Fluid Simulation with Lattice Boltzmann Methods Implemented on GPUs Using CUDA

Pablo R. Rinaldi[1,2], Enzo A. Dari[3], Marcelo J. Vénere[3,2] and Alejandro Clausse[1,2,3],

[1]CONICET, [2]UNCPBA, [3]CNEA,
{prinaldi, venerem, clausse}@exa.unicen.edu.ar, darie@cab.cnea.gov.ar

Abstract. This work presents a parallel shared-memory implementation