A Problem-Based Learning Approach to GPU Computing

Robert Geist
School of Computing
Clemson University
Clemson, SC, USA
29634-0974
geist@clemson.edu

Joshua A. Levine
School of Computing
Clemson University
Clemson, SC, USA
29634-0974
levinej@clemson.edu

James Westall
School of Computing
Clemson University
Clemson, SC, USA
29634-0974
westall@clemson.edu

ABSTRACT

Compared to CPUs, modern GPUs exhibit a high ratio of computing performance per watt, and so current supercomputer designs often include multiple racks of GPUs in order to achieve high teraflop counts at minimal energy cost. GPU programming is thus becoming increasingly important, and yet it remains a challenging task. This paper describes a course in GPU programming for senior undergraduates and first-year graduates that has been taught at Clemson University annually since 2010. The course uses problem-based learning, with focus on a large, real-world problem, in particular, a system for parallel solution of partial differential equations. Although the system for solving PDEs is useful in its own right, the problem is used as a vehicle in which to explore design issues that face those attempting to achieve new levels of performance on SIMD architectures.

Categories and Subject Descriptors

C.1.2 [**Multiple Data Stream Architectures**]: Single-instruction-stream, multiple-data-stream processors (SIMD); K.3.2 [**Computer and Information Science Education**]: Computer science education

General Terms

Algorithms, Languages

Keywords

GPU, problem-based learning, lattice-Boltzmann, τέχνη

1. INTRODUCTION

Graphics Processing Unit (GPU) was a term introduced in the late 1990s by NVIDIA when "VGA controller" was deemed an inadequate description of the graphics hardware found in the typical desktop PC [12]. Since that time, GPU designs have undergone a series of dramatic transformations that have collectively produced the spectacular graphics capability that is now available for the PC at very low cost. In 2001, user programmability of part of the graphics pipeline first appeared in NVIDIA's GeForce3 and ATI's Radeon 8500. This programmability was restricted to a particular sub-engine of the GPU called the vertex processor, but in 2002, programmability was extended to the sub-engine called the fragment processor, which gave programmers per-pixel control. Researchers soon realized that these highly-parallel processors could be effectively employed in solving non-graphics problems, and a community of developers interested in general-purpose computation on graphics processing units (GPGPU) quickly emerged (www.gpgpu.org). The principal motivation was (and is) floating point performance. As an example, the NVIDIA Titan X GPU has a theoretical peak of 7 teraflops, which is more than an order of magnitude higher than the theoretical peak of a high-end CPU, the Intel Haswell E5-2699V3.

Energy consumption is also a motivating factor. It is well understood that significant new results in computational science will require simulations of systems that exceed the size of most current systems by several orders of magnitude, which in turn will require orders of magnitude increases in computational power. Peter Kogge, recipient of the IEEE CS 2012 Seymour Cray Award and 2014 Charles Babbage Award, phrased it thus: "You could build a computer that runs 100 times faster than Cray's 1-petaflop Blue Waters machine, but you'd need a good-sized nuclear power plant next door, not to mention a huge, dedicated cooling system." GPUs provide not only strong computational performance, but high performance per watt. The Titan Cray XK7 supercomputer at Oak Ridge, which is the top-ranked supercomputer in the U.S., is based on NVIDIA GPUs. It provides a theoretical peak of 27 petaflops, and yet it is also ranked in the top 100 on the Green 500 list of the most energy-efficient supercomputers in the world.

Nevertheless, extracting full performance from a GPU is hardly straightforward. Parallel algorithms are necessary but far from sufficient. Careful layout of both control flow patterns and memory access patterns is required to avoid flow divergence and bank conflicts, which can severely stall computational threads. Memory hierarchies, memory staging techniques, and the available synchronization primitives must be thoroughly understood.

Teaching GPU programming is thus a challenging task and yet one that is becoming increasingly important. The purpose of this paper is to describe the approach taken in CPSC 4780/6780 at Clemson University. This course is aimed at senior undergraduates and beginning graduate students who have had calculus, linear algebra, data structures, and significant experience in developing non-trivial code in C and C++. It has been taught each year since 2010.

2. THE τέχνη METHOD

The τέχνη method [6, 8, 9, 11] was developed at Clemson University and has been in use for more than a decade. The method is

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
EduHPC2015 November 15-20, 2015, Austin, TX USA
Copyright 2015 ACM. ISBN 978-1-4503-3961-2/15/11 $15.00
DOI: http://dx.doi.org/10.1145/2831425.2833197

built on a foundation of cognitive constructivism and draws directly from Piaget, Dewey, and Rousseau in its basic tenets:

- Learning is an active process of constructing individual knowledge.
- Learning occurs when observations differ from expectations, and new models must be constructed as accommodations.
- Teaching is the process of invoking and supporting these constructions.

Nevertheless, this foundation alone is an insufficient specification of method, and thus we specify pillars, upon which courses should be designed and taught:

- Problem-based Learning.
  Problem-based learning is well-known [10], and its use is wide-spread. Carefully designed problems demand that learners acquire self-directed learning strategies, critical knowledge, and the problem-solving proficiency needed for effective progress in deriving solutions. The τέχνη method differs from other problem-based learning methods in the size and scope of the problems addressed. It calls for one, large-scale problem per semester. Nevertheless, the single problem is always presented in several phases so that solution components may be independently developed and tested.
- Visual Problem Domain.
  Problems should have some tangible connection with computer graphics and/or visualization. Society is increasingly visually-oriented, and visual problems quickly capture the attention and interest of young students. In his discussions on visual communication, Cunningham [7] observed that the tools of thought for effective problem solving, as well the attendant communication skills for group efforts, were extremely well-supported by computer graphics.
- Cognitive Apprenticeship and Cognitive Authenticity.
  Resnick [23] observed that the time-honored master-apprentice relationship could be transferred from the arena of physical skills to that of cognitive skills, given an appropriately designed learning environment. Process is the key, rather than any artificial balance in the roles of master and apprentice. There is great value in observation of a master at work. Original solutions to challenging problems, including missteps and erasures, carry a vitality that is missing in solutions that have been cleaned and polished for presentation in more formal settings. Such vitality will only appear if the cognitive demands on the master are authentic. Thus the ideal, semester-long problems are those that challenge both student and instructor. The students then learn of the tools available and develop the skills to use them within a context of purpose.

