Implementing the lattice Boltzmann model on commodity graphics hardware

Journal of Statistical Mechanics: Theory and Experiment
An IOP and SISSA journal

Implementing the lattice Boltzmann model on commodity graphics hardware

Arie Kaufman, Zhe Fan and Kaloian Petkov

Center for Visual Computing and Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-4400, USA
E-mail: ari@cs.sunysb.edu, fzhe@cs.sunysb.edu and kpetkov@cs.sunysb.edu

Received 4 November 2008
Accepted 24 March 2009
Published 18 June 2009

Online at stacks.iop.org/JSTAT/2009/P06016
doi:10.1088/1742-5468/2009/06/P06016

Abstract. Modern graphics processing units (GPUs) can perform general-purpose computations in addition to the native specialized graphics operations. Due to the highly parallel nature of graphics processing, the GPU has evolved into a many-core coprocessor that supports high data parallelism. Its performance has been growing at a rate of squared Moore's law, and its peak floating point performance exceeds that of the CPU by an order of magnitude. Therefore, it is a viable platform for time-sensitive and computationally intensive applications.

The lattice Boltzmann model (LBM) computations are carried out via linear operations at discrete lattice sites, which can be implemented efficiently using a GPU-based architecture. Our simulations produce results comparable to the CPU version while improving performance by an order of magnitude. We have demonstrated that the GPU is well suited for interactive simulations in many applications, including simulating fire, smoke, lightweight objects in wind, jellyfish swimming in water, and heat shimmering and mirage (using the hybrid thermal LBM).

We further advocate the use of a GPU cluster for large scale LBM simulations and for high performance computing. The Stony Brook Visual Computing Cluster has been the platform for several applications, including simulations of real-time plume dispersion in complex urban environments and thermal fluid dynamics in a pressurized water reactor.

Major GPU vendors have been targeting the high performance computing market with GPU hardware implementations. Software toolkits such as NVIDIA CUDA provide a convenient development platform that abstracts the GPU and allows access to its underlying stream computing architecture. However, software programming for a GPU cluster remains a challenging task. We have therefore developed the Zippy framework to simplify GPU cluster programming. Zippy is based on global arrays combined with the stream programming model and it hides the low-level details of the underlying cluster architecture.

Keywords: lattice Boltzmann methods

Contents

1. Introduction

The graphics processing unit (GPU) is the processor inside a commodity 3D graphics card that has been designed for accelerating rasterization-based graphics rendering. Computer games and virtual environments utilize dense 3D meshes and enormous amounts of graphics data that have to be rendered at real-time speeds (no less than 15 frames per second). These requirements have long surpassed the capabilities of software-based renderers and fueled the growth of hardware-based solutions.

Figure 1 shows a comparison of the recent growth in computational power of GPUs and CPUs. During the 2003–2008 period, the GPU computational throughput has increased 73 times and on average has doubled every 11 months. In contrast, the CPU

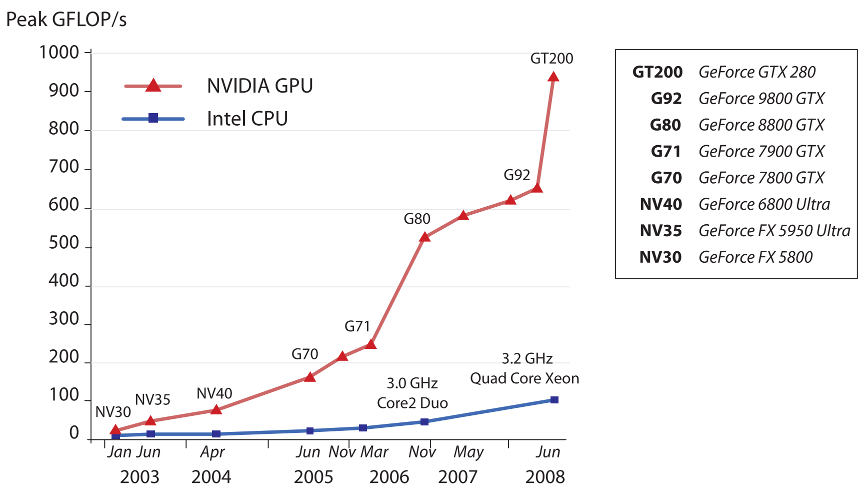


Figure 1. A comparison of the growth of the computational power of GPUs (green) and CPUs (blue). (This figure has been adapted from [14].)

computational power, which doubled every 20 months on average (approximating Moore's law), has increased only 10 times. The high GPU computational power and its fast growth have been made possible by the explicit data parallelism inherent in the rasterization-based rendering workload. Contiguous data elements such as vertices, fragments, and geometry primitives are processed in a SIMD fashion where an operation may take as input thousands of elements. These operations are relatively simple and their code contains only a small number of branches. In addition, data elements are often processed independently of each other: the results computed for one pixel are not needed by other pixels in the same rendering pass. Therefore, GPU operations have well understood and predictable memory access patterns that allow efficient pre-fetching.

Modern GPUs have been designed with very simple computational cores that forgo the complex caching and flow control needed in CPU architectures [14]. The data parallelism present in rasterization-based rendering operations favors a large number of cores working in parallel on very high bandwidth memory. Since the architecture has only limited caching capability, the latency of memory accesses is instead hidden via dense calculations. As an example, the contemporary NVIDIA flagship product, the GeForce GTX 280, contains 240 thread processors that collectively deliver a theoretical peak performance of 933 GFLOPS for single-precision computations. To match this high computational power, the GPU is coupled with very fast on-board texture memory with bandwidth of 142 GB s^{-1}, which is an order of magnitude higher than the bandwidth available in many CPU systems.

The flexibility and programmability of GPUs have increased with each new hardware generation. The graphics pipeline supports high-level shading languages, such as GLSL [17], Cg [10], and HLSL [19], which provide a C-like programming environment. These languages allow graphics developers to design customized vertex, geometry, and fragment shaders that replace the fixed-function transformation and lighting units in older non-programmable hardware. Because of this programmability and the high computational performance, general-purpose computation using GPUs (GPGPU) [15] has become an active area of research. Target applications span a variety of fields and include physically-based simulations, linear algebra operations, fast Fourier

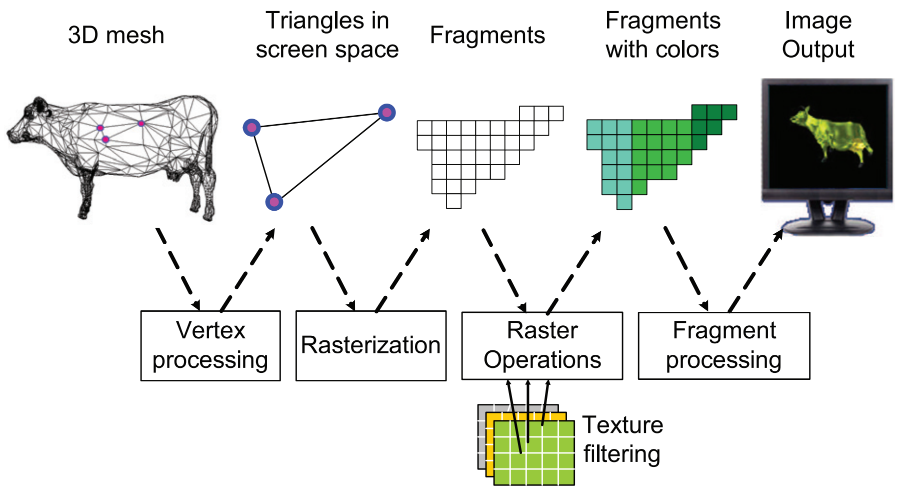


Figure 2. A simplified illustration of the graphics pipeline.

transform, image/volume processing and reconstruction, registration, volume rendering, flow visualization and database operations.

We present a survey of efficient implementations of the lattice Boltzmann model (LBM) [20] for interactive flow simulations on the GPU and on GPU clusters. Since the LBM models the dynamics of fluid particle distribution functions at a mesoscopic level, boundary conditions are easy to handle, even for complex moving shapes, by manipulating the individual particle distributions. Also, the method allows the modeling of interactions within multi-component and multi-phase fluids. These two features make the LBM attractive for many applications, where interactions with the simulation domain are typically important. Furthermore, because LBM computations are explicitly data parallel, we can perform the simulation very efficiently on the GPU architecture. Such a system can achieve physically accurate results at potentially interactive speeds in computer graphics, prediction, and science applications. The GPU programming techniques that we discuss involve approaches based on the graphics pipeline, our Zippy framework and a new implementation using the NVIDIA CUDA programming environment. Our experimental results cover several generations of graphics hardware, spanning the entire timeline presented in figure 1, and include previously unpublished performance figures.

2. Background of GPU

The rasterization-based rendering on the GPU involves a sequence of processing stages that run in parallel and in a fixed order, known as the graphics pipeline (see figure 2). The first stage of the pipeline is the *vertex processing*. The input to this stage is a 3D polygonal mesh and the world coordinates of each vertex are transformed to a 2D screen position. Lighting and texture coordinates associated with each vertex are also evaluated. During the *rasterization* stage the transformed vertices are grouped into rendering primitives, such as triangles which are then scanline-converted to a set of fragments in screen space. Each fragment stores the state information needed to compute a pixel value. In the *fragment processing* stage, the texture coordinates associated with each fragment are used to fetch the appropriate texels (texture pixels) from a set of textures residing in the on-board texture memory. Mathematical operations may also be performed to determine the resulting color for the fragment. Finally, various *raster operations* such as depth testing, stencil testing, and alpha blending determine whether and how the fragment is used to update a pixel in the frame buffer.

