To appear in The Visual Computer

Implementing Lattice Boltzmann Computation on Graphics Hardware

Wei Li, Xiaoming Wei, and Arie Kaufman

Center for Visual Computing (CVC) and
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

Abstract

LBM is a physically-based approach that simulates the microscopic movement of fluid particles by simple, identical and local rules. We accelerate the computation of the LBM on general-purpose graphics hardware, by grouping particle packets into 2D textures and mapping the Boltzmann equations completely to the rasterization and frame buffer operations. We apply stitching and packing to further improve the performance. In addition, we propose techniques, namely range scaling and range separation, that systematically transform variables into the range required by graphics hardware and thus prevent overflow. These approaches can be extended to a compiler that automatically translates general calculations to operations on graphics hardware.

Keywords: Graphics hardware, Lattice Boltzmann Method, flow simulation.

1 Introduction

Simulations of fluid behavior are in great demand in film making, as well as in visual simulations such as animation design, texture synthesis, flight simulation, and scientific visualization. In Computational Fluid Dynamics (CFD), fluid properties, such as density and velocity are typically described by the Navier-Stokes (NS) equations, which have nonlinear terms making them too expensive to solve numerically in real time. Instead of calculating the macroscopic equations, we can simulate the linear, microscopic LBM [1, 9, 11] to satisfy the NS equations. The fluid flow consists of many tiny flow particles and the collective behavior of these microscopic particles results in the macroscopic dynamics of a fluid. Since the macroscopic dynamics of fluid is insensitive to the underlying details in microscopic physics [**?**], this gives us the possibility of using a simplified microscopic kinetic-type model to simulate its movements.

The LBM can be understood as a Cellular Automata representing discrete packets moving on a discrete lattice at discrete time steps. The calculation is performed on a regular grid. At each grid cell, there are variables indicating the status of the grid point. All the cells modify their status at each time step based on linear and local rules. Although faster than other solutions to the NS equations, the computation of the LBM is still slow. The calculations at each point are simple, but there is usually a large amount of cells. Therefore, a practical use of the LBM typically demands parallel supercomputers [9, 11].

Commodity graphics hardware can perform pixel-oriented operations very efficiently. Not only the operations are pipelined in dedicated hardware, but there are usually up to four color channels and multiple pixel pipelines that essentially provide parallel processing. The speed of graphics hardware doubles approximately every six months which is much faster than the improving rate of CPU. Inspired by the performance of graphics hardware and the resemblance in the computation pattern between the LBM and the rasterization stage, we propose to accelerate the LBM on commodity graphics hardware by storing packets of the LBM as textures and translating the Boltzmann equations into rendering operations of the texture units and the frame buffer. We further apply stitching to reduce the overhead of texture switching and packing to reduce the memory requirement as well as to exploit parallelism of the four color channels. In addition, we present range scaling that transforms the range of arbitrary variables and any intermediate results to fulfill the requirement of graphics hardware. The techniques guarantee no overflow no matter how the variables are evaluated while the scaling factors are chosen to take the full precision of the hardware. We use LBM equations as an example to show how the scaling is applied and observe only up to 1% of error. In addition, the range separation proposed in this paper overcomes the non-negative limits required in certain stages of the graphics pipeline.

Although we focus on the LBM and its applications to visual simulation of fluids and smoke, our approach is extendable to any cellular-automata-typed calculations. We expect that the techniques proposed in the paper, such as range scaling and range separation, be developed into a compiler that automatically generates rendering instructions equivalent to various general computations with the range of all variables and intermediate and final results properly transformed.

The rest of the paper is organized as follows. First, we review related work. In Section 3 we present range scaling and range separation which are critical components of our approach in accelerating general computations on graphics hardware. In Sections 4 and 5, after a brief introduction of the theory of the LBM, we present our methods of mapping the LBM equations as multi-pass rasterization operations and how the range scaling is applied to the LBM. In Section 6, we give the experimental results.

2 Related Work

Graphics hardware has been extended to various applications beyond its originally-expected usage. Examples include matrix multiplication [10], 3D convolution [6], morphological operations such as dilation and erosion [7], computation of Voronoi diagrams and proximity queries [4, 5], voxelization [2], algebraic reconstruction [12], and volumetric deformation [16].

Graphics architecture, such as OpenGL, can be treated as a general SIMD computer [13]. Various computations are implemented as the operations of the texture mapping unit and the frame buffer. Final results are obtained in one or more rendering passes. Both Peercy et al. [13] and Proudfoot et al. [14] have developed languages for programmable procedural shading systems as well as compilers that automatically generate instructions corresponding

[1]Email:{liwei,wxiaomin,ari}@cs.sunysb.edu

to rendering operations on graphics hardware. However, to apply these ideas to other applications, the limited value range and accuracy of graphics hardware have to be considered. Trendall et al. [17] gave several formulas for scaled and biased functions whose value ranges are within the limits and applied the method to the computation of interactive caustics.

Some of the above applications scale and shift the variables in their computations so that they fit into the value range of the graphics hardware. Their scaling and shifting parameters are chosen either trivially or empirically. The range scaling proposed in this paper provides a systematic way for mapping general computations onto graphics hardware which guarantees that all the inputs and outputs as well as the intermediate results are not clamped by the hardware, in addition to exploiting the hardware precision as much as possible.

There are a few papers on accelerating flow visualization on graphics hardware. Heidrich et al. [3] exploit pixel texture to compute line integral convolution, which is a technique for visualizing vector data. Jobard et al. [8] translate texture advection computations to frame buffer operations to accelerate the visualization of the motion of 2D flows. Weiskopf et al. [19] extend Jobard et al.'s work to 3D flows. They also take advantage of the newly available OpenGL extensions, namely offset texture and dependent texture. All these techniques are for the visualization of fluid with given velocity fields. In contrast, this paper focuses on the simulation, specifically the generation of the fields, such as velocity, that are required for the visualization. By employing similar visualization techniques, our approach can map the whole simulation and visualization onto the graphics hardware without the need of transferring data back and force between the host memory and the graphics memory. Harris et al [**?**] implement coupled map lattice (CML), an variation of cellular automata, on graphics hardware, which has similar motivation and applications to this paper. We expect our work on LBM eventually result in a compiler of general computations to graphics hardware, therefore one of the focus of the paper is how to systematically mapping general equations to rendering operations considering the range and precision limitation of the graphics hardware. Besides, we propose techniques such as texture packing and stitching to further accelerate the execution on hardware.

