International Journal of Computational Fluid Dynamics

ISSN: 1061-8562 (Print) 1029-0257 (Online) Journal homepage: http://www.tandfonline.com/loi/gcfd20

TeraFLOP computing on a desktop PC with GPUs for 3D CFD

J. Tölke & M. Krafczyk

To cite this article: J. Tölke & M. Krafczyk (2008) TeraFLOP computing on a desktop PC with GPUs for 3D CFD, International Journal of Computational Fluid Dynamics, 22:7, 443-456, DOI: 10.1080/10618560802238275

To link to this article: http://dx.doi.org/10.1080/10618560802238275

International Journal of Computational Fluid Dynamics
Vol. 22, No. 7, August 2008, 443–456

Taylor & Francis
Taylor & Francis Group

TeraFLOP computing on a desktop PC with GPUs for 3D CFD

J. Tölke* and M. Krafczyk

Institute for Computer Based Modeling in Civil Engineering, TU Braunschweig, Braunschweig, Germany

(*Received 13 May 2008; final version received 29 May 2008*)

A very efficient implementation of a lattice Boltzmann (LB) kernel in 3D on a graphical processing unit using the compute unified device architecture 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 magnitude with respect to the computational performance of a PC. A non-trivial example shows the performance of the LB implementation, which is based on a D3Q13 model that is described in detail.

Keywords: 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 (Bolz *et al.* 2003, Krüger and Westermann 2003, Buck *et al.* 2004). 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) (NVIDIA 2008). In comparison, the theoretical peak performance of the Intel Core 2 Duo 3 GHz is only 24 GFLOPS for double precision and 48 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.

Due to the facts that lattice Boltzmann methods (LBM) operate on a finite difference grid, are explicit in nature and usually require only next neighbour interaction, they are very suitable for the implementation on GPUs. In Li *et al.* (2003), 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 rasterisation 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 (Chu and Tai 2005), dispersion simulation and visualisation for urban security (Qiu *et al.* 2004), simulation of soap bubbles (Wei *et al.* 2004), simulation of miscible binary mixtures (Zhu *et al.* 2006), melting and flowing in multiphase environment (Zhao *et al.* 2006) and visual simulation of heat shimmering and mirage (Zhao *et al.* 2007). Even GPU clusters have been assembled for general-purpose computations (Fan *et al.* 2004) and LB simulations have been performed. An implementation of a Navier–Stokes solver on a GPU can be found in Wu *et al.* (2004). Nevertheless, all these applications use a programming style close to the hardware especially developed for graphics applications.

The remainder of the paper is organised as follows. The graphics hardware is shortly sketched in Section 2. The compute unified device architecture (CUDA) programming technology is presented in Section 3. The D3Q13 LB model is described in Section 4. The implementation of this LB model using CUDA is described in Section 5 and an example is given in Section 6. The performance of the approach is discussed in Section 7. Finally, Section 8 concludes the paper and provides a short outlook.

2. nVIDIA G80: the parallel stream processor

The G80-chip on an nVIDIA 8800 Ultra graphics card has 16 multiprocessors with 8 processors each, for a total of 128 processors. These are generalised 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 two clock cycles

*Corresponding author. Email: toelke@irmb.tu-bs.de

ISSN 1061-8562 print/ISSN 1029-0257 online

DOI: 10.1080/10618560802238275
http://www.informaworld.com

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, two floating point operations (FLOP) per cycle (one addition and multiplication), we obtain $2 \times 1.6 \times 128 \approx 410$ GFLOPS. Each multiprocessor has a single instruction, multiple data architecture.

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 $384/8 \times 1.1 \times 2 \approx 104$ double datarate RAM (DDR) 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 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.

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.

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 technology gives computationally intensive applications access to the processing power of nVIDIA 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 fast Fourier transform (FFT) and basic linear algebra subprograms (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

In this subsection, only a small subset of the application programming interface (API) needed for the LB kernel is discussed following NVIDIA (2008). 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 optimised API calls that utilise the device's high-performance direct memory access engines.

3.2.1 Thread block

A thread block is a batch of threads that can cooperate together by efficiently sharing data through some fast shared memory and synchronising their execution to coordinate memory accesses by specifying synchronisation 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 3D array and identify each thread using a three-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.

3.2.2 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

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 GHz)	48	10.7 (7)	4000	1000
NEC SX-8R A (Single node, 8 CPUs)	281	563	128,000	Expensive
nVIDIA 8800 Ultra (shader: 1.6 GHz)	410	104	768	500

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 synchronise with each other. Each block is identified by its block ID. An application can also specify a grid as a 2D array and identify each block using a two-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 65,535 in each dimension. In Figure 1, a 2 × 3 grid of thread blocks of size (3,1,1) and their indexing is shown.

