Comput Visual Sci (2010) 13:29–39
DOI 10.1007/s00791-008-0120-2

REGULAR ARTICLE

Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA

Jonas Tölke

Received: 31 May 2007 / Accepted: 7 February 2008 / Published online: 24 July 2008

Abstract In this article a very efficient implementation of a 2D-Lattice Boltzmann kernel using the Compute Unified Device Architecture (CUDA™) interface developed by nVIDIA® is presented. By exploiting the explicit parallelism exposed in the graphics hardware we obtain more than one order in performance gain compared to standard CPUs. A non-trivial example, the flow through a generic porous medium, shows the performance of the implementation.

1 Introduction

A Graphical Processing Unit (GPU) is specifically designed to be extremely fast at processing large graphics data sets (e.g., polygons and pixels) for rendering tasks. The use of the GPU to accelerate non-graphics computations has drawn much attention [2,3,14]. This is due to the fact that the computational power of GPUs has exceeded that of PC-based CPUs by more than one order of magnitude while being available for a comparable price. For example the recently released nVIDIA GeForce 8800 Ultra has been observed to deliver over 4×10^{11} single precision (32-bit) floating operations per second (400 GFLOPS) [18]. In comparison, the theoretical peak performance of the Intel Core 2 Duo 2.4 GHz is only 19.2 GFLOPS for double and 38.4 GFLOPS for single precision. Also the bandwidth to the memory interface is much larger: Memory bandwidth for desktop computers ranges from 5.3 to 10.7 GB/s, whereas the nVIDIA GeForce 8800 Ultra delivers up to 104 GB/s.

Communicated by G. Wittum.

J. Tölke (✉)
Institute for computer based modeling in civil engineering,
TU Braunschweig, Pockelstr. 3, 38106 Braunschweig, Germany
e-mail: toelke@cab.bau.tu-bs.de

Due to the facts that Lattice Boltzmann (LB) methods operate on a finite difference grid, are explicit in nature and require only next neighbor interaction they are very suitable for the implementation on GPUs. In [16] the computation of the LBM is accelerated 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. A speedup of at least one order of magnitude could be achieved compared to an implementation on a CPU. Applications for LB simulations in graphics hardware range from real-time ink dispersion in absorbent paper [5], dispersion simulation and visualization for urban security [20], simulation of soap bubbles [24], simulation of miscible binary mixtures [29], melting and flowing in multiphase environment [28] and visual simulation of heat shimmering and mirage [27]. Even GPU clusters have been assembled for general-purpose computation [8] and LB simulation have been performed. An implementation of a Navier–Stokes solver on a GPU can be found in [26]. Nevertheless all these applications use a programming style close to the hardware especially developed for graphics applications.

The remainder of the paper is organized as follows: In Sect. 2 the graphics hardware is shortly sketched, in Sect. 3 the Compute Unified Device Architecture (CUDA) programming technology is presented, in Sect. 4 the D2Q9 LB model is described, in Sect. 5 the implementation of this model using CUDA is described, Sect. 6 discusses the performance of the approach, Sect. 7 gives an example and Sect. 8 concludes the paper and gives a short outlook.

2 nVIDIA—G80: the parallel stream processor

The G80-chip on a nVIDIA 8800 Ultra graphics card has 16 multiprocessors with 8 processors each, for a total of 128

processors. These are generalized floating-point processors capable of operating on 8-,16- and 32-bit integer types and 16- and 32-bit floating point types. Each multiprocessor has a memory of 16 KB size that is shared by the processors within the multiprocessor. Access to a location in this shared memory has a latency of only 2 clock cycles allowing fast nonlocal operations. The processors are clocked (Shader Clock) at 1.6 GHz, giving the GeForce 8800 Ultra a tremendous amount of floating-point processing power. Assuming 2 floating point operations per cycle (one addition and multiplication) we obtain $2 \times 1.6 \times 128$ GFLOPS = 410 GFLOPS. Each multiprocessor has a Single Instruction, Multiple Data architecture (SIMD).

The multiprocessors are connected by a crossbar-style switch to six Render Output Unit (ROP) partitions. Each ROP partition has its own L2 cache and an interface to device memory that is 64-bits wide. In total, that gives the G80 a 384-bit path to memory with a clock frequency of 1100 MHz. This results in a theoretical memory bandwidth of $384/8 \times 1.1 \times 2$ (DDR) GB/s = 104 GB/s. In practice 80% of this value can be achieved for simple copy throughput. The transfer rates over the PCI-E bus are dependent on the system configuration. Assuming PCI-Ex16, the transfer speed is 1.5 GB/s for peagable memory and 3.0 GB/s for pinned memory. The available amount of memory is 768 MB. The nVIDIA Quadro GPUs deliver memory up to 2 GB. There is also new product line called "NVIDIA Tesla" (also based on the G80 chip) especially designed for High performance computing.

3 nVIDIA CUDA

3.1 Introduction

The nVIDIA CUDA technology [18] is a fundamentally new computing architecture that enables the GPU to solve complex computational problems. Compute Unified Device Architecture (CUDA) technology gives computationally intensive applications access to the processing power of nVIDIA graphics processing units (GPUs) through a new programming interface. Software development is strongly simplified by using the standard C language. The CUDA Toolkit is a complete software development solution for programming CUDA-enabled GPUs. The Toolkit includes standard FFT and BLAS libraries, a C-compiler for the nVIDIA GPU and a runtime driver. CUDA technology is currently supported on the Linux and Microsoft Windows XP operating systems. We used the version 1.1 for the implementation.

3.2 Application programming interface (API)

In this subsection only a small subset of the API following [18] needed for the LB kernel is discussed. The GPU is viewed as a compute device capable of executing a very high number of threads in parallel. It operates as a coprocessor to the main CPU called host. Data-parallel, compute-intensive portions of applications running on the host are off-loaded onto the device by using a function that is executed on the device as many different threads. Both the host and the device maintain their own DRAM, referred to as host memory and device memory, respectively. One can copy data from one DRAM to the other through optimized API calls that utilize the device's high-performance Direct Memory Access (DMA) engines.

