Towards three-dimensional teraflop CFD computing on a desktop PC using graphics hardware

J. Tölke [a,*], M. Krafczyk [a],

[a] *Institute for Computational Modeling in Civil Engineering, TU Braunschweig*

Abstract

A very efficient implementation of a Lattice Boltzmann (LB) kernel in three dimensions on a graphical processing unit (GPU) using the Compute Unified Device Architecture (CUDA) interface developed by nVIDIA is presented. By exploiting the explicit parallelism offered by the graphics hardware we obtain an efficiency gain of up to two orders of magitude with respect to the computational performance of a PC. A non-trivial example shows the performance of the lattice Boltzmann implementation, which is based on a D3Q13 model that is being described in detail.

Key words: Lattice Boltzmann model, D3Q13 model, Graphical Processing Unit, High Performance Computing

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 [1–3]. 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) [4]. In comparison, the theoretical peak performance of the Intel Core 2 Duo 3.0 GHz is only 24 GFLOPS for double precision and 48 GFLOPS for single

* Corresponding author, email: toelke@cab.bau.tu-bs.de

precision. Also the bandwidth to the memory interface is much larger: Memory bandwidth for desktop computers ranges from 5.3 to 10.7 GigaByte per second (GB/s), whereas the nVIDIA GeForce 8800 Ultra delivers up to 104 GB/s.

Due to the facts that Lattice Boltzmann methods (LBM) operate on a finite difference grid, are explicit in nature and usually require only next neighbor interaction they are very suitable for the implementation on GPUs. In [5] the computation of the LBM is accelerated on general-purpose graphics hardware by mapping the primary LB variables to 2D textures and the Boltzmann equations completely to rasterization and frame buffer operations. A speedup of at least one order of magnitude could be achieved compared to a implementation on a CPU. Applications for LB simulations in graphics hardware range from real-time ink dispersion in absorbent paper [6], dispersion simulation and visualization for urban security [7], simulation of soap bubbles [8], simulation of miscible binary mixtures [9], melting and flowing in multiphase environment [10] and visual simulation of heat shimmering and mirage [11]. Even GPU clusters have been assembled for general-purpose computations [12] and LB simulations have been performed. An implementation of a Navier-Stokes solver on a GPU can be found in [13]. 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 section 2 the graphics hardware is shortly sketched, in section 3 the Compute Unified Device Architecture (CUDA) programming technology is presented, in section 4 the D3Q13 LB model is described, in section 5 the implementation of this model using CUDA is described, section 6 gives an example, section 7 discusses the performance of the approach and section 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 1.6GHz, 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 pageable memory and 3.0 GB/s for pinned memory. The available amount of memory is 768 MegaByte (MB). The nVIDIA Quadro GPUs deliver memory up to 2 GigaByte (GB). There is also new product line called 'NVIDIA Tesla' (also based on the G80 chip) especially designed for high performance computing .

In table 1 the theoretical peak performance PEAK, the theoretical bandwidth to memory interface MBW, the amount of main memory MEM and the price of different systems are given. The theoretical bandwidth for copy throughput assuming a write allocate strategy for the scalar CPU architectures (additional cache line load is performed on a write miss) is given in brackets. This comparison definitely shows that the G80 chip offers an outstanding PEAK/EURO and MBW/EURO ratio.

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

Platform	PEAK [GFLOPS]	MBW [GB/s]	MEM [MB]	price [Euro]
Intel Core 2 Duo (3.0 GHz)	48.0	10.7 (7.0)	4 000	1 000
NEC SX-8R A (Single node, 8 CPUs)	281.0	563.0	128 000	expensive
nVIDIA 8800 Ultra (shader: 1.6Ghz)	410.0	104.0	768	500

3 nVIDIA CUDA: The GPU programming technology

3.1 Introduction

The nVIDIA CUDA technology is a fundamentally new computing architecture that enables the GPU to solve complex computational problems. CUDA

(Compute Unified Device Architecture) 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 Linux and Microsoft Windows XP operating systems.

3.2 Application Programming Interface (API)

In this subsection only a small subset of the API needed for the LB kernel is discussed following [4]. 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 transferred to 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 devices 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

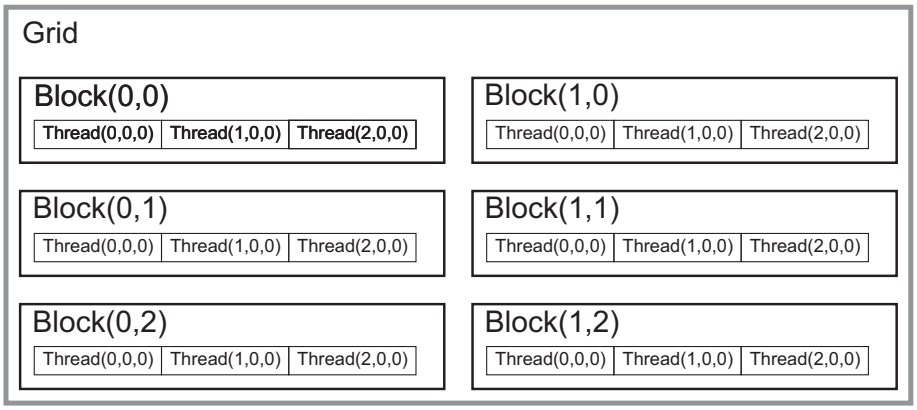

Fig. 1. A 2×3 grid of thread blocks of size $(3, 1, 1)$

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. In Fig. 1 a 2×3 grid of thread blocks of size $(3, 1, 1)$ and their indexing is shown.

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` $\times$ `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` $\times$ `Db.y` $\times$ `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 (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 either
  - `cudaMemcpyHostToHost`,
  - `cudaMemcpyHostToDevice`,
  - `cudaMemcpyDeviceToHost`,
  - `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 addess `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/sec [14]. 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.

4 Lattice Boltzmann method: The D3Q13 model

The Lattice Boltzmann method is a numerical method to solve the Navier-Stokes equations [15–17], where mass fractions (with unit $\mathsf{kg\,m^{-3}}$) propagate and collide on a regular grid. In the following discussion the font bold sans serif ($\mathbf{x}$) represents a three-dimensional vector in space and the font bold with serif $\boldsymbol{f}$ a b-dimensional vector, where b is the number of microscopic velocities. We use the $D3Q13$ model [18] which is probably the model with the minimal set of velocities in three dimensions to obtain the correct Navier-Stokes equation. It is also a very efficient model in terms of memory consumption, since due to a decoupling in two independent lattices it is possible to delete half of the nodes. It has the following microscopic velocities,

$$\begin{aligned}\{\mathbf{e}_i, i = 0, \ldots, 12\} &= \{\mathbf{e}_r, \mathbf{e}_{ne}, \mathbf{e}_{sw}, \mathbf{e}_{se}, \mathbf{e}_{nw}, \mathbf{e}_{te}, \mathbf{e}_{bw}, \mathbf{e}_{be}, \mathbf{e}_{tw}, \mathbf{e}_{tn}, \mathbf{e}_{bs}, \mathbf{e}_{bn}, \mathbf{e}_{ts}, \} \\ &= \left\{ \begin{array}{cccccccccccc} 0 & c & -c & c & -c & c & -c & c & -c & 0 & 0 & 0 & 0 \\ 0 & c & -c & -c & c & 0 & 0 & 0 & 0 & c & -c & c & -c \\ 0 & 0 & 0 & 0 & 0 & c & -c & -c & c & c & -c & -c & c \end{array} \right\}\end{aligned} \tag{1}$$

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

$$f_i(t + \Delta t, \mathbf{x} + \mathbf{e}_i \Delta t) = f_i(t, \mathbf{x}) + \Omega_i, \quad i = 0, \ldots, 12 \tag{2}$$

where f_i are mass fractions with unit $\mathsf{kg\,m^{-3}}$ propagating with microscopic velocity $\mathbf{e}_i$ and $\boldsymbol{\Omega}$ is the collision operator. The microscopic velocities or the mass fractions are also labeled depending on their direction **r**est, **n**orth**e**ast, **s**outh**w**est, **s**outh**e**ast, **n**orth**w**est, **t**op**e**ast, **b**ottom**w**est, **b**ottom**e**ast, **t**op**w**est, **t**op**n**orth, **b**ottoms**s**outh, **b**ottom**n**orth, **t**ops**s**outh as f_r, f_{ne}, f_{sw}, f_{se}, f_{nw}, f_{te},

f_{bw}, f_{be}, f_{tw}, f_{tn}, f_{bs}, f_{bn}, f_{ts}. The collision operator is given by

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

where M is the transformation matrix given in appendix A and $\boldsymbol{k}$ is the change of mass fractions in moment space.