Early graphics cards for commodity PCs implemented the graphics pipeline through fixed-function hardware. In 1999, NVIDIA introduced the term GPU with the GeForce 256. In addition to the vertex processing capability through a T&L unit (Transform and Lighting), the GeForce 256 provides register combiners that give developers limited control over the fragment processing portion of the pipeline. In 2000, Microsoft released the DirectX 8.0 API, which provided a standard interface for writing *vertex shaders* and *fragment shaders*. The shader is a set of instructions that specifies a custom rendering effect and replaces fixed-function processing in the pipeline. Early shaders were developed using assembly style instructions and pioneered the introduction of the programmable graphics pipeline into the consumer space. Early specifications limited vertex shaders to 128 instructions and pixel shaders to 12 instructions on a fixed-point pipeline. Contemporary hardware supports 65535 static instructions and unlimited dynamic instructions through branching, a large register file and full 32-bit precision throughout the graphics pipeline. High-level shading languages such as GLSL [17], Cg [10], and HLSL [19] have greatly simplified the creation of shaders by providing C-like languages for graphics processing and optimizing compilers. Recently, a new stage, the *geometry shader*, has been added into the graphics pipeline. While the vertex and fragment shaders can only process data sent to the GPU by the host application, a geometry shader can dynamically generate new graphics primitives in an iterative fashion.

To process large graphics data sets at real-time speed, modern GPUs are data driven and emphasize data parallelism. For instance, the NVIDIA GeForce 7900 GTX has 8 vertex processors and 24 fragment processors that evaluate shaders in a SIMD fashion. These processors support four-dimensional vectors (representing the xyzw homogeneous coordinates or the RGBA color channels) and a four-component vector floating point SIMD instruction set for computation. Later graphics hardware such as the NVIDIA GeForce 8800 GTX and GeForce 9800 GTX employ a unified shader model in which the 128 scalar processors are dynamically allocated to vertex, geometry, and fragment processing. The newer NVIDIA GeForce GTX 280 graphics board contains 240 scalar processors.

A wide range of general-purpose computation applications have been developed on the GPU, such as physically-based simulation, flow visualization, volume rendering, ray tracing, computed tomography (CT) volume reconstruction, volume segmentation, volume registration, computational geometry, image processing, collision detection, dense matrix multiplication and database operations (see also [12, 15, 16]). Programming languages, such as Brook [1], Sh [11], PeakStream, and RapidMind, have been proposed for GPGPU programming. Their goal is to abstract the GPU as a stream coprocessor and hide the underlying graphics pipeline from the programmer. NVIDIA CUDA [14] is a contemporary development environment that uses the C language with extensions for stream computing. On supported hardware, CUDA interfaces with the GPU directly and overcomes some of the limitations imposed by the graphics pipeline. In particular, the CUDA kernels have no limits on the instruction count and have limited support for memory scatter operations.

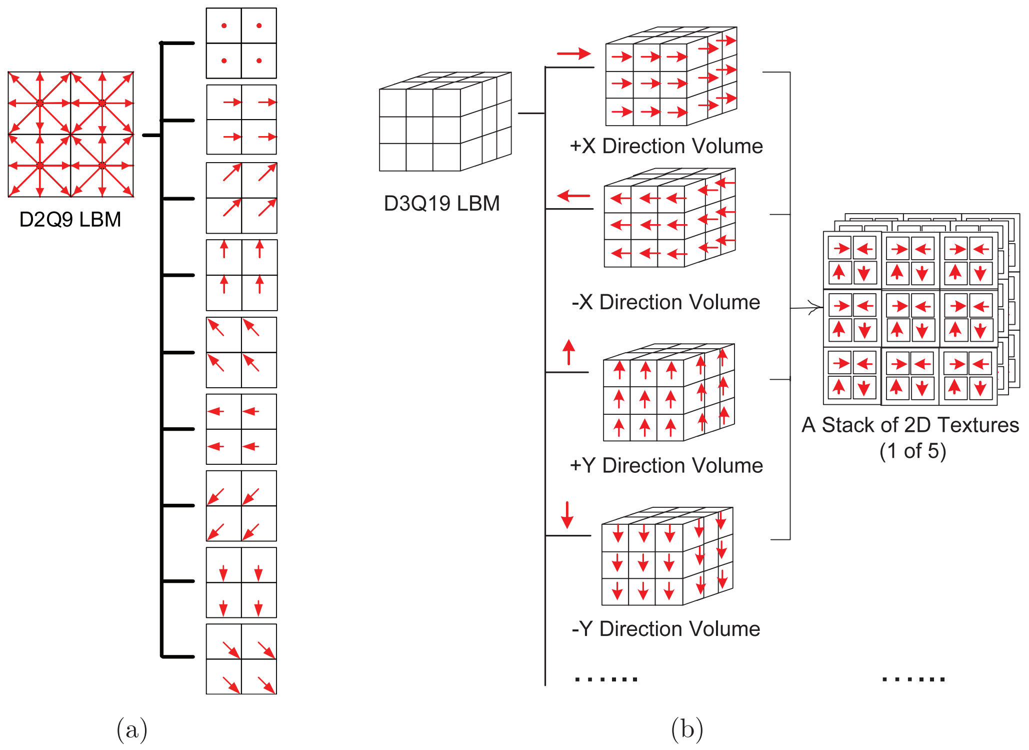


Figure 3. (a) Division of the D2Q9 model, according to its velocity directions. (b) Division of the D3Q19 model. We pack groups of four direction volumes into one stack of 2D textures.

The NVIDIA GeForce GTX 200 series also supports double-precision computation in CUDA kernels.

3. LBM on the GPU

3.1. Algorithm overview

To compute the LBM equations on the GPU, we divide the LBM lattice and group the particle distributions f_i into arrays according to their velocity vectors $\mathbf{c}_i$. All the particle distributions of the same velocity vector are grouped into the same array, while preserving the spatial relationship of the original model. For a 2D simulation domain, we store the arrays as a set of 2D textures. Figure 3(a) shows the division of a 2D model D2Q9 according to its velocity directions.

We use a similar approach for the 3D domain. Each of the 19 particle distributions f_i in the D3Q19 LBM is represented by a scalar volume. As shown in figure 3(b), we pack groups of four volumes into one stack of 2D textures, using the four-color components of each texel. Therefore, the 19 distribution values can be packed into 5 stacks of 2D textures.

All other variables needed for the LBM, such as the density ρ, the velocity $\mathbf{u}$, and the equilibrium distributions f_i^{eq} are stored similarly in 2D textures. To initiate the LBM computation, we set the rendering viewport in OpenGL to the size of the 2D texture and render a rectangle that completely covers the viewport under orthographic projection.

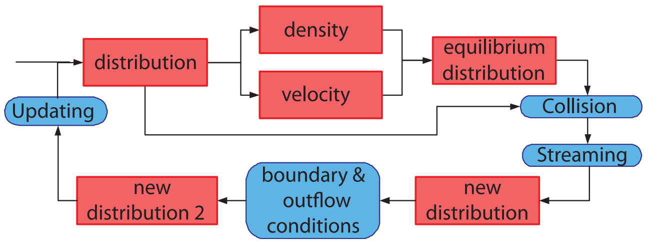


Figure 4. Flow chart of the LBM computation on the GPU. (Red boxes indicate textures; blue round boxes indicate operations.)

During the rasterization stage of the graphics pipeline, a single fragment is created for each lattice site represented in the textures. The LBM equations are implemented by a set of fragment shaders, and during a single rendering pass a shader is evaluated once for each fragment in the pipeline. Figure 4 shows the data flow of the LBM computation on the GPU, where the textures storing lattice properties are represented by red boxes and the operations are represented by blue rounded boxes. Each operation involves the execution of a shader program that samples the textures bound as input and renders to one or more output textures.

With the flexibility of modern GPUs, a translation of the LBM operations shown in figure 4 to fragment shaders is fairly straightforward when using a high-level shading language such as Cg [10]. However, achieving high performance on the GPU requires additional optimizations that are specific to the underlying hardware architecture.

3.2. Data packing

During the simulation, textures are updated dynamically at every step by copying from or binding to the frame buffer (or the pixel buffer). In the graphics hardware, it is most efficient to use RGBA textures where each texel contains four channels (Red, Green, Blue and Alpha) which can store up to four scalar values or a 4D vector.

The first optimization that we propose is packing different variables into the same texel. For example, we pack four particle distributions with different velocities into a single RGBA texel. In addition, we store together variables that are involved in the same LBM equations in order to reduce the number of textures that need to be activated and the number of texture fetches. This strategy also improves the data locality and the cache coherency during texture sampling operations in the fragment shader.

Table 1 lists the contents of the textures packed with the variables of the D3Q19 model, including densities, velocities and particle distributions. In texture $\mathbf{u}\rho$, v_x, v_y, and v_z are the three components of the velocity stored in the RGB channels, while the density ρ is stored in the Alpha (A) channel. The rows in table 1 for textures *Tex0* through *Tex4* show the packing patterns of both the particle distributions f_i and the equilibrium distributions f_i^{eq}. $f_{(x,y,z)}$ is the distribution in the direction of (x, y, z). Note that we pack distributions of the opposite directions in pairs into the same texture. This approach improves efficiency during handling of complex or moving boundaries where neighboring distributions at opposite directions are needed to compute particle distributions on boundary links. Also, when programming the fragment pipeline, we

Table 1. LBM variables of the D3Q19 model packed in textures.