Thread Block. A thread block is a batch of threads that can cooperate together by efficiently sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses by specifying synchronization points in the kernel. Each thread is identified by its thread ID, which is the thread number within the block. An application can also specify a block as a three-dimensional array and identify each thread using a 3-component index. The layout of a block is specified in a function call to the device by a variable type dim3, which contains three integers defining the extensions in x,y,z. If one integer is not specified, it is set to one. Inside the function the built-in global variable blockDim contains the dimensions of the block. The built-in global variable threadIdx is of type uint3 (also a type composed of three integers) and contains the thread index within the block. To exploit the hardware efficiently a thread block should contain at least 64 threads and not more than 512.

Grid of thread blocks. There is a limited maximum number of threads (in the current CUDA Version 512) that a block can contain. This number can be smaller due to the amount of local and shared memory used. However, blocks that execute the same kernel can be batched together into a grid of blocks, so that the total number of threads that can be launched in a single kernel invocation is much larger. This comes at the expense of reduced thread cooperation, because threads in different thread blocks from the same grid cannot communicate and synchronize with each other. Each block is identified by its block ID. An application can also specify a grid as a two-dimensional array and identify each block using a 2-component index. The layout of a grid is specified in a function call to the device by a variable type dim3, which contains two integers defining the extensions in x,y. The third integer is set to one. Inside the function the built-in global variable gridDim contains the dimensions of the grid. The built-in global variable blockIdx is of type uint3 and contains the block index within the grid. The different blocks of a grid can run in parallel and to exploit the hardware efficiently at least 16 blocks per grid should be used. For future devices this value may increase. The present upper limit for the number of blocks is 65535 in each dimension.

Function type qualifiers.

- The _device_ qualifier declares a function that is executed on the device and callable from the device only.
- The _global_ qualifier declares a function as being a kernel. Such a function is executed on the device and callable from the host only. Any call to a _global_ function must specify the execution configuration for that call. The execution configuration defines the dimension of the grid and blocks that will be used to execute the function on the device. It is specified by inserting an expression of the form <<< Dg, Db>>> between the function name and the parenthesized argument list, where Dg is of type dim3 and specifies the dimension and size of the grid, such that Dg.x × Dg.y equals the number of blocks being launched. Db is also of type dim3 and specifies the dimension and size of each block, such that Db.x × Db.y × Db.z equals the number of threads per block;
- The _host_ qualifier declares a function that is executed on the host and callable from the host only.

Variable type qualifiers.

- The _device_ qualifier declares a variable that resides in global memory space of the device. It is accessible from all the threads within the grid (with a latency of about 200–300 clock cycles) and from the host through the runtime library.
- The _shared_ qualifier declares a variable that resides in the shared memory space of a thread block and is only accessible from all the threads within the block (but with a latency of only 2 clock cycles).

Memory management.

- cudaError_t cudaMalloc(void** devPtr, size_t count) allocates count bytes of linear memory on the device and returns in *devPtr a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable.
- cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind) copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind is one of
  - cudaMemcpyHostToHost,
  - cudaMemcpyHostToDevice,
  - cudaMemcpyDeviceToHost or
  - cudaMemcpyDeviceToDevice

  and specifies the direction of the copy.

Both functions can only be called on the host.

Synchronization. The function void _syncthreads() synchronizes all threads in a block. Once all threads have reached this point, execution resumes normally. This function can only be used in device functions.

3.3 Memory bandwidth

The effective bandwidth of each memory space depends significantly on the memory access pattern. Since device memory is of much higher latency and lower bandwidth than on-chip shared memory, device memory accesses should be arranged so that simultaneous memory accesses of one block can be coalesced into a single contiguous, aligned memory access. This means that each block thread number N should access element N at byte address BaseAddress + sizeof(type)*N, where N starts from zero and sizeof(type) is equal to 4, 8, 16. Moreover BaseAddress should be aligned to 16*sizeof(type) bytes, otherwise memory bandwidth performance breaks down to about 10 GB/s [23]. Any address of a variable residing in global memory or returned by one of the memory allocation routines is always aligned to satisfy the memory alignment constraint.

3.4 Small example

In this subsection a simple example is sketched. Each element of a float matrix of size nx,ny is initialized with 1.0 and then multiplied by 0.5. The data layout is done in a way that a good performance can be achieved. The elements of the matrix are stored in a one-dimensional array of size nx × ny, where the access pattern for element (x,y) in the 1d-array is k = nx × y + x. We collect our data for thread processing along the x-axis for contiguous memory access. The layout of each block is (num_threads, 1, 1) so each block contains num_threads elements and represents just a part of one line of the matrix. The grid of blocks is defined as (nx/num_threads,ny). In this simple configuration nx,ny and num_threads have to be a multiple of 16 and nx ≥ num_threads.

Excerpts of the host code read as follows:

```
...
// allocation of host memory
float* fH  = (float*) malloc
                (mem_size_Mat);
// initialize host memory
for(y=0 ; y< ny ; y++){
 for(x=0 ; x< nx ; x++){
  k = nx*y+x;
  fH[k]=1.0;
 }
}
// allocate device memory
cudaMalloc((void**) &f0, mem_size_Mat);
```

```
cudaMalloc((void**) &f1, mem_size_Mat);
// copy host memory to device
cudaMemcpy(f0,  fH,  mem_size_Mat,
  cudaMemcpyHostToDevice);
// setup execution parameters
dim3 threads(num_threads, 1, 1);
dim3 grid(nx/num_threads, ny);
//Execute the kernel
Kernel<<< grid, threads >>>
    ( nx, f0,f1);
...
```

The device code (kernel) is very simple and reads as follows:

```
_global_ void Kernel(int nx,float* f0,
    float* f1)
{
  // number of threads
  int num_threads = blockDim.x;
  // Thread index
  int tx = threadIdx.x;
  // Block index x
  int bx = blockIdx.x;
  // x-Index
  int x = tx +  bx*num_threads;
  // Block index y = y-Index
  int y =  blockIdx.y;
  // f0[k]:Load data from device memory
  // f1[k]:Write data to device memory
  int k = nx*y + x;
  f1[k]=0.5*f0[k];
}
```

With this setup a performance of 72 GB/s is achieved corresponding to 70% of the theoretical maximum bandwidth. With further optimizations (essentially using a strided loop for the line segments to avoid function calls) it is possible to obtain 87 GB/s corresponding to 83% of the maximum bandwidth.