3 General Computations on Graphics Hardware

We use the rasterization units (e.g., Nvidia's register combiners) and the frame buffer in graphics hardware to implement addition, subtraction and multiplication. Division and other more complicated calculations are replaced with lookup tables storing precomputed values. Input values are stored in textures and output values are either copied from the frame buffer to textures or written directly to the textures with the render-to-texture extension.

Values in graphics hardware are clamped to either $[0, 1]$ or $[-1, 1]$, depending on the stage of the graphics pipeline. Therefore, we need to transform the value ranges of all the inputs and outputs. Trendall et al. [17] also mapped the lower bound of the range to 0 by biasing for better accuracy. In contrast, we generally avoid introducing any bias during the mapping. (The reason is explained in section 3.2). That is, we only apply scaling to change the numerical ranges.

Range scaling can be considered as a simulation of floating point on fixed-point hardware, whereas floating point texture and frame buffer have been suggested for future hardware [15]. However, even if the support to floating point were available in the rasterization stage of the graphics hardware, it would be significantly slower and could take more texture memory than its fixed-point counter part. The range scaling proposed in this paper makes no assumption of the precision of the hardware.

We always prefer the rasterization units because they are more flexible than the frame buffer. However, distributing certain operations to the frame buffer reduces the number of rendering passes and the need of copying the frame buffer contents into textures.

3.1 Range Scaling

The scaling factors should be carefully selected so that no clamping error occurs and the computation exploits the full precision of the hardware. For any input or output function $f(x)$, we divide it with its maximal absolute value and obtain a scaled function $\widetilde{f(x)}$, such that:

$$f^{max}\widetilde{f(x)} = f(x) \tag{1}$$

where $f^{max} = \max_x(|f(x)|)$ which we refer to as the left-hand scalar. Obviously, $\widetilde{f(x)} \in [-1, 1]$. We then use $\widetilde{f(x)}$ throughout the hardware pipeline.

We should also make sure that during the computation of $\widetilde{f(x)}$, no intermediate result is clamped. We denote $U(f)$ as the maximal absolute value of all intermediate results during the evaluation of $f(x)$, no matter what computation order is taken if the computation contains multiple operations. It is easy to see that $U(f) \geq f^{max}$. If we multiply $1/U(f)$ on the right-hand-side before computing, we guarantee that no overflow occurs. We refer to $U(f)$ as the right-hand scalar.

If a function $s(x)$ is a weighted sum of several other functions, $s(x) = \sum_i k_i f_i(x)$, we compute the scaled function $\widetilde{s(x)}$ as follows:

$$\widetilde{s(x)} = \frac{U(s)}{s^{max}} \sum_i \frac{k_i f_i^{max}}{U(s)} \widetilde{f_i(x)} \tag{2}$$

The left-hand scalar and the upper bound of the intermediate values are computed as:

$$\begin{aligned} s^{max} &= \max(\sum_i k_i f_i^{max} (\exists x, f_i(x) > 0), \\ &\qquad \sum_i k_i f_i^{max} (\exists x, f_i(x) < 0)) \end{aligned} \tag{3}$$

$$\begin{aligned} U(s) &= \max(\sum_i k_i U(f_i) (\exists x, f_i(x) > 0), \\ &\qquad \sum_i k_i U(f_i) (\exists x, f_i(x) < 0)) \end{aligned} \tag{4}$$

That is, each $\widetilde{f_i(x)}$ is scaled by $k_i f_i^{max}/U(s)$ before summation. We know $|k_i f_i^{max}/U(s)| \leq 1$. The sum is then multiplied by a constant $U(s)/s^{max}$. Note that $U(s)/s^{max} \geq 1$. To multiply with a factor larger than 1, we have two choices: (1) utilizing the output scale mapping of the register combiners; (2) applying multiplication and addition or using dot product. s^{max} is obtained by summing the maximal values of positive and negative entries separately and selecting the sum with the larger absolute value. $U(s)$ is computed similarly. Since $s(x)$ may not reach the computed maximal value s^{max}, it is better to replace s^{max} with the actual largest absolute value $\widetilde{s^{max}}$ such that $\widetilde{s^{max}} < s^{max}$. To measure $\widetilde{s^{max}}$, we implement a software-only version of the same calculations on the CPU and feed into various input values.

If $p(x)$ is the product of several functions, $p(x) = k \sum_i f_i(x)$. We have

$$\widetilde{p(x)} = \frac{U(p)}{p^{max}} k \prod_i \frac{f_i^{max}}{U(f_i)} \widetilde{f_i(x)} \tag{5}$$

where

$$p^{max} = k \prod_i f_i^{max} \tag{6}$$

$$U(p) = k \prod_i U(f_i) \tag{7}$$

If the function $g(x)$ is an input for a rendering pass, we speculate that $U(g) = g^{max}$. Apparently, if all $f_i(x)$ are inputs, $p^{max} = U(p) = k \prod_i f_i^{max}$.

In each rendering pass, the hardware performs a mix of additions (subtractions) and multiplications [1]. We first obtain the absolute maximal values of inputs either by prior knowledge or measurement of software simulation on the CPU. Then, before evaluating a function $f(x)$, we compute its left-hand and right-hand scalars by grouping its right-hand side into sums and products and by recursively applying Equations 3, 4, 6 and 7. Next, we divide the right-hand side by $U(f)$ and distribute $U(f)$ to each input functions according to Equations 2 and 5. All the scale coefficients of the inputs are computed in software.

