Mathematical Simulations on Programmable Graphics Hardware: The two-dimensional diffusion equation

Peter Schmitt
Department of Computer Graphics and Animation, DePaul University

Keywords: Partial Differential Equations, programmable graphics hardware, The diffusion equation

ABSTRACT

In this case study, we present a method of using programmable graphics hardware for general purpose computing. We implemented a finite differencing solution to the two-dimensional diffusion equation. Running time per computational step and numerical accuracy are compared to a traditional CPU-approach towards solving the diffusion equation.

1.0 INTRODUCTION

Many fields in science and engineering use computers to predict or model real-world phenomenon. This is also known as numerical simulation or **scientific computing**[1]. Computers conduct a set of mathematical calculations to generate data. Such data is converted to a readable format and analyzed by humans. For example, a weather report on television or in the newspaper relays information generated by computers into a human-readable format.

Each calculation on a computer takes a finite amount of time. Researchers or scientists continuously develop more complex mathematical models that require hundreds of millions of calculations for a particular problem. Fortunately, the capabilities of computer hardware have increased. This allows for more complicated models that require many mathematical calculations to generate meaningful data.

Ideally, researchers have access to infinite computational resources to conduct mathematical simulations. However, computational power is limited by money for new hardware and physical space constraints for that hardware. Existing computer hardware can be utilized as a coprocessor, leaving the main **CPU** available for additional computation.

Computer graphics hardware (also known as Graphics Processing Units or **GPUs**) is becoming increasingly powerful in terms of number of calculations per second[2]. Such hardware is driven by the consumer video game industry and is consistently lowering in price. Improvements in software and hardware make it possible to utilize **GPUs for general-purpose computing**. This work will focus on using GPUs for physics simulations of heat flow throughout a two-dimensional surface.

2.0 BACKGROUND

Our system explores mathematical simulation as an example of the power of GPUs as coprocessors. Therefore, we briefly highlight the underlying mathematics of the diffusion equation and explore numerical accuracy.

2.1 PARTIAL DIFFERENTIAL EQUATIONS

Partial differential equations (PDE) is a subfield of mathematics. PDEs are often used by physicists and engineers to develop a mathematical understanding of natural phenomenon. The diffusion equation—a specific PDE—is used to model heat flow through an object[7].

We examine the diffusion equation because it is an easy PDE that serves as a good starting point for mathematical modeling. The diffusion equation can be expanded to model fluid flow, chemical or

1 All bold terms are defined in 8.0: Glossary.

cellular concentrations and much more. Therefore, the techniques suggested by this paper can be expanded to aid in numerical computation using GPUs for a variety of applications. Although analytic solutions to the diffusion equation exist, our study is limited to numerical simulation using GPUs.

The **two-dimensional diffusion equation** models heat flow through a flat two-dimensional surface e.g. a table top or sheet of metal. Using computer software, energy is calculated for every position on a surface for any specific time, t. Mathematically, this is represented as U(x, y, t) where U = energy. The abstract mathematical definition of the diffusion equation is listed in (1).

$$\frac{\partial U}{\partial t} = D\nabla^2 U \tag{1}$$

$$\text{where } \nabla^2 U = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} \text{ and } U = U(x, y, t)$$

2.2 NUMERICAL TECHNIQUES

Numerical simulations use computer systems to generate mathematical data. The scientific community traditionally uses two types of computer systems for numerical simulations. Either a single (desktop) computer or a set of interconnected computers (also known as parallel or high performance computing) scientific software. Desktop computers are small and relatively portable, but computational power is limited by the power of its hardware. Supercomputers can be extremely powerful but are very expensive and take up a large amount of space and electricity to run.

Numerical techniques are used to convert an abstract mathematical equation to meaningful numbers. Finite differencing is a specific numerical technique frequently used to solve PDEs such as the diffusion equation [13].

This study uses a numerical technique **Forward Time Centered Space** (FTCS) finite differencing. FTCS is an iterative method used to used to generate a numerical solution to the diffusion equation**.**

FTCS updates data at a position by simulating the Laplacian or ∇^2 . Equation 2 lists the approximation of the Laplacian. The finite difference is often called a stencil, as the position is updated by accessing its neighbors. Figure 1 displays the stencil over a grid of data.