The foundation and the pillars together offer a sufficiently tight specification of method under which new, τέχνη-compliant courses may be designed. Multiple published reports (see [6, 8, 9, 11]) attest to the efficacy of the method in delivering the two principal accommodations: an improved ability to solve real-world computational problems and a more positive attitude toward computing as a discipline. Problem selection is, of course, the key.

3. PROBLEM SELECTION

Students enrolled in the course come from a wide range of degree programs across the science and engineering disciplines. To describe basic syntax and provide elementary visualization capabilities, toy examples in CUDA, OpenCL, and OpenGL are provided. Nevertheless, for the principal problem of the course, we seek a substantial challenge that will be of broad interest. The current problem for the course is the development of a method for parallel solution of partial differential equations (PDEs).

Although not all students have had a course in differential equations, we find this to be no drawback whatsoever. Differential equations and their importance in modeling and simulation are easily described to students with calculus background in less than one lecture. The development then proceeds through solutions of classes of problems of increasing spatial dimension.

Lattice Boltzmann (LB) methods are an alternative to finite element methods (FEMs) that are particularly well-suited to this task. They have provided significant successes in modeling fluid flows and associated transport phenomena [5, 16, 17, 24, 25]. They provide stability, accuracy, and computational efficiency comparable to FEMs, but they offer significant advantages in ease of implementation, parallelization, and an ability to handle interfacial dynamics and complex boundaries. The principal drawback to the methods, compared to FEMs, is the counter-intuitive direction of the derivation they require. Differential equations describing the macroscopic system behavior are derived (emerge) from a postulated computational update, rather than the reverse. In the classroom setting, this provides a natural division of labor in which the instructor may focus principally on method derivations, and the students may focus principally on method implementations. Sample implementations of the solutions presented here are available on the first author's website. Those for the 1D and 2D methods are in OpenCL, and that for the 3D method is in CUDA. We now sketch the derivations of a 1D, a 2D, and a 3D method.

3.1 A 1-Dimensional Diffusion Model

Heat flow in a 1-dimensional rod is described by the so-called *diffusion equation*. If $\rho(x,t)$ denotes heat energy (temperature × constant heat capacity) at site x at time t, then the change in energy is described by

$$\frac{\partial\rho(x,t)}{\partial t} = D\frac{\partial^2\rho(x,t)}{\partial x^2} \tag{1}$$

where D is called the diffusion constant. It characterizes the thermal conductivity of the material in the rod.

Assume we have a 1-dimensional lattice with spacing λ and that we will conduct synchronous updates to lattice values with time step τ. Let $f_\pm(x,t)$ denote the energy at lattice site x, time t, flowing in direction ± 1, and assume, perhaps due to site collisions, that at each time step a fraction σ of $f_\pm(x,t)$ continues in the current direction, and the remainder reverses direction. The postulated fundamental update is thus

$$\begin{pmatrix} f_+(x+\lambda,t+\tau) \\ f_-(x-\lambda,t+\tau) \end{pmatrix} = \begin{pmatrix} \sigma & 1-\sigma \\ 1-\sigma & \sigma \end{pmatrix} \begin{pmatrix} f_+(x,t) \\ f_-(x,t) \end{pmatrix}$$

It will be most convenient to write this in incremental form,

$$\begin{pmatrix} f_+(x+\lambda,t+\tau) - f_+(x,t) \\ f_-(x-\lambda,t+\tau) - f_-(x,t) \end{pmatrix} = \Omega \begin{pmatrix} f_+(x,t) \\ f_-(x,t) \end{pmatrix} \tag{2}$$

where $\Omega = \begin{pmatrix} \sigma-1 & 1-\sigma \\ 1-\sigma & \sigma-1 \end{pmatrix}$.

Our real interest is the macroscopic behavior of total site energy, $\rho(x,t) = f_+(x,t) + f_-(x,t)$, as lattice spacing and time step approach zero. We assume that τ approaches 0 faster than λ. Specifically, we write $\lambda = \varepsilon\lambda_0$ and $\tau = \varepsilon^2\tau_0$ for any small $\varepsilon > 0$.

Our final assumption is that heat flow can be written as a small perturbation about a local equilibrium, in particular, $f_\pm = f_\pm^0 +$

$\varepsilon f^1_\pm + \varepsilon^2 f^2_\pm + ...$ where $f^0_+ + f^0_- = \rho$, and ε is the mean free path between collisions (Knudsen number). This is the so-called Chapman-Enskog expansion from statistical mechanics [5].

If we now apply a (two variable) Taylor expansion to the left side of (2) we obtain:

$$\begin{pmatrix} \lambda\frac{\partial f_+}{\partial x} + \tau\frac{\partial f_+}{\partial t} + (\lambda^2/2)\frac{\partial^2 f_+}{\partial x^2} + \lambda\tau\frac{\partial^2 f_+}{\partial x\partial t} + ... \\ -\lambda\frac{\partial f_-}{\partial x} + \tau\frac{\partial f_-}{\partial t} + (\lambda^2/2)\frac{\partial^2 f_-}{\partial x^2} - \lambda\tau\frac{\partial^2 f_-}{\partial x\partial t} + ... \end{pmatrix} = \Omega \begin{pmatrix} f_+ \\ f_- \end{pmatrix} \tag{3}$$

The principal step in the derivation of the diffusion equation is now at hand: we substitute ε-based expressions for λ, τ, and $f_\pm$ into (3), and equate coefficients of like powers of ε. For coefficients of the constant term (ε^0) we obtain:

$$0 = \Omega \begin{pmatrix} f^0_+ \\ f^0_- \end{pmatrix} \tag{4}$$

Since Ω has eigenvalues 0 and $2\sigma - 2$ with eigenvectors $(1,1)$ and $(1,-1)$, we can conclude that $(f^0_+, f^0_-) = K(1,1)$. Further, since components sum to ρ, we must have

$$(f^0_+, f^0_-) = (\rho/2, \rho/2) \tag{5}$$

For coefficients of ε^1, we obtain:

$$\begin{pmatrix} \lambda_0\frac{\partial f^0_+}{\partial x} \\ -\lambda_0\frac{\partial f^0_-}{\partial x} \end{pmatrix} = \Omega \begin{pmatrix} f^1_+ \\ f^1_- \end{pmatrix} \tag{6}$$

From (5)

$$\begin{pmatrix} (\lambda_0/2)\frac{\partial \rho}{\partial x} \\ -(\lambda_0/2)\frac{\partial \rho}{\partial x} \end{pmatrix} = \Omega \begin{pmatrix} f^1_+ \\ f^1_- \end{pmatrix} \tag{7}$$