The moments $\boldsymbol{m}$ of the mass fractions are given by

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

where the moment ρ of zero order is the density and the moments $(\rho_0 u_x, \rho_0 u_y, \rho_0 u_z)$ of first order are the momentum. The moments $e, p_{xx}, p_{ww}, p_{xy}, p_{yz}, p_{xz}$ of second order are related to the viscous stress tensor by

$$\begin{aligned}
\sigma_{xx} &= 2\,\nu\,\rho_0\,\frac{4}{3}(2\,u_x^2 - u_y^2 - u_z^2 - p_{xx}/\rho_0)/(8\nu + c^2\,\Delta t)\\
\sigma_{yy} &= 2\,\nu\,\rho_0\,\frac{2}{3}(4\,u_y^2 - 2\,u_x^2 - 2\,u_z^2 + p_{xx}/\rho_0 - 3\,p_{ww}/\rho_0)/(8\nu + c^2\,\Delta t)\\
\sigma_{zz} &= 2\,\nu\,\rho_0\,\frac{2}{3}\,(4\,u_z^2 - 2\,u_x^2 - 2\,u_y^2 + p_{xx}/\rho_0 + 3\,p_{ww}/\rho_0)/(8\nu + c^2\,\Delta t)\\
\sigma_{xy} &= \nu\,\rho_0\;(u_x\,u_y - p_{xy}/\rho_0)/(\nu + c^2\,\Delta t\,/\,4)\\
\sigma_{yz} &= \nu\,\rho_0\;(u_y\,u_z - p_{yz}/\rho_0)/(\nu + c^2\,\Delta t\,/\,4)\\
\sigma_{xz} &= \nu\,\rho_0\;(u_x\,u_z - p_{xz}/\rho_0)/(\nu + c^2\,\Delta t\,/\,4)
\end{aligned} \tag{5}$$

The moments h_x, h_y, h_z of third order are related to second derivates of the flow field.

The vector $\boldsymbol{k}$ is given by

$$\begin{aligned}
k_0 &= 0,\;\; k_1 = 0\;\; k_2 = 0,\;\; k_3 = 0\\
k_4 &= k_e = -s_e\left(e - (-\frac{11}{2}c^2\,\rho + \frac{13}{2}\,\rho_0\,(u_x^2 + u_y^2 + u_z^2)\right)\\
k_5 &= k_{xx} = -s_\nu\left(p_{xx} - \rho_0\,(2\,u_x^2 - u_y^2 - u_z^2)\right)\\
k_6 &= k_{ww} = -s_\nu\left(p_{ww} - \rho_0\,(u_y^2 - u_z^2)\right)\\
k_7 &= k_{xy} = -s\prime_\nu\,(p_{xy} - \rho\,u_x\,u_y)\\
k_8 &= k_{yz} = -s\prime_\nu\,(p_{yz} - \rho\,u_y\,u_z)\\
k_9 &= k_{xz} = -s\prime_\nu\,(p_{xz} - \rho\,u_x\,u_z)\\
k_{10} &= k_{hx} = -s_h\,h_x\\
k_{11} &= k_{hy} = -s_h\,h_y\\
k_{12} &= k_{hz} = -s_h\,h_z
\end{aligned} \tag{6}$$

where $s_e, s_\nu, s\prime_\nu, s_h$ are relaxation rates explained in more detail below.

Performing either a Chapman-Enskog [15] or an asymptotic expansion [19,20] of equation (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 relaxation rates s_ν and $s\prime_\nu$ are related to the viscosity by

$$\begin{aligned} s_\nu &= \frac{2}{8\frac{\nu}{c^2\,\Delta t}+1} \\ s\prime_\nu &= \frac{2}{4\frac{\nu}{c^2\,\Delta t}+1}. \end{aligned} \tag{7}$$

The collision rates s_e and s_h are not relevant for the incompressible limit of the Navier-Stokes equations and can be chosen in the range $[0, 2]$ to improve stability [21]. The optimal values for the MRT model depend on the specific system under consideration (geometry, initial and boundary conditions) and can not be computed in advance. A good choice is to set these values to one.

The hydrodynamic pressure is given by

$$p = \frac{c^2}{3}\,\rho = \frac{13}{33}\rho_0(u_x^2+u_y^2+u_z^2) - \frac{2}{33}\,e. \tag{8}$$

4.1 Boundary conditions

In our implementation we mark nodes as fluid, solid or boundary condition nodes. Solid walls and velocity boundary conditions are implemented by applying the simple bounce back rule for the mass fractions:

$$f_i(t+\Delta t, \mathbf{x}) = f_{\hat{i}}(t, \mathbf{x}) + \frac{\rho_0}{4\,c^2}\mathbf{e}_i\,\mathbf{U}_0(\mathbf{x}+\frac{1}{2}\mathbf{e}_i\,\Delta t), \tag{9}$$

where $\mathbf{U}_0$ is the prescribed velocity and f_i is the incoming mass fraction and $f_{\hat{i}}$ the anti-parallel outgoing mass fraction (see Fig. 2). If the boundary is not located exactly in the middle $\mathbf{x}+\frac{1}{2}\mathbf{e}_i\,\Delta t$ of the link i the boundary condition is only first order accurate. For higher order boundary conditions we refer to [22].

4.2 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 mass fraction $f_{\hat{i}}(t, \mathbf{x}_f)$ and $f_i(t+\Delta t, \mathbf{x}_f)$ hitting the boundary [23] as shown in Fig. 2. The momentum change can be computed by regarding the mass fraction before and after hitting the boundary:

$$\mathbf{F}_k(t+\Delta t/2) = -\frac{V}{\Delta t}\mathbf{e}_i\left(f_i(t+\Delta t, \mathbf{x}_f) + f_{\hat{i}}(t, \mathbf{x}_f)\right), \tag{10}$$

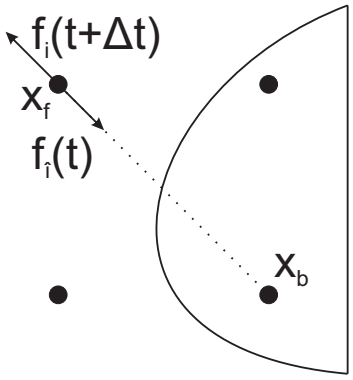

Fig. 2. Momentum transfer on fixed obstacles

where V is the volume of the unit cell. Note that for our implementation the unit call is a rhombic dodecahedron (see section 5.1) and the volume $V = 2h^3$. Drag and lift forces on the whole obstacle are computed by summing up all contributions $\mathbf{F}_k$,

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

where $\mathfrak{C}$ is the set of all links cut by the obstacle and the sum considers only boundary nodes $\mathbf{x}_f$.

5 Implementation of a Lattice Boltzmann kernel

A detailed overview of efficient implementation approaches of LB kernels for CPUs is given in [24]. Since the architecture of the GPU is different, also the implementation is different from a design optimized for CPUs. As GPUs have no cache hierarchy the layout of the data structures has to be designed to exploit the memory bandwidth. In contrast to CPU design where one has to avoid powers of two in the leading dimension of an array to avoid cache trashing effects, the opposite is true for the GPU. Here memory addresses have to be aligned as discussed in section 3.3.