3.2.3 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 parenthesised 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.

3.2.4 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 two clock cycles).

3.2.5 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.

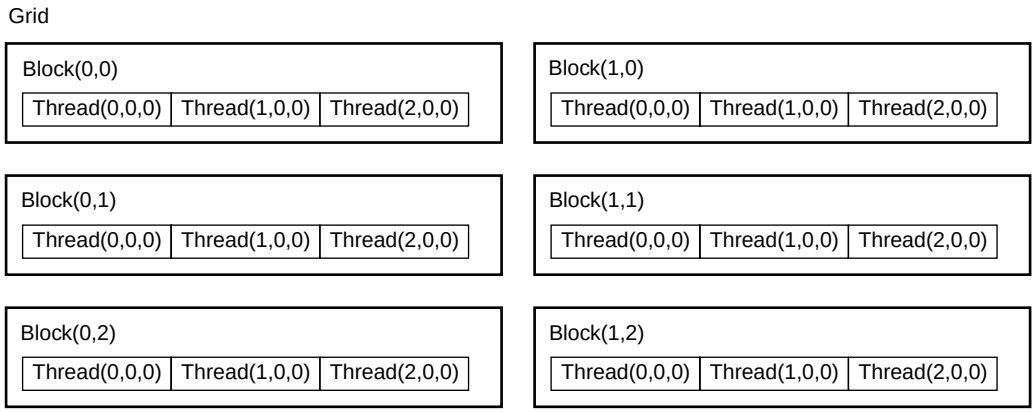

Figure 1. A 2 × 3 grid of thread blocks of size (3,1,1).

3.2.6 Synchronisation

The function `void _syncthreads()` synchronises 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 (Tutubalina 2008). 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. LBM: the D3Q13 model

The LBM is a numerical method to solve the Navier–Stokes equations (Frisch *et al.* 1987, Benzi *et al.* 1992, Chen and Doolen 1998), where mass fractions (with unit $\mathrm{kg\,m^{-3}}$) propagate and collide on a regular grid. In the following discussion, the font bold sans serif ($\mathbf{x}$) represents a 3D vector in space and the font bold with serif $\mathbf{f}$ a b-dimensional vector, where b is the number of microscopic velocities. We use the $D3Q13$ model (d'Humières *et al.* 2001) which is probably the model with the minimal set of velocities in 3D 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,

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

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 LB equation is

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

where f_i are mass fractions with unit $\mathrm{kg\,m^{-3}}$ propagating with microscopic velocity $\mathbf{e}_i$ and $\mathbf{\Omega}$ is the collision operator. The microscopic velocities or the mass fractions are also labelled 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**ottom**s**outh, **b**ottom**n**orth, **t**op**s**outh as $f_{\mathrm{r}}, f_{\mathrm{ne}}, f_{\mathrm{sw}}, f_{\mathrm{se}}, f_{\mathrm{nw}}, f_{\mathrm{te}}, f_{\mathrm{bw}}, f_{\mathrm{be}}, f_{\mathrm{tw}}, f_{\mathrm{tn}}, f_{\mathrm{bs}}, f_{\mathrm{bn}}$ and f_{ts}. The collision operator is given by

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

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

The moments m of the mass fractions are given by

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

where the moment ρ of zero order is the density variation 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

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

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

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

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

where s_e, s_ν, s'_ν, s_h are relaxation rates explained in more detail below.

Performing either a Chapman–Enskog (Frisch *et al.* 1987) or an asymptotic expansion (Inamuro *et al.* 1997, Junk *et al.* 2005) of Equation (2), it can be shown that the LBM 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'_ν are related to the viscosity by

$$
s_\nu = \frac{2}{8(\nu/c^2\Delta t) + 1}, \quad s'_\nu = \frac{2}{4(\nu/c^2\Delta t) + 1}. \tag{7}
$$

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 (Lallemand and Luo 2000). The optimal values for the MRT model depend 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.

The hydrodynamic pressure is given by

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

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:

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

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 Figure 2). If the boundary is not located exactly in the middle $\mathbf{x} + (1/2)\mathbf{e}_i\Delta t$ of the link i the boundary condition is only first order accurate. For higher order boundary conditions, we refer to Ginzburg and d'Humières (2003).