Although Ω cannot be inverted, the left side of (7) is a multiple of (1,-1), an eigenvector whose eigenvalue is $2\sigma - 2$. We conclude

$$(f^1_+, f^1_-) = (\frac{\lambda_0}{4\sigma-4}\frac{\partial\rho}{\partial x}, \frac{-\lambda_0}{4\sigma-4}\frac{\partial\rho}{\partial x}) \tag{8}$$

Finally, for the coefficients of ε^2 in (3) we obtain:

$$\begin{pmatrix} \tau_0\frac{\partial f^0_+}{\partial t} + \lambda_0\frac{\partial f^1_+}{\partial x} + (\lambda_0^2/2)\frac{\partial^2 f^0_+}{\partial x^2} \\ \tau_0\frac{\partial f^0_-}{\partial t} - \lambda_0\frac{\partial f^1_-}{\partial x} + (\lambda_0^2/2)\frac{\partial^2 f^0_-}{\partial x^2} \end{pmatrix} = \Omega \begin{pmatrix} f^2_+ \\ f^2_- \end{pmatrix} \tag{9}$$

Here we can rely on the fact that the column sums of Ω are zero. If we substitute expressions for $f^0_\pm$ (5) and $f^1_\pm$ (8) into (9) and sum, we arrive at

$$\frac{\partial\rho}{\partial t} - \left(\frac{\lambda_0^2}{\tau_0}\right)\left(\frac{\sigma}{2-2\sigma}\right)\frac{\partial^2\rho}{\partial x^2} = 0 \tag{10}$$

or, equivalently,

$$\frac{\partial\rho}{\partial t} = \left(\frac{\lambda^2}{\tau}\right)\left(\frac{\sigma}{2-2\sigma}\right)\frac{\partial^2\rho}{\partial x^2} \tag{11}$$

the 1-D diffusion equation, with diffusion coefficient

$$D = \left(\frac{\lambda^2}{\tau}\right)\left(\frac{\sigma}{2-2\sigma}\right)$$

Thus the desired flow behavior (1) can be effected by implementing a highly-parallel, computational update (2). The simplicity of the computation is seen in the OpenCL kernel, shown in Figure 1. There is one thread per lattice site. To achieve synchronous updates, two buffers of flow values at the lattice sites are maintained, and successive calls to the kernel alternate reading from one and writing to the other, as shown in the calling CPU code of Figure 2. In Figure 3 we show the results of running this code for a rod of length 2m, with a lattice spacing $\lambda = (1/1024)$m, time step $\tau = (\lambda^2/2)$s, $\sigma = 1/2$, and final time 0.25s. The initial conditions are given by a linear heat ramp that peaks at the center, as shown in red in Figure 3.

```
#define store(node,dir) (DIRECTIONS*(node)+(dir))

__kernel void update(__global double* from, __global double* to,
        __global int* dist, __constant double* omega)
{
unsigned int j = get_global_id(0);
int k, n, t, ci[DIRECTIONS] = {1, -1};
double new_energy;

for(k=0;k<DIRECTIONS;k++){
        t = j + ci[k];
        if(t<=0 || t>=LENGTH) continue;
        new_energy = from[store(j,k)];
        for(n=0;n<DIRECTIONS;n++){
                new_energy += omega[store(k,n)]*from[store(j,n)];
                }
        to[store(t,k)] = new_energy;
        }
}
```

Figure 1: Kernel code.

```
void run_updates()
{
size_t gws[1] = {LENGTH};
size_t lws[1] = {THREAD_BLOCK_SIZE};
int from = 0, to = 1;
double t;

for(t=0.0;t<FINAL_TIME;t+=tau,from = 1-from, to = 1-to){
        clSetKernelArg(update,0,sizeof(cl_mem),(void *)&flow[from]);
        clSetKernelArg(update,1,sizeof(cl_mem),(void *)&flow[to]);
        clEnqueueNDRangeKernel(q,update,1,NULL,gws,lws,0,0,NULL);
        }
clEnqueueReadBuffer(q,flow[to],CL_TRUE,0,LENGTH*sizeof(double),
        host_flow,0,NULL,NULL);
}
```

Figure 2: Calling code.

Before turning to higher dimensions, it is worth observing that the 1-dimensional diffusion equation is one of those relatively rare cases in which an analytic solution is available [3], and so we can check the accuracy of the lattice-Boltzmann method just presented. If $g(x)$ is the initial heat energy over the rod of length l, and

$$B_k = \frac{2}{l}\int_0^l g(x)\sin(\frac{k\pi x}{l})$$

then the general solution is

$$\rho(x,t) = \sum_{m=0}^{\infty} e^{-D(\frac{m\pi}{l})^2 t} B_m \sin(\frac{m\pi x}{l})$$

For the case at hand, we obtain

$$\rho(x,t) = \sum_{k=0}^{\infty}(-1)^k\frac{8}{((2k+1)\pi)^2}e^{-(((2k+1)\pi)^2 t/4)}\sin(\frac{(2k+1)\pi x}{2}). \tag{12}$$

With only 100 terms from this series, we find the results match those from the LB solution to 6 decimal places at all positions and all times.

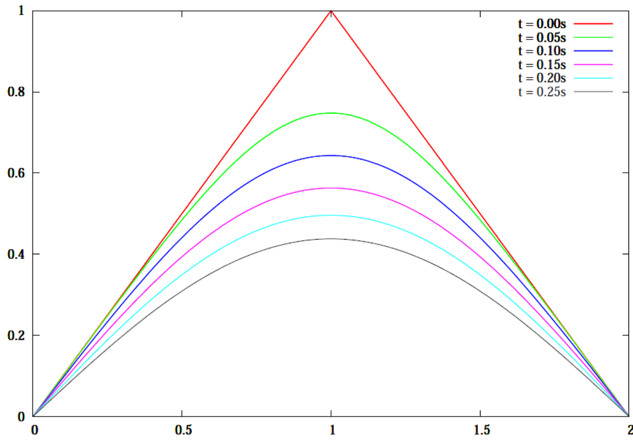

Figure 3: Heat Diffusion in a Rod.