5.1 Memory layout for the D3Q13 model

In a simple matrix based memory layout for the D3Q13 model the mass fractions are stored in a matrix $\mathrm{m}(nx, ny, nz, b)$ and are related to their position in the lattice through $x = h \times i, y = h \times j, z = h \times k$, where $i \in [1, nx]$, $j \in [1, ny]$ and $k \in [1, nz]$ are the indices. In the propagation step the mass fractions are shifted in the 13 directions (f_r is not shifted but copied to the same location) and stored in a second matrix at the right location. The full

lattice is composed of cubes generating a space filling comb. The basic cube has coordinates ($\pm$ 1/2, $\pm$1/2, $\pm$1/2) h.

A careful inspection of the connection graph of the lattice reveals that the lattice can be split into two totally independent sublattices consisting of the nodes with $i + j + k$ even for one and odd for the other [18]. Geometrical transformations or the possibility to run two simulations simultaneously on the grid to remove this staggered invariant are proposed in [18].

Here we propose another option not using a matrix layout. It is possible to just use only the lattice composed of the nodes with $i+j+k$ even. Using only half of the nodes the basic unit cell becomes a rhombic dodecahedron shown in Fig. 3. It is a Catalan solid with 12 rhombic faces, 24 edges and 14 vertices. The vertices are given by ($\pm$ 1,$\pm$ 1,$\pm$ 1) h, ($\pm$ 1, 0, 0)h, (0, $\pm$ 1, 0)h, (0, 0, $\pm$ 1)h. The rhombic dodecahedra honeycomb (see Fig. 4) is a space-filling tessellation (or honeycomb) in Euclidean 3-space. It is the Voronoi diagram of the face-centered cubic sphere-packing, which is believed to be the densest possible packing of identical spheres in ordinary space. The honeycomb is cell-transitive, face-transitive and edge-transitive meaning that all cells, faces and edges are the same. It is not vertex-transitive, as it has two kinds of vertices. The vertices with the obtuse rhombic face angles have 4 cells. The vertices with the acute rhombic face angles have 6 cells. The volume of the rhombic dodecahedron is given by

$$V = \frac{16}{9}\sqrt{3}a^3, \tag{12}$$

where a is the length of one edge. For the unit cell $a = \sqrt{3\,(h/2)^2}$ and therefore $V = 2\,h^3$.

For the D3Q13 LB model we have 13 mass fractions which have to be shifted in 13 different directions. We store the mass fractions in $2 \times 13 = 26$ 1-D arrays, one set for the current time step and one set for the new time step. This layout corresponds to the propagation optimized layout discussed in [24]. The element $m = nx \times (ny \times k + j) + i$ in each of the 1-D arrays is related to the position in space (x, y, z) by

$$\begin{aligned} a &= \begin{cases} 0 & \text{if j even and k even} \\ 0 & \text{if j odd and k odd} \\ 1 & \text{if j odd and k even} \\ 1 & \text{if j even and k odd} \end{cases} \\ x &= h\,(a + 2\,i) \\ y &= h\,j \\ z &= h\,k \end{aligned} \tag{13}$$

The values nx, ny and nz define the extensions of the grid. Note that $x =$

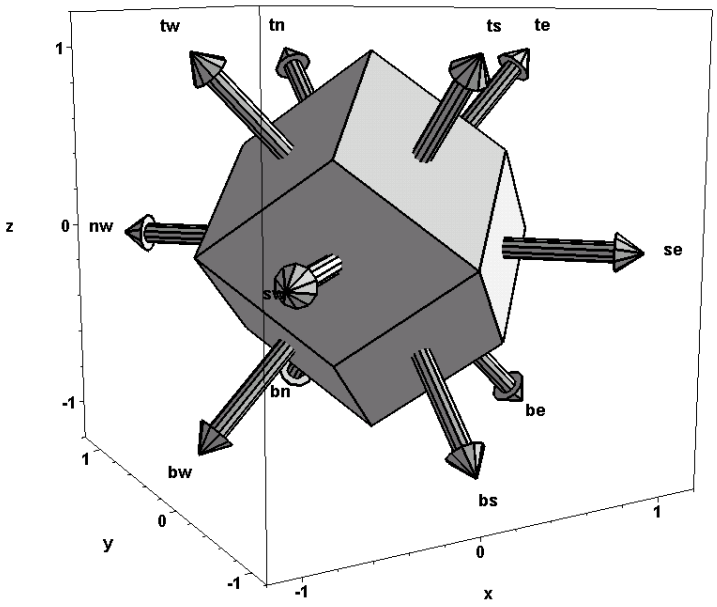

Fig. 3. Basic Unit cell for D3Q13 model: rhombic dodecahedron

$h\,(a+2\,i)$ and thus we have half the nodes in contrast to the full lattice.

The addressing scheme for the 1D-vector and the position in space (x,y,z) is computed in C-code as

```
int m = nx*(ny*k + j) + i;
float x =  h * ( (j&0x1)^(k&0x1) + i*2 );
float y =  h * j;
float z =  h * k;
```

The position mm in the 1D-vector of the neighbor $x + \Delta t\, e_{x,l}, y + \Delta t\, e_{y,l}, z + \Delta t\, e_{z,l}$ can be computed by

```
int xi = ( j&0x1)^(k&0x1)  + i*2;
int knew = k+ez[l];
int jnew = j+ey[l];
int a = ( jnew&0x1)^( knew&0x1);
int inew =  (xi+ex[l]-a)/2;
int mm = nx*(ny*(knew) +  jnew) + inew;
```

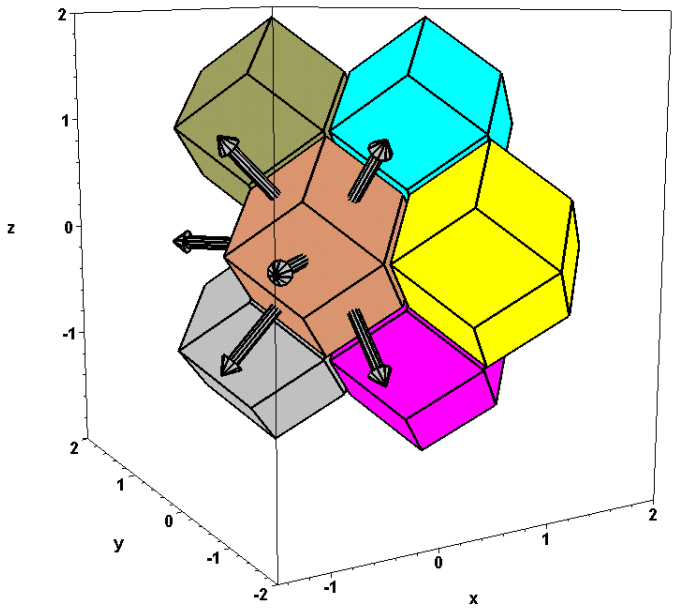

Fig. 4. rhombic dodecahedra honeycomb

5.2 Single precision versus double precision

The precision of float (32-bit) is 8 digits and of double (64-bit) is 16 digits. So mass- and momentum conservation is locally only guaranteed up to this precision. We experienced no problems in terms of accuracy for the simulations we did run up to now. In [25] the flow through a generic porous 2D-medium (square array of 324 circles) was computed up to a relative error of 2.4E-03 using single precision and simple bounce back. Problems reported with single precision and LB simulations [26] were often due to the fact that the original compressible model was used and the mass fractions had a constant part of $\mathcal{O}(1)$ related to the constant part of the density and a fluctuating part of $\mathcal{O}(h^2)$ related to the pressure. This was numerically very unsatisfactory. With the incompressible model [27] also used in our work, this deficiency is removed.

The stability is not as high as for simulations using double precision, but the breakdown is close in terms of the achievable Reynolds-number. The authors believe that a careful implementation of the collision operator is more useful to improve stability than just to switch to double precision.