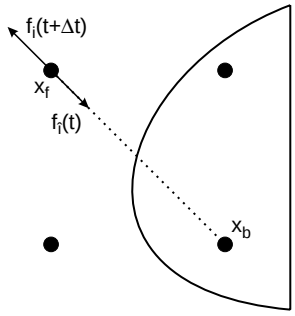

Figure 2. Momentum transfer on fixed obstacles.

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 (Nguyen and Ladd 2004) as shown in Figure 2. The momentum change can be computed by regarding the mass fraction before and after hitting the boundary:

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

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 LB kernel

A detailed overview of efficient implementation approaches of LB kernels for CPUs is given in Wellein *et al.* (2006). Since the architecture of the GPU is different, the implementation is also different from a design optimised 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 $m(nx, ny, nz, b)$ 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 indexes. 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 (d'Humières *et al.* 2001). Geometrical transformations or the possibility to run two simulations simultaneously on the grid to remove this staggered invariant are proposed in (d'Humières *et al.* 2001).

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 Figure 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 Figure 4) is a space-filling tessellation (or honeycomb) in

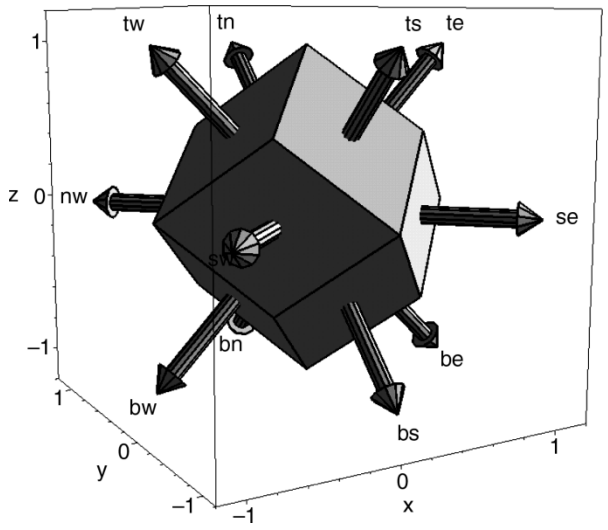

Figure 3. Basic unit cell for D3Q13 model: rhombic dodecahedron.

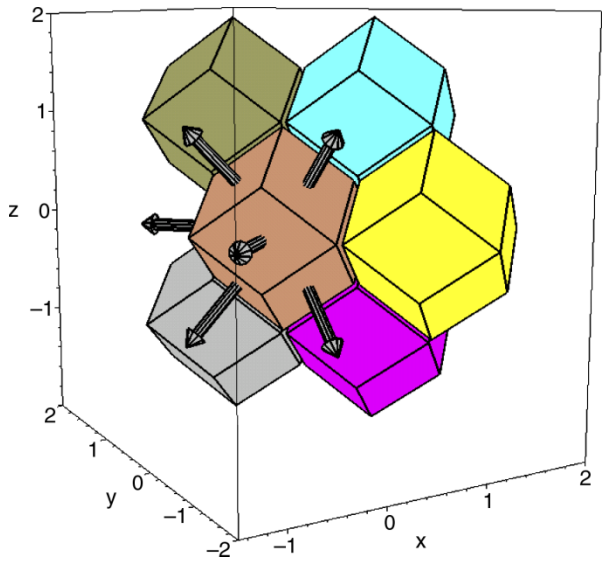

Figure 4. Rhombic dodecahedra honeycomb.

Euclidean 3-space. It is the Voronoi diagram of the face-centred cubic sphere packing, which is believed to be the densest possible packing of identical spheres in ordinary space. The honeycomb is cell-, face- 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 four cells. The vertices with the acute rhombic face angles have six 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, $V = 2h^3$.

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 $2 \times 13 = 26$ 1D arrays, one set for the current time step and one set for the new time step. This layout corresponds to the propagation optimised layout discussed by Wellein *et al.* (2006). The element $m = nx \times (ny \times k + j) + i$ in each of the 1D arrays is related to the position in space (x, y, z) by

$$a = \begin{cases} 0 & \text{if } j \text{ even and } k \text{ even} \\ 0 & \text{if } j \text{ odd and } k \text{ odd} \\ 1 & \text{if } j \text{ odd and } k \text{ even} \\ 1 & \text{if } j \text{ even and } k \text{ odd} \end{cases} \tag{13}$$