Texture	R	G	B	A
$\mathbf{u}\rho$	v_x	v_y	v_z	ρ
Tex0	$f_{(1,0,0)}$	$f_{(-1,0,0)}$	$f_{(0,1,0)}$	$f_{(0,-1,0)}$
Tex1	$f_{(1,1,0)}$	$f_{(-1,-1,0)}$	$f_{(1,-1,0)}$	$f_{(-1,1,0)}$
Tex2	$f_{(1,0,1)}$	$f_{(-1,0,-1)}$	$f_{(1,0,-1)}$	$f_{(-1,0,1)}$
Tex3	$f_{(0,1,1)}$	$f_{(0,-1,-1)}$	$f_{(0,1,-1)}$	$f_{(0,-1,1)}$
Tex4	$f_{(0,0,1)}$	$f_{(0,0,-1)}$	$f_{(0,0,0)}$	Unused

typically need to pass the corresponding velocity vectors $\mathbf{e}_i$ as arguments to the fragment program. When opposite distributions are always neighbors, only two of the $\mathbf{e}_i$ vectors are needed, while the other two are easily inferred.

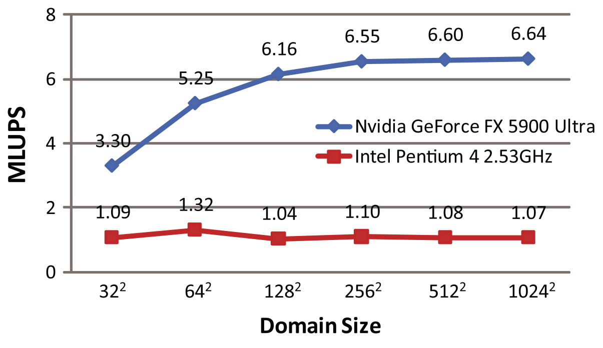
Another optimization is related to the representation of the 3D simulation domain in the GPU memory. Instead of using a stack of 2D textures to represent a volume, we tile the slices to create a single larger 2D texture [6, 8]. One advantage of this 'flat volume' approach is the reduced number of texture switches during the execution of the fragment shaders. It also reduces the number of proxy quads used during rendering. Specifically, a $W \times H \times D$ volume is stored as a $(W * d1) \times (H * d2)$ texture, where $D = d1 * d2$. We choose $d1$ to be the factor of D that is the closest to $\sqrt{D}$.

During the streaming operation in the LBM, each particle distribution with non-zero velocity propagates to a neighboring lattice position at every time step of the simulation. On contemporary GPUs, the texture sampling unit can obtain a texel at an arbitrary position indicated by texture coordinates that are computed in the fragment program. When a distribution f is propagated along vector $\mathbf{e}$, a texel is fetched from the distribution texture at the current lattice position minus $\mathbf{e}$. Since the four channels are packed with four distributions with different velocity vectors, four fetches are needed for each fragment.

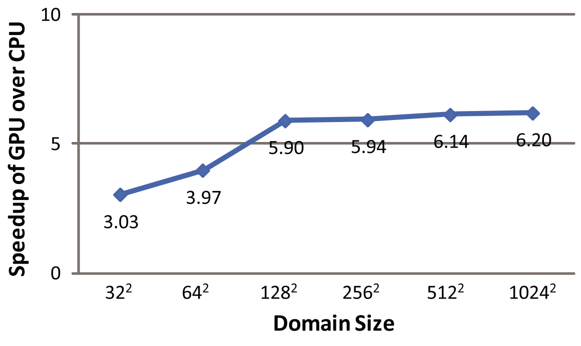
To propagate using the flat volume, for each channel, we can add the 3D position of the fragment with the negative of the corresponding $\mathbf{e}_i$, then convert to the texture coordinate space of the flat volume before fetching. However, this approach computes the conversion four times per fragment. One optimization that we apply is to push the coordinate conversions to the vertex level, since for each channel inside a slice, the velocity vectors are the same. We can either assign to each vertex of the proxy quad four texture coordinates containing the converted values, or generate the texture coordinates with a vertex program. With this approach, the correct texel positions for all lattice sites are computed efficiently by linear interpolation during the rasterization phase of the graphics pipeline.

3.3. Experimental results

We have implemented a GPU-based LBM framework for fluid simulations using OpenGL and the Cg shading language. The target platform contains a single NVIDIA GeForce FX 5900 Ultra graphics board with a core speed of 400 MHz and 256MB DDR SDRAM running at 425 MHz. The host contains a Pentium IV 2.53 GHz processor with 1 GB PC800 RDRAM. All GPU computations use full 32-bit precision throughout the pipeline. For comparison, we have also implemented a software version of the LBM simulation using

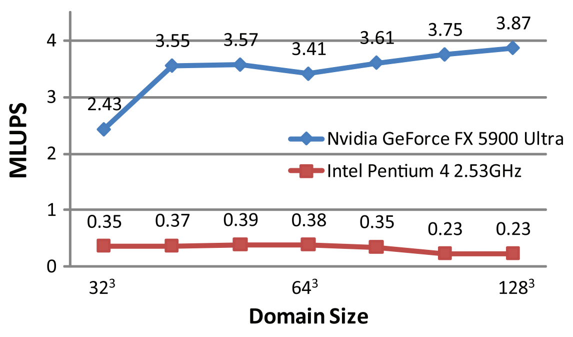


(a)

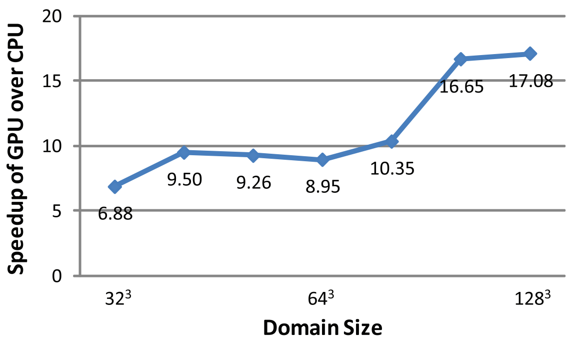


(b)

Figure 5. Performance results for the D2Q9 LBM simulation. (a) LBM performance in MLUPS. (b) Speedup on the GPU versus CPU.



(a)



(b)

Figure 6. Performance results for the D3Q19 LBM simulation. (a) LBM performance in MLUPS. (b) Speedup on the GPU versus CPU.

single-precision floating point arithmetic. The peak theoretical performance for the GPU is 28 GFLOPs, compared to 5 GFLOPs for the CPU.

Figure 5(a) shows the LBM performance in MLUPS (mega-lattice site updates per second) for the LBM D2Q9 model as a function of the lattice size for both the CPU and the GPU versions. The test case is the lid-driven cavity flow benchmark. In figure 5(b) we plot the GPU versus CPU speedup factor for our implementations. For domain sizes equal to or greater than 128^2, the unoptimized GPU simulation is approximately six times faster than the equivalent unoptimized CPU simulation. The GPU is less effective for smaller problem sizes due to fixed overhead in the GPU pipeline. In particular, texture switching and OpenGL context switching are costly operations that are independent of the lattice size. For GPU computations on large textures, the fixed cost of these operations is amortized over the longer running time of the simulation and the efficiency per lattice site is improved.

In figure 6 we compare the CPU and GPU performances for the D3Q19 LBM model. In comparison to the 2D simulation case, each lattice site requires more computation and more data access operations. Therefore, the GPU implementation achieves an even higher speedup factor. In terms of computational accuracy, the 32-bit single-precision

GPU pipeline offers sufficient accuracy for many LBM flow simulations in interactive applications. A GPU-based LBM implementation has been used for interactive modeling of 3D natural phenomena, such as soap bubbles and feathers in the wind [23], fire [26], smoke propagation [25], flow on curved surfaces [4] and heat shimmering and mirage [24].

4. LBM on a GPU cluster

Because of the attractive FLOPS/dollar ratio and the rapid evolution of GPUs in terms of features, programmability and performance, a GPU cluster seems very promising for data-intensive scientific computing. In particular, it can outperform CPU clusters at a similar price point. We built the Stony Brook Visual Computing Cluster in 2004, following this with a significant expansion in 2005, and implemented on it the LBM for flow simulations, along with a variety of other applications. The cluster contains 64 nodes connected by a Gigabit Ethernet switch or Infiniband. Each of the first 32 nodes is an HP workstation equipped with two Intel Xeon 2.4 GHz processors, 2.5 GB of memory and a GeForce FX 5800 Ultra GPU with 128 MB on-board memory, which is used for GPU-based cluster computations. The remaining 32 nodes contain dual Intel Xeon 3.6 GHz CPUs and high performance NVIDIA Quadro FX 4500 GPUs. We use MPI for run time data transfer across network nodes. Sharing the GPU memory content between nodes also requires managing data transfers between the GPU and the host processor.