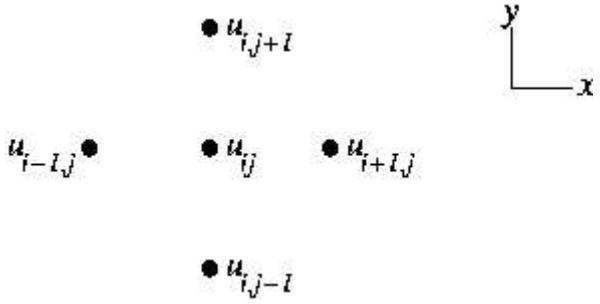

Figure 1: Laplacian stencil. Datapoint at U_{ij} is updated by applying the FTCS finite difference stencil. (Figure courtesy of Indiana University [9])

2.3 GPU ARCHITECTURE

A brief discussion on recent GPU architecture is available in [4]. This work makes use of the programmable vertex and the fragment or pixel pipelines. The vertex pipeline gives control of lighting and transformations, whereas the fragment pipeline gives control of per-pixel operations, such as texturing.

Vertex and fragment shader code is written under the OpenGL 2.0 specification. This programming specification provides direct access to graphics hardware. Our work uses an NVidia 6800 series card [3]. This work is not currently general enough to be ported directly to a non-NVidia card due to card-specific extensions that are used in this paper.

2.4 PREVIOUS WORK

Recent work makes use of programmable graphics hardware for simulations of the diffusion equation. Rumpf and Strzodka developed a diffusion method to aid in image and surface processing [16]. The paper fails to explore the methods used to generate data using the vertex and pixel pipelines. Additionally, the Rumpf and Strzodka was published in 2000, years before the high-level **OpenGL 2.0 shading language**.

Harris et. al. implemented the diffusion equation as a part of a larger effort to model cellular automaton[5]. Harris models a more advanced PDE that may make it difficult to understand how to implement such as system using OpenGL.

Goodnight, Woolley et. al. implemented a multigrid solver for a series of PDEs using programmable hardware[4]. Goodnight et. al. explore the viability of such a model, but fail to describe the underlying techniques to implement their simulation.

3.0 MATHEMATICAL MODEL

The problem is discretized, or split into a two-dimensional grid of points, as in Figure 1. Several parameters guide the diffusion. Table 1 on the following page lists each parameter and details what its effect is.

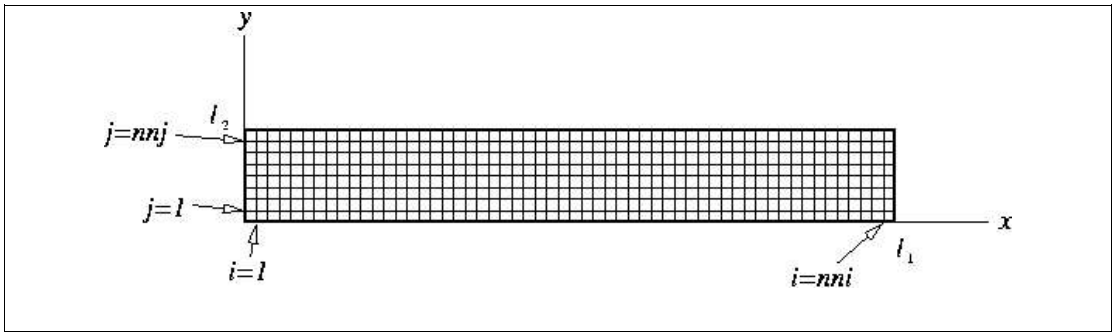

Figure 1: Discretized plate. Heat plate is simulated as a grid of finite points. (Figure courtesy of Indiana University [9])

Table 1: Diffusion parameters

Parameter	***Effect***
Diffusion Coefficient	Number that determines how much the material resists a change in temperature. For instance, copper heats up [faster/slower] than lead
Initial condition	The initial temperature distribution of the grid. This can be a single number or a function, such as $\sin(x^2+y^2)$.
Length (l_2) and Width (l_1)	How large the heat plate is
Number of grid points along length (nni) Number of grid points along width (nnj)	Once the grid has been discretized, how many data points will there be? (note: more grid points makes for a more accurate model)

4.0 IMPLEMENTATION

OpenGL, a programming application programmer interface (**API**) is used to access the GPU and display information to the screen. The programmer writes a C/C++ OpenGL application to access vertex and fragment shaders that carry out the numerical simulation. Pseudocode for our C/C++ OpenGL implementation is available in Figure 2.

```
Setup environment
        Set up OpenGL window
        Create pixel buffer # 1
        Create pixel buffer #2
        Load initial condition in to pixel buffer 1
Loop through simulation, advancing one time step on each iteration
        Computation: Loop through each point on grid and
                Vertex & Fragment Shader 1:
                        Read from pixel buffer 1
                Vertex & Fragment Shader 2:
                        Apply Laplacian on current grid point.
                        Write new value to pixel buffer 2
        Optionally display the solution using a traditional OpenGL window
```

Figure 2: Implementation pseudocode

4.1 OPENGL & THE PIXEL BUFFER

During computation, the diffusion data is stored as a set of textures resident in high-speed memory on the GPU. In order to update the simulation, OpenGL is used to write directly to a texture. For enhanced performance, off-screen rendering is used to write to directly a texture as data is calculated. As of OpenGL version 1.3, **pixel buffers** (p-buffers) provide for such a capability [12]. Implementation instructions for Render to texture is available in [19].