3.2 Range Separation

Conventional graphics hardware only supports a numerical range of [0,1]. Recent OpenGL extensions, such as Nvidia's register combiners, expand the range to [-1, 1] in the rasterization stage, whereas the final combiner and the frame buffer are still limited to [0, 1]. Hence, negative values have to be transformed before they are sent to the final combiner or the frame buffer. A solution is to apply bias and scaling to transform the range to [0, 1] before entering the final combiner stage and the frame buffer. However, it is then very difficult to utilize the final combiner or the frame buffer for multiplication or addition on the biased variables. Consequently, the number of rendering passes and the times of backing-up the contents of the frame buffer are likely to increase.

We propose to separate the positive and negative ranges and avoid biasing as an alternative solution. Similar to other approaches, we scale all the variables to fit into the range of [-1, 1] as the first step. If all the values of a variable are constantly non-negative or non-positive, it is trivial to map them to [0, 1].

To handle a variable containing both positive and negative values, we divide the range of [-1, 1] into two parts, [-1, 0] and [0, 1]. For arbitrary function f, we have:

$$f = [f]^+ + [f]^- = [f]^+ - [-f]^+ \tag{8}$$

where $[\,]^+$ and $[\,]^-$ denote the clamping to [0, 1] and [-1, 0], respectively. Obviously, both $[f]^+$ and $[-f]^+$ contain only 0 or positive values.

Let f and g be two functions. Addition (subtraction) , multiplication (scalar and dot product) and division are then performed as follows:

$$\begin{aligned} f + g &= ([f]^+ + [g]^+) - ([-f]^+ + [-g]^+) && (9)\\ fg &= [f]^+[g]^+ + [-f]^+[-g]^+ \\ &\quad -([f]^+[-g]^+ + [-f]^+[g]^+) && (10)\\ \frac{f}{g} &= \frac{fg}{g^2} && (11)\end{aligned}$$

Note that in Equation 11, g^2 can be computed as:

$$g^2 = ([g]^+)^2 + ([-g]^+)^2 \tag{12}$$

[1]Other computations are implemented with lookup table, hence are treated as inputs

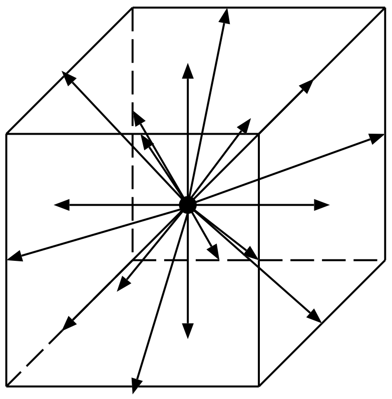


Figure 1: The D3Q19 lattice geometry. The velocity directions of the 18 moving packet distribution are shown as arrows.

Range separation introduces more calculations than the unseparated version, but recall that scaling and biasing requires additional operations as well. In practice, we choose either biasing and scaling or range separation depending on which can be executed in fewer rendering passes and involves fewer texture units. We make a choice considering also the fact that range separation provides one additional bit of precision.

4 Lattice Boltzmann Method

Now, let's review the principles of the lattice Boltzmann method [1, 9, 11] and see what computations are needed for the simulation of the LBM. The LBM consists of a regular grid and a set of packet distribution values. Each packet distribution f_{qi} corresponds to a velocity direction vector $\overrightarrow{e_{qi}}$ shooting from a node to its neighbor. The index *qi* describes the D-dimensional sub-lattice where q is the sub-lattice level and i enumerates the sub-lattice vectors. Figure 1 depicts a single node of the D3Q19 model (19 packets in 3D space), while the left part of Figure 2 shows four grid nodes of the D2Q9 (9 packets in 2D space). The arrows in the figures represent the $\overrightarrow{e_{qi}}$ vectors.

The LBM updates the packet distribution values at each node based on two rules: collision and propagation. Collision describes the redistribution of packets at each local node. Propagation means the packets move to the nearest neighbor along the velocity directions. These two rules can be described by the following equations:

$$collision: f_{qi}^{new}(\overrightarrow{x}, t) - f_{qi}(\overrightarrow{x}, t) = \Omega_{qi} \tag{13}$$

$$propagation: f_{qi}(\overrightarrow{x} + \overrightarrow{e_{qi}}, t+1) = f_{qi}^{new}(\overrightarrow{x}, t) \tag{14}$$

where Ω is a general collision operator. Since components of $\overrightarrow{e_{qi}}$ can only be choosen from {-1, 0, 1}, the propagation is local.

The density and velocity are calculated from the packet distributions as follows:

$$\rho = \sum_{qi} f_{qi} \tag{15}$$

$$\overrightarrow{v} = \frac{1}{\rho} \sum_{qi} f_{qi}\overrightarrow{e_{qi}} \tag{16}$$

The collision operator is selected in a way that mass and momentum are conserved locally. Suppose that there is always a local equilibrium particle distribution f_{qi}^{eq} dependent only on the conserved quantities ρ and $\overrightarrow{v}$, then the collision step is changed to:

$$f_{qi}^{new}(\overrightarrow{x},t) - f_{qi}(\overrightarrow{x},t) = -\frac{1}{\tau}(f_{qi}(\overrightarrow{x},t) - f_{qi}^{eq}(\rho,\overrightarrow{v})) \quad (17)$$

where τ is the relaxation time scale. f_{qi}^{eq} is decided by the following equation:

$$\begin{aligned} f_{qi}^{eq}(\rho,\overrightarrow{v}) = \rho(A_q + B_q < \overrightarrow{e_{qi}}, \overrightarrow{v} > + \\ C_q < \overrightarrow{e_{qi}}, \overrightarrow{v} >^2 + D_q < \overrightarrow{v}, \overrightarrow{v} >) \quad (18) \end{aligned}$$

$< \overrightarrow{x}, \overrightarrow{y} >$ denotes the dot product between two vectors $\overrightarrow{x}$ and $\overrightarrow{y}$. The constants A_q to D_q depend on the employed lattice geometry.

The simulation of the LBM then proceeds as follows: (1) compute density according to Equation 15; (2) compute velocity (Equation 16); (3) compute equilibrium distribution (Equation 18); (4) update distributions by Equation 17 and go back to step (1). More details on the LBM model can be found in [18].