4 Lattice Boltzmann method

The Lattice Boltzmann method is a numerical method to solve the Navier–Stokes equations [1,4,9], where particle distribution functions (mass fractions) propagate and collide on a regular grid. In the following $\mathbf{x}$ represents a two-dimensional vector in space and $\boldsymbol{f}$ a b-dimensional vector, where b is the number of microscopic velocities. We discuss the $d2q9$ model [19] with the following microscopic velocities,

$$\{\mathbf{e}_i, i = 0, \ldots, 8\} = \begin{Bmatrix} 0 & c & 0 & -c & 0 & c & -c & -c & c \\ 0 & 0 & c & 0 & -c & c & c & -c & -c \end{Bmatrix} \tag{1}$$

generating a space-filling lattice with a nodal distance $\Delta x = c\Delta t$, where c is a constant microscopic velocity and Δt the time step. The lattice Boltzmann equation is

$$f_i(t + \Delta t, \mathbf{x} + \mathbf{e}_i \Delta t) = f_i(t, \mathbf{x}) + \Omega_i, \quad i = 0, \ldots, 8 \tag{2}$$

where f_i are the particle distribution functions with unit $\mathrm{kg\,m^{-3}}$ propagating with speed $\mathbf{e}_i$ and $\boldsymbol{\Omega}$ is the collision operator. The distribution functions are also labeled depending on their direction (**r**est, **e**ast, **n**orth, **w**est, **s**outh, **n**orth**e**ast, **n**orth**w**est, **s**outh**w**est, **s**outh**e**ast) as fr, fe, fn, fw, fs, fne, fnw, fsw, fse.

We use a modified version of the multi-relaxation time (MRT) model [6,7,15,22]. The collision operator is

$$\boldsymbol{\Omega} = \mathsf{M}^{-1}\, \boldsymbol{k}, \tag{3}$$

where M is the transformation matrix given in Appendix A and $\boldsymbol{k}$ is the change of distribution functions in moment space.

The moments $\boldsymbol{m}$ of the distribution functions are given with

$$\boldsymbol{m} = \mathsf{M}\boldsymbol{f} := (\rho, \rho_0 u_x, \rho_0 u_y, e, p_{xx}, p_{xy}, h_x, h_y, \epsilon), \tag{4}$$

where ρ is a density variation, $(\rho_0 u_x, \rho_0 u_y)$ is the momentum and ρ_0 is a constant reference density. The moments e, p_{xx}, p_{xy} of second order are related to the strain rate tensor by

$$\begin{aligned} \partial_x u_x =& \frac{s_e}{4c^2\Delta t}\left(3(u_x^2 + u_y^2) - \frac{e}{\rho_0}\right) \\ &+ \frac{3s_\nu}{4c^2\Delta t}\left(u_x^2 - u_y^2 - \frac{p_{xx}}{\rho_0}\right) \\ \partial_y u_y =& \frac{s_e}{4c^2\Delta t}\left(3(u_x^2 + u_y^2) - \frac{e}{\rho_0}\right) \\ &+ \frac{3s_\nu}{4c^2\Delta t}\left(u_y^2 - u_x^2 + \frac{p_{xx}}{\rho_0}\right) \\ \partial_y u_x + \partial_x u_y =& \frac{3s_\nu}{c^2\Delta t} u_x u_y - \frac{p_{xy}}{\rho_0}, \end{aligned} \tag{5}$$

where s_ν and s_e are relaxation rates. Moments h_x, h_y and ϵ are of third and fourth order. Vector $\boldsymbol{k}$ is given with

$$\begin{aligned} k_0 &= 0 \\ k_1 &= g_x \Delta t \\ k_2 &= g_y \Delta t \\ k_3 = k_e &= -s_e\left(e - 3\rho_0(u_x^2 + u_y^2)\right) \\ k_4 = k_{xx} &= -s_\nu\left(p_{xx} - \rho_0(u_x^2 - u_y^2)\right) \\ k_5 = k_{xy} &= -s_\nu\left(p_{xy} - \rho_0 u_x u_y\right) \\ k_6 = k_{hx} &= -s_h h_x \\ k_7 = k_{hy} &= -s_h h_y \\ k_8 = k_\epsilon &= -s_\epsilon \epsilon, \end{aligned} \tag{6}$$

where $\mathbf{G} = (g_x, g_y)$ is a body force with unit $\mathsf{kg\,s^{-2}\,m^{-2}}$ and s_e and s_h are relaxation rates related to the higher order moments.

Performing either an Chapman–Enskog [9] or an asymptotic expansion [12,13] of Eq. (2), it can be shown that the LB-Method is a scheme of first order in time and second order in space for the incompressible Navier–Stokes equations in the low Mach number limit. The kinematic viscosity is related to the relaxation rate s_ν by

$$\nu = c^2 \Delta t \left(\frac{1}{3 s_\nu} - \frac{1}{6} \right). \tag{7}$$

The hydrodynamic pressure is given by

$$p = \frac{c^2}{3} \rho. \tag{8}$$

The collision rates s_e, s_h and s_ϵ are not relevant for the incompressible limit of the Navier–Stokes equations and can be chosen in the range $[0, 2]$. They can be tuned to improve stability [15], where the optimal values depend for the MRT model on the specific system under consideration (geometry, initial and boundary conditions) and cannot be computed in advance. A good choice is to set these values to one. If we omit the quadratic terms in Eq. (6) the Stokes equations result. The corresponding kernel is labeled as MRTL later in the text. For Stokes flow a good choice for the relaxation rates is [10]

$$s_e = s_\epsilon = s_\nu, \quad s_h = 8 \frac{(2 - s_\nu)}{(8 - s_\nu)}. \tag{9}$$

If we set all relaxation rates to the same value the usual LBGK collision [19] operator is obtained. The corresponding kernels are labeled as LBGK and the linear variant LBGKL.

Boundary conditions. In our implementation we use a voxel matrix indicating the type of cell node: inflow, outflow, solid or fluid node. Solid walls are implemented by applying the simple bounce back rule for the distribution functions: on a solid node all distribution functions are inverted meaning that fe-fw, fn-fs, fne-fsw, fnw-fse are interchanged. Pressure or velocity boundary conditions are implemented by extrapolating the moments to the boundary node, setting either the density or the momentum to the desired value and transforming back with $\boldsymbol{f} = \mathsf{M}^{-1}\boldsymbol{m}$. For higher order boundary conditions we refer to [10].