Rendering to a p-buffer is similar to rendering to a regular OpenGL window. To render to the p-Buffer, one must get a device and rendering context from the native windowing system. This is achieved on a Microsoft Windows-based system through a series of wgl[..] commands. The exact procedure to set up p-buffers is available in [18].

Standard OpenGL stores texture data as a series of floating point values constrained between [0..1][2] Like most scientific datasets, the diffusion equation generates data that may extend outside the [0..1] range. Additionally, standard OpenGL does not store texture data in full a 32-bit IEEE floating point format. This restriction limits the **precision** of texture memory for scientific computing.

nVidia has developed a custom **OpenGL extension** to overcome many of the drawbacks of the standard OpenGL model for handling texture data. **nv_float_buffer** gives the programmer the ability to read and write unconstrained floating point data [extension registry]. This removes the [0..1] restriction and provides for 32-bit high-precision floating point data. Using the nv_float_buffer extension, data is constrained between [$10^{-5}..10^{37}$] [8].

Standard OpenGL **texture coordinates** are clamped in the range [0..1]x[0..1] for two-dimensional textures that store temperature data. The application writes to a texture using p-buffers. Texture coordinates for p-Buffers are stored [0..width]x[0..height]. Additionally, the standard OpenGL implementation stores textures in memory constrained by power of two-dimensions (2^6 x 2^6 or 64x64, for example). One may want to run a simulation on a non-uniform grid e.g. 300X50. Using a power of two texture may allocate large quantities of unused texture memory.

Another nVidia extension is used to alleviate the problems of clamped texture coordinates and non power of two textures: the nVidia-specific **nv_texture_rectangle** extension. This extension allows for [0..width]x[0..height] texture coordinates and non power of two textures.

4.2 GLSL SHADERS

The application uses three pairs of vertex and fragment shaders. Each shader is developed in the **OpenGL Shader Language (GLSL)** [15]. Each shader pair carries out one of three basic operations on the data.

First, data is initialized into texture memory. Initial and boundary conditions are then set on a grid of data. The initial condition operation is applied once, at the beginning of program execution.

2 [0..1] means that the value is any number between 0 and 1.

Secondly, data is copied from the current solution into a temporary texture. This copying is necessary to read and write to texture memory using p-buffers. Data is copied by reading from a texture and writing to a pixel buffer.

Finally, an iteration of the current solution is computed. The discretized Laplacian operator (∇^2 , see equation (1)) is applied with the FTCS stencil (Figure 1). Data from the previous time step is read from a texture and used to update to a pixel buffer, which stores the discretized grid of data.

Using nv_texture_rectangle in a fragment shader requires a nVidia-specific **sampler** and texture access function. The fragment shader must have a uniform sampler2DRect to access the memory in which texture data is stored. Texture data is accessed with the texture2DRect(...) function. The nVidia GLSL Release Notes [14] document nv_texture_rectangle access in a GLSL fragment shader.

4.3 DISPLAY

Optionally, a fourth vertex/fragment shader pair can be used to display data as a solution to the diffusion equation is calculated for each iteration. The display vertex and fragment shader pair takes a texture from a p-buffer as input. Data at each point is compared to a ramp function[3] within the shader and drawn to the window.

For each iteration of display to the user, OpenGL must undergo a context switch. In other words, rather than writing to a p-buffer, data is drawn directly to the window. The OpenGL state is modified to draw to window, rather than utilizing off-screen rendering with p-buffers. Displaying data is a time-intensive operation due to this context switching for each time the data is drawn directly to window.

5.0 DISCUSSION

Comparison software has been developed at Indiana University in C utilizing serial and parallel computing [9]. Runtime of the serial and GPU implementations are comparable. The GPU implementation is much faster than the parallel implementation. This is most likely due to the simple nature of the diffusion equation. A more advanced problem may run much more efficiently on a massively parallel system.

Programmable graphics hardware show promise in scientific computing. Price and performance are currently driven by the rapidly expanding video game industry. Our work suggests that commodity graphics cards are a cost-effective co-processor.

Setting up a basic framework to access programmable capabilities of graphics hardware is time consuming. However, once such a framework has been set up, one can easily implement a variety of numerical computations by developing different GLSL shaders. Our system has a set of eight total shaders (4 vertex and 4 fragment shaders) consisting of less than 100 lines of high-level shader code total. The number of lines of code are comparable to serial and parallel implementations of the diffusion equation. For more information on the serial and parallel implementation, refer to [9].