5 Mapping LBM to Graphics Hardware

5.1 Algorithm Overview

To compute the LBM equations on graphics hardware, we divide the LBM grid and group the packet distributions f_{qi} into arrays according to their velocity directions. All the packet distributions with the same velocity direction are grouped into the same array, while keeping the neighboring relationship of the original model. Figure 2 shows the division of a 2D model. We then store the arrays as 2D textures. For a 2D model, all such arrays are naturally 2D, while for a 3D model, each array forms a volume and is stored as a stack of 2D textures. The idea of the stack of 2D textures is from 2D texture-based volume rendering, but note that we don't need three replicated copies of the dataset.

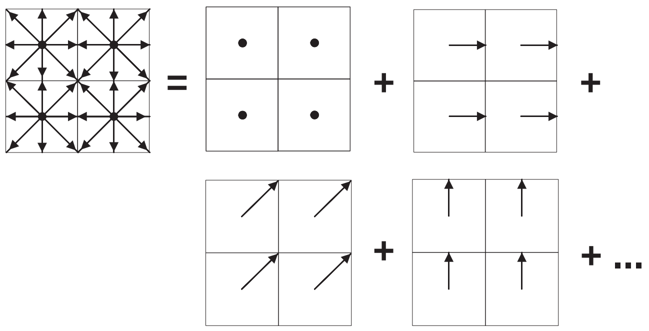


Figure 2: Division of the D2Q9 model. Only 4 grids out of 9 are shown. Packets are grouped according to their velocity directions.

All the other variables, the density ρ, the velocity $\overrightarrow{v}$ and the equilibrium distributions f_{qi}^{eq} are stored similarly in 2D textures. We project multiple textured rectangles with the color-encoded densities, velocities and distributions. For convenience, the rectangles are parallel to the viewing plane and are rendered orthogonally. Therefore, the texture space has the same resolution as the image space and the interpolation mode is set to nearest-neighbor.

As shown in Figure 3, the textures of the packet distributions are the inputs. Density and velocity are then computed from the distribution textures. Next, the equilibrium distribution textures are obtained from the densities and the velocities. According to the propagation equation, new distributions are computed from the distributions and the equilibrium distributions. Finally, we apply the boundary conditions and update the distribution textures. The updated distribution textures are then used as inputs for the next simulation step.

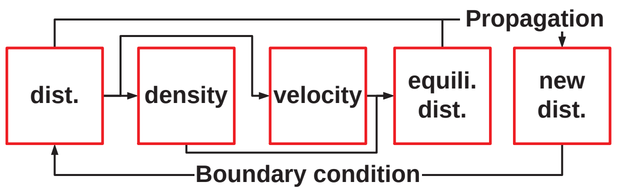


Figure 3: The data flow of the hardware accelerated LBM computations.

To reduce the overhead of switching between textures, we stitch multiple textures representing packet distributions with the same velocity direction into one larger texture. The left part of Figure 5 shows an example, in which, every four slices are stitched into a larger texture. The pipeline depicted in Figure 3 is then operated on the stitched textures.

5.2 Propagation

According to Equation 14, each packet distribution having non-zero velocity propagates to the neighboring grid every time step. Since we group packets based on their velocity directions, the propagation is accomplished by shifting distribution textures in the direction of the associated velocity. We decompose the velocity into two parts, the velocity component within the slice (*in-slice velocity*) and the velocity component orthogonal to the slice (*orthogonal velocity*). The propagation is done for the two velocity components independently. To propagate in the direction of the in-slice velocity, we simply translate the texture of distributions appropriately, as shown in Figure 4.

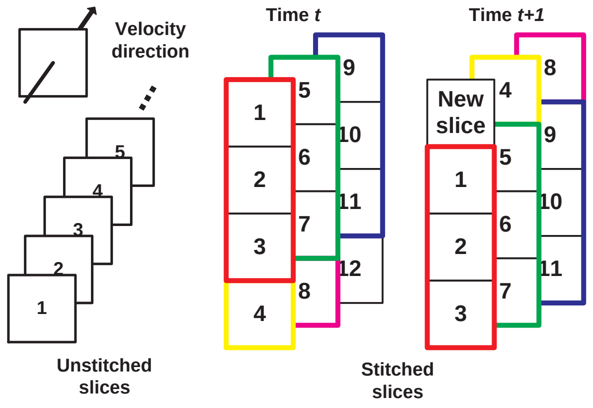


Figure 4: Propagation of the packet distributions along the direction of the velocity component orthogonal to the slices.

If we don't stitch multiple slices into one texture, the propagation in the direction of the orthogonal velocity is done simply by renaming the distribution textures. Because of the stitching, we need to apply translation inside the stitched textures as well as copying sub-textures to other stitched textures. Figure 5 shows the out-of slice propagation for stitched slices. The indexed blocks denote the slices storing packet distributions. The rectangles in thicker lines mark the sub-textures that are propagated. For example, the sub-texture composed of slices 1 to 3 is shifted down by the size of one slice in the Y dimension. Slices 4 and 8 are moved to the next textures. Note that in time step $t+1$, a new slice is added owing to the inlet or the boundary condition, while a block (12) has moved out of the framework and is discarded.

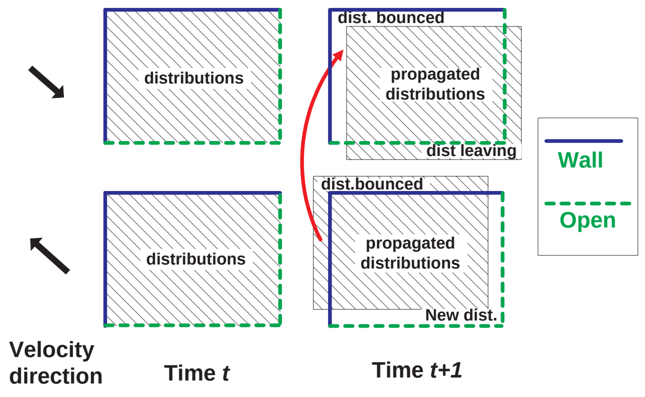


Figure 5: Propagation of the packet distributions along the direction of the in-slice velocity and application of boundary condition. The rectangular edges in blue thick lines are the boundaries.

