hahn, S., Oskin, M., Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis, In *Proc. of ACM/IEEE MICRO-35*, Nov. 2002