3.2 A 2-Dimensional Wave Model

The movement of waves on the surface of a body of water is described by the wave equation,

$$\partial^2 h(\mathbf{r},t)/\partial t^2 = c^2 \nabla^2 h(\mathbf{r},t) \tag{13}$$

where $h(\mathbf{r},t)$ denotes the height of the wave at site $\mathbf{r} = (r_x, r_y)$ at time t, c is the wave speed (phase velocity), and $\nabla = (\partial/\partial x, \partial/\partial y)$, is the vector differential operator.

Although water waves are 3-dimensional objects, we can restrict development to 2D by considering flows to be displacements from a mean level. In this case, we can use a rectangular grid with four, unit-length directions, $\mathbf{c_i}$, $i = 1,\ldots,4$, and a single zero-length direction, $\mathbf{c_0}$, as shown in Figure 4. Although 2D, rectangular grids can generate anisotropic flows for certain LB models, we will see that anisotropy is avoided here through a careful choice of the site update matrix, Ω.

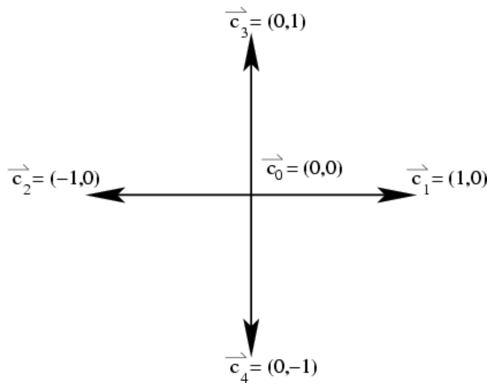

Figure 4: Model grid.

We assume a lattice spacing, λ, a time step, τ, unit velocity $v = (\lambda/\tau)$, and velocity vectors $\mathbf{v_i} = v\mathbf{c_i}$, $i = 0,\ldots,4$. We further assume that $h(\mathbf{r},t)$, the wave height at site $\mathbf{r}$ and time t, comprises 5 directional flows,

$$h(\mathbf{r},t) = \sum_{i=0}^{4} f_i(\mathbf{r},t) \tag{14}$$

where $f_i(\mathbf{r},t)$, represents the mass flow at location $\mathbf{r}$ at time t moving in direction $\mathbf{c_i}$. The velocity field is then

$$\mathbf{u}(\mathbf{r},t) = (\sum_{i=0}^{4} \mathbf{v_i} f_i(\mathbf{r},t))/(\sum_{i=0}^{4} f_i(\mathbf{r},t)) \tag{15}$$

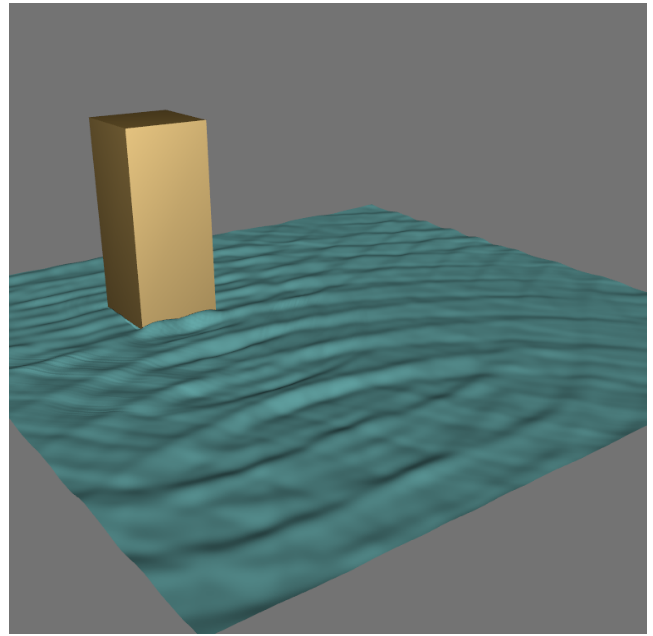

Figure 5: Wave Model Visualization.

The fundamental system update equation (basis for simulation) is given by:

$$f_i(\mathbf{r}+\lambda\mathbf{c_i}, t+\tau) = f_i(\mathbf{r},t) + \Omega_i \cdot f(\mathbf{r},t), \qquad \text{i = 0,1, ..., 4} \tag{16}$$

where Ω_i is the ith row of a matrix $\Omega : \Re^5 \to \Re^5$, which is a *collision matrix* in the sense that $\Omega_{i,j}$ represents the deflection of flow f_j into the ith direction. Once Ω is specified, equation (16) is, essentially, the entire computational model. Starting with initial conditions, we apply (16) synchronously to all lattice sites and then generate the new wave height field at time $t = t + \tau$ by (14).

Given a choice for Ω, students can begin implementation immediately. A typical result is shown in Figure 5, a frame from an animation where the wave surface is rendered as an OpenGL quadrilateral mesh.

The boundary conditions are periodic (toroidal) and reflection from the obstacle is achieved by simply bouncing back any flow that would otherwise enter the obstacle.

Per the τέχνη method, the visualization component is an important part of the project. To include the visualization with OpenGL, four kernels are required:

1. The *update* kernel is entirely analogous to that in the heat diffusion model. It uses the collision matrix to update site directional flows and distribute these to neighbor sites, as shown in (16).

2. The *heights* kernel sums the directional flows at each site. Waves on water surfaces are most often composites of waves moving at differing speeds, and so all directional flows for waves of all speeds at the site are included.

3. The *normals* kernel computes an approximate surface normal at each site using the results of the heights kernel at the site and at each neighbor.

4. The *colors* kernel uses the computed normal at each site, together with a light direction and a view direction to compute a color per site.

The heights and the colors can be written directly to a buffer object in GPU memory that is shared between OpenCL and OpenGL so that the entire surface can be rendered with a single (CPU) call to the OpenGL function *glDrawElements* that avoids transferring any information back to main memory.

Clearly, the choice of Ω determines the properties of the system. Some important constraints on this choice can be specified immediately. From (16) we have:

- conservation of mass: $\sum_{i=0}^{4} \Omega_i \cdot f(\mathbf{r},t) = 0$
- conservation of momentum: $\sum_{i=0}^{4} \mathbf{v_i}\Omega_i \cdot f(\mathbf{r},t) = (0,0)$

The principal constraint is that the limiting behavior of (16) as $\lambda, \tau \rightarrow 0$ should be a recognizable wave equation.