$$x = h(a + 2i) \quad y = hj \quad z = hk.$$

The values nx, ny and nz define the extensions of the grid. Note that $x = h(a + 2i)$ 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 neighbour $x + \Delta te_{x,l}$, $y + \Delta te_{y,l}$, $z + \Delta te_{z,l}$ 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)^(knew0x1);
int inew = (xi + ex[l]-a)/2;
int mm = nx*(ny*(knew) + jnew) + inew;
```

5.2 Single precision vs. double precision

The precision of float (32-bit) is 8 digits and of double (64-bit) are 16 digits. So mass- and momentum-conservation are locally guaranteed only up to this precision. We experienced no problems in terms of accuracy for the simulations we ran up to now. In Tölke (2008), the flow through a generic porous 2D medium (square array of 324 circles) was computed up to a relative error of 2.4×10^3 using single precision and simple bounce back. Problems reported with single precision and LB simulations (Skordos 1993) 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 (He and Luo 1997) 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.

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 restriction to 256 threads in our case come 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 set-up is possible. For a more flexible set-up in 2D, we refer to Tölke (2008). In Figure 5, the set-up 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` define 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 do not 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)`. An excerpt of the main loop is given below.

```
...
//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
```

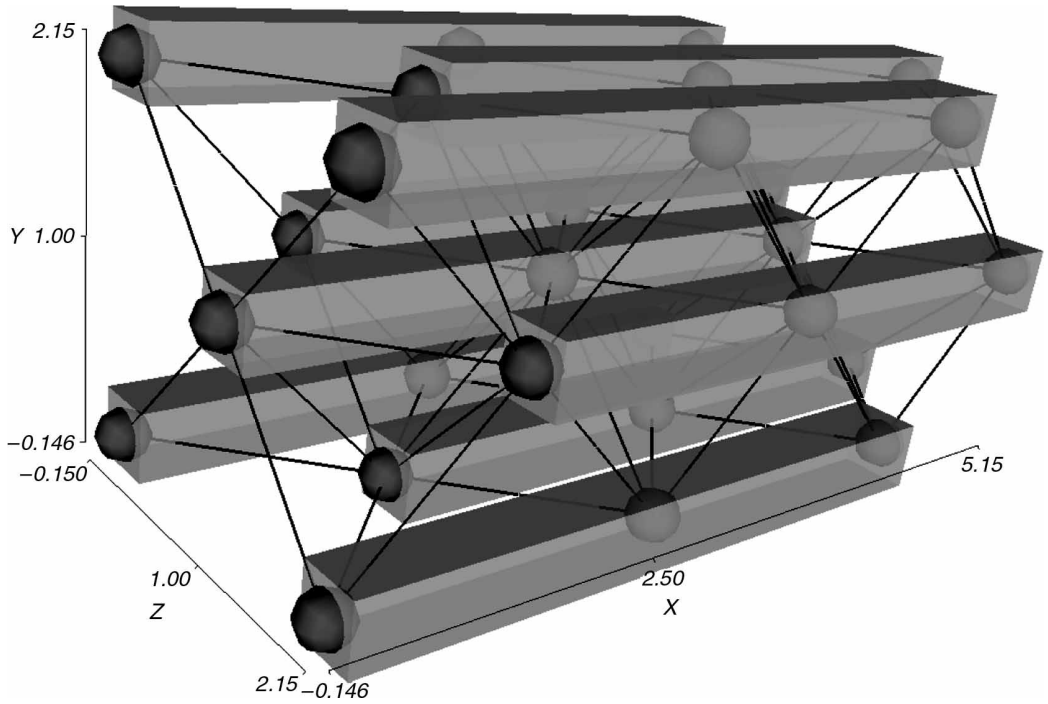

Figure 5. Mapping of physical lattice to computational grid.

```
    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 directly to the device memory leads to a substantial performance breakdown and the bandwidth is restricted to 10 GB/s. 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 Figure 6, the propagation is shown for the mass fractions f_{ne} in north-east direction in a plane $z = \text{const}$. 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 the bounce back rule for non-slip nodes or the velocity boundary condition is also integrated by an `if`-statement.

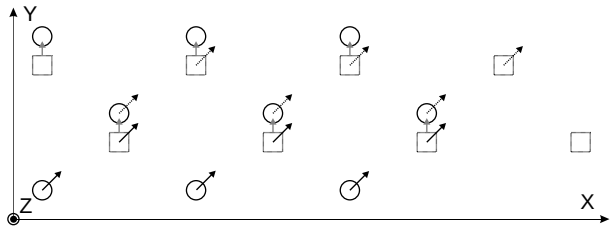

Figure 6. Propagation of mass fractions f_{ne} in north-east direction using shared memory: circles represent lattice as hold in device memory, squares represent shared memory. First, the mass fractions are written to shared memory and then transferred back uniformly to device memory.