3 Ramp function maps a color to a numerical value in a smooth, continuous manner.

5.1 SUGGESTIONS

Graphics card companies such as ATI and nVidia could develop graphics accelerators geared towards scientific computing. [4] suggests that memory bandwidth is the largest bottleneck in graphics hardware. A larger memory capacity – although not very useful for today's video games – may provide for larger and more complicated simulations.

5.2 FUTURE WORK

A research project at Stanford University, BrookGPU [1], may help the scientific community make more use of graphics hardware as a coprocessor. This C-style coding style may make it easier for scientists to port their applications to a GPU. Developing a solution to the diffusion equation in BrookGPU may help us determine whether or not GPUs coprocessors are feasible in the scientific community.

We would like to implement a more advanced mathematical model of the diffusion equation. Such as an implementation with **heat sources** and **sinks**, more interesting initial and boundary conditions, and/or implementing specific heat across a surface being modeled. A more complicated model may provide for a more accurate real-world example.

GPU hardware and programmable shaders is a rapidly growing market in the computer graphics industry. For instance, GLSL is only one year old. According to our literature, the first uses of GPUs for scientific computing dates back to 2000. It would be interesting to execute numerical simulations using the latest techniques available over a set of years. This may help form a trend on the viability of the use of graphics hardware to aid in numerical simulations.

6.0 ACKNOWLEDGMENTS

I thank Dr. John McDonald for his knowledge and patience which motivated my interest in using a GPU as a coprocessor for computation of the diffusion equation. I would also like to extend special thanks to Dr. Goedde for introducing me to differential equations and Mike Dvorak for support in developing a parallel solution to the diffusion equation with a high-performance Linux cluster.

7.0 REFERENCES

[1] "BrookGPU." Stanford University Computer Graphics Department. *http://graphics.stanford.edu/projects/brookgpu/*. Accessed March 1, 2005.

[2] C. Donner, H. Jensen. "Faster GPU Computations Using Adaptive Refinement" SIGGRAPH 2004

[3] GeForce 6 Series *http://www.nvidia.com/page/geforce6.html* Accessed 15 March 2005.

[4] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, G. Humphreys. "A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware"

Eurographics 2003.

[5] M. Harris, G. Coombe, T. Scheuermann, A. Lastra. "Physically-Based Visual Simulation on Graphics Hardware" Eurographics 2002.

[6] M. Harris. GPGPU: General-Purpose Computation Using Graphics Hardware

[7] M. Keane. "A Very Applied First Course in Partial Differential Equations" 2002, Prentice Hall, Upper Saddle River, NJ.

[8] W. Kernighan, D. Ritchie. "The C Programing Language." 1988 Prentice Hall, Englewood Cliffs, NJ.

[9] MPI – Diffusion Equation Example, Indiana University Research and Academic Computing Center *http://www.iu.edu/~rac/hpc/mpi_tutorial/9-0.html* Accessed 15 Mar 2005.

[10] "OpenGL Extension Registry: nv_float_buffer" *http://oss.sgi.com/projects/ogl-sample/registry/NV/float_buffer.txt*. Accessed 1 March 2005

[11] "OpenGL Extension Registry: nv_texture_rectangle" *http://oss.sgi.com/projects/ogl-sample/registry/NV/texture_rectangle.txt*. Accessed 1 March 2005

[12] Pixel Buffers, OpenGL.org *http://www.opengl.org/resources/tutorials/sig99/advanced99/notes/node213.html*. Accessed 15 Mar 2005.

[13] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. Numerical Recipes in C, pages 834-837. Cambridge University Press, 1988.

[14] "GLSL Release Notes for Release 60." Nvidia website. *http://download.nvidia.com/developer/GLSL/GLSL%20Release%20Notes%20for%20Release%2060.pdf*. 30 Aug 2004. Accessed 24 Feb. 2005

[15] R. Rost, J. Kessenich, B. Lichtenbelt. "OpenGL Shading Language" 2004 Addison-Wesley, Boston, MA.

[16] M. Rumpf, R. Strzodka. "Nonlinear Diffusion in Graphics Hardware" Proceedings of EG/IEEE TCVG Symposium on Visualization 2001.

[17] Eric W. Weisstein. "Relaxation Methods." From MathWorld--A Wolfram Web Resource. *http://mathworld.wolfram.com/RelaxationMethods.html* Accessed 21 Feb 2005

[18] Chris Wynn. "Using P-Buffers for Off-Screen Rendering in OpenGL" *http://developer.nvidia.com/attach/6534*. Accessed 20 Feb 2005.

[19] Chris Wynn. "Render To Texture" *http://developer.nvidia.com/attach/6725*. Accessed 20 Feb 2005.