We choose to specify

$$\Omega = \begin{pmatrix} -4K & 2-4K & 2-4K & 2-4K & 2-4K \\ K & K-1 & K-1 & K & K \\ K & K-1 & K-1 & K & K \\ K & K & K & K-1 & K-1 \\ K & K & K & K-1 & K-1 \end{pmatrix} \quad (17)$$

where $K \in (0, 1/2]$ is a parameter. This choice ultimately yields a limiting wave equation with speed (phase velocity) $v\sqrt{K}$. The key to this choice, as we will show, is that 0 is a triple eigenvalue of Ω and that the eigenvectors $e_0 = (2-4K, K, K, K, K)$, $e_1 = (0, 1, -1, 0, 0)$, and $e_2 = (0, 0, 0, 1, -1)$ span the null space.

The derivation of wave equation from (16) is completely independent of the implementation of the code for its numerical solution. Nevertheless, it is essential science, and so the derivation is presented in the class, and the students are quizzed on components thereof.

We begin with a standard Chapman-Enskog expansion [5]. If we apply a Taylor expansion to the basic update equation (16) we obtain:

$$[(\lambda \mathbf{c_i}, \tau) \cdot \nabla] f_i(\mathbf{r},t) + \frac{[(\lambda \mathbf{c_i}, \tau) \cdot \nabla]^2}{2!} f_i(\mathbf{r},t) + \ldots = \Omega_i \cdot f(\mathbf{r},t) \quad (18)$$

As noted, we want to consider the limiting behavior here as $\lambda, \tau \rightarrow 0$; they can, of course, approach at different rates, but we assume they do not. Specifically, we write

$$t = \frac{s}{\varepsilon} \qquad \text{where} \quad s = o(\varepsilon)$$

$$\mathbf{r} = \frac{\mathbf{q}}{\varepsilon} \qquad \text{where} \quad \mathbf{q} = o(\varepsilon)$$

and where the limit of interest is $\varepsilon \rightarrow 0$. Then

$$\frac{\partial}{\partial t} = \varepsilon \frac{\partial}{\partial s}$$

$$\frac{\partial}{\partial r_\alpha} = \varepsilon \frac{\partial}{\partial q_\alpha} \qquad \text{for} \quad \alpha \in \{x, y\}$$

So

$$\nabla = (\partial/\partial r_x, \partial/\partial r_y, \partial/\partial t) = \varepsilon(\partial/\partial q_x, \partial/\partial q_y, \partial/\partial s) \quad (19)$$

We also assume that the solution, $f(\mathbf{r},t)$, is a small perturbation on this same scale about some local equilibrium, i.e.,

$$f(\mathbf{r},t) = f^0(\mathbf{r},t) + \varepsilon f^1(\mathbf{r},t) + \varepsilon^2 f^2(\mathbf{r},t) + \ldots \quad (20)$$

To qualify as a local equilibrium, f^0 must carry the macroscopic quantities of interest, that is,

$$h(\mathbf{r},t) = \sum_{i=0}^{4} f_i^0(\mathbf{r},t) \quad (21)$$

and

$$\mathbf{u}(\mathbf{r},t) = (\sum_{i=0}^{4} \mathbf{v_i} f_i^0(\mathbf{r},t)) / (\sum_{i=0}^{4} f_i^0(\mathbf{r},t)) \quad (22)$$

For the chosen Ω, these two conditions uniquely determine f^0. Since f^0 is an equilibrium, it is in the null space of Ω, and so we can write $f^0 = Ae_0 + Be_1 + Ce_2$. Then (21) and (22) together provide 3 independent equations in A, B, and C. The result is:

$$f_i^0(\mathbf{r},t) = \begin{cases} h(\mathbf{r},t)(1-2K) & i = 0 \\ (h(\mathbf{r},t)/2)\left[K + (\mathbf{v_i} \cdot \mathbf{u}(\mathbf{r},t))/v^2\right] & i = 1,2,3,4 \end{cases} \quad (23)$$

We now insert (19) and (20) into (18), and then sum (18) over $i = 0, 1, \ldots, 4$, divide by τ, and equate coefficients of ε^1. We obtain

$$\left(\frac{\partial}{\partial q_x}, \frac{\partial}{\partial q_y}\right) \cdot \sum_{i=0}^{4} \mathbf{v_i} f_i^0(\mathbf{r},t) + \frac{\partial}{\partial s} \sum_{i=0}^{4} f_i^0(\mathbf{r},t) = 0 \quad (24)$$

and so, after multiplying by ε,

$$\partial h(\mathbf{r},t)/\partial t + \nabla_{\mathbf{r}} \cdot [h(\mathbf{r},t)\mathbf{u}(\mathbf{r},t)] = 0 \quad (25)$$

This is an important result in its own right; it is called the *continuity equation*.

Next, if we multiply (18) by $\mathbf{v_i} = (v_{ix}, v_{iy})$, sum over $i = 0, 1, \ldots, 4$, divide by τ, and again equate coefficients of ε^1, we obtain a pair of equations for $\alpha \in \{\text{x,y}\}$:

$$\frac{\partial}{\partial s} \sum_{i=0}^{4} v_{i\alpha} f_i^0(\mathbf{r},t) + \frac{\partial}{\partial q_x} \sum_{i=0}^{4} v_{i\alpha} v_{ix} f_i^0(\mathbf{r},t) + \frac{\partial}{\partial q_y} \sum_{i=0}^{4} v_{i\alpha} v_{iy} f_i^0(\mathbf{r},t) = 0 \quad (26)$$

where the right hand side vanishes due to conservation of momentum. This pair can be expressed as

$$\frac{\partial}{\partial t}[h(\mathbf{r},t)\mathbf{u}(\mathbf{r},t))] + \nabla_{\mathbf{r}} \cdot \Pi^0(\mathbf{r},t)) = \mathbf{0} \quad (27)$$

where

$$\Pi^0_{\alpha\beta} = \sum_{i=0}^{4} v_{i\alpha} v_{i\beta} f_i^0(\mathbf{r},t), \quad (28)$$

for $\alpha, \beta \in \{\text{x,y}\}$, is called the *momentum tensor*. Note that for the limited set of directions we use, $v_{i\alpha} v_{i\beta} = v^2 \delta_{\alpha\beta}$, and so Π^0 is diagonal. The explicit expression for f^0 in (23) allows an important simplification. We have $\Pi^0_{xx} = \Pi^0_{yy} = Kv^2 h(\mathbf{r},t)$, and so

$$\frac{\partial}{\partial t}[h(\mathbf{r},t)\mathbf{u}(\mathbf{r},t))] + Kv^2 \nabla_{\mathbf{r}} h(\mathbf{r},t) = \mathbf{0} \quad (29)$$

This is called the *Euler equation of hydrodynamics*.