5.3 Boundary Condition

Packet distributions on the boundary should be handled differently from the internal ones. A general approach is to compute the new distributions for the boundary distributions, then set the new values into the distributions textures. The computation can be done with either the CPU or the graphics hardware.

Bounce-back boundary condition can be easily handled by the graphics hardware. Because the particles are grouped according to their velocity directions, we simply copy the boundary packet distributions to the texture of the opposite velocity direction. Similar to propagation, the bounce-back is treated for in-slice velocity and orthogonal velocity separately. For a boundary face unparallel to the slices, the intersection of the face with a slice is a line segment (or a curve, if we allow non-planar boundary face). We set the distributions next to the intersection by drawing texture strips which are just one texel wide. Figure 4 shows the bounce-back from the left and the top walls. Note that the distributions leaving one slice become the new distributions of the slice with the opposite velocity direction. For a boundary face parallel to the slices, usually a 2D texture needs to be updated. In Figure 5, the block marked ”new slice” is obtained from the slice at the same position but with an opposite direction, or it is set to the inlet distributions if the slice is adjacent to an inlet face.

5.4 Packing

In our preliminary work [18] of hardware accelerated LBM, we stored only the distributions of the same direction in a single texture. Due to the restriction of the current graphics hardware and the considerations of efficiency, every distribution texture is in the format of RGBA. Hence, each f_{qi} is replicated 4 times into the RGBA channels, and the operations over the distributions are duplicated as well.

In this paper, we pack four f_{qi}s from different directions as an RGBA texel. That is, a single texture is composed of four distribution arrays with different velocity directions. This packing scheme reduces the memory requirement of distributions to nearly $1/4$ of the design without packing.

To compute density ρ (refer to Equation 15), we use a dot-product to add the distributions stored in different color channels, as shown in Figure 6. The packing essentially reduces the number of operations to one quarter for the computation of density. Multiple distribution textures are added together with the OpenGL extensions of multi-textures and the register combiners. In addition, we also utilize the additive blending of the frame buffer to make it unnecessary to backup the intermediate contents by copying the frame buffer to a texture or switching to different frame (pixel) buffer.

The calculation of velocities is a little more complicated owing to the packing, since each distribution needs to be multiplied with its own direction vector $\overrightarrow{e_{qi}}$ and the RGB components can't be read completely individually in the current implementation of the register combiners. Therefore, we need to dot-product the distributions in the RGB channels with (1, 0, 0) or (0, 1, 0) to separate the distributions. The value in the blue channel is extracted with an alpha combiner. As shown in Figure 7, we add eight distribution slices (stored in two textures) weighted by the corresponding $\overrightarrow{e_{qi}}$ with six combiner stages. Here we apply a trick on the final combiner stage so that it adds four inputs. Note that the inputs B and C to the final combiner stage are scaled by 2 and A is set to 0.5, while the other two inputs of the final combiner stage are sent to Spare0 and Secondary. Again, we use additive blending of the frame buffer to add consecutive outputs from the final combiner to avoid copying the frame buffer. Range separation are applied to $\overrightarrow{e_{qi}}$ so that negative values are not clamped. That is, we compute $\overrightarrow{v}^+$ and $\overrightarrow{v}^-$ separately and later add them together.

5.5 Scaling of the LBM Equations

In this section, we show how to apply the range transformation described in Section 3 to the LBM equations. Assume f_q^{max} is the left-hand scalar of the packet distributions and the equilibrium packet distributions of sub-lattice q. We define the scaled distributions $\widetilde{f_{qi}}$ and the scaled density $\widetilde{\rho}$ as:

$$\widetilde{f_{qi}} = \frac{1}{f_q^{max}} f_{qi} \tag{19}$$

$$\widetilde{\rho} = \frac{\rho}{\rho^{max}} = \sum_{qi} \frac{f_q^{max}}{\rho^{max}} \widetilde{f_{qi}} \tag{20}$$

Since all the f_{qi} are positive inputs, $\rho^{max} = U(\rho) = \sum_{qi} f_q^{max}$. We also define:

$$\frac{1}{\widetilde{\rho}'} = \frac{\rho^{min}}{\widetilde{\rho}\rho^{max}} \tag{21}$$

where ρ^{min} is the lower bound of the density and $\frac{1}{\widetilde{\rho}'} \in [0, 1]$.

According to Equation 2 and the symmetry of the LBM, the right-hand factor of the scaled velocity $U(\overrightarrow{v})$ is:

$$U(\overrightarrow{v}) = \frac{1}{\rho^{min}} \max_b \sum_{qi} f_q^{max} \{\overrightarrow{e_{qi}}[b] > 0\} \tag{22}$$

where b is the dimension index of vector $\overrightarrow{e_{qi}}$. Note that $U(\overrightarrow{v})$ and v^{max} are scalars instead of vectors. Then, the scaled velocity is computed as:

$$\widetilde{\overrightarrow{v}} = \frac{\overrightarrow{v}}{v^{max}} = \frac{U(\overrightarrow{v})}{v^{max}} \frac{1}{\widetilde{\rho}'} \sum_{qi} (\frac{f_q^{max}}{U(\overrightarrow{v})\rho^{min}} \overrightarrow{e_{qi}}) \widetilde{f_{qi}} \tag{23}$$

With such range scaling, Equations 17 and 18 become:

$$\widetilde{f_{qi}}(\overrightarrow{x} + \overrightarrow{e_{qi}}, t+1) = \widetilde{f_{qi}}(\overrightarrow{x}, t) - \frac{1}{\tau}(\widetilde{f_{qi}}(\overrightarrow{x}, t) - \widetilde{f_{qi}^{eq}}) \tag{24}$$