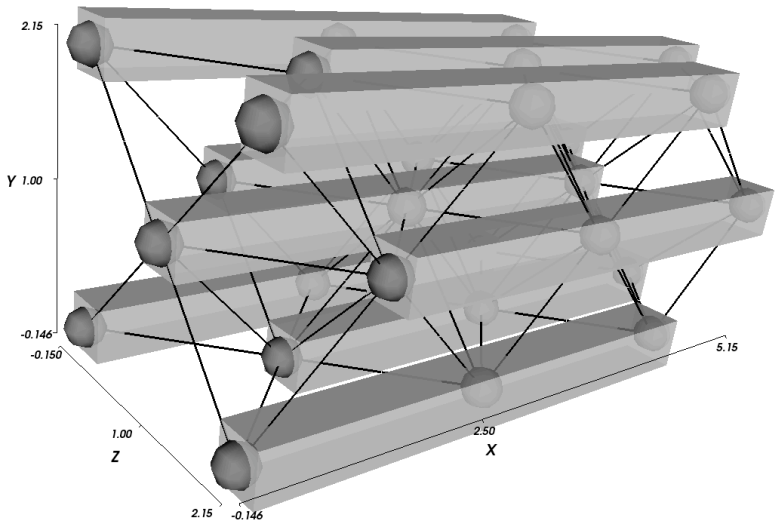

Fig. 5. Mapping of physical lattice to computational grid

5.3 Implementation using CUDA

To obtain a good memory throughput we load and store the mass fractions in complete lines along the x-direction. One block is thus configured to contain all the nodes along one line in x-direction as threads. This restricts the extension of the x-direction to $nx \in [16, 256]$, where nx should be a multiple of 16. This restriction comes from the fact that a certain number of threads is needed to run efficiently and that a maximum number of threads (512) is supported. The restrictions to 256 threads in our case comes from the fact that only a certain amount of registers, local and shared memory is available and that restricts the number of threads to this value. Note that due to the layout proposed in section 5.1 the lattice extension in x-direction is $2 \times nx \times h$.

The grid of thread blocks is defined by the number of nodes ny and nz along the y- and z-direction. The number of blocks in the grid should be larger than 16 to run efficiently. Note that despite the restrictions a very flexible setup is possible. For a more flexible setup in 2 dimensions we refer to [25]. In Fig. 5 the setup for a domain defined by $nx = 3, ny = 3, nz = 3$ is shown. The quadratic tubes indicate one block of threads.

To allow a uniform propagation without incorporation of if-statements a ghost layer in y- and z-direction is added and the value of `startoffy=1` `startoffz=1`. In the subsequent examples `nnx`, `nny` and `nnz` defines the domain and `nx=nnx`, `ny=nny+startoffy` and `nz=nnz+startoffz` the grid including ghost layers. This allows an efficient shift of the mass fractions in the propagation direction. We don't need a ghost layer in x-direction since we use shared memory buffers for the propagation.

In the time loop the kernel function `LBKernel` is responsible for collision and propagation. The layout of each block is `(num_threads, 1, 1)` and the grid of blocks is defined as `(ny,nz)`. Below an excerpt of the main loop is given.

```
...
//mass fractions
typedef struct  Distri{
    float* f[13];
} Distributions;
...
//allocate  Distributions d0,d1
...
    // setup execution parameters for one thread block
    dim3 threads(nnx, 1, 1);
    // configuration of the grid of blocks
    dim3 grid(nny, nnz);
...
    //time loop
    for(t=0;t<=tend;t++){
        //Switch pointers
        if(t%2==0){
            dold=d0;
            dnew=d1;
        }
        else{
            dold=d1;
            dnew=d0;
        }
         // execute the kernel: Collision+Propagation
        LBKernel<<< grid, threads >>> (nx, ny, geoD, dold, dnew);
        if(t%tpost ==0){
            //Copy to CPU, Postprocessing
        }
}
```

LBKernel:
We loop over the nodes in x-direction indexing as given by equation (13), so that contiguous memory access is possible when loading the current time step. We combine collision and propagation and have to shift the propagations to the correct locations. Here care has to be taken: The mass fractions f_r, f_{tn}, f_{bs}, f_{bn}, f_{ts} (the fraction with no shift, and the fractions not going to the east or west direction) can be directly written to the device memory since they are aligned to a location in memory at `16*sizeof(type)` bytes. For the other mass fractions this is not always true anymore, since they are shifted `sizeof(type)` bytes to the east or west for some configurations. Writing them

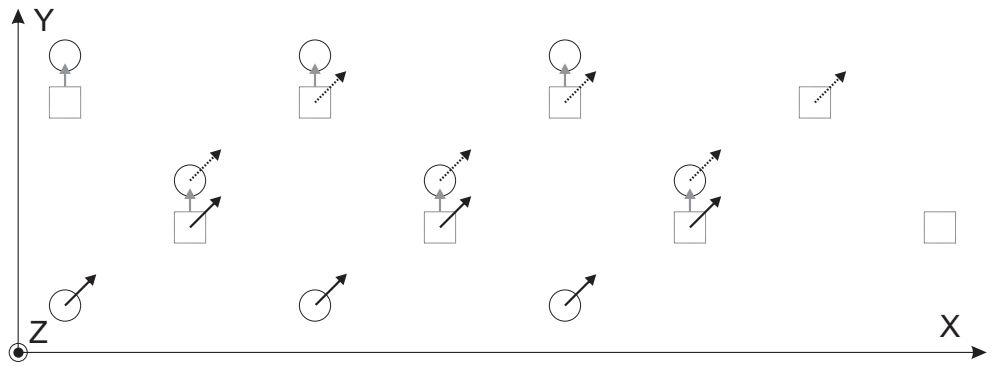

Fig. 6. Propagation of mass fractions f_{ne} in north-east direction using shared memory: circles represent lattice as hold in device memory, squares represent shared memory. First the mass fractions are written to shared memory and then transferred back uniformly to device memory

directly to the device memory leads to a substantial performance breakdown and the bandwidth is restricted to 10 GB/sec. To avoid this problem, we allocate shared memory for the mass fractions, propagate them using this fast memory and write back these values to the device memory uniformly without a shift. In Fig. 6 the propagation is shown for the mass fractions f_{ne} in north-east direction in a plane z =const. Note that the x-rows are staggered due to the topology and geometry of the D3Q13-model. The lowest and the middle row propagate the mass fractions represented by black arrows to the shared memory location represented by squares. The lowest row propagates the mass fraction without shift in memory location, the middle with a shift in east direction. The shared memory is then transferred back to device memory as indicated by the gray arrows. Note that for the lowest and then every second row shared memory is not needed, but we did implement it to not disturb the code with additional `if`-statements. In appendix B an excerpt of `LBKernel` is given. In this kernel function also the bounce back rule for non-slip nodes or the velocity boundary condition is integrated by an `if`-statement.

6 Example: moving sphere in a circular pipe

An approximate solution [28] for the dimensionless drag coefficient for a sphere moving with speed U_0 in an infinite fluid is given by

$$c_d = \frac{24}{\mathrm{Re}} \, (1 + 0.15 \, \mathrm{Re}^{0.687}). \tag{14}$$

The Reynolds-number is defined as

$$\mathrm{Re} = \frac{U_0 \, d}{\nu}, \tag{15}$$

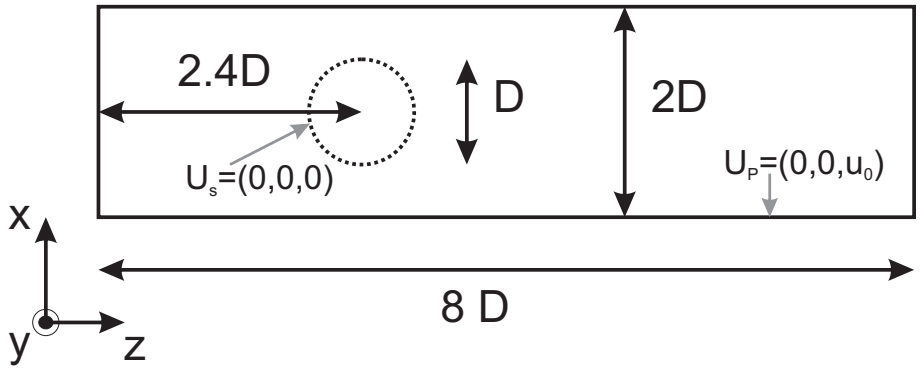

Fig. 7. Moving sphere in a pipe, setup for numerical simulation

where d is the diameter of the sphere and ν the kinematic viscosity. The drag force F_d exerted on the sphere is

$$F_d = c_d \, \frac{1}{2} \, \rho_0 \, U_0^2 \, \pi \, \frac{d^2}{4}. \tag{16}$$

The relative error of approximation (14) is $\pm$ 5 % for Re $<$ 800.