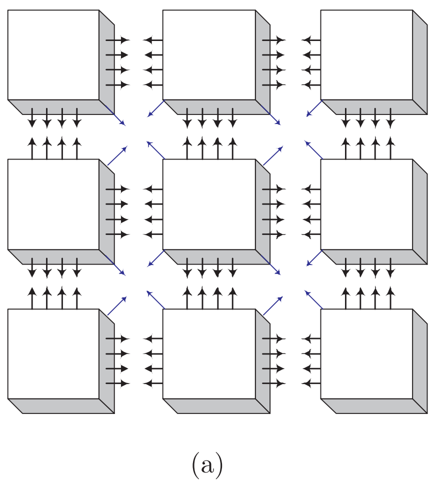
4.1. Domain partitioning

To implement the LBM on the GPU cluster, we decompose the LBM lattice space into 3D sub-domains, each processed by a single GPU. The computation therefore contains two levels of parallelism. At the coarse level, multiple GPUs synchronize velocity distributions at the border sites of sub-domains with adjacent network nodes during every simulation step. At the fine level, each GPU executes the LBM computation kernels in SIMD fashion on the lattice sites of its sub-domain, as described in section 3. Figure 7(a) illustrates the communication pattern where boundary particle distributions stream to axial nearest neighbor nodes and diagonal second-nearest neighbor nodes. The pattern is scalable to 3D sub-domains. Communication involves three steps: (1) distributions are read from the GPU memory into CPU memory; (2) data packets are transferred through the network to the system memory of the appropriate neighboring nodes; (3) particle distributions are then written to the GPU memory of the neighboring node.

4.2. Optimization of inter-GPU communication

The primary challenge in implementing the LBM on the GPU cluster is to minimize the communication cost—the time taken for network communication and for transferring data between the GPU and the CPU memory. One approach to hiding network latency is to overlap the LBM computation with the data transfer since the CPU is idle during the execution of the GPU kernels. The GPU can compute a single simulation step very quickly, and therefore additional optimizations of network performance are required.

Evaluating the impact of communication switching time using both MPI and TCP/IP sockets shows that: (1) during communication between two nodes, if a third node tries to



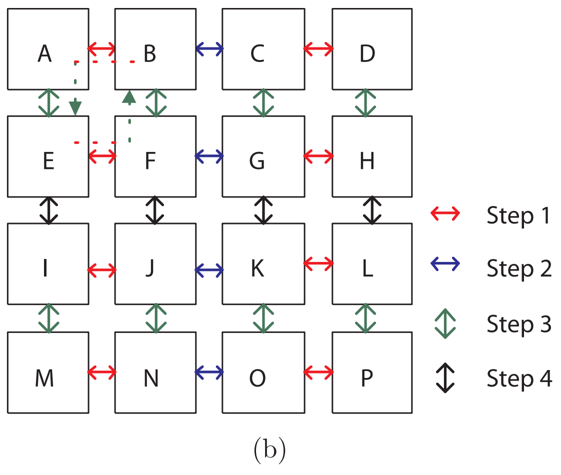


Figure 7. (a) The LBM lattice is decomposed into sub-domains and each sub-domain is processed by one GPU. Arrows show the communication among GPUs. (b) The optimized communication schedule and pattern of the parallel LBM simulation. Different colors indicate different steps in the schedule.

send data to either one, the interruption may dramatically reduce transfer performance, and (2) assuming a constant communication data size, increasing the number of receiving nodes has a considerable negative effect on communication time. To address these issues, we have designed communication schedules [21] that reduce the likelihood of interruptions and simplify the communication pattern of the parallel LBM simulation. This pattern is illustrated in figure 7(b) and involves transfers between independent axial pairs of nodes in multiple steps. For diagonal synchronization we send data along the axial connections during the existing steps of the schedule.

We also discovered that for simulations with a small number of nodes (less than 16), synchronizing the nodes by calling MPI_barrier() at the end of each scheduled step improves the network performance. However, if more than 16 nodes are used, the increased overhead of the barrier synchronization negates the performance gains from using a synchronized schedule.

The speed of data transfer between GPU and CPU memory along the AGP or the newer PCI-Express buses represents another bandwidth limitation. The velocity distributions that stream out of each sub-domain are stored in different texels and different channels in multiple textures. We have designed fragment programs which are executed during every time step to gather all required data in a single texture. With this approach we minimize the overhead of initializing multiple read operations, and more efficiently utilize the available memory bandwidth.

4.3. Experimental results

We have evaluated the performance of our LBM simulation on implementations using the CPUs and the GPUs in our cluster for the lid-driven cavity flow problem. Note that although each node contains two CPUs, for the purpose of a fair comparison, we used only one simulation thread per node for the computation.

J. Stat. Mech. (2009) P06016

Table 2. Per step execution times (in ms) for the CPU and GPU clusters and the GPU/CPU speedup factor. (Each node computes an 80^3 sub-domain of the lattice.)

		GPU cluster				
Nodes	CPU cluster	Computation	AGP	Network non-O (total)	Total	GPU/CPU speedup
1	1420	214	—	—	214	6.64
2	1424	216	13	0 (38)	229	6.22
4	1430	224	42	0 (47)	266	5.38
8	1429	222	50	0 (68)	272	5.25
12	1431	230	50	0 (80)	280	5.11
16	1433	235	50	0 (85)	285	5.03
20	1436	237	50	0 (87)	287	5.00
24	1437	238	50	0 (90)	288	4.99
28	1439	237	50	11 (131)	298	4.83
30	1440	237	50	25 (145)	312	4.62
32	1440	237	49	31 (151)	317	4.54

In table 2, we report the execution time in milliseconds per simulation step (averaged over 500 steps) for different cluster configurations. Each node evaluates a 80^3 sub-domain and the sub-domains are arranged in two dimensions. Our code also supports efficient sub-domain arrangements in three dimensions. The timing for the CPU cluster simulation (shown in column 2 of table 2) includes only computation time because the network communication time is overlapped with the computation by using a second thread for network communication. The total time for the GPU cluster simulation is shown in column 6 (labeled 'Total') and includes computation time, local memory transfer time across the GPU's AGP bus, and non-overlapping (non-O) network communication time.

Network communication time is plotted as a function of the number of nodes in figure 8(b). GPU computation of the collision operation for inner cells of its sub-domain (which takes approximately 120 ms) is overlapped with the network communication. Column 5 shows the non-overlapped cost along with a total network communication time in parentheses.

Figure 8(a) shows the GPU cluster speedup factor as a function of the number of nodes in the network. When only a single node is used, the speedup factor is 6.64. This value projects the theoretical maximum for the GPU cluster/CPU cluster speedup factor that could be reached under optimal communication conditions. When the number of nodes is below 28, network communication is fully overlapped with the GPU computations.

4.4. Application: dispersion simulation in New York city

As an example application for LBM simulation on a GPU cluster, we have simulated airborne contaminant dispersion in the Times Square area of New York City. The simulation area extends north from 38th Street to 59th Street, and east from 8th Avenue to Park Avenue. It covers an area of about 1.66 km × 1.13 km that consists of 91 blocks and over 850 buildings. Our model features a polygonal mesh with 1 m resolution and synthetic facade textures generated from high resolution photographs. The LBM

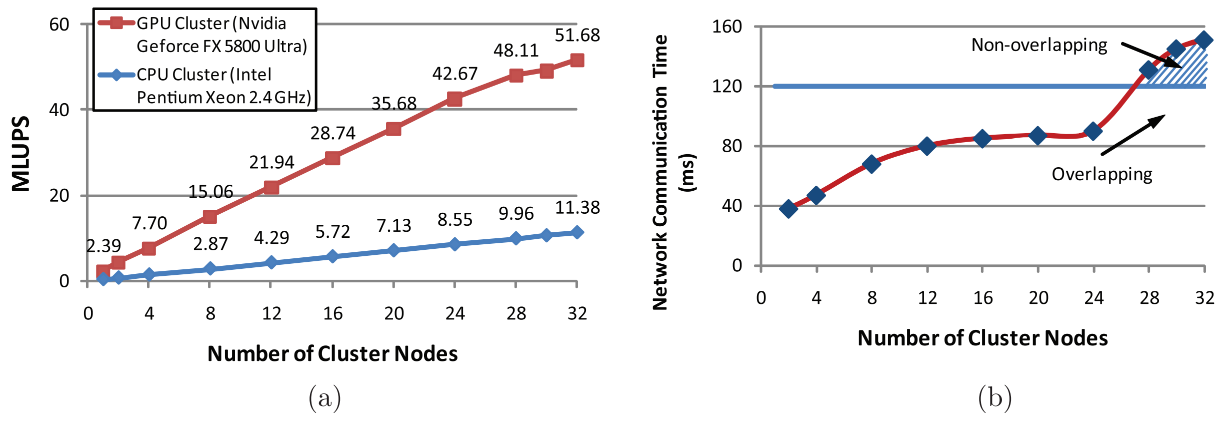


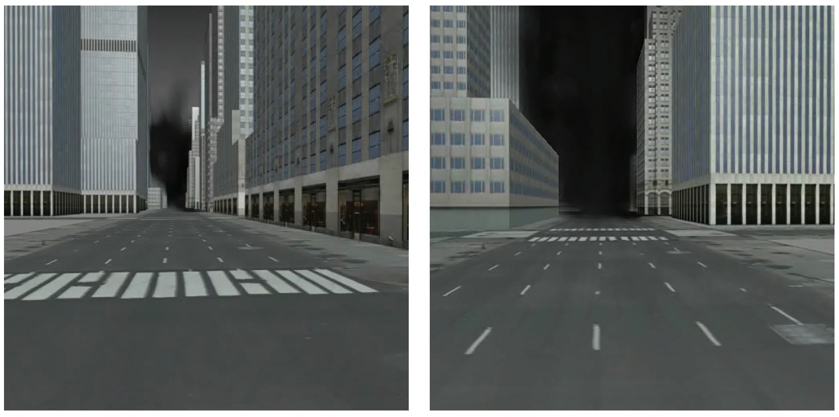
Figure 8. LBM cluster performance on the CPU and the GPU. Each node computes a 80^3 sub-domain and problem size scales with the number of nodes. Each GPU has a theoretical peak performance of 26 GFLOPs, compared to 5 GFLOPs for each CPU. (a) LBM cluster performance in MLUPS. (b) Network communication time for GPU cluster in ms.

can easily accommodate the complex-shape boundaries of urban environments that are characterized by sky-scrapers and deep urban canyons. For large scale simulations of this kind, the combined computational speed of the GPU cluster and the linear nature of the LBM model create a powerful tool that can meet the requirements of both speed and accuracy. Beyond enhancing our understanding of the fluid dynamics processes governing dispersion, the simulation supports the prediction of airborne contaminant propagation so that emergency responders can more effectively engage their resources in response to urban accidents or attacks.