Finally, if we differentiate (25) with respect to t, differentiate (29) with respect to $\mathbf{r}$, and subtract, we obtain

$$\partial^2 h(\mathbf{r},t)/\partial t^2 - Kv^2 \nabla^2_{\mathbf{r}} h(\mathbf{r},t) = \mathbf{0} \quad (30)$$

the classical wave equation with wave speed $v\sqrt{K}$.

This model offers a chance to illustrate simultaneous thread-level and vector-level parallelism. As noted earlier, real water waves are a composite of waves traveling at different speeds that move through one another. The students are required to implement this model for at least eight simultaneous waves of different speeds.

```
#define to(i,j,k) to[((i)*(WIDTH*DIRECTIONS)+(k)*WIDTH+(j))]

__kernel void heights(__global float4* rbuffer, __global float8*
to)
{ int j = get_global_id(0);
int i = get_global_id(1);
int k, n;
float u, v, h;
float8 new_height;

new_height = (float8)(0.0f);
for(n=0;n<DIRECTIONS;n++) new_height += to(i,j,n);
h = new_height.s0 + new_height.s1 + new_height.s2 + new_height.s3 +
    new_height.s4 + new_height.s5 + new_height.s6 + new_height.s7;
u = SCALE*((j/(float)(WIDTH-1))-0.5f);
v = SCALE*((i/(float)(LENGTH-1))-0.5f);
rbuffer[i*WIDTH+j] = (float4)(u, h, v, 1.0f);
}
```

Figure 6: Heights kernel code.

The OpenCL language supports large vector operations, and so the collision matrix, Ω, and all the directional flows, $f_i(\mathbf{r},t)$, can be treated as vectors, with each vector component handling a different wave speed. The *heights* kernel for such an implementation might then resemble the code of Figure 6. Thus it first adds five vectors of dimension eight and then sums the vector components to yield the final wave height, which it stores in the render buffer along with the associated 2D lattice components.

For additional details on selecting wave speeds and targeting wave spectra, see [14].

3.3 A 3-Dimensional Airflow Model.

In higher dimensions, the proof that a relatively simple computational update has a given partial differential equation as its limit can be quite tedious, e.g., [15], but successes are numerous, e.g., [1, 5, 13, 18, 20, 19]. The most common target in three dimensions is the Navier-Stokes equation:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u} = -(1/\rho)\nabla p - \nu\nabla^2(\mathbf{u}) \qquad (31)$$

where $\mathbf{u}$ denotes fluid velocity, p is pressure, and ν is viscosity. Multiple flow directions on an LB lattice are required, and the most common choice is probably the 27 lattice points in a cube of unit (λ) radius, for which the fundamental update becomes

$$f_i(\mathbf{x}+\lambda\mathbf{c}_i, t+\tau) = f_i(\mathbf{x},t) + \Omega_i(f(\mathbf{x},t)), \qquad \text{i} = 0,1,\ldots,26 \quad (32)$$

where Ω_i is a local collision operator that describes scattering in each direction at each site, and $\mathbf{c}_i$ denotes the ith direction, which we enumerate as:

$$\mathbf{c}_i = \begin{cases} (0,0,0) & i=0 \\ (\pm1,0,0),(0,\pm1,0),(0,0,\pm1) & i=1,..,6 \\ (\pm1,\pm1,0),(\pm1,0,\pm1),(0,\pm1,\pm1) & i=7,..,18 \\ (\pm1,\pm1,\pm1) & i=19,..,26 \end{cases} \qquad (33)$$

As in the one dimensional case, $\rho(\mathbf{x},t) = \sum_i f_i(\mathbf{x},t)$, and per-site velocity is calculated as expected, $\mathbf{u}(\mathbf{x},t) = \sum_i v\mathbf{c}_i f_i(\mathbf{x},t)/\rho(\mathbf{x},t)$, where $v = \lambda/\tau$ is the link speed, which can be fixed at 1 for computational convenience.

An important simplification was suggested almost simultaneously in [4] and [22]: the collision operator, $\Omega_i(f(\mathbf{x},t))$ in (32), can be replaced by $-\omega(f_i(\mathbf{x},t) - f_i^0(\mathbf{x},t))$, where $f_i^0(\mathbf{x},t)$ represents the continuous Maxwell-Boltzmann equilibrium distribution and ω is a relaxation parameter that controls the speed at which solutions converge to the equilibrium. In the case of the 3D lattice with 27 flow

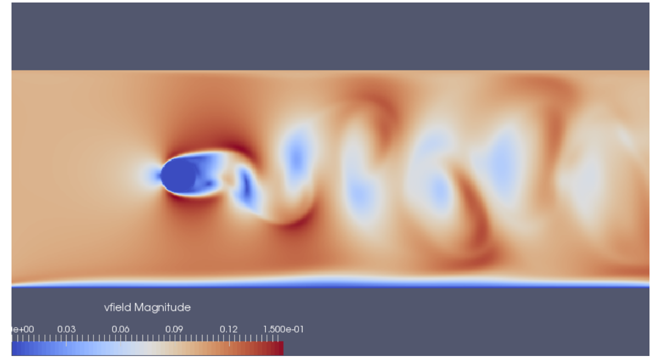

Figure 7: Velocity magnitude of airflow over a cylinder.

directions (called D3Q27), the equilibrium distribution is taken as a second-order, discrete approximation to the continuous Maxwell-Boltzmann distribution:

$$f_i^0(\mathbf{x},t) = \rho w_i \left[1 + \frac{\mathbf{c}_i\cdot\mathbf{u}}{3} + \frac{9(\mathbf{c}_i\cdot\mathbf{u})^2}{2} - \frac{3\mathbf{u}\cdot\mathbf{u}}{2}\right]$$

where the link weight, w_i, is $8/27, 2/27, 1/54, 1/216$ for links of length $0, 1, \sqrt{2}, \sqrt{3}$ lattice units.

Under this simplification, the computational update (32) still yields the Navier-Stokes equation (31) in the limit, but it offers both reduced computational effort and direct control over the resulting kinematic viscosity [5]:

$$\nu = \frac{1}{3}\left(\frac{1}{\omega} - \frac{1}{2}\right) \qquad (34)$$

Although the derivation of Navier-Stokes is not presented in class, students are encouraged to experiment numerically with 3D air flow over simple geometries using (32).