For a moving sphere in an infinite pipe the influence of the wall can be taken into account by [29]

$$c_{d,W50} = c_d + \frac{24}{Re}(K-1), \tag{17}$$

where K is given by [30]

$$K = \frac{1 - 0.75857\,\lambda^5}{1 - 2.1050\,\lambda + 2.0865\,\lambda^3 - 1.7068\,\lambda^5 + 0.72603\,\lambda^6} \tag{18}$$

and $\lambda = \frac{d}{D}$ is the ratio of the diameters of the sphere and the pipe. Approximation (17) has a relative error of$\pm$ 5 % for Re $<$ 50 and $\lambda < 0.6$. In the range 100 $<$ Re $<$ 800 the dimensionless drag coefficient is given by

$$c_{d,W800} = k_f \, c_d, \tag{19}$$

where k_f is given by [31]

$$k_f = \frac{1}{1 - 1.6\,\lambda^{1.6}}. \tag{20}$$

Approximation (19) has a relative error of $\pm$ 6 % for $\lambda < 0.6$.

We choose a coordinate system moving with the sphere, leading to a setup shown in Fig. 7. No-slip conditions are imposed on the boundary of the sphere and velocity boundary conditions on the inflow, outflow and on the boundary of the pipe.

6.1 Moving sphere in a pipe at Re=1

We simulate the moving sphere for a low Reynolds-number with three different resolutions. We increase mesh resolution and simultaneously reduce the Mach number by lowering u_0. The force on the sphere is computed using equation (11) and the drag coefficient is obtained from equation (16). The reference values for the drag coefficient is $c_{d,W} = 144.48$. In Table 2 the results are given. The criterion for a steady state was that the fourth digit in $c_{d,W}$ did not change anymore. The number of iterations to reach steady state is given as # iter, the number of nodes of the whole domain as # nodes=$nx \times ny \times nz$ and the device memory used on the GPU is given as Mem. The performance P is defined in section 7 and given in Mega-Lattice updates per second (MLUPS) where as basis # nodes is used. Mega-Fluid-Lattice updates per second (MFLUPS) represents a value, where only the fluid nodes are counted. The difference between MLUPS and MFLUPS is approximately $\pi/4$ in this case (ratio of circle to square), since the nodes outside the pipe and inside the sphere are irrelevant.

One can clearly observe a convergent behavior for $c_{d,W}$ with increasing mesh resolution.

Table 2
Moving sphere at Re=1, relative errors and other values for different mesh sizes

domain size	u_0 [$\mathsf{m\,s^{-1}}$]	ν [$\mathsf{m^2\,s^{-1}}$]	D [m]	# iter [-]	$c_{d,W}$ [-]	Rel. Err. [-]
$32^2 \times 128$	0.004	0.0595	14.88	40 000	152.2	5.3 %
$64^2 \times 256$	0.002	0.0605	30.24	80 000	146.6	1.5 %
$128^2 \times 512$	0.001	0.0610	60.96	260 000	145.3	0.6 %

domain size	$nx \times ny \times nz$	P [MLUPS]	P [MFLUPS]	# nodes [-]	Mem [MB]
$32^2 \times 128$	16×32×128	**239**	188	65 536	23
$64^2 \times 256$	32×64×256	**386**	303	524 288	118
$128^2 \times 512$	64×128×512	**582**	457	4 194 304	693

6.2 Moving sphere in a pipe at Re=10, 50, 100, 200, 300 and 400

We use a grid resolution of $128^2 \times 512$ to simulate the moving sphere at different Reynolds-numbers. The diameter D of the sphere is 60.95 m and the velocity boundary condition is $u_0 = 0.02\ \mathsf{m\,s^{-1}}$. In Table 3 the Reynolds-number Re, the kinematic viscosity ν, the wall clock time WCT in seconds, the number of time steps #iter, the numerical drag coefficient $c_{d,W}$, the reference drag

coefficient $c_{d,W,Ref.}$, the ratio of pressure drag to viscous drag and the relative error are given. The pressure and viscous drag was computed by two methods:

A For the pressure drag we initialized the fluid nodes close to the boundary with the equilibrium moments, where the density was the computed one and the velocity set to zero. Then we applied the momentum transfer using equation 2.

B We did a numerical integration over the sphere: We computed the pressure and the elements of the stress tensor using equations (5) and (8) of the node closest to a $d\mathbf{S}$-Element of the sphere. Then we did a projection of the pressure tensor using the normal of $d\mathbf{S}$.

A) and B) yielded results which differ at most by 10 %, in Table 3 the values for method A) are given.

For $Re = 300$ and $Re = 400$ the flow field becomes instationary. In Fig. 8 streamlines for the stationary case Re=200 and for the instationary cases Re=300 and 400 are shown, where for the instationary cases a snapshot of the flow field at the end of the simulation has been used to generate the streamlines. In Fig. 9 the drag coefficient over time is given, where T_{ref} is 1000 sec and corresponds to 1000 time steps. The amplitude of the oscillation is roughly 1 % of the average value for Re=300. In Fig. 10 the amplitude of the oscillation over the Strouhal number $St = f \times D \,/\, u_0$ is given, where f is the frequency. For Re=300 a peak at St=0.2 and for the case Re=400 a peak at St=0.03 and a more or less pronounced peak at St=0.22 can be observed.

In an advanced experimental setup [32] for a sphere in uniform flow a value of Re≈300 for the onset of vortex shedding and a shedding frequency in the range of 0.15-0.18 was observed. In numerical studies of the flow around a sphere [33,34], where the blockage ration was small, a value Re=280 for the onset of vortex shedding was observed. Also the amplitude of the oscillation of the drag coefficient was roughly 1 % of the average value for Re=300.

7 Performance

The performance of the Lattice Boltzmann method can be measured in **L**attice **U**pdates **P**er **S**econd (LUPS) and is either limited by available memory bandwidth or peak performance. A rough estimation of the attainable maximum performance P in LUPS is given by

$$\mathsf{P} = \min \left\{ \frac{\mathsf{MBW}}{\mathsf{NB}} \,,\, \frac{\mathsf{PEAK}}{\mathsf{NF}} \right\} \tag{21}$$

Table 3
Moving sphere at different Re, grid resolution $128^2 \times 512$

Re [-]	ν [$\mathsf{m^2\,s^{-1}}$]	WCT [s]	# iter[-]	$c_{d,W}$[-]	$c_{d,W,Ref.}$[-]	$\frac{p.drag}{v.drag}$	Rel. Err. [-]
10	0.121920	106	15 000	14.74	15.84	0.93	6.9 %
50	0.024384	415	59 000	3.697	3.876	1.15	4.6 %
100	0.012192	520	74 000	2.380	2.312	1.43	2.9 %
200	0.006096	774	110 000	1.679	1.706	1.90	1.6 %
300	0.004064	2100 [1]	300 000	1.440 [2]	1.448	2.35	0.6 %
400	0.003048	2800 [1]	400 000	1.305 [3]	1.296	2.82	0.7 %

[1] instationary flow field, time required to reach oscillatory state from initial uniform flow field (no disturbance imposed)
[2] average value, $t = 280 \ldots 2000\ T_{ref}$
[3] average value, $t = 200 \ldots 3000\ T_{ref}$

where NB is the number of bytes per cell and time step to be transferred from/to main memory and NF is the number of floating point operations per cell and time step. Considering the memory bandwidth as the limiting factor we find NB = (14 (read) + 13 (write)) $\times$ 4 bytes = 108 bytes per cell for the D3Q13 model. While memory bandwidth is given by the architecture, the average number NF of floating point operations (FLOP) per cell depends on processor details, compiler and the implementation. We assume for the D3Q13 model 150 additions and 30 multiplications and choose NF $= (30 + 30) + 2 \times (150 - 30) = 300$FLOP since the peak performance can only be achieved if the processors can do an addition and multiplication simultaneously.