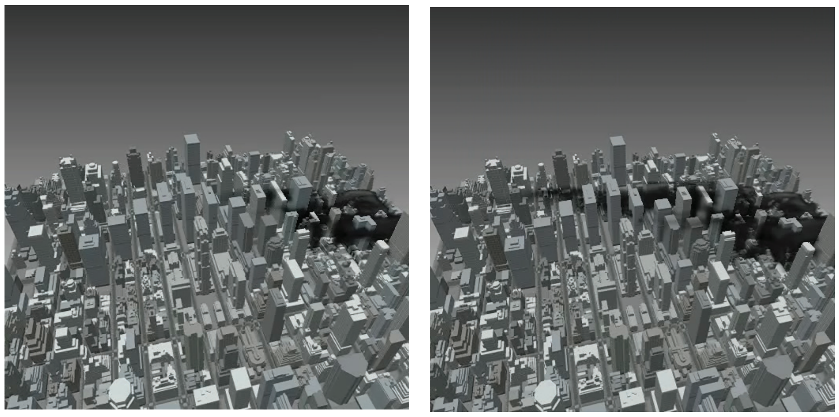
We model the flow using the D3Q19 BGK LBM with a $480 \times 400 \times 80$ lattice. This simulation is executed on 30 nodes of the GPU cluster where each node computes an 80^3 sub-domain. The urban model is rotated to align it with the LBM domain axes and occupies a lattice area of 480×400 at the base of the simulation domain. As a result, the simulation resolution is about 3.8 m per lattice spacing. We simulate a north-eastern wind with a velocity boundary condition on the northern side of the LBM domain. The LBM flow model executes at 0.31 s/step on 30 nodes of the GPU cluster for a total performance of 49.53 MLUPS. The combined theoretical peak performance of the GPU cluster is $30 \times 26 = 780$ GFLOPs. After 1000 steps of the LBM computation, the pollution tracer particles begin to propagate along the LBM lattice links according to transition probabilities obtained from the LBM velocity distributions [9]. Figure 9 shows snapshots of a smoke dispersion simulation in the Times Square area of New York City.

4.5. Application: thermal fluid dynamics in a pressurized water reactor

We present a simulation and visualization system for a critical application—the analysis of the thermal fluid dynamics inside a pressurized water reactor (PWR) of a nuclear power plant when cold water is injected into the reactor vessel. We have worked closely with PWR scientific engineers and have developed a visual simulation system [2] that employs a hybrid thermal lattice Boltzmann method (HTLBM) [7] for modeling of the thermal fluid



(a)



(b)

Figure 9. Smoke dispersion simulated in the Times Square area of New York City. (a) Street-level view. (b) Bird's-eye view.

dynamics. In particular, the simulation demonstrates the formation of cold-water plumes in the reactor vessel. One challenge is that the thermal fluid dynamics need to be computed in the special geometry of an irregularly shaped PWR. When using a straightforward rectangular domain that encompasses the entire input mesh (figure 10), only 5.8% of the lattice cells are relevant to the simulation. Therefore, for efficient computation and memory consumption, we propose an LBM implementation of an irregularly shaped simulation domain on the GPU cluster which packs only the non-empty LBM cells in the GPU texture memory. This approach significantly improves the simulation performance in terms of storage and computation time.

We classify the fluid and empty cells in a preprocessing step. Cells close to the vessel wall are detected by an intersection test of the mesh and the simulation cells. Given these border cells and an interior cell as the seed, a region-growing algorithm based on a depth-first search identifies all interior cells. The interior and border cells are then compactly stored in memory and an indexing table is constructed to translate between cell indices and positions in the simulation domain.

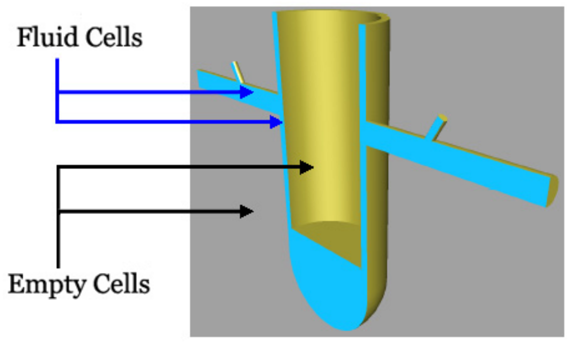


Figure 10. The irregular shape of the reactor vessel. Only 5.8% of the cells are useful if a rectangular region is used as the simulation domain.

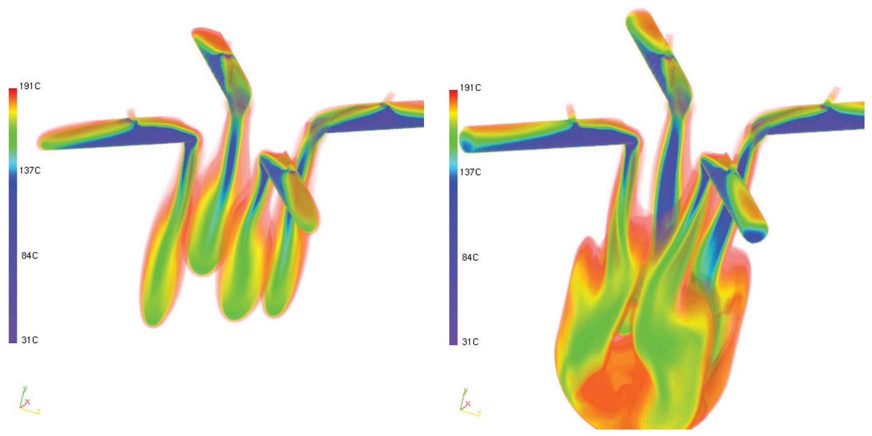
Table 3. Average computation time (in ms) of a single LBM step tested on 12 cluster nodes with and without GPU acceleration.

Resolution	CPU cluster	GPU cluster	Speedup
$800 \times 600 \times 480$	8759	2810	3.1
$600 \times 450 \times 360$	3927	1297	3.0
$400 \times 300 \times 240$	1150	405	2.8

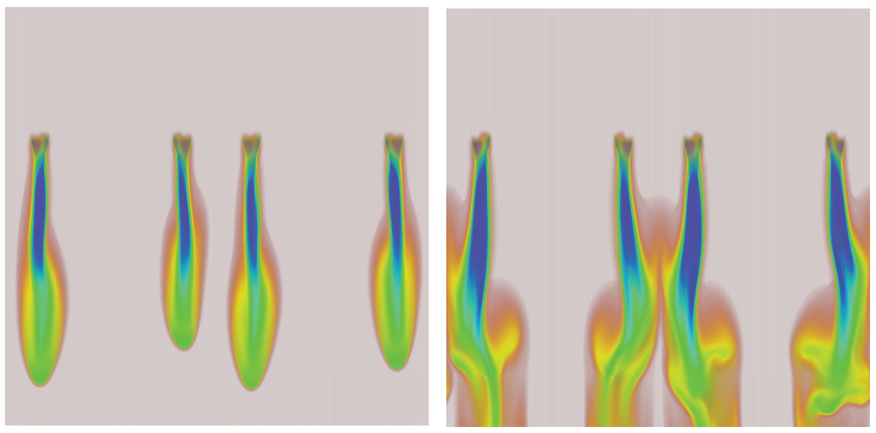
Table 3 reports the simulation performance of our cluster with and without GPU acceleration. Using 12 CPU nodes, the simulation on a $800 \times 600 \times 480$ grid requires 8.8 s per simulation step. The GPU cluster implementation using the same number of nodes executes in 2.8 s per simulation step. Each cluster node is equipped with an NVIDIA Quadro FX 4500 GPU and an Intel Xeon 3.6 GHz CPU. The peak theoretical performance per device is 200 GFLOPs and 7 GFLOPs respectively. A set of interactive visualization tools, such as side-view slices, 3D volume rendering, thermal layers rendering, and panorama rendering, are collectively provided to visualize the structure and dynamics of the temperature field in the vessel, as shown in figure 11. Our GPU-based implementation is not particularly optimized and its performance is presented here only in comparison to the unoptimized CPU version. Also, the irregular grid employed with our hybrid thermal LBM code is less suitable for GPU acceleration since it requires indirect addressing of the simulation domain.

5. Zippy framework for GPU cluster programming

Programming for GPU clusters involves working at a low level which is error-prone, and presents challenges in terms of obtaining optimal performance. The applications presented in section 4.4 and section 4.5 both involve a significant programming effort to achieve good performance in a multi-GPU environment. We have introduced the Zippy framework to simplify application development for GPU-based clusters. Our system employs global arrays (GA) to simplify the communication, synchronization, and collaboration among multiple GPUs. At the same time, it exposes data locality to the programmer to allow improved performance and scalability. In essence, we preserve the advantages of



(a)



(b)

Figure 11. Two steps of the simulation of cold water injected into the reactor vessel. (a) Thermal layers of the heat distribution. (b) Panoramic view of the downcomer.

the established MPI+stream computing and distributed shared memory (DSM) models, respectively.