Forces on fixed obstacles. The force $\mathbf{F}_k$ acting on a boundary cut by a link k between $\mathbf{x}_f$ and $\mathbf{x}_b$ results from the momentum-exchange between the particle distribution $f_{\hat{i}}(t, \mathbf{x}_f)$ and $f_i(t + \Delta t, \mathbf{x}_f)$ hitting the boundary [17] as shown in Fig. 1. The momentum change can be computed by regarding the distribution function before and after hitting the boundary. Since the amplitude of $f_{\hat{i}}$ is not altered by the simple bounce back, $\mathbf{F}_k$ can be computed as

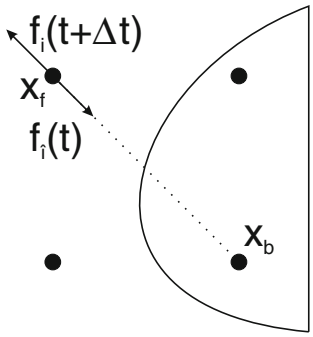

Fig. 1 Momentum transfer on fixed obstacles

$$\mathbf{F}_k(t + \Delta t/2) = -2 \frac{\Delta x^2 \, l_z}{\Delta t} \mathbf{e}_i \, f_i(t + \Delta t, \mathbf{x}_f), \tag{10}$$

where l_z is the length in the third dimension. Drag and lift forces on the whole obstacle are computed by summing up all contributions $\mathbf{F}_k$,

$$\mathbf{F} = \sum_{k \in \mathfrak{C}} \mathbf{F}_k + \mathbf{F}_{\text{body}}, \tag{11}$$

where the sum goes through all the cut links $k \in \mathfrak{C}$ for all boundary nodes $\mathbf{x}_f$ of the obstacle. If a body force $\mathbf{G}$ is applied momentum conservation requires that one adds the force exerted on the body by integrating over the area A of the body

$$\mathbf{F}_{\text{body}} = l_z \int_A -\mathbf{G} \, dA \tag{12}$$

For a detailed discussion concerning momentum transfer we refer to [10].

5 Implementation of a Lattice Boltzmann kernel

A detailed overview of the efficient implementation of LB kernels for CPUs is given in [25]. Since the architecture of the GPU is different, also the implementation is different from a design optimized for CPUs. We have no cache hierarchy, so the layout of the data structures has to be designed in a way that the memory bandwidth is exploited. In contrast to CPU design where one has to avoid powers of two in the leading dimension of an array for not having cache trashing effects, the opposite is true for the GPU. Here memory addresses have to be aligned as discussed in Sect. 3.3.

In the LB method 9 particle distribution functions have to be shifted in 9 different directions. We allocate $2 \times 9 = 18$ 1-D arrays, one set for the current time step and one set for the new time step with the restriction that nx,ny have to be a multiple of 16. Also the arrays are allocated with an offset

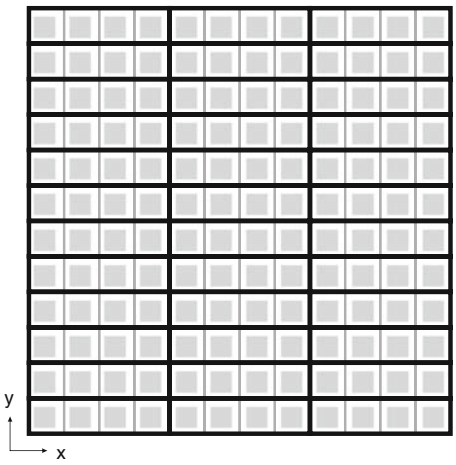

Fig. 2 Grid configuration for LBCollProp for a 12 × 12 matrix and 4 threads: grid configuration is represented by *black lines* (12 × 4), the thread partition within one block is represented by *gray lines*

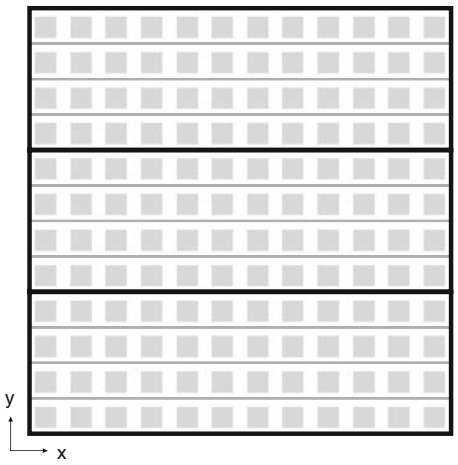

Fig. 3 Grid configuration for LBExchange and LBBC for a 12 × 12 matrix and 4 threads: grid configuration is represented by *black lines* (3 × 1), the thread partition within one block is represented by *gray lines*

startoff(=16) in y-direction to allow an efficient shift of the distribution functions in the propagation direction with north and south parts. Distributions with east and west parts are propagated using shared memory explained later.

In the time loop we have three kernel functions:

- LBCollProp: This kernel function is responsible for collision and propagation of the fluid and no-slip nodes. The layout of each block is (num_threads, 1, 1) and the grid of blocks is defined as (nx/num_threads,ny). The configuration for a 12 × 12 matrix and 4 threads is shown in Fig. 2.
- LBExchange: This kernel function synchronizes the distributions across the borders of the thread blocks. The layout of each block is the same as before (num_threads, 1, 1), but the configuration of the grid is different: We process each row of the matrix sequentially but the different rows are independent and can be processed in parallel. So each thread is responsible for one row and the grid is defined as (1,ny/num_threads). The configuration for a 12 × 12 matrix and 4 threads is shown in Fig. 3.
- LBBC: This kernel function is responsible for the inlet and outlet boundary conditions and has the same configuration as LBExchange.