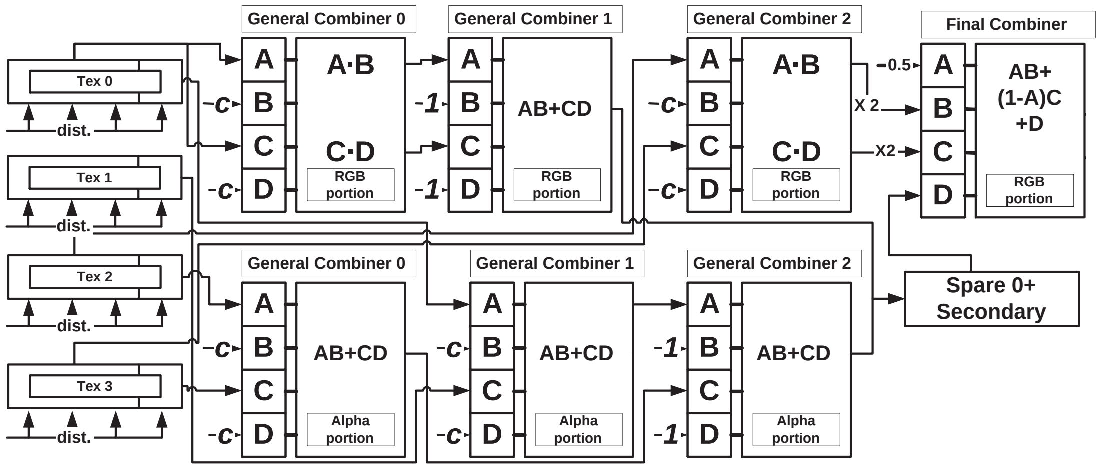


Figure 6: The configuration of the register combiners for computing density from packed distributions. c denotes constants dependent on qi and the scaling factors

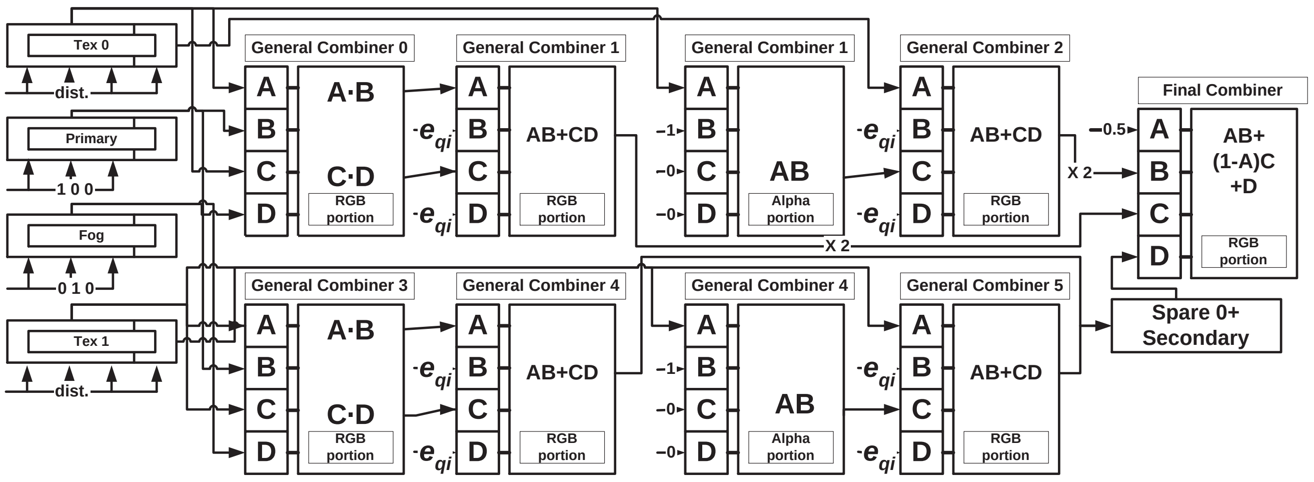


Figure 7: The configuration of the register combiners for computing velocity from packed distributions.

$$\widetilde{f_{qi}^{eq}} = \frac{U(f_q^{eq})}{f_q^{max}} \tilde{\rho}\Big(\frac{A_q \rho^{max}}{U(f_q^{eq})} + \frac{2B_q v^{max} \rho^{max}}{U(f_q^{eq})} < \frac{\overrightarrow{e_{qi}}}{2}, \widetilde{\vec{v}} > + \frac{4C_q (v^{max})^2 \rho^{max}}{U(f_q^{eq})} < \frac{\overrightarrow{e_{qi}}}{2}, \widetilde{\vec{v}} >^2 + \frac{4D_q (v^{max})^2 \rho^{max}}{U(f_q^{eq})} < \frac{\widetilde{\vec{v}}}{2}, \frac{\widetilde{\vec{v}}}{2} >\Big) \quad (25)$$

For D3Q19 model, $A_q \geq 0$, $B_q \geq 0$, $C_q \geq 0$ and $D_q \leq 0$, hence:

$$U(f_{qi}^{eq}) = \max((\rho^{max} A_q + 2B_q v^{max} \rho^{max} + 4C_q (v^{max})^2 \rho^{max}), (2B_q v^{max} \rho^{max} - 4D_q (v^{max})^2 \rho^{max})) \quad (26)$$

Note that in Equation 25, we scaled the vectors before the dot products. The scaling factor is chosen to be a power of two so that it is easy to implement it in hardware.

6 Experimental Results

We have implemented our techniques on an Nvidia GeForce4 Ti 4600 card that has 128MB of memory. For comparison, we also implemented the LBM in software on a PC with a 1.6Ghz P4 processor and 512MB DDR memory. We use the D3Q19 model throughout the experiments.

6.1 Accuracy

A major concern about using graphics hardware for general computation is the accuracy. Most graphics hardware supports only 8 bits per color channel. There have been a few limited supports of 16-bit textures but are too restricted for a relatively complicated application such as the LBM simulation. Fortunately, the variables of the LBM fall into a small numerical range which makes the range scaling effective. Besides, the property of the LBM, that the macroscopic dynamics is insensitive to the underlying details of the microscopic physics [?], relaxes the requirement on the accuracy of the computation.