5.1. Programming model

General-purpose computation on a GPU cluster often involves a two-layer communication among GPUs. If a GPU needs data from the memory of another GPU, the data on the source node are downloaded from the GPU memory to the CPU memory, transferred through the network to the CPU memory of the destination node, and then uploaded to the destination GPU memory. This communication pattern further increases the complexity of programming with MPI. However, we believe that to provide a virtually shared space that eases the communication among multiple GPUs, it is important to make the NUMA (non-uniform memory access) and the two-level parallelism hierarchy explicit to programmers.

Table 4 lists the theoretical bandwidth and latency of the GPU memory, PCI-Express bus, and the network. The aggregate performance numbers for memory access

Table 4. Theoretical bandwidth and latency in a GPU cluster, which demonstrates a NUMA characteristic.

	Bandwidth (GB s^{-1})	Latency (μs)
GPU memory	100	0.05
PCI-Express	4	0.5
Infiniband	2.5	1
GigaE	0.1	50
Remote GPU memory on Infiniband cluster	2.5	2
Remote GPU memory on GigaE cluster	0.1	51

between GPUs are shown in the last two rows and are especially significant for our applications. In comparison to the local GPU memory case, the bandwidth is 2–3 orders of magnitude lower and the latency is 2–4 orders of magnitude higher. In addition, due to hardware limitations, a GPU cannot transfer data through the PCI-Express bus during computations, which further increases memory access latency. Because of this NUMA characteristic of the GPU cluster, our Zippy API differentiates between the shared space and the local space and exposes data locality to the programmer which potentially allows optimal performance. In particular, to alleviate the effects of the high remote GPU memory access latency, shared space should be accessed only with coarse-grained operations such as block copy, and stream computing kernels should only operate in the local space. This is in contrast to the DSM model, in which GPU computations can implicitly trigger inter-GPU communication.

Because of the large performance difference between local and remote memory transfers, Zippy abstracts the GPU cluster with a two-level parallelism hierarchy. The coarse level in the hierarchy supports the collaboration between multiple GPUs using the GA model [13]. This model has seen numerous applications in the parallel computing community, including matrix multiplication, computational chemistry and physics, and electronic structure. The GA data structure and a set of shared memory style functions simplify data transfers between cluster nodes in the GA space. In addition, GA acknowledges the NUMA characteristic of distributed memory architectures and exposes to the programmer data locality and the management of communications. The original GA toolkit was proposed for PC clusters and supercomputers. We have implemented the two-level parallelism hierarchy through a new object-oriented library that encapsulates and abstracts the low-level details of GPU computations and the required two-layer communication.

At the fine level in the hierarchy, existing GPGPU toolkits based on stream computing, such as Cg [10], GLSL [17], HLSL [19], Brook [1], Sh [11], NVIDIA CUDA [14], and RapidMind, provide efficient ways to develop computation kernels. Unlike the MPI+stream computing method, the Zippy API seamlessly integrates the fine-grained parallelism supported by these toolkits with the coarse-grained parallelism of the GPU cluster. This is possible because arrays are the fundamental data structures in GA and also a natural data container for stream computing. In fact, our framework eliminates the need for manual management of threads, handling of individual GPUs and performing

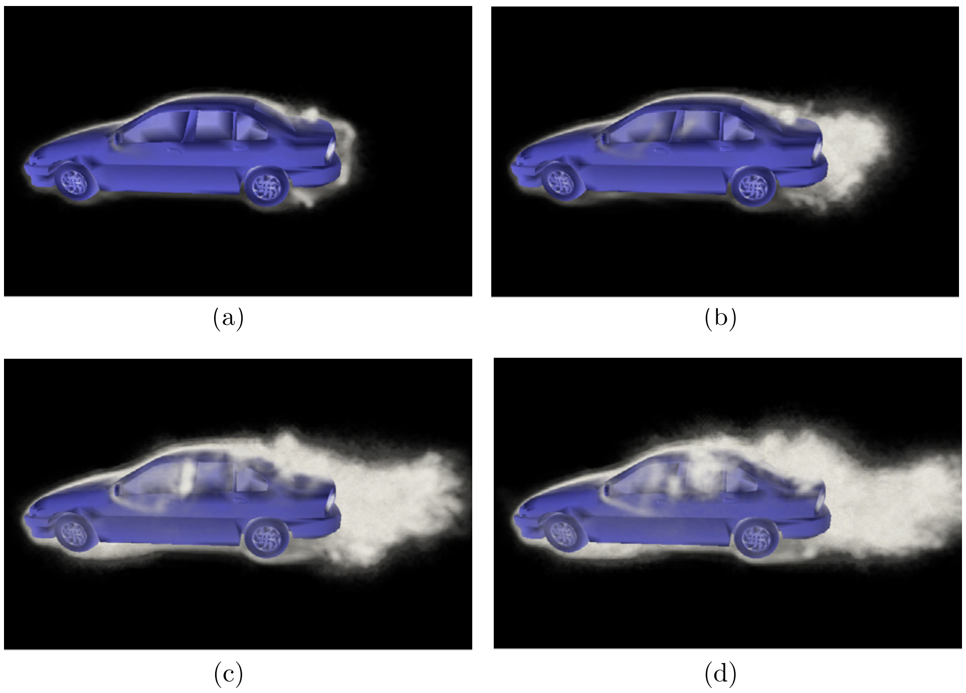


Figure 12. A sequence of snapshots of the LBM flow simulation. Turbulence in the system is visualized using volume rendering of the vorticity magnitude.

inter-GPU data transfers. Although kernels in Zippy are currently implemented in the Cg language, which is a C-like, graphics-oriented shading language, the resulting programming model is general enough that it can support a variety of data-parallel applications outside the graphics field. Target applications for the framework include LBM, linear algebra and medical imaging.

5.2. Experimental results

We have implemented the GPU-based LBM simulation described in section 4 using the Zippy framework. The program is tested on our Gigabit Ethernet cluster using the cluster nodes that contain a NVIDIA Quadro FX 4500 GPU on the PCI-Express bus and dual Intel Xeon 3.6 GHz CPUs. The simulation operations are defined in the global space and each GPU operates on a local chunk of the simulation domain. Each local chunk contains a layer of ghost cells defined with built-in constructs of the Zippy API and communication between the computational nodes relies on non-blocking update function calls. Our initial single-GPU implementation contains 750 lines of C++ code and 550 lines of Cg code. The global array support and the ghost cell update functions require only 100 additional lines of C++ code. The simulation and visualization modules are both developed using Zippy and use common data structures, which avoid costly memory copy operations during rendering. Figure 12 displays snapshots of the simulation running on a $400\times200\times200$ grid. For an image size of 800^2, the overall speed is about 4.5 frames s^{-1}, including rendering, on a cluster of 16 GPUs. The LBM performance is 80 MLUPS. Compared to our previous MPI-based implementation on the same GPU cluster, the new implementation achieves

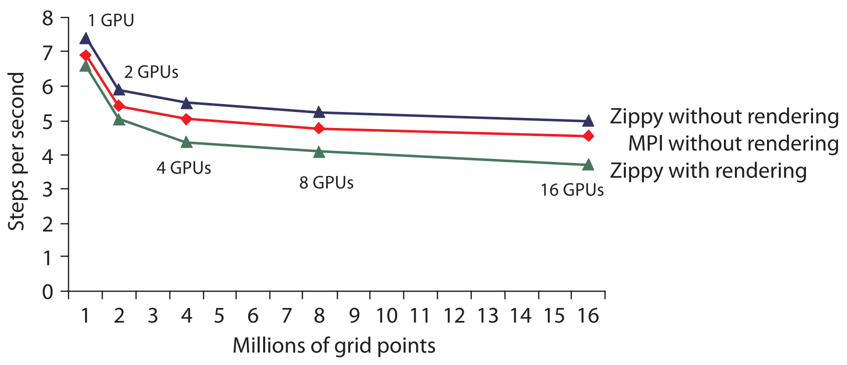


Figure 13. Performance comparison between the MPI and Zippy implementations on the same GPU cluster. Each GPU manages a 100^3 sub-grid and the problem size increases with the number of GPUs.

better performance, as shown in figure 13, with significantly reduced programming effort.

Details of the implementation and additional analysis of the Zippy API are available in Fan *et al* [3]. This paper also presents performance analysis for volume rendering and iso-surface extraction applications implemented on the GPU cluster with our framework.

6. LBM implementation with CUDA

6.1. Programming model

General programming for a GPU architecture requires intimate knowledge of the graphics pipeline and associated graphics primitives, which hinders the efforts of developers who lack this background. NVIDIA CUDA (compute unified device architecture) is a framework that abstracts supported GPUs using the standard C language with extensions for stream computing. These extensions allow for example the definition of device versus host routines and the execution of computational kernels in parallel. The framework also includes optimized FFT and BLAS implementations.

The GPU is a compute device that (1) is a coprocessor with the CPU or host, (2) has its own on-board device memory, and (3) can execute many threads in parallel. Data-parallel algorithms are defined in the form of kernels, which run in parallel over the threads. The GPU threads are very lightweight compared to CPU threads with very little creation overhead, since the GPU must keep 100s of threads in flight to efficiently utilize its available computational resources. Data are transferred between the host memory and the device memory through a set of highly optimized function calls and the framework supports a number of different memory layouts. For example, data can be copied to 'linear global memory', which supports read–write access, but the memory access is not cached. In contrast, read-only 'texture memory' supports linear data interpolation and may offer higher bandwidth due to caching.