So depending on the operations to carry out one can take advantage of the dynamic configuration of the thread blocks and the grid. Below an excerpt of the main loop is given.

```
...
//allocate  fr0,fe0,fn0,fw0,fs0,fne0,fnw0,fsw0,fse0
//and  fr1,fe1,fn1,fw1,fs1,fne1,fnw1,fsw1,fse1
...
dim3 threads(num_threads, 1, 1);
dim3 grid(nx/num_threads, ny);
dim3 grid1(1, ny/num_threads);
```

```
...
for(t=0;t<=tend;t++)
{
  //Set Pointers
  if(t%2==0)
  {
   frOld=fr0;feOld=fe0;fnOld=fn0;...
   frNew=fr1;feNew=fe1;fnNew=fn1;...
  }
  else{
   frOld=fr1;feOld=fe1;fnOld=fn1;...
   frNew=fr0;feNew=fe0;fnNew=fn0;...
  }

  //collision and propagation
  LBCollProp<<< grid, threads >>> (nx, ny,
    startoff,  s, geoD,
    frOld,  feOld,  fnOld,  fwOld,  fsOld,
    fneOld,  fnwOld,  fswOld,  fseOld,
    frNew,  feNew,  fnNew,  fwNew,  fsNew,
    fneNew,  fnwNew,  fswNew,  fseNew);

  //synchronize distributions across
  // thread blocks
  LBExchange<<< grid1, threads >>> (nx, ny,
    startoff, nx/num_threads, feNew, fwNew,
    fneNew,  fnwNew,  fswNew,  fseNew);

  //impose boundary conditions on
  //in- and outlet
  LBBC<<< grid1, threads >>> ( nx, ny,
   startoff, geoD,
   frNew,  feNew,  fnNew,  fwNew,  fsNew,
   fneNew,  fnwNew,  fswNew,  fseNew);

   //Postprocess results
   ...
  }
}
```

LBCollProp *:* We loop over the nodes with the indexing as described in Sect. 3.4, so that contiguous memory access for the 1-D arrays is possible when loading the current time

Table 1 Peak performance, memory bandwidth and price of different platforms

Platform	PEAK (Gflop/s)	TMBW (GB/s)	MMBW (GB/s)	MEM (MB)	price (Euro)
Intel Core 2 Duo (2.4 GHz)	38.4	8.5	5.2	2, 000	1,000
NEC SX-8R A (8 CPUs)	281.0	563.0	–	128, 000	–
nVIDIA 8800 Ultra	410.0	104	87	768	500

step. We combine collision and propagation and have to shift the propagations to the right locations. Here care has to be taken: The particle distribution functions f_r, f_n and f_s (the rest particle with no shift, and the particles going to the north and south direction) can be directly written to the device memory since they are aligned to a location in memory at 16*sizeof(type) bytes. But for the other distribution functions this is not true anymore, since they are shifted sizeof(type) bytes to the east or west. If we write them directly to the device memory performance breaks down and the bandwidth is restricted to 10 GB/s. The trick is here to allocate shared memory for the distribution functions, to propagate them using this fast shared memory and to write back these values to the device memory without a shift. Since shared memory is only global within a block, distributions leaving the border are reinjected at the opposite side and stored there. In Appendix B an excerpt of LBCollProp is given. In this kernel function also the bounce back rule for non-slip walls is integrated by an if-statement.

LBExchange*:* After applying LBCollProp we have to exchange the values stored at the boundaries of a block (in our case this are only the starting and ending point of one line) in the kernel function LBExchange. Each row of the matrix is processed sequentially using two for-loops, one for the distributions going to the east and one for the distributions going to the west. In Appendix C an excerpt of LBExchange is given.

6 Performance

In Table 1 the peak performance PEAK, the theoretical bandwidth to memory interface TMBW, the achievable bandwidth MMBW for simple copy throughput, the amount of main memory MEM and the price of different systems are given. For the peak performance we observed that the theoretical and the achievable peak performance are very close, if the number of multiplications and additions are equal. This comparison shows definitely that the G80 chip offers the best PEAK/EURO and MMBW/EURO ratio.

Performance is either limited by available memory bandwidth or peak performance. Thus, the attainable maximum performance P in LUPS is given as

$$\mathsf{P} = \min\left\{\frac{\mathsf{MMBW}}{\mathsf{NB}}, \frac{\mathsf{PEAK}}{\mathsf{NF}}\right\} \tag{13}$$

where NB is the number of bytes per cell to be transferred from/to main memory and NF is the number of floating point operations per cell. Considering the memory bandwidth as the limiting factor we find NB = (10 (read) + 9 (write)) × 4 bytes = 76 bytes per cell for the D2Q9 model. While memory bandwidth is given by the architecture, the average number of floating point operations NF per cell depends on processor details and compiler. The D2Q9 LBGK,Lin/LBGK/MRT model has approximately 45/65/130 additions and 30/35/30 multiplications, so we choose NF = 90/130/260 since the peak performance can only be achieved if the processors can do an addition and multiplication simultaneously. In Table 2 the resulting lattice updates per second (LUPS) are given for different kernels and platforms. The problem computed was a driven cavity discretized on a mesh of size $2048^2/2051^2$ (avoid cache trashing for CPU). The number of threads for the GPU was 128. In Table 3 the exploitation of peak performance and maximum bandwidth of memory interface for a CPU and GPU platform is given, where as basis for the maximum bandwidth of memory interface MMBW was chosen. For both kernels LBGKL and MRT the limiting factor is the memory bandwidth, for the GPU as well as for the CPU. For the kernel LBGKL the are very good with 51% for the CPU and 59% for the GPU. For the more complex kernel MRT the values are lower but nevertheless show a very good utilization of the performance delivered by the hardware.

Discussion of the performance gain versus coding strategies. The coding strategy presented here is more difficult than a straightforward implementation of a lattice Boltzmann kernel. This is usually done with a separate collision and propagation step. The best performance one can expect theoretically with this approach is two times worse than the approach presented here, since one has to load and store the

Table 2 LUPS in Mio. for different kernels and platforms

Platform	LBGKL	LBGK	MRT
Intel Core 2 Duo (2.4 GHz)	35	25	23
nVIDIA 8800 Ultra	670	568	527

Table 3 Exploitation of peak performance and maximum bandwidth of memory interface for different platforms

Platform	PEAK-LBGKL(%)	MMBW-LBGKL(%)	PEAK-MRT(%)	MMBW-MRT(%)
Intel Core 2 Duo (2.4 GHz)	8	51	16	34
nVIDIA 8800 Ultra	15	59	33	46

distributions two times per time step and the memory bandwidth is the limiting factor. In reality the situation is even worse: In [11] a LB kernel with separate collision and propagation step is implemented on a ClearSpeed Advance™ Accelerator Board. It is reported that the propagation step takes approximately three times longer than the collision step. Concerning the propagation step it is mentioned that "the main disadvantage in comparison to the collision step is, that the distribution functions of the neighbors cannot be copied in blocks". So the extra coding effort for the approach presented here pays well.