Figures 8a and 8b shows two color-encoded velocity slices extracted from a 3D LBM simulation. Figure 8a is computed by the CPU with floating-point accuracy and Figure 8b is obtained with the hardware approach described in the paper. Figure 8c shows the exaggerated error image with all pixel values scaled up by 10. All the values are transformed to the range of [0, 255] for display. Visually, it is difficult to see any difference between Figure 8a and 8b. Actually, the maximal pixel-wise difference is less than 1% after one step of simulation.

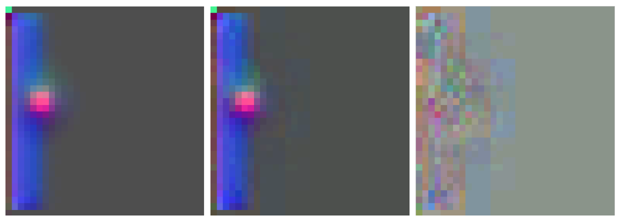


Figure 8: Velocity slice computed by (a) software, (b) graphics hardware. (c) The difference image between (a) and (b) scaled up by a factor of ten.

6.2 Performance

Stitching smaller textures into bigger ones significantly reduces the overhead of texture switching. Before testing the performance of our hardware implementation of LBM, we first determine how big the stitched textures should be. Figure 9 shows the relationship between the area of the stitched textures and the number of cells that the hardware can handle per second. For our hardware configuration, a texture of the size of 512×512 performs the best. In fact, for textures smaller than 256×256, the computation time is nearly independent on the size of the textures. Note that we restrict the dimensions of the textures to be powers of two, although we could exploit the non-power-of-two-texture extension to achieve even better results.

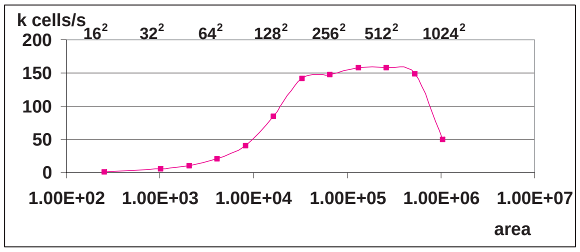


Figure 9: Number of cells processed per second as a function of the area of the stitched textures. Note that the X axis is in Logarithmic scale.

Figure 10 compares the time (in seconds) per step of the hardware LBM with a software implementation. The statistics does not include the time for rendering. The ”Stitching” curve refers to the performance after stitching small ones into 512×512 textures, while ”No Stitching” does not. Note that the hardware accelerated technique wins in speed for any size of the model, except that for the 16^3 grid, the ”No Stitching” method is slower than software. However, simply by stitching the sixteen 16×16 textures into one 64×64 textures gains a speedup factor of 12. Figure 10 is in logarithmic scale for both the axes. Note that stitching is very effective for grids equal to or smaller than 128^3.

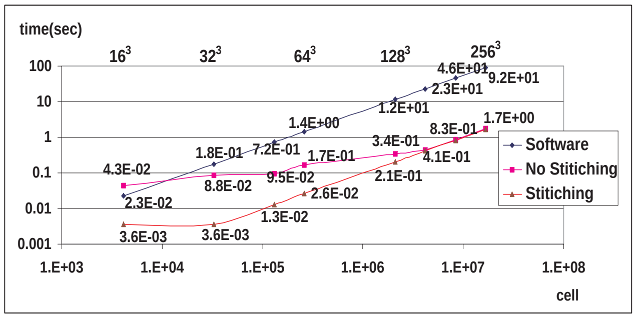


Figure 10: Time per step of the LBM compution with graphics hardware (with or without stitching) and software.

The speedup factor of the hardware accelerated method against the software approach is more clearly shown in Figure 11. The proposed method is at least 50 times faster than its software counterpart except for the 16^3 model. Note that the memory requirement of a 256^3 sized model is much larger than the memory on the graphics board. Actually, only the distributions for a D3Q19 model need $256^3 \times 19 = 319$M. Our implementation still achives 1.7 second per step for a 256^3 model, which is acceptable for an interactive application.

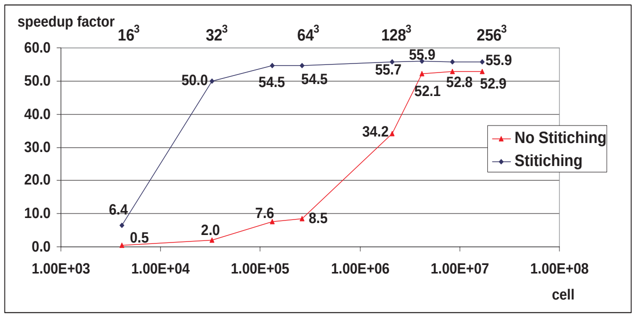
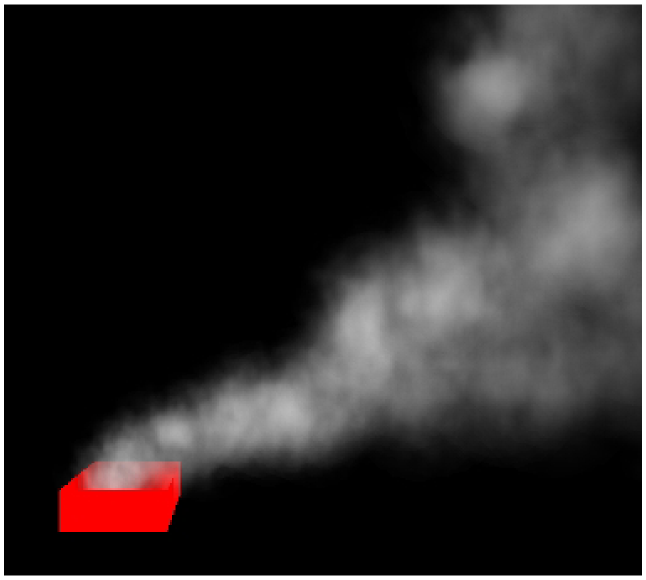