At the lowest level, kernel executions in CUDA are grouped into *thread blocks*, in which the individual threads can synchronize their execution using a sync barrier and share data through very fast per-block shared memory. The size of the thread blocks can be defined with 1D, 2D or 3D vectors depending on application requirements and each thread is assigned a thread ID that can be queried at run time. At a higher level, individual thread blocks are grouped into grids, whose size can again be defined using a multi-dimensional vector. This hierarchical organization can greatly simplify the processing of multi-dimensional data. More information, including code examples, is available in the NVIDIA CUDA Programming Guide [14].

6.2. LBM with CUDA

Our CUDA implementation of LBM is built as an extension to the CPU-based framework. The simulation involves the lid-driven cavity flow problem on the D3Q19 LBM. During initialization, memory is allocated on the GPU and all LBM control structures are copied to constant memory. The host is also responsible for initializing the geometry array and the two distributions arrays. Thread blocks are defined so that the threads in each block access elements along a single axis-aligned line of the simulation domain. Such an organization allows for efficient reading and writing of the density values [22]. The computation grid is defined using the remaining two dimensions of the domain. Listing 1 contains the relevant parts of the host code.

Listing 1. Sections of the host code for LBM.

```
struct Distributions {
        float* f [19] ;
} ;

int3 domainSize ;
Distributions mDensities, mDensitiesNew ;
int* mGeometry ;

__constant__ float3 D3Q19Links[19] ;    // velocity vectors
__constant__ int3 D3Q19Neighbors[19] ; // neighbor offset indices
__constant__ float D3Q19Weights[19] ;  // velocity weights for equilibrium calculations
__constant__ float D3Q19Omega ;         // relaxation constant
...

void OneStepCUDA ()
{
    // Configure the grid
    dim3 blockSize (domainSize.x , 1, 1) ;
    dim3 gridSize (domainSize.y, domainSize.z , 1) ;

    // Execute the LBM kernel
    LBMOneStep<<<gridSize, blockSize>>> (domainSize, mGeometry, mDensities, mDensitiesNew) ;

    // Swap distributions arrays
}
```

Our CUDA kernel combines the collision and propagation steps of the LBM and its implementation is a straightforward extension of our CPU framework. Most of the code is reused from the CPU kernel and the resulting application achieves performance that is an order of magnitude higher than that of our CPU application. The main performance bottleneck is the streaming step, during which densities are copied to neighboring lattice sites. Because of the memory layout and grid configuration, all memory writes with offsets $(0, 0, 0)$, $(0, 0, \pm 1)$, $(0, \pm 1, 0)$, and $(0, \pm 1, \pm 1)$ are coalesced and executed very efficiently.

However, the remaining 10 propagations are offset along the x direction, which forces writes to misaligned memory locations within a single thread block. These non-coalesced memory transactions may reduce memory bandwidth by an order of magnitude. The hardware capabilities of a CUDA device are described in terms of computation capability versions, which are listed in the NVIDIA CUDA Programming Guide [14]. In general, devices with compute capability 1.0 and 1.1 can coalesce memory transactions only when threads within a block access memory words sequentially.

We employ shared memory during the propagation step, which achieves coalesced memory writes for all propagation operations and greatly improves the computational throughput. Each thread copies the critical densities to the appropriate offset in the allocated shared memory instead of the global memory. After the synchronization barrier, each thread within the block writes to global memory at sequentially increasing memory offsets, which achieves coalescing of the memory writes. Listing 2 illustrates the use of shared memory within the CUDA kernel.

Listing 2. CUDA kernel for LBM collision and propagation.

```
__global__ static void LBMOneStep (/*...*/)
{
    float f [19] ;   // Storage for the distributions of the current lattice site

    // Load and process the current lattice site (perform relaxation, apply boundary conditions)
    // ...

    // Allocate shared memory for streaming
    __shared__ float sharedf1[THREAD_COUNT+2] ;
    __shared__ float sharedf2[THREAD_COUNT+2] ;
    // ...

    // Copy critical densities to shared memory
    sf1 [tx+2] = f [1] ;
    sf2 [tx] = f [2] ;
    // ...

    // Synchronize thread execution so that access to shared memory is hazard-free
    __syncthreads () ;

    // Propagate densities for which the velocity is perpendicular to the x-axis (writes are coalesced)
    (densitiesNew.f [0])[ GetStorageIndex (GetNeighbor(pos3D, dims, 0), dims)] = f[0] ;
    (densitiesNew.f [3])[ GetStorageIndex (GetNeighbor(pos3D, dims, 3), dims)] = f[3] ;
    // ...

    // Propagate remaining densities using shared memory
    (densitiesNew.f [1])[ pos1D] = sharedf1[tx+1] ;
    (densitiesNew.f [2])[ pos1D] = sharedf2[tx+1] ;
    // ...
}
```

As an additional optimization, we allocate single float variables instead of float arrays in the CUDA kernel, which ensures that the variables are kept in registers instead of spilling into local memory. This optimization may not be feasible for more complex simulations or larger domain sizes. Table 5 compares the performances of our straightforward implementation and the optimized version.

Table 5. The effect of memory access optimizations on CUDA performance, expressed in MLUPS.

	Unoptimized				Optimized			
CUDA device	32^3	64^3	96^3	128^3	32^3	64^3	96^3	128^3
NVIDIA Quadro FX 4600 (CUDA)	28	33	32	20	101	144	127	134
NVIDIA Tesla C1060 (CUDA)	59	64	60	46	232	240	278	233

Table 6. LBM performance in MLUPS for different architectures and domain sizes.

	Domain size				
Architecture	16^3	32^3	64^3	96^3	128^3
Intel Core Duo 1.86GHz	0.6	0.5	0.5	0.5	0.5
Intel Xeon E5420	2.0	1.7	1.4	1.4	1.4
NVIDIA Quadro FX 2500M (Zippy)	2.2	15	25	24	24
NVIDIA Quadro FX 4600 (Zippy)	6.4	37	94	91	89
NVIDIA Quadro FX 4600 (CUDA)	—	101	144	127	134
NVIDIA Tesla C1060 (CUDA)	—	232	240	278	233

6.3. Experimental results

We have compared the performance of our Zippy-based implementation of LBM against that of the new code based on CUDA. In both cases we have implemented the SRT-LBM on the D3Q19 Cartesian lattice and the applications do not perform any data collection or rendering of the resulting velocity fields. We have also compared the performance against that of a CPU implementation of LBM in C++, which involves a similar computational load without any particular optimizations to the computation kernel or the memory layout. We have implemented multi-threading using OpenMP directives on the main computational loop.

Our benchmarks were conducted across dual-core and quad-core CPU systems and across three generations of graphics processors. The hardware includes an Intel Core Duo 1.86 GHz machine with an NVIDIA Quadro FX 2500M GPU (230 GFLOPs theoretical peak performance) and an Intel Xeon Quad-core 2.5 GHz workstation with an NVIDIA Quadro FX 4600 and an NVIDIA Tesla C1060 board (40 GFLOPs, 518 GFLOPs and 933 GFLOPs theoretical peak performance respectively). Table 6 lists the performances of our implementations in MLUPS. Figure 14 illustrates the performance scaling of the GPU and CPU architectures for different domain sizes.

It is clear that a CUDA implementation of LBM provides an increase in performance over an implementation that relies on the graphics pipeline. In particular, performance for relatively small problem sizes is improved dramatically. Nevertheless, efficient use of the GPU requires a sufficient number of threads per block and at least 100 blocks per grid [14]. This is in contrast to the case for CPUs, where the increased domain size lowers the effectiveness of memory caching and thus lowers performance. Another difference is that the GPU is generally a memory-bound architecture, in terms of both memory bandwidth and memory size. Obtaining high memory throughput generally involves the use of the

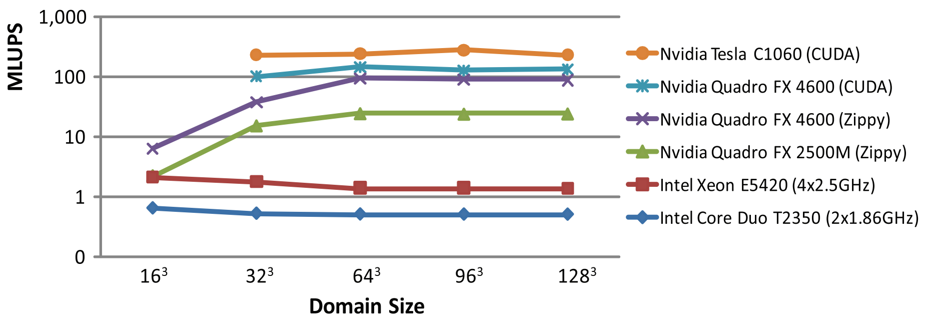


Figure 14. Performance scaling for CPU and GPU architectures.

fast shared memory, which is a limited resource per multiprocessor. Additional memory constraints are placed on the number of available registers per block and the amount of global memory. The limited scalability of GPUs also has broad implications for the use of GPU clusters. Even when using CUDA as the back-end computation interface, the limited range of efficient problem sizes translates into poor scalability for fixed-size problems. On the other hand, GPU clusters can exhibit close to linear performance increase when the problem size grows with the introduction of new GPU nodes (see section 5.2).