In Table 4 the Performance P in LUPS for different mesh sizes for a driven cavity problem is given. As discussed in section 5.3 the value of nx defines the number of threads and (ny, nz) the grid of thread blocks. The best performance is achieved with 64 threads and large domains. A reduction of the performance is observed for a small number of threads and small domains. Taking the value P=592 MLUPS as a reference value the exploitation of the performance delivered by the hardware (44 % of the peak performance and **61 % (!)** of the maximum memory bandwidth) is very satisfactory and shows a good balance between floating point computing power and memory bandwidth. In [24] very efficient CPU-implementations of the D3Q19 model are discussed. We give some values for comparison and for details we refer to [24]. Note that the data transfer volume is for the D3Q19-model (double precision implementation) is $2 \times 19/13 = 2.92$ times higher. The performance in MLUPS was for Intel Xeon (3.4GHz):P=4.8, Intel Itanium 2 (1.4GHz): P=7.6 and for the vector machine NEC SX6+ (565MHz):P=41.3.

Table 4
LUPS in Mio. for different mesh sizes and number of threads

ny × nz \ nx	16	32	64	80	128	192	256
32 ×32	231	392	570	446	523	444	476
64 ×64	239	378	565	472	546	454	483
128×128	230	384	592	478	549	452	483

8 Summary and Outlook

The CUDA technology in combination with the computational approach presented here yields a very efficient LB simulator in terms of the price to performance ratio. 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 be estimated by D = *solid nodes / all nodes*. A more sophisticated approach would decompose the domain in smaller blocks and mask blocks where no computation is needed.

The current implementation could be extended to other discretization stencils such as D3Q15 and D3Q19, but due to the fact that the memory consumption will more than double these models are of limited use for present GPUs.

With the CUDA technology it is also possible to access several GPUs on one host allowing for teraflop simulations on a desktop PC. 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.

A Orthogonal eigenvectors and Transformation Matrix

The eigenvectors $\{\boldsymbol{Q}_k, k = 0 \ldots 12\}$ of the collision operator are orthogonal with respect to the inner product $< \boldsymbol{Q}_i, \boldsymbol{Q}_j >$ and are given by

$$Q_{0,i} = 1 = (1,1,1,1,1,1,1,1,1,1,1,1,1) \tag{A.1}$$
$$Q_{1,i} = e_{x,i} = c \cdot (0,1,-1,1,-1,1,-1,1,-1,0,0,0,0) \tag{A.2}$$
$$Q_{2,i} = e_{y,i} = c \cdot (0,1,-1,-1,1,0,0,0,0,1,-1,1,-1) \tag{A.3}$$
$$Q_{3,i} = e_{z,i} = c \cdot (0,0,0,0,0,1,-1,-1,1,1,-1,-1,1) \tag{A.4}$$
$$Q_{4,i} = \frac{13}{2}\,\mathbf{e}^2 - 12\,c^2 = c^2 \cdot (-12,1,1,1,1,1,1,1,1,1,1,1,1) \tag{A.5}$$
$$Q_{5,i} = 3\,e_{x,i}^2 - \mathbf{e}^2 = c^2 \cdot (0,1,1,1,1,1,1,1,1,-2,-2,-2,-2]) \tag{A.6}$$
$$Q_{6,i} = e_{y,i}^2 - e_{z,i}^2 = c^2 \cdot (0,1,1,1,1,-1,-1,-1,-1,0,0,0,0) \tag{A.7}$$
$$Q_{7,i} = e_{x,i}\,e_{y,i} = c^2 \cdot (0,1,1,-1,-1,0,0,0,0,0,0,0,0) \tag{A.8}$$
$$Q_{8,i} = e_{y,i}\,e_{z,i} = c^2 \cdot (0,0,0,0,0,0,0,0,0,1,1,-1,-1) \tag{A.9}$$
$$Q_{9,i} = e_{x,i}\,e_{z,i} = c^2 \cdot (0,0,0,0,0,1,1,-1,-1,0,0,0,0) \tag{A.10}$$
$$Q_{10,i} = e_{x,i}\,(e_{y,i}^2 - e_{z,i}^2) = c^3 \cdot (0,1,-1,1,-1,-1,1,-1,1,0,0,0,0) \tag{A.11}$$
$$Q_{11,i} = e_{y,i}\,(e_{z,i}^2 - e_{x,i}^2) = c^3 \cdot (0,-1,1,1,-1,0,0,0,0,1,-1,1,-1) \tag{A.12}$$
$$Q_{12,i} = e_{z,i}\,(e_{x,i}^2 - e_{y,i}^2) = c^3 \cdot (0,0,0,0,0,1,-1,-1,1,-1,1,1,-1)\,. \tag{A.13}$$

where $\mathbf{e}^2 = (e_{x,i}^2 + e_{y,i}^2 + e_{z,i}^2)$. The transformation matrix M is composed of the eigenvectors $\mathsf{M}_{\mathsf{ki}} = Q_{k,i}$.

B Kernel function `LBKernel`

```
__global__ void LBKernel(  int nx, int ny, unsigned int* geoD,
            Distributions dold, Distributions dnew)
{
    //geoD: integer matrix indicating the node type
    // (fluid,solid or boundary condition)
    //dold: 'Old' distribution functions
    //dnew: 'New'  distribution functions

    // Thread index
    int tx = threadIdx.x;

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

    // Block index y in the grid
```

```
int by = blockIdx.y;

// Global x-Index
int x = tx;

// Global y-Index
int y = bx + startoffy;

// Global z-Index
int z = by + startoffz;

unsigned int GEO;
float f_R,f_NE,f_SW,f_SE,f_NW,f_TE,f_BW,
    f_BE,f_TW,f_TN,f_BS,f_BN,f_TS;

// Shared memory for propagation in direction with east/west parts
__shared__ float fo_SE[THREAD_NUM+1];
__shared__ float fo_NE[THREAD_NUM+1];
__shared__ float fo_NW[THREAD_NUM+1];
__shared__ float fo_SW[THREAD_NUM+1];
__shared__ float fo_BE[THREAD_NUM+1];
__shared__ float fo_TE[THREAD_NUM+1];
__shared__ float fo_BW[THREAD_NUM+1];
__shared__ float fo_TW[THREAD_NUM+1];

// Index in 1d-vector
int k = nx*(ny*z + y) + x;

// Load data from device memory to local memory
GEO = geoD[k];
f_R  = (dold.f[ dirR])[k];
f_NE = (dold.f[dirNE])[k];
f_SW = (dold.f[dirSW])[k];
f_SE = (dold.f[dirSE])[k];
f_NW = (dold.f[dirNW])[k];
f_TE = (dold.f[dirTE])[k];
f_BW = (dold.f[dirBW])[k];
f_BE = (dold.f[dirBE])[k];
f_TW = (dold.f[dirTW])[k];
f_TN = (dold.f[dirTN])[k];
f_BS = (dold.f[dirBS])[k];
f_BN = (dold.f[dirBN])[k];
f_TS = (dold.f[dirTS])[k];

if(GEO == GEO_FLUID){
```

```
//Collision
...
}
else if(GEO == GEO_SOLID){
//Bounce Back
...
}
else if(GEO == GEO_INLET){
//Velocity Boundary Condition
...
}
// Propagation via shared memory for mass fractions
// with East or West part.
// Due to the memory layout the shift in East direction is
// either zero or one and vice versa for the West direction
//
int shiftE = ((y-startoffy)&0x1) ^ ((z-startoffz)&0x1);
int shiftW = 0x1 & (~shiftE);

int txE = tx+shiftE;
int txW = tx-shiftW;

fo_SE[txE]   = f_SE;
fo_NE[txE]   = f_NE;
fo_NW[txW+1] = f_NW;
fo_SW[txW+1] = f_SW;
fo_BE[txE]   = f_BE;
fo_TE[txE]   = f_TE;
fo_BW[txW+1] = f_BW;
fo_TW[txW+1] = f_TW;

__syncthreads();

// write data to device memory
//Propagation by computing correct index
int nxny = nx*ny;
int kn  = k + nx;
int ks  = k - nx;
int kt  = k + nxny;
int kb  = k - nxny;
int kts = k + nxny -nx;
int ktn = k + nxny +nx;
int kbs = k - nxny -nx;
int kbn = k - nxny +nx;
```

```
    (dnew.f[ dirR])[k]    = f_R;
    (dnew.f[ dirNE])[kn]  = fo_NE[tx];
    (dnew.f[ dirNW])[kn]  = fo_NW[tx+1];
    (dnew.f[ dirSE])[ks]  = fo_SE[tx];
    (dnew.f[ dirSW])[ks]  = fo_SW[tx+1];
    (dnew.f[ dirTE])[kt]  = fo_TE[tx];
    (dnew.f[ dirTW])[kt]  = fo_TW[tx+1];
    (dnew.f[ dirBE])[kb]  = fo_BE[tx];
    (dnew.f[ dirBW])[kb]  = fo_BW[tx+1];
    (dnew.f[ dirTS])[kts] = f_TS;
    (dnew.f[ dirTN])[ktn] = f_TN;
    (dnew.f[ dirBS])[kbs] = f_BS;
    (dnew.f[ dirBN])[kbn] = f_BN;
}
```