6. Example: moving sphere in a circular pipe

An approximate solution (Schiller and Naumann 1933) for the dimensionless drag coefficient for a sphere moving with speed U_0 in an infinite fluid is given by

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

The Reynolds number is defined as

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

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 Fayon and Happel (1960)

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

where K is given by Haberman and Sayre (1958)

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

and $\lambda = d/D$ 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 Clift *et al.* (1978)

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

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 set-up shown in Figure 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.

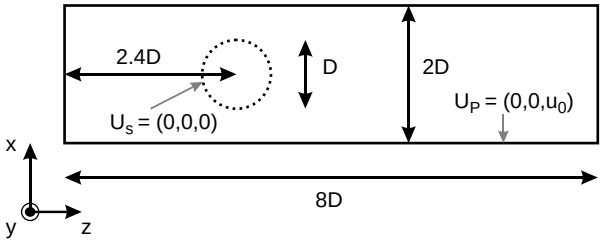

Figure 7. Moving sphere in a pipe, setup for numerical simulation.

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 are $c_{d,W} = 144.48$. 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 a basis #nodes is used. Mega-fluid-lattice updates per second (MFLUPS) represent 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 behaviour for $c_{d,W}$ with increasing mesh resolution.

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.

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

Domain size	u_0 (m s^{-1})	ν (m^2 s^{-1})	d (m)	# iter (−)	$c_{d,W}$ (−)	Rel. err. (−)	$nx \times ny \times nz$	P (MLUPS)	P (MFLUPS)	# nodes (−)	Mem (MB)
$32^2 \times 128$	0.004	0.0595	14.88	40,000	152.2	5.3%	16 × 32 × 128	239	188	65,536	23
$64^2 \times 256$	0.002	0.0605	30.24	80,000	146.6	1.5%	32 × 64 × 256	386	303	524,288	118
$128^2 \times 512$	0.001	0.0610	60.96	260,000	145.3	0.6%	64 × 128 × 512	582	457	4,194,304	693

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

Re (−)	ν ($m^2\,s^{-1}$)	WCT (s)	# iter (−)	$c_{d,W}$ (−)	$c_{d,W,\mathrm{Ref}}$ (−)	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] Nonstationary flow field, time required to reach oscillatory state from initial uniform flow field (no disturbance imposed).
[2] Average value, $t = 280 \ldots 2000\ T_{\mathrm{ref}}$.
[3] Average value, $t = 200 \ldots 3000\ T_{\mathrm{ref}}$.

The diameter D of the sphere is 60.95 m and the velocity boundary condition is $u_0 = 0.02\,\mathrm{m\,s^{-1}}$. 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 were computed by two methods:

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

In an advanced experimental set-up (Sakamoto and Haniu 1990) for a sphere in uniform flow a value of $Re \approx 300$, 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 (Johnson and Patel 1999, Tomboulides and Orszag 2000), where the blockage ration was small, a value of $Re = 280$ 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 LBM can be measured in lattice updates per second (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

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

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 FLOP per cell and time step. Considering the memory bandwidth as the limiting factor, we find $\mathsf{NB} = (14\ (\mathrm{read}) + 13\ (\mathrm{write})) \times 4$ bytes $= 108$ bytes per cell for the D3Q13 model. While memory bandwidth is given by the architecture, the average number NF of FLOP per cell depends on processor details, compiler and the implementation. We assume for the D3Q13 model 150 additions and 30 multiplications and choose $\mathsf{NF} = (30 + 30) + 2 \times (150 - 30) = 300\mathrm{FLOP}$, 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 $\mathsf{P} = 592$ 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 Wellein *et al.* (2006), very efficient CPU-implementations of the D3Q19 model are