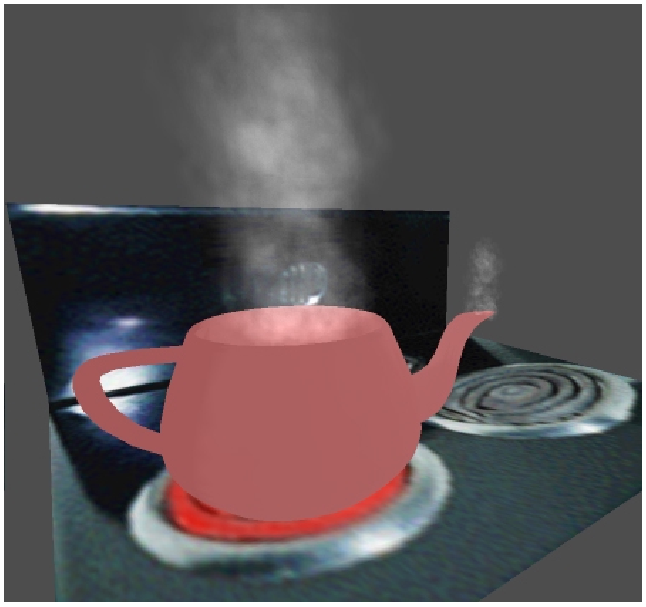
Figure 11: Speedup factor of the hardware accelerated method to the software method.

6.3 Application

We visualize the simulation results by either directly showing the color-encoded velocity field (as in Figure 8) or by injecting particles into the system from an inlet. These particles are considered massless, that is, they do not affect the flow calculation. The particles are advected according to the velocity of the grid cell. The number of particles depends on the density of the fluid. We then render the scene with texture splats [18]. Figure 12a shows an image of smoke emanating from a chimney and then blown by the wind. The left side of the grid is assigned a speed along the X axis to model the effect of wind. We also incorporate an upward force due to the difference in the temperature field. Figure 12b shows the result of hot steam rising up from a teapot and its spout according to a velocity field simulated with our LBM approach and hardware acceleration. Both the smoke and the steam are simulated on a 32^3 grid.



(a) Smoke emanating from a chimney and then blown up by the wind.



(b) Hot steam rising up from a teapot and its spout.

Figure 12: Applications of the hardware accelerated LBM.

7 Discussion

In this paper, we presented the algorithms for implementing the Lattice Boltzmann Model on commodity graphics hardware. Experimental results show that the LBM can be simulated on current low-cost computers in real time for a grid size of up to 64^3 and interactively for a grid size of 128^3.

Although we focused on the LBM, our techniques can be extended to other computations. It is also possible to generalize our methods into a framework of accelerating a large variety of applications on conventional graphics hardware and its future enhanced versions. One of our planned direction is a development environment including a language describing general parallel computations and a compiler that automatically translates code written in the language into available operations on graphics hardware. We would also like to develop a debugger for conveniently inspecting the intermediate results of the graphics pipeline.

Acknowledgement

This work has been supported by an ONR grant N000140110034. We thank Klaus Mueller and Suzi Yoakum-Stover for helpful discussions.

References

[1] S. Chen and G. D. Doolean. Lattice boltzmann method for fluid flows. *Annu. Rev. Fluid Mech.*, 30:329–364, 1998.

[2] S. Fang and H. Chen. Hardware accelerated voxelization. *Computers & Graphics*, 24(3):433–442, June 2000.

[3] W. Heidrich, R. Westermann, H.-P. Seidel, and T. Ertl. Applications of pixel textures in visualization and realistic image synthesis. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 127–134, April 1999.

[4] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of SIGGRAPH 99*, pages 277–286, August 1999.

[5] K. Hoff, A. Zaferakis, M. C. Lin, and D. Manocha. Fast and simple 2D geometric proximity queries using graphics hardware. In *2001 ACM Symposium on Interactive 3D Graphics*, pages 145–148, March 2001.

[6] M. Hopf and T. Ertl. Accelerating 3d convolution using graphics hardware. In *IEEE Visualization '99*, pages 471–474, October 1999.

[7] M. Hopf and T. Ertl. Accelerating morphological analysis with graphics hardware. In *Workshop on Vision, Modelling, and Visualization VMV '00*, pages 337–345, 2000.

[8] B. Jobard, G. Erlebacher, and M. Y. Hussaini. Hardware-accelerated texture advection for unsteady flow visualization. In *IEEE Visualization 2000*, pages 155–162, October 2000.

[9] B. D. Kandhai. *Large Scale Lattice-Boltzmann Simulations*. PhD thesis, University of Amsterdam, December 1999.

[10] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. *The International Conference for High Performance Computing and Communications*, 2001.

[11] D. Muders. *Three-Dimensional Parallel Lattice Boltzmann Hydrodynamics Simulations of Turbulent Flows in Interstellar Dark Clouds*. PhD thesis, University at Bonn, August 1995.

[12] K. Mueller and R. Yagel. On the use of graphics hardware to accelerate algebraic reconstruction methods. In *SPIE Medical Imaging Conference*, 1999.

[13] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. In *Proceedings of ACM SIGGRAPH 2000*, pages 425–432, July 2000.

[14] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of ACM SIGGRAPH 2001*, pages 159–170, August 2001.

[15] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programable hardware. In *Proceedings of ACM SIGGRAPH 2002*, August 2002.

[16] C. Rezk-Salama, M. Scheuering, G. Soza, and G. Greiner. Fast volumetric deformation on general purpose hardware. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2001.

[17] C. Trendall and A. J. Stewart. General calculations using graphics hardware with applications to interactive caustics. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 287–298, June 2000.

[18] X. Wei, W. Li, K. Mueller, and A. Kaufman. Simulating fire with texture splats. In *Proceedings IEEE Visualization '02*, 2002.

[19] D. Weiskopf, M. Hopf, and T. Ertl. Hardware-accelerated visualization of time-varying 2D and 3D vector fields by texture advection via programmable per-pixel operations. In *Workshop on Vision, Modeling, and Visualization VMV '01*, pages 439–446, 2001.