References

[1] J. Bolz, I. Farmer, E. Grinspun, P. Schröder, Sparse matrix solvers on the GPU: conjugate gradients and multigrid, ACM Trans. Graph. (SIGGRAPH) 22 (3) (2003) 917–924.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, Brook for GPUs: Stream Computing on Graphics Hardware, ACM Trans. Graph. (SIGGRAPH), to appear.

[3] J. Krüger, R. Westermann, Linear algebra operators for GPU implementation of numerical algorithms, ACM Trans. Graph. (SIGGRAPH) 22 (3) (2003) 908–916.

[4] NVIDIA CUDA Programming Guide.
URL `http://developer.download.nvidia.com`

[5] W. Li, X. Wei, A. Kaufman, Implementing Lattice Boltzmann Computation on Graphics Hardware, The Visual Computer 19 (7-8) (2003) 444–456.

[6] N. Chu, C.-L. Tai, MoXi: Real-Time Ink Dispersion in Absorbent Paper, ACM Transactions on Graphics 24 (3) (2005) 504–511.

[7] F. Qiu, Y. Zhao, Z. Fan, X. Wei, H. Lorenz, J. Wang, S. Yoakum-Stover, A. E. Kaufman, K. Mueller, Dispersion Simulation and Visualization For Urban Security., in: IEEE Visualization, 2004, pp. 553–560.

[8] X. Wei, Y. Zhao, Z. Fan, W. Li, F. Qiu, S. Yoakum-Stover, A. Kaufman, Lattice-based Flow Field Modeling, IEEE Transactions on Visualization and Computer Graphics 10 (6) (2004) 719–729.

[9] H. Zhu, X. Liu, Y. Liu, E. Wu, Simulation of miscible binary mixtures based on lattice Boltzmann method, Comp. Anim. Virtual Worlds 17 (2006) 403–410.

[10] Y. Zhao, L. Wang, F. Qiu, A. Kaufman, K. Mueller, Melting and Flowing in Multiphase Environments, Computers & Graphics 30 (4) (2006) 519–528.

[11] Y. Zhao, Y. Han, Z. Fan, F. Qiu, Y.-C. Kuo, A. Kaufman, K. Mueller, Visual simulation of heat shimmering and mirage, IEEE Transactions on Visualization and Computer Graphics 13 (1) (2007) 179–189.

[12] Z. Fan, F. Qiu, A. E. Kaufman, S. Yoakum-Stover, GPU Cluster for High Performance Computing, in: Proceedings of ACM/IEEE Supercomputing Conference 2004, 2004, pp. 47–59.

[13] E. Wu, Y. Liu, X. Liu, An improved study of real-time fluid simulation on GPU, Comp. Anim. Virtual Worlds 15 (2004) 139–146.

[14] A. Tutubalina, 8800 GTX Performance tests, in Russian.
URL `http://blog.lexa.ru/2007/03/08/nvidia_8800gtx_skorost__chtenija_tekstur.html`

[15] U. Frisch, D. d'Humires, B. Hasslacher, P. Lallemand, Y. Pomeau, J.-P. Rivet, Lattice gas hydrodynamics in two and three dimensions, Complex Systems 1 (1987) 75–136.

[16] R. Benzi, S. Succi, M. Vergassola, The lattice Boltzmann equation: theory and applications, Phys. Rep. 222, 3 (1992) 147–197.

[17] S. Chen, G. Doolen, Lattice Boltzmann method for fluid flows, Annu. Rev. Fluid Mech. 30 (1998) 329–364.

[18] D. d'Humières, M. Bouzidi, P. Lallemand, Thirteen-velocity three-dimensional lattice Boltzmann model, Phys. Rev. E 63 (6) (2001) 066702.

[19] T. Inamuro, M. Yoshino, F. Ogino, Accuarcy of the lattice Boltzmann method for small Knudsen number with finite Reynolds number, Phys. Fluids 9 (1997) 3535–3542.

[20] M. Junk, A. Klar, L. Luo, Asymptotic analysis of the lattice Boltzmann equation, Journal Comp. Phys. 210 (2005) 676–704.

[21] P. Lallemand, L.-S. Luo, Theory of the lattice Boltzmann method: Dispersion, dissipation, isotropy, Galilean invariance, and stability, Phys. Rev. E 61 (6) (2000) 6546–6562.

[22] I. Ginzburg, D. d'Humières, Multireflection boundary conditions for lattice Boltzmann models, Phys. Rev. E 68 (2003) 066614.

[23] N.-Q. Nguyen, A. Ladd, Sedimentation of hard-sphere suspensions at low Reynolds number, J. Fluid Mech. 525 (2004) 73–104.

[24] G. Wellein, T. Zeiser, G. Hager, S. Donath, On the single processor performance of simple lattice Boltzmann kernels, Computers & Fluids 35 (8-9) (2006) 910–919.

[25] J. Tölke, Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture, submitted to Computing and Visualization in Science, 2007.

[26] P. A. Skordos, Initial and Boundary Conditions for the Lattice Boltzmann Method, Physical Review E 48 (6) (1993) 4823–4842.

[27] X. He, L.-S. Luo, Lattice Boltzmann model for the incompressible Navier-Stokes equation, J. Stat. Phys. 88 (1997) 927–944.

[28] L. Schiller, A. Z. Naumann, Über die grundlegenden Berechnungen bei der Schwerkraftaufbereitung, Zeitschrift Des Vereines Deutscher Ingenieure 77 (12) (1933) 318–320.

[29] A. Fayon, J. Happel, Effect of a cylindrical boundary on fixed rigid sphere in a moving viscous fluid, AIChE J. 6 (1) (1960) 55–58.

[30] W. L. Haberman, R. M. Sayre, Motion Of Rigid And Fluid Spheres In Stationary and Moving Liquids Inside Cylindrical Tubes, David Taylor Model Basin Report No. 1143, US Navy Department, Washington DC (1958).

[31] R. Clift, J. R. Grace, M. E. Weber, Bubbles, Drops and Particles, Academic Press, 1978.

[32] H. Sakamoto, H. Haniu, A study on vortex shedding from spheres in a uniform flow, ASME Transactions Journal of Fluids Engineering 112 (1990) 386–392.

[33] T. A. Johnson, V. C. Patel, Flow past a sphere up to a Reynolds number of 300, Journal of Fluid Mechanics 378 (1999) 19–70.

[34] A. G. Tomboulides, S. A. Orszag, Numerical investigation of transitional and weak turbulent flow past a sphere, Journal of Fluid Mechanics 416 (2000) 45–73.