7 Example: square array of cylinders

The purpose of this section is to show that nontrivial setups can be handled by the present approach. In Fig. 4 a square array of cylinders is shown. The solid volume fraction θ is given with

$$\theta = \pi \frac{r^2}{a^2}. \tag{14}$$

The force F exerted on one particle is given with

$$F = \frac{4\,\pi\,\nu\,\rho_0\,U\,l_z}{k}, \tag{15}$$

where U is Darcy (volume averaged) velocity. An analytical expression for k is given with [21]

$$k = -\frac{1}{2}\ln\theta - 0.738 + \theta - 0.887\theta^2 + 2.039\theta^3 - 2.421\theta^4 + \mathcal{O}(\theta^5). \tag{16}$$

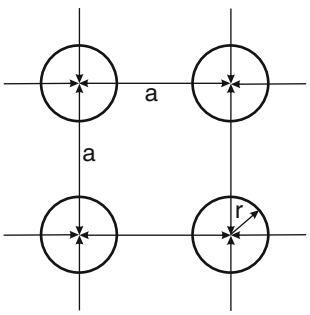

Fig. 4 square array of circles

For the numerical simulation it is possible to reduce the domain under consideration by exploiting symmetries, so that only one half of one obstacle has to be considered. But this is not the goal here, we want to show that it is possible to compute some type of porous media with the present approach efficiently. So we choose the following setup: The domain under consideration is square with length $L = 180\,\mathrm{m}$. The values $a = 10\,\mathrm{m}$ and $r = 2\,\mathrm{m}$ yielding a value of $\theta = 0.1256637$ and 324 obstacles. The viscosity was set to $\nu = 1/6\,\mathrm{m}^2\,\mathrm{s}^{-1}$, the density to $\rho_0 = 1\,\mathrm{kg\,m}^{-3}$ and the body force to $g = 2.5E-4\,\mathrm{kg\,m}^{-2}\,\mathrm{s}^{-2}$.

For the numerical simulation we used the linear MRTL model and used the relaxation rates given by Eq. (9). The force on the obstacles is computed either numerically with Eq. (11) or analytically for one obstacle with $F_a = a^2 l_z g_x$. The Darcy velocity is computed by averaging the numerical solution in all lattice sites and the numerical value for k is computed with Eq. (15).

In Table 4 the relative error with respect to the solid fraction θ and the value k for different mesh resolution are given. The numerical solid fraction is computed by adding all solid voxels and dividing by the total number of voxels. Due to the fact that we use only the simple bounce back boundary condition, the error in k is strongly related to the error θ_{err}, so that for $N = 256$ k_{err} is smaller than for $N = 512$. Nevertheless one can clearly observe a convergent behavior.

In Table 5 the LUPS for different mesh sizes and number of threads are given. The best performance is achieved with 128 threads and large domains and is only 10% slower than the driven cavity example. The reduction of performance for grid size 512 is related to the implementation and grid configuration of LBExchange and LBBC as shown in Fig. 3. For an efficient utilization of the hardware at least 16 grid blocks

Table 4 Porous medium, relative errors for different mesh sizes

Mesh size	θ_{err}	k_{err}
128^2	1.88E-01	2.06E-01
256^2	9.05E-04	1.73E-02
512^2	1.81E-02	1.87E-02
1024^2	5.99E-04	5.61E-03
2048^2	8.70E-04	4.78E-03
3072^2	2.36E-04	2.37E-03

Table 5 Porous medium, LUPS in Mio. for different mesh sizes and number of threads

Number of threads	32	64	128	192	256
Mesh size					
512^2	180	218	217	221	294
1024^2	281	398	391	357	360
2048^2	274	430	466	434	438
3072^2	248	413	481	452	452

with 64 threads have to run in parallel, leading to an extent of the domain in y-direction of 1024.

8 Discussion and outlook

The CUDA technology in combination with the approach presented here yields a very efficient LB simulator with a very good utilization of the performance delivered by the hardware. One key issue is to do the propagation via the fast shared memory and to read and to write data from and to memory only at blocks aligned to $16 \times$ sizeof(float).

The present approach can also handle domains with a large number of obstacles, the performance degradation D is only due to the amount of solid nodes, where no computation is needed but performed in the current implementation. D can estimated by $\mathsf{D} = \mathit{solid\ nodes} / \mathit{all\ nodes}$. A more sophisticated approach would decompose the domain in smaller blocks and mask blocks where no computation is needed.

With the CUDA technology it is also possible to access several GPUs on one host. It is possible to handle each GPU by a CPU thread. The communication is done by reading and writing memory from/to the host and GPU. First results are very promising and are subject to a future publication.

Acknowledgments Many thanks to Gerhard Wellein for the help given to tune the CPU code.

Appendix A: Orthogonal eigenvectors and transformation matrix

The eigenvectors $\{\boldsymbol{Q}_k, k = 0 \ldots 8\}$ of the collision operator are orthogonal with respect to the inner product $\langle \boldsymbol{Q}_i, \mathsf{W}, \boldsymbol{Q}_j \rangle$. The matrix W is diagonal and has the following weights $\boldsymbol{w}$ on its diagonal:

$$\boldsymbol{w} = \left(\frac{4}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}\right). \tag{17}$$

The eigenvectors are given with

$$Q_{0,i} = 1 \tag{18}$$
$$Q_{1,i} = e_{x,i} \tag{19}$$
$$Q_{2,i} = e_{y,i} \tag{20}$$
$$Q_{3,i} = 3\,(e_{x,i}^2 + e_{y,i}^2) - 2\,c^2 \tag{21}$$
$$Q_{4,i} = e_{x,i}^2 - e_{y,i}^2 \tag{22}$$
$$Q_{5,i} = e_{x,i} e_{y,i} \tag{23}$$
$$Q_{6,i} = (3(e_{x,i}^2 + e_{y,i}^2) - 4c^2)\, e_{x,i} \tag{24}$$
$$Q_{7,i} = (3(e_{x,i}^2 + e_{y,i}^2 - 4c^2)\, e_{y,i} \tag{25}$$
$$Q_{8,i} = \frac{1}{2}(9(e_{x,i}^2 + e_{y,i}^2)^2 - 15c^2(e_{x,i}^2 + e_{y,i}^2) + 2c^4). \tag{26}$$