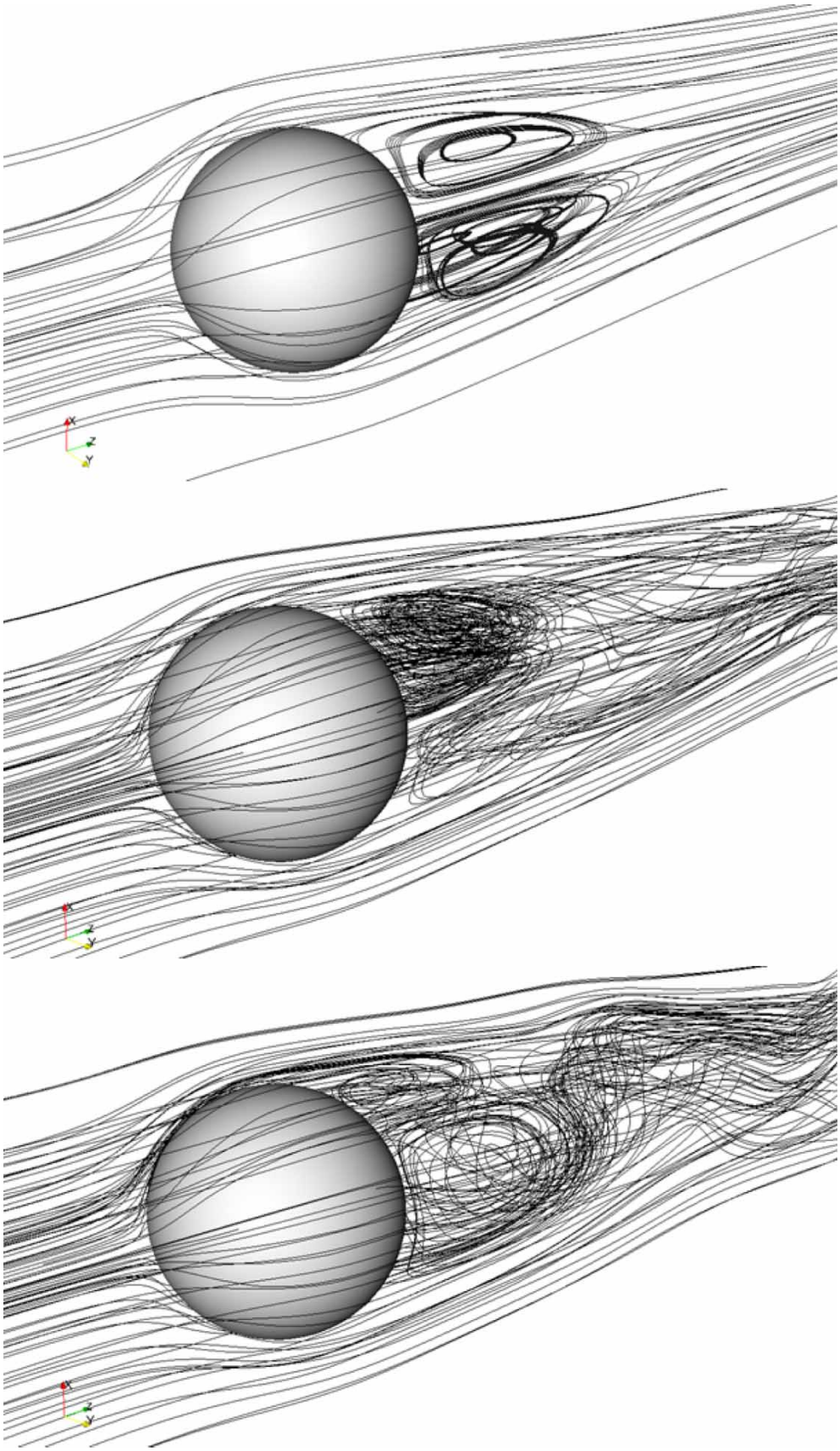

Figure 8. Streamlines for $Re = 200$, 300 and 400.

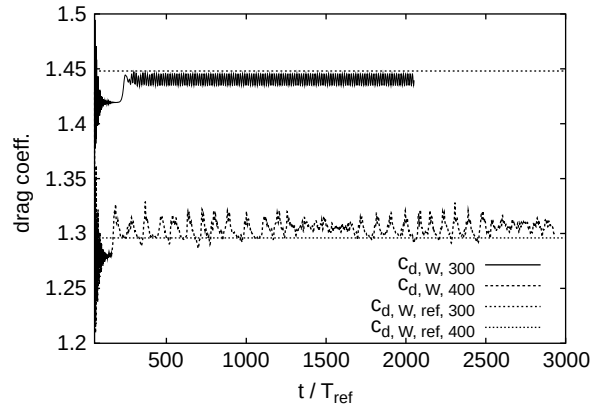

Figure 9. Drag coefficient for $Re = 300$ and 400 over time.

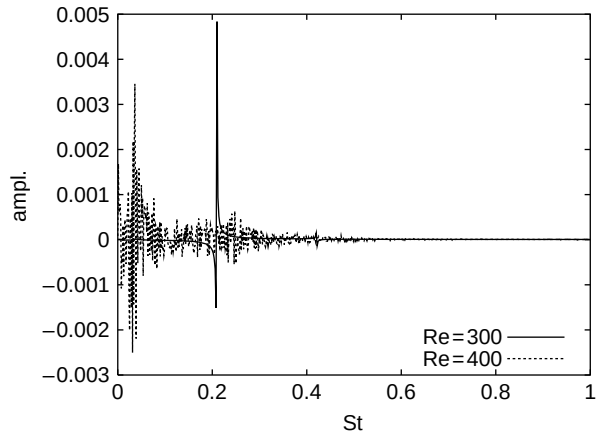

Figure 10. Amplitude of oscillation over Strouhal number.