Suppose we use a lattice spacing $\lambda = 1$mm, a time step $\tau = 1$ms, and place a cylinder that measures 40mm in diameter in a 3D grid of dimension $800\times256\times256$. The viscosity of air at $300K^\circ$ is $15.68\times10^{-6}m^2/s$. If we express this in units of λ^2/τ, and solve (34), we obtain a relaxation parameter $\omega = 1.82802$. Using this value in (32), we can obtain a 3D velocity field of air flow over the cylinder.

The parameters have been chosen here to illustrate a particularly compelling visual phenomenon called the *von Karman vortex street*. In Figure 7 we show a single frame of a slice of this velocity field, and in Figure 8 we show the same image with line integral convolution (LIC) applied. Visualization of 3D vector fields, such as this one, is important, but it need not require extensive coding. Students can simply modify their LB flow code to periodically write out slices of the vector field in some standard format, such as VTK [21], and then use open source tools, such as Paraview [2] for the visualization. This was the method used to produce Figures 7 and 8. The vortices that are being shed behind the cylinder are evident in the figures.

4. DISCUSSION

Although the 1D and 2D problems presented earlier easily admit real-time solutions, even for large grids, the 3D air flow problem is computationally intensive. Thus, while the students are busy implementing a 2D wave model solution, classroom discussions focus on performance issues, including flow divergence, memory bank conflicts, coalescing memory accesses, effective use of shared (on-chip) memory, multi-processor occupancy, and available profiling

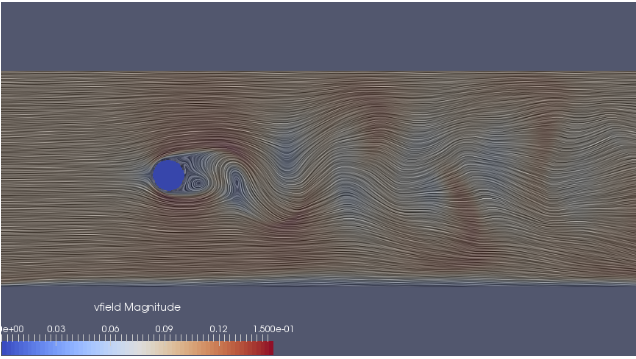

Figure 8: Velocity magnitude of airflow over a cylinder, augmented with line integral convolution.

tools. On NVIDIA cards, profiling is enabled by simply setting three environment variables:

```
export COMPUTE_PROFILE=1
export COMPUTE_PROFILE_LOG=myprofile_output
export COMPUTE_PROFILE_CONFIG=myprofile_config
```

The named output file will receive results, and the named configuration file is a text file listing desired outputs, such as GPU execution time, CPU execution time, multi-processor occupancy, registers used per thread, memory transfer sizes, and active *warps*. A *warp* is the minimum block of threads that will synchronously execute a single instruction in parallel.

Some dramatic performance effects are easily illustrated within the 3D air flow code. For instance, simply rearranging linear storage can have such effects. Although directional airflow storage on the card in the 3D case is most naturally aligned, effectively, as

$$flow[WIDTH][HEIGHT][DEPTH][DIRECTIONS] \tag{35}$$

a significantly preferable alignment is

$$flow[WIDTH][HEIGHT][DIRECTIONS][DEPTH] \tag{36}$$

On NVIDIA cards, when the threads in a *warp* access consecutive four-byte words in memory, and those words are cache-aligned with the 128-byte lines of the L1 cache, all 128 bytes are fetched in a single memory access. Each thread in the airflow code updates values at a single (i,j,k) lattice site and must access all 27 directional flows. Thus, with the standard alignment (35), where those directional flows occupy the consecutive memory words, threads (i,j,k) and $(i,j,k+1)$, which are likely to be grouped in a *warp*, will definitely not access consecutive words. With the preferred alignment (36), it is likely that they will.

The effect of such changes, and hence the importance of memory alignment, can be easily illustrated. In class, we execute the airflow code with both the standard alignment (35) and the preferred alignment (36) and capture the performance. The sample airflow code uses three kernels, here called *cascade*, *stream*, and *bounce*. The *cascade* kernel computes only the local update per lattice site, i.e., the right-hand side of (32). The *stream* kernel distributes these values to the neighboring nodes, i.e., completes the assignment in (32). The *bounce* kernel reflects airflow from the surface of the cylinder and from the floor of the wind tunnel. In Table 1 we show average execution times for these kernels under both alignments. Thus, a simple realignment of storage afforded us a 79% reduction in kernel execution time.

For other applications, modifications such as this can generate

kernel	standard alignment	preferred alignment
stream	2,300,410.48	284,842.17
cascade	569,319.16	316,735.24
bounce	3,834.98	1,459.72
total	2,873,564.62	603,037.13

Table 1: Average kernel execution time (microseconds)

orders of magnitude speedup in kernel execution, as seen in [17].

Finally, a significant limitation of current GPUs is the size of onboard, global memory. The previously mentioned NVIDIA Titan X, which by some measures is the top performing card, has only 12GB. Thus an important extension to our 3D airflow model includes a discussion of methods for extending the models across multiple cards, both within the same compute node, i.e., across the PCIe bus, and across multiple compute nodes via Infiniband. Across a PCIe bus, direct card-to-card memory access is invoked in CUDA with *cudaDeviceEnablePeerAccess*. After this call, the memory on both cards is directly addressable by either card. Across Infiniband nodes, remote DMA (RDMA) is available for direct transfer from card memory to card memory via *MPISend* and *MPIReceive*. Profiling tools quickly show the cost of such transfers. If the cost is significant, the transfers can usually be overlapped with computation.

5. LIMITATIONS

The proposed method for teaching GPU programming differs from more conventional approaches in which breadth of GPU feature coverage is emphasized at some expense in problem depth (e.g. CS 179, Cal. Tech.) or those where the emphasis is placed on acceleration of algorithms from computer graphics (e.g. CIS 565, U. Penn.). Our approach sacrifices some breadth of topic coverage, e.g., we do not cover CUDA streams, an important topic for overlapping data transfer with computation, and the visual artifacts from our course, though informative, cannot aesthetically match those from a course dedicated to computer graphics. Further, the proposed approach clearly requires that the instructor be well-versed in solving PDEs as well as having solid understanding of most of the features of current GPU architectures.