The Transformation matrix M is composed of the eigenvectors $\mathsf{M}_{\mathsf{ki}} = Q_{k,i}$:

$$\mathsf{M} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & c & 0 & -c & 0 & c & -c & -c & c \\ 0 & 0 & c & 0 & -c & c & c & -c & -c \\ -2c^2 & c^2 & c^2 & c^2 & c^2 & 4c^2 & 4c^2 & 4c^2 & 4c^2 \\ 0 & c^2 & -c^2 & c^2 & -c^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & c^2 & -c^2 & c^2 & -c^2 \\ 0 & -c^3 & 0 & c^3 & 0 & 2c^3 & -2c^3 & -2c^3 & 2c^3 \\ 0 & 0 & -c^3 & 0 & c^3 & 2c^3 & 2c^3 & -2c^3 & -2c^3 \\ c^4 & -2c^4 & -2c^4 & -2c^4 & -2c^4 & 4c^4 & 4c^4 & 4c^4 & 4c^4 \end{bmatrix} \tag{27}$$

Appendix B: Kernel function LBCollProp

```
__global__ void LBCollProp(int nx, int ny,
int startoff, float4 s, unsigned int* geoD,
float* fr0, float* fe0, float* fn0,
float* fw0, float* fs0, float* fne0,
float* fnw0, float* fsw0, float* fse0,
float* fr1, float* fe1, float* fn1,
float* fw1, float* fs1, float* fne1,
float* fnw1, float* fsw1, float* fse1)
{
  // number of threads
  int num_threads = blockDim.x;

  // local thread index
  int tx = threadIdx.x;

  // Block index in x
  int bx = blockIdx.x;

  // Block index in y
  int by = blockIdx.y;
```

```
// Global x-Index
int xStart = tx +  bx*num_threads;

// Global y-Index
int yStart = by + startoff;

// Index k in 1D-arrays
int k = nx*yStart+xStart;

//Shared memory for propagation
__shared__ float F_OUT_E[THREAD_NUM];
__shared__ float F_OUT_W[THREAD_NUM];
__shared__ float F_OUT_NE[THREAD_NUM];
__shared__ float F_OUT_NW[THREAD_NUM];
__shared__ float F_OUT_SW[THREAD_NUM];
__shared__ float F_OUT_SE[THREAD_NUM];

...
//load fr0[k],fe0[k],fn0[k],fw0[k],fs0[k],
//fne0[k],fnw0[k],fsw0[k],fse0[k]
//to local variables F_IN_R,F_IN_E,F_IN_N,
//F_IN_W,F_IN_S,F_IN_NE,F_IN_NW,F_IN_SW,F_IN_SE
...

if(geoD[k] == GEO_FLUID)
{
  //collision:
  //modify F_IN_R,F_IN_E,...,F_IN_SE
  ...
}
else if(geoD[k]  == GEO_SOLID)
{
  //bounce back:
  //modify F_IN_R,F_IN_E,...,F_IN_SE
  ...
}

//Propagation using shared memory for
//distributions having a shift
//in east or west direction
if(tx==0)
{
  F_OUT_E [tx+1]=F_IN_E;
  F_OUT_NE[tx+1]=F_IN_NE;
  F_OUT_SE[tx+1]=F_IN_SE;

  //store distribution leaving
  //the domain across the west border
  F_OUT_W [num_threads-1]=F_IN_W;
  F_OUT_NW[num_threads-1]=F_IN_NW;
  F_OUT_SW[num_threads-1]=F_IN_SW;

}
else if(tx==num_threads-1)
{
  //store distribution leaving
  //the domain across the east border
  F_OUT_E [0]=F_IN_E;
  F_OUT_NE[0]=F_IN_NE;
  F_OUT_SE[0]=F_IN_SE;

  F_OUT_W [tx-1]=F_IN_W;
  F_OUT_NW[tx-1]=F_IN_NW;
  F_OUT_SW[tx-1]=F_IN_SW;
}
else{
  F_OUT_E [tx+1]=F_IN_E;
  F_OUT_NE[tx+1]=F_IN_NE;
  F_OUT_SE[tx+1]=F_IN_SE;
  F_OUT_W [tx-1]=F_IN_W;
  F_OUT_NW[tx-1]=F_IN_NW;
  F_OUT_SW[tx-1]=F_IN_SW;
}

// synchronize threads
__syncthreads();

// write data back to device memory
fr1[k]=F_IN_R;
fe1[k]=F_OUT_E[tx];
fw1[k]=F_OUT_W[tx];

k = nx*(yStart+1) + xStart;
fn1[k]=F_IN_N;
fne1[k]=F_OUT_NE[tx];
fnw1[k]=F_OUT_NW[tx];

k = nx*(yStart-1) + xStart;
fs1[k]=F_IN_S;
fsw1[k]=F_OUT_SW[tx];
fse1[k]=F_OUT_SE[tx];

}
```

Appendix C: Kernel function LBExchange

```
__global__ void LBExchange(int nx, int ny,
int Startoff,  int nbx, float* fe1, float* fw1,
float* fne1, float* fnw1, float* fsw1, float* fse1)
{
 //number of elements of one thread block in LBCollProp
 int num_threads = THREAD_NUM;

 //number of threads in this function
 int num_threads1 = blockDim.x;

 // Block index y
 int by = blockIdx.y;

 // local thread index
 int tx = threadIdx.x;

 int bx;
 int xStart, yStart;
 int xStartW, xTargetW;
 int xStartE, xTargetE;
 int kStartW, kTargetW;
 int kStartE, kTargetE;

 for(bx=0; bx<nbx ;bx++)
 {
  xStart = bx*num_threads;
  xStartW = xStart+2*num_threads-1;
  xTargetW  = xStartW-num_threads;
  yStart = by*num_threads1 + Startoff + tx;
  kStartW = nx*yStart+xStartW;
  kTargetW  = nx*yStart+xTargetW;

  fw1 [kTargetW]  = fw1[kStartW];
  fnw1[kTargetW]  = fnw1[kStartW];
  fsw1[kTargetW]  = fsw1[kStartW];
 }

 for(bx=nbx-1; bx>=0 ;bx--)
```

```
 {
  xStart = bx*num_threads;
  xStartE = xStart;
  xTargetE  = xStartE+num_threads;
  yStart = by*num_threads1 + Startoff + tx;
  kStartE = nx*yStart+xStartE;
  kTargetE  = nx*yStart+xTargetE;

  fe1 [kTargetE]  = fe1[kStartE];
  fne1[kTargetE] = fne1[kStartE];
  fse1[kTargetE] = fse1[kStartE];
 }
}
```