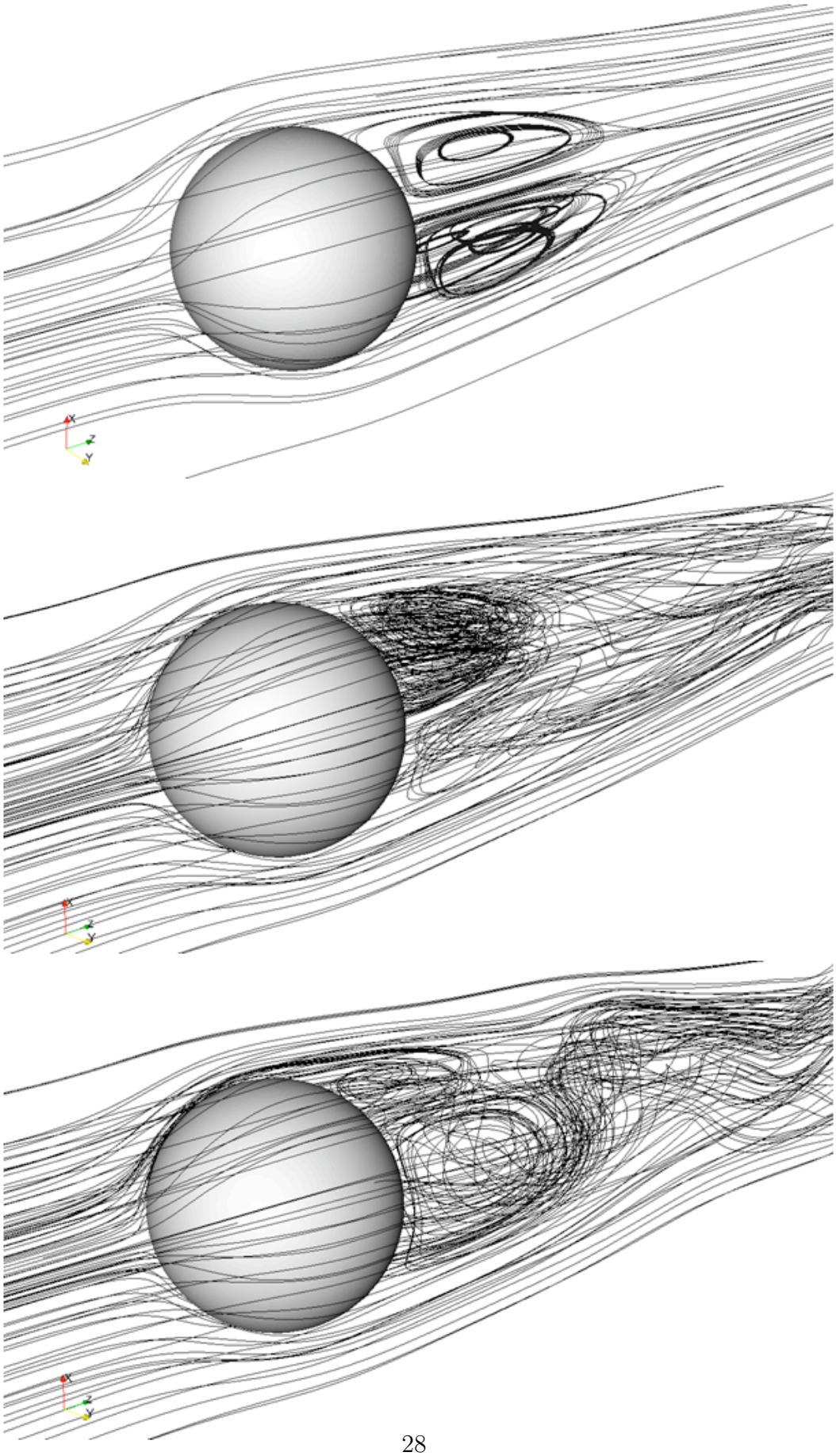

Fig. 8. Streamlines for Re=200, 300 and 400

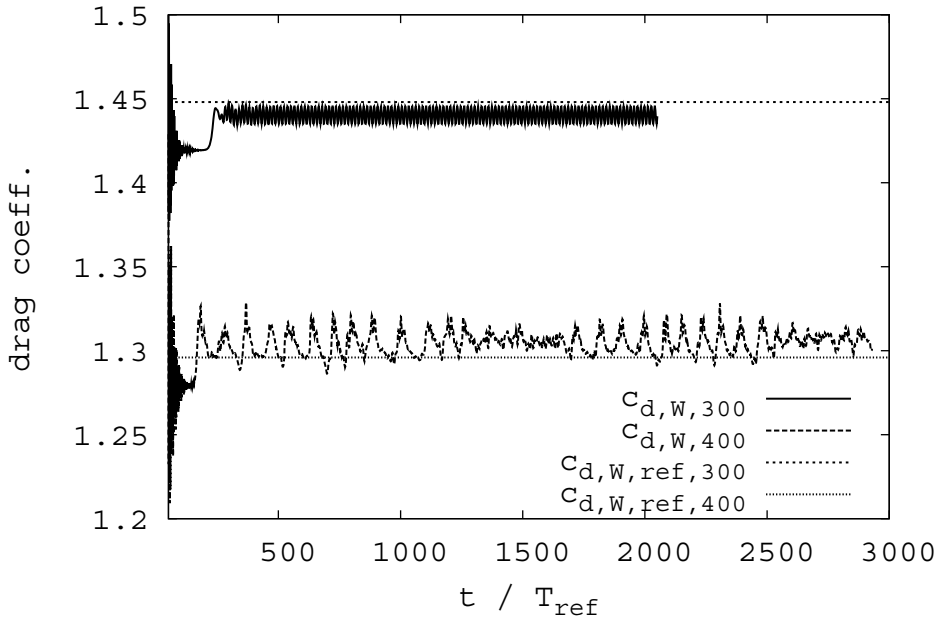

Fig. 9. Drag coefficient for Re=300 and 400 over time

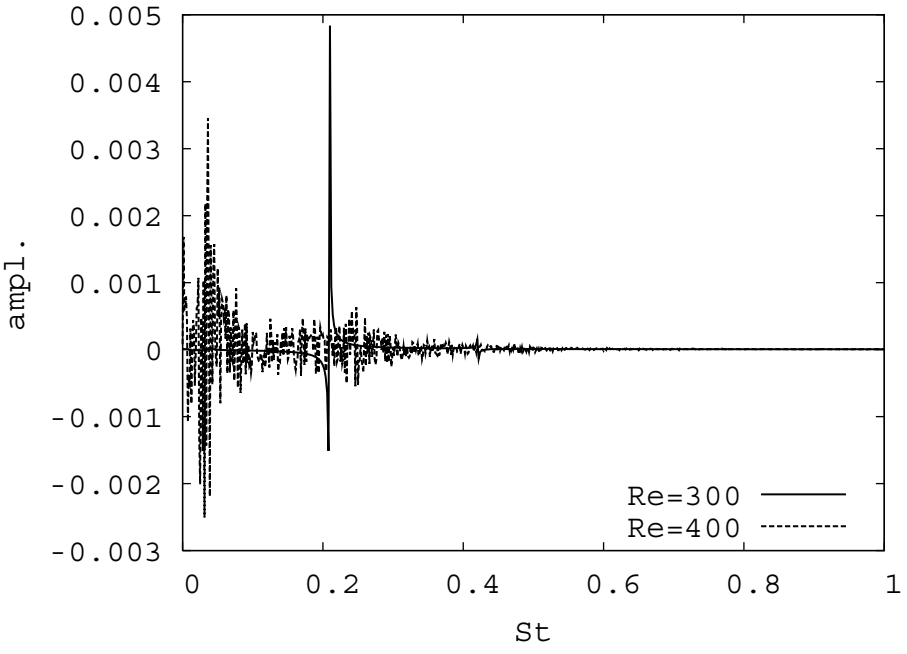

Fig. 10. Amplitude of oscillation over Strouhal number