Finally, numerous side-by-side studies in which the τέχνη method has been compared to a more conventional approach [6, 8, 9, 11] have demonstrated the efficacy of the τέχνη method when applied to a variety of undergraduate courses, but no such study has yet been conducted for this course, which is delivered at a higher level than those in the studies cited. Such a side-by-side study remains as important future work.

6. CONCLUSIONS

We have described CPSC 4780/6780, a course in GPU programming that has been taught at Clemson University every year since 2010. GPU programming is becoming increasingly important due to the significant advantage GPUs provide, when compared to CPUs, in computing power per watt. Thus it is no surprise that the course, though strictly an elective, has seen increasing enrollment and now numbers approximately 30 students per year.

The course design uses problem-based learning. Toy problems are used to introduce syntax and features of the relevant languages, in this case OpenCL, OpenGL, and CUDA, but the principal focus is on a single, real-world problem. The current problem choice is the development of a system for numerical solution of partial differential equations. The lattice-Boltzmann method is used, due

to simplicity of design and ease of adaptation to the SIMD design of GPU architectures.

The problem solutions provide value in their own right, but the problem serves a larger purpose in that it is a vehicle by which we can convey, with tangible examples, the issues involved today in using GPUs to achieve high performance in solving real problems.

7. ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under Grant Nos. IIS-1314757 and CNS-1126344. The School of Computing at Clemson University is an NVIDIA GPU Research Center and an NVIDIA GPU Teaching Center. Thanks to David Leubke, Cliff Woolley, and Chandra Cheij of NVIDIA for their support in providing hardware and technical consultation.

8. REFERENCES

[1] E. Aharonov and D. Rothman. Non-newtonian flow through porous media: A lattice-boltzmann method. *Geophysical Research Letters*, 20:679–682, April 1993.

[2] U. Ayachit. *The Paraview Guide*. KitWare, Inc., January 2015.

[3] V. Celli. Phys 311 notes. http://galileo.phys.virginia.edu/classes/311/notes/dimension/node8.html, July 1997.

[4] S. Chen, H. Chen, D. Martinez, and W. Matthaeus. Lattice boltzmann model for simulation of magnetohydrodynamics. *Phys. Rev. Lett.*, 67:3776–3779, 1991.

[5] B. Chopard and M. Droz. *Cellular Automata Modeling of Physical Systems*. Cambridge Univ. Press, Cambridge, UK, 1998.

[6] C. Corsi, R. Geist, and D. Lingerfelt. A virtual graphics card for teaching device driver design. In *Proc. ACM SIGCSE Technical Symp. on Computer Science Education (SIGCSE 2014)*, pages 555–560, Atlanta, Georgia, March 2014.

[7] S. Cunningham. Graphical problem solving and visual communication in the beginning computer graphics course. *ACM SIGCSE Bulletin*, 34(1):181 – 185, 2002.

[8] T. Davis, R. Geist, S. Matzko, and J. Westall. τέχνη: A first step. *ACM SIGCSE Bulletin (Proc. ACM SIGCSE Annual Conf.)*, 36(1):125 – 129, 2004.

[9] T. Davis, R. Geist, S. Matzko, and J. Westall. τέχνη: Trial phase for the new curriculum. *ACM SIGCSE Bulletin (Proc. ACM SIGCSE Annual Conf.)*, 39(1):415 – 419, 2007.

[10] B. Duch, S. Gron, and D. Allen. *The Power of Problem-Based Learning*. Stylus Publishing, LLC, Sterling, VA, 2001.

[11] A. Duchowski, R. Geist, R. Schalkoff, and J. Westall. τέχνη trees: A new course in data structures. In *Proc. ACM SIGCSE Technical Symp. on Computer Science Education (SIGCSE 2011)*, pages 341–346, Dallas, Texas, March 2011.

[12] R. Fernando and M. Kilgard. *The Cg Tutorial*. Addison Wesley, Boston, MA, 2003.

[13] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the navier-stokes equation. *Physical Review Letters*, 56:1505–1508, April 1986.

[14] R. Geist, C. Corsi, J. Tessendorf, and J. Westall. Lattice-Boltzmann water waves. In G. B. et al., editor, *ISVC 2010 Part I, LNCS 6453, (Proc. 6th Int. Symp. on Visual Computing)*, pages 74–85, Las Vegas, Nevada, November 2010. Springer, Heidelberg.

[15] R. Geist, K. Rasche, J. Westall, and R. Schalkoff. Lattice-boltzmann lighting. In *Rendering Techniques 2004 (Proc. Eurographics Symposium on Rendering)*, pages 355 – 362,423, Norrkőping, Sweden, June 2004.

[16] R. Geist, J. Steele, and J. Westall. Convective clouds. In *Natural Phenomena 2007 (Proc. of the Eurographics Workshop on Natural Phenomena)*, pages 23 – 30, 83, and back cover, Prague, Czech Republic, September 2007.

[17] R. Geist and J. Westall. Lattice-Boltzmann lighting models. In W. mei Hwu, editor, *GPU Computing GEMS Emerald Ed.*, volume 1, chapter 25. Morgan Kaufmann, Feb. 2011.

[18] L. Giraud, D. d'Humieres, and P. Lallemand. A lattice boltzmann model for jeffreys viscoelastic fluid. *Europhysics Letters*, 42:625–630, June 1998.

[19] X. He, S. Chen, and G. Doolen. A novel thermal model for the lattice boltzmann method in incompressible limit. *Journal of Computational Physics*, 146:282–300, 1998.

[20] F. Higuera, S. Succi, and R. Benzi. Lattice gas dynamics with enhanced collisions. *Europhysics Letters*, 9:345–349, June 1989.

[21] KitWare. *The VTK User's Guide*. KitWare, Inc., March 2010.

[22] Y. Qian, D. d'HumiÃÍres, and P. Lallemand. Lattice bgk models for navier-stokes equation. *Europhysics Letters*, 17(6):479–484, 1992.

[23] L. B. Resnick. Learning in school and out. *Educational Researcher*, 16(9):13–20, 1987.

[24] N. Thürey, U. Rüde, and M. Stamminger. Animation of open water phenomena with coupled shallow water and free surface simulations. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 157–164, Vienna, Austria, 2006.

[25] X. Wei, W. Li, K. Mueller, and A. Kaufman. The lattice-boltzmann method for gaseous phenomena. *IEEE Transactions on Visualization and Computer Graphics*, 10(2), 2004.