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

$ny \times nz \backslash nx$	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

discussed. We give some values for comparison and for details, we refer to Wellein *et al.* (2006). Note that the data transfer volume for the D3Q19-model (double precision implementation) is $2 \times 19/13 = 2.92$ times higher. The performance in MLUPS was for Intel Xeon (3.4 GHz): $\mathsf{P} = 4.8$, Intel Itanium 2 (1.4 GHz): $\mathsf{P} = 7.6$ and for the vector machine NEC SX6 + (565 MHz): $\mathsf{P} = 41.3$.

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 write data from and to memory only at blocks aligned to `16 × 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 $\mathsf{D} =$ 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 discretisation 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.

References

Benzi, R., Succi, S. and Vergassola, M., 1992. The lattice Boltzmann equation: theory and applications. *Physics Reports*, 222 (3), 147–197.

Bolz, J., *et al.*, 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics (SIGGRAPH)*, 22 (3), 917–924.

Buck, I., *et al.*, 2004. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics (SIGGRAPH)*, 23, 777–786, http://graphics.stanford.edu/papers/brookgpu/.

Chen, S. and Doolen, G., 1998. Lattice Boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, 30, 329–364.

Chu, N. and Tai, C.L., 2005. MoXi: real-time ink dispersion in absorbent paper. *ACM Transactions on Graphics*, 24 (3), 504–511.

Clift, R., Grace, J.R. and Weber, M.E., 1978. *Bubbles, drops and particles*. New York: Academic Press.

d'Humières, D., Bouzidi, M. and Lallemand, P., 2001. Thirteen-velocity three-dimensional lattice Boltzmann model. *Physical Review E*, 63 (6), 066702.

Fan, Z., *et al.*, 2004. GPU cluster for high performance computing, In: Proceedings of ACM/IEEE Supercomputing Conference, 6–12 November, 47–59.

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

Frisch, U., *et al.*, 1987. Lattice gas hydrodynamics in two and three dimensions. *Complex Systems*, 1, 75–136.

Ginzburg, I. and d'Humières, D., 2003. Multireflection boundary conditions for lattice Boltzmann models. *Physical Review E*, 68, 066614.

Haberman, W.L. and Sayre, R.M., 1958. 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.

He, X. and Luo, L.S., 1997. Lattice Boltzmann model for the incompressible Navier–Stokes equation. *Journal of Statistical Physics*, 88, 927–944.