References

1. Benzi, R., Succi, S., Vergassola, M.: The lattice Boltzmann equation: theory and applications. Phys. Rep. **222**(3), 147–197 (1992)
2. Bolz, J., Farmer, I., Grinspun, E., Schröder, P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. ACM Trans. Graph. (SIGGRAPH) **22**(3), 917–924 (2003)
3. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: Stream Computing on Graphics Hardware. ACM Trans. Graph. **23**, 777–786 (2004). http://graphics.stanford.edu/papers/brookgpu/
4. Chen, S., Doolen, G.: Lattice Boltzmann method for fluid flows. Annu. Rev. Fluid Mech. **30**, 329–364 (1998)
5. Chu, N., Tai, C.L.: Moxi: Real-time ink dispersion in absorbent paper. ACM Trans. Graph. **24**(3), 504–511 (2005)
6. d'Humières, D.: Generalized lattice–Boltzmann equations. In: Shizgal, B.D., Weave, D.P. (eds.) Rarefied Gas Dynamics: Theory and Simulations. Prog. Astronaut. Aeronaut., vol. 159 pp. 450–458. AIAA Washington, DC (1992)
7. d'Humières, D., Ginzburg, I., Krafczyk, M., Lallemand, P., Luo, L.S.: Multiple-relaxation-time lattice Boltzmann models in three-dimensions. Philo. Trans. R. Soc. Lond. A **360**, 437–451 (2002)
8. Fan, Z., Qiu, F., Kaufman, A.E., Yoakum-Stover, S.: Gpu cluster for high performance computing. In: Proceedings of ACM/IEEE Supercomputing Conference 2004, pp. 47–59 (2004)
9. Frisch, U., d'HumiFres, D., Hasslacher, B., Lallemand, P., Pomeau, Y., Rivet, J.P.: Lattice gas hydrodynamics in two and three dimensions. Comp. Syst. **1**, 75–136 (1987)
10. Ginzburg, I., d'Humières, D.: Multireflection boundary conditions for lattice Boltzmann models. Phys. Rev. E **68**, 066,614 (2003)
11. Heuveline, V., Weiß, J.P.: A Parallel Implementation of a Lattice Boltzmann Method on the ClearSpeed Advance™ Accelerator Board. Tech. rep., Rechenzentrum Universität Karlsruhe (2007). http://www.rz.uni-karlsruhe.de/download/RZ-TR-2007-1.pdf
12. Inamuro, T., Yoshino, M., Ogino, F.: Accuarcy of the lattice boltzmann method for small knudsen number with finite reynolds number. Phys. Fluids **9**, 3535 (1997)
13. Junk, M., Klar, A., Luo, L.: Asymptotic analysis of the lattice boltzmann equation. J. Comp. Phys. **210**, 676 (2005)
14. Krüger, J., Westermann, R.: Linear algebra operators for GPU implementation of numerical algorithms. ACM Trans. Graph. (SIGGRAPH) **22**(3), 908–916 (2003)
15. Lallemand, P., Luo, L.S.: Theory of the lattice Boltzmann method: Dispersion, dissipation, isotropy, Galilean invariance, and stability. Phys. Rev. E **61**(6), 6546–6562 (2000)
16. Li, W., Wei, X., Kaufman, A.: Implementing Lattice Boltzmann Computation on Graphics Hardware. Vis. Comput. **19**(7-8), 444–456 (2003)
17. Nguyen, N.Q., Ladd, A.: Sedimentation of hard-sphere suspensions at low Reynolds number. J. Fluid Mech. **525**, 73–104 (2004)
18. NVIDIA CUDA Programming Guide. http://developer.download.nvidia.com
19. Qian, Y.H., d'Humières, D., Lallemand, P.: Lattice BGK models for Navier-Stokes equation. Europhys. Lett. **17**, 479–484 (1992)
20. Qiu, F., Zhao, Y., Fan, Z., Wei, X., Lorenz, H., Wang, J., Yoakum-Stover, S., Kaufman, A.E., Mueller, K.: Dispersion simulation and visualization for urban security. In: IEEE Visualization, pp. 553–560 (2004)
21. Sangani, A.S., Acrivos, A.: Slow flow past periodic arrays of cylinders with application to heat-transfer. Int. J. Multiph. Flow **8**(3), 193–206 (1982)
22. Tölke, J.: A thermal model based on the lattice Boltzmann method for low Mach number compressible flows. J. Comput. Theor. Nanosci. **3**(4), 579–587 (2006)
23. Tutubalina, A.: 8800 gtx performance tests. http://blog.lexa.ru/2007/03/08/nvidia_8800gtx_skorost__chtenija_tekstur.html. In Russian
24. Wei, X., Zhao, Y., Fan, Z., Li, W., Qiu, F., Yoakum-Stover, S., Kaufman, A.: Lattice-based flow field modeling. IEEE Trans. Vis. Comput. Graphics **10**(6), 719–729 (2004)
25. Wellein, G., Zeiser, T., Hager, G., Donath, S.: On the single processor performance of simple lattice Boltzmann kernels. Comput. Fluids **35**(8–9), 910–919 (2006)
26. Wu, E., Liu, Y., Liu, X.: An improved study of real-time fluid simulation on GPU. Comp. Anim. Virtual Worlds **15**, 139–146 (2004)
27. Zhao, Y., Han, Y., Fan, Z., Qiu, F., Kuo, Y.C., Kaufman, A., Mueller, K.: Visual simulation of heat shimmering and mirage. IEEE Trans. Vis. Comput. Graph. **13**(1), 179–189 (2007)
28. Zhao, Y., Wang, L., Qiu, F., Kaufman, A., Mueller, K.: Melting and flowing in multiphase environments. Comput. Graph. **30**(4), 519–528 (2006)
29. Zhu, H., Liu, X., Liu, Y., Wu, E.: Simulation of miscible binary mixtures based on lattice Boltzmann method. Comp. Anim. Virtual Worlds **17**, 403–410 (2006)