One feature of particular interest for scientific computing is the support for extended precision operations under CUDA. In particular, the compute capability 1.3 profile for CUDA hardware introduces support for 64-bit precision computations [14] at the cost of reduced peak throughput. For computationally bound kernels, the stated peak performance of the NVIDIA Tesla C1060 for double-precision computations is 1/12th of the peak single-precision performance. In our optimized LBM kernel, the performance decreases by approximately a factor of 2 down to 121 MLUPS for the 96^3 domain. In our experience, the single-precision arithmetic of the GPU architecture provides sufficiently accurate and stable results in the LBM for interactive applications.

7. Outlook and future work

Recently, major GPU vendors have targeted the high performance computing (HPC) market. For example, NVIDIA has announced the Tesla S1070 which is a 1U rack-mount server with 4 GPUs dedicated to general-purpose computations. Both the NVIDIA Tesla S1070 and the AMD FireStream 9170 support double-precision floating point arithmetic which addresses a major limitation of older hardware in terms of GPGPU applications. Researchers have accelerated a wide range of general-purpose applications on commodity GPUs, but a GPU cluster is necessary to be competitive against large scale hardware.

Beside the GPU, there have been other multi-core architectures that are populating HPC systems. Examples are the multi-core CPU, Cell BE, and ClearSpeed (see table 7). In addition, Intel introduced the Larrabee many-core visual computing architecture at Siggraph 2008 [18], which employs multiple x86 cores for computer graphics and other highly parallel workloads. As processor parallelism is becoming more ubiquitous, LBM and other discrete simulation techniques will benefit from the emerging multi-core era.

Compared with the GPUs, the CPUs are more flexible and can support a wider range of applications at the cost of greatly increased chip complexity. Specifically, programs that require complicated control flows and large data caches to achieve optimal performance

Table 7. Performance characteristics on GPUs and other multi-core processors. (Thermal design power (TDP) refers to the maximum amount of thermal power that the processors may potentially dissipate under load.)

			Theoretical GFLOPS		
	Number of cores	Clock (GHz)	32-bit	64-bit	TDP (W)
GeForce GTX 280	240	1.3	933	117	236
GeForce 9800 GTX	128	1.7	653	N/A	156
Quad-core xeon	4	3.2	102	51	150
Cell BE	$8+1$	3.2	205	14.6	110
ClearSpeed Advance e710	192	0.25	96	96	25

are better suited for CPU-based implementations. On the other hand, contemporary GPUs have a significantly larger number of cores and devote a higher percentage of their transistors to floating point operations. Therefore, the GPU provides higher computational parallelism and delivers better performance than the CPU for certain applications. The LBM computation is an example of this kind, as the computation kernels are local and linear. As both CPUs and GPUs have their advantages and disadvantages, a hybrid CPU + GPU node architecture may be a promising future direction. Whether the CPU and GPU should be loosely or tightly coupled is an arguable point that requires further research.

In terms of GPU programming, the transition from C code to CUDA is fairly straightforward, although additional optimizations are often required to achieve optimal GPU performance. In many cases, the optimizations involve the non-trivial use of the fast shared memory for more efficient transactions to the uncached global memory. On the other hand, implementations using the Zippy framework with kernels based on Cg require a more substantial rewrite of the original code. The advantage is that the final applications provide good performance on a single GPU and scale well on GPU clusters. In addition, the back-end of the Zippy framework can be implemented with CUDA, leveraging the ease of writing computation kernels in CUDA instead of Cg, the increased performance, and the support for the new features of emerging GPU architectures.

Hybrid clusters with multiple CPUs and multiple GPU at each cluster node may also be a trend for future high performance computing. Currently, our GPU cluster implementations use the CPU only for network communication and GPU management; the computational power of the CPU, especially in multi-core systems, has not been fully utilized. A possible solution is to combine the vertical communication of Sequoia [5] and the horizontal communication of Zippy, which would allow the programmer to exploit multiple memory levels on the cluster, including GPU memories, system memories, and disks. Specifically, the global arrays can be defined at any level so that the computational power of both the CPU and GPU architectures can be exploited. In addition, multiple GPUs can work in parallel on each node in the cluster. The communication costs within a single node are lower than network communication costs. Therefore, designing and programming the GPU cluster as a three-level parallelism architecture will further increase the scalability in terms of the number of GPUs.

The GPU cluster is a bandwidth-starved architecture. Our experiments have shown that network communication is a major performance bottleneck even with high computational density kernels. We have been implementing a higher bandwidth and lower latency network, such as Infiniband, for our GPU cluster. Furthermore, GPU-based compression and decompression algorithms can reduce the data transfer sizes. As computation is relatively inexpensive on the GPU architecture, we can trade computation for communication performance by using a real-time compression/decompression scheme.

Acknowledgments

This work was supported by grants from NSF (CCF-0702699), from the US Nuclear Regulatory Commission (NRC) through Information Systems Laboratories, Inc. (ISL), and from the Environment Measurement Lab (EML) of the Department of Homeland Security.

References

[1] Buck I, Foley T, Horn D, Sugerman J, Fatahalian K, Houston M and Hanrahan P, *Brook for GPUs: stream computing on graphics hardware*, 2004 *ACM Trans. Graph.* **23** 777
[2] Fan Z, Kuo Y, Zhao Y, Qiu F, Kaufman A and Arcieri B, *Visual simulation of thermal fluid dynamics in a pressurized water reactor*, 2009 *The Visual Computer* online first doi:10.1007/s00371-008-0309-x
[3] Fan Z, Qiu F and Kaufman A, *Zippy: a framework for computation and visualization on a GPU cluster*, 2008 *Comput. Graph. Forum* **27** 341
[4] Fan Z, Zhao Y, Kaufman A and He Y, *Adapted unstructured LBM for flow simulation on curved surfaces*, 2005 *ACM SIGGRAPH/EUROGRAPHICS Symp. on Computer Animation* pp 245–54
[5] Fatahalian K, Knight T J, Houston M, Erez M, Horn D R, Leem L, Park J Y, Ren M, Aiken A, Dally W J and Hanrahan P, *Sequoia: programming the memory hierarchy*, 2006 *ACM/IEEE Supercomputing Conf.* p 4
[6] Harris M, Baxter W V, Scheuermann T and Lastra A, *Simulation of cloud dynamics on graphics hardware*, 2003 *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware* pp 92–101
[7] Lallemand P and Luo L-S, *Theory of the lattice Boltzmann method: acoustic and thermal properties in two and three dimensions*, 2003 *Phys. Rev.* E **68** 036706
[8] Li W, Wei X and Kaufman A, *Implementing lattice Boltzmann computation on graphics hardware*, 2003 *Vis. Comput.* **19** 444
[9] Lowe C P and Succi S, 2002 *Go-with-the-Flow Lattice Boltzmann Methods for Tracer Dynamics* (Berlin: Springer)
[10] Mark W R, Glanville R S, Akeley K and Kilgard M J, *Cg: a system for programming graphics hardware in a C-like language*, 2003 *ACM Trans. Graph.* **22** 896
[11] McCool M, Du Toit S, Popa T, Chan B and Moule K, *Shader algebra*, 2004 *ACM Trans. Graph.* **23** 787
[12] Nguyen H, 2007 *GPU Gems 3* (Reading, MA: Addison-Wesley)
[13] Nieplocha J, Palmer B, Tipparaju V, Krishnan M, Trease H and Apra E, *Advances, applications and performance of the global arrays shared memory programming toolkit*, 2006 *Int. J. High Perform. Comput. Appl.* **20** 203
[14] NVIDIA, 2008 *Compute Unified Device Architecture Programming Guide* Version 2.0, http://www.nvidia.com/object/cuda_develop.html
[15] Owens J D, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn A E and Purcell T J, *A survey of general-purpose computation on graphics hardware*, 2007 *Comput. Graph. Forum* **26** 80
[16] Pharr M and Fernando R, 2005 *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* (Reading, MA: Addison-Wesley)
[17] Rost R J, 2006 *OpenGL Shading Language* 2nd edn (Reading, MA: Addison-Wesley)
[18] Seiler L *et al*, *Larrabee: a many-core x86 architecture for visual computing*, 2008 *ACM Trans. Graph.* **27**
[19] St-Laurent S, 2005 *The Complete Effect and HLSL Guide* (Redmond, WA: Paradoxal Press)
[20] Succi S, *The lattice Boltzmann equation for fluid dynamics and beyond*, 2001 *Numerical Mathematics and Scientific Computation* (Oxford: Oxford University Press)
[21] Tam A T C and Wang C-L, *Contention-aware communication schedule for high-speed communication*, 2003 *Cluster Comput.* **6** 339

[22] Tölke J and Krafczyk M, *Towards three-dimensional teraflop CFD computing on a desktop PC using graphics hardware*, 2008 *Technical Report* Institute for Computational Modeling in Civil Engineering, TU Braunschweig

[23] Wei X, Zhao Y, Fan Z, Li W, Qiu F, Yoakum-Stover S and Kaufman A, *Lattice-based flow field modeling*, 2004 *IEEE Trans. Vis. Comput. Graph.* **10** 719

[24] Zhao Y, Han Y, Fan Z, Qiu F, Kuo Y, Kaufman A E and Mueller K, *Visual simulation of heat shimmering and mirage*, 2007 *IEEE Trans. Vis. Comput. Graph.* **13** 179

[25] Zhao Y, Qiu F, Fan Z and Kaufman A, *Flow simulation with locally-refined LBM*, 2007 *ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games* pp 181–8

[26] Zhao Y, Wei X, Fan Z, Kaufman A and Qin H, *Voxels on fire*, 2003 *Proc. 14th IEEE Visualization* doi:10.1109/VIS.2003.10009