Inamuro, T., Yoshino, M. and Ogino, F., 1997. Accuarcy of the lattice Boltzmann method for small Knudsen number with finite Reynolds number. *Physics of Fluids*, 9, 3535–3542.
Johnson, T.A. and Patel, V.C., 1999. Flow past a sphere up to a Reynolds number of 300. *Journal of Fluid Mechanics*, 378, 19–70.
Junk, M., Klar, A. and Luo, L., 2005. Asymptotic analysis of the lattice Boltzmann equation. *Journal of Computational Physics*, 210, 676–704.
Krüger, J. and Westermann, R., 2003. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (SIGGRAPH)*, 22 (3), 908–916.
Lallemand, P. and Luo, L.S., 2000. Theory of the lattice Boltzmann method: dispersion, dissipation, isotropy, Galilean invariance and stability. *Physical Review E*, 61 (6), 6546–6562.
Li, W., Wei, X. and Kaufman, A., 2003. Implementing lattice Boltzmann computation on graphics hardware. *The Visual Computer*, 19 (7–8), 444–456.
Nguyen, N.Q. and Ladd, A., 2004. Sedimentation of hard-sphere suspensions at low Reynolds number. *Journal of Fluid Mechanics*, 525, 73–104.
NVIDIA, NVIDIA CUDA programming guide, 2008.
Qiu, F., *et al.*, 2004. Dispersion simulation and visualization for urban security. *IEEE Visualization*, 553–560.
Sakamoto, H. and Haniu, H., 1990. A study on vortex shedding from spheres in a uniform flow. *ASME Transactions Journal of Fluids Engineering*, 112, 386–392.
Schiller, L. and Naumann, A.Z., 1933. Über die grundlegenden Berechnungen bei der Schwerkraftaufbereitung. *Zeitschrift Des Vereines Deutscher Ingenieure*, 77 (12), 318–320.
Skordos, P.A., 1993. Initial and boundary conditions for the lattice Boltzmann method. *Physical Review E*, 48 (6), 4823–4842.
Tölke, J., 2008. Implementation of a lattice Boltzmann kernel using the compute unified device architecture. *Computing and Visualization in Science*, accepted.
Tomboulides, A.G. and Orszag, S.A., 2000. Numerical investigation of transitional and weak turbulent flow past a sphere. *Journal of Fluid Mechanics*, 416, 45–73.
Tutubalina, A., 2008. 8800 GTX performance tests, In Russian URL. http://blog.lexa.ru/2007/03/08/nvidia_8800gtx_skorost_chtenija_tekstur.html
Wei, X., *et al.*, 2004. Lattice-based flow field modeling. *IEEE Transactions on Visualization and Computer Graphics*, 10 (6), 719–729.
Wellein, G., *et al.*, 2006. On the single processor performance of simple lattice Boltzmann kernels. *Computers & Fluids*, 35 (8–9), 910–919.
Wu, E., Liu, Y. and Liu, X., 2004. An improved study of real-time fluid simulation on GPU. *Computer Animation and Virtual Worlds*, 15, 139–146.
Zhao, Y., *et al.*, 2006. Melting and flowing in multiphase environments. *Computers & Graphics*, 30 (4), 519–528.
Zhao, Y., *et al.*, 2007. Visual simulation of heat shimmering and mirage. *IEEE Transactions on Visualization and Computer Graphics*, 13 (1), 179–189.
Zhu, H., *et al.*, 2006. Simulation of miscible binary mixtures based on lattice Boltzmann method. *Computer Animation and Virtual Worlds*, 17, 403–410.

Appendix A. Orthogonal eigenvectors and transformation matrix

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

$$Q_{0,i} = 1 = (1,1,1,1,1,1,1,1,1,1,1,1,1), \tag{A1}$$

$$Q_{1,i} = e_{x,i} = c \cdot (0,1,-1,1,-1,1,-1,1,-1,0,0,0,0), \tag{A2}$$

$$Q_{2,i} = e_{y,i} = c \cdot (0,1,-1,-1,1,0,0,0,0,1,-1,1,-1), \tag{A3}$$

$$Q_{3,i} = e_{z,i} = c \cdot (0,0,0,0,0,1,-1,-1,1,1,-1,-1,1), \tag{A4}$$

$$Q_{4,i} = \frac{13}{2}\mathbf{e}^2 - 12c^2 = c^2 \cdot (-12,1,1,1,1,1,1,1,1,1,1,1,1), \tag{A5}$$

$$Q_{5,i} = 3e_{x,i}^2 - \mathbf{e}^2 = c^2 \cdot (0,1,1,1,1,1,1,1,1,1,-2,-2,-2,-2), \tag{A6}$$

$$Q_{6,i} = e_{y,i}^2 - e_{z,i}^2 = c^2 \cdot (0,1,1,1,1,-1,-1,-1,-1,0,0,0,0), \tag{A7}$$

$$Q_{7,i} = e_{x,i}e_{y,i} = c^2 \cdot (0,1,1,-1,-1,0,0,0,0,0,0,0,0), \tag{A8}$$

$$Q_{8,i} = e_{y,i}e_{z,i} = c^2 \cdot (0,0,0,0,0,0,0,0,0,1,1,-1,-1), \tag{A9}$$

$$Q_{9,i} = e_{x,i}e_{z,i} = c^2 \cdot (0,0,0,0,0,1,1,-1,-1,0,0,0,0), \tag{A10}$$

$$Q_{10,i} = e_{x,i}\left(e_{y,i}^2 - e_{z,i}^2\right) = c^3 \cdot (0,1,-1,1,-1,-1,1,-1,1,0,0,0,0), \tag{A11}$$

$$Q_{11,i} = e_{y,i}\left(e_{z,i}^2 - e_{x,i}^2\right) = c^3 \cdot (0,-1,1,1,-1,0,0,0,0,1,-1,1,-1), \tag{A12}$$

$$Q_{12,i} = e_{z,i}\left(e_{x,i}^2 - e_{y,i}^2\right) = c^3 \cdot (0,0,0,0,0,1,-1,-1,1,-1,1,1,-1). \tag{A13}$$

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

Appendix 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)&0 × 1)^((z-
startoffz)&0 × 1);
  int shiftW = 0 × 1 & (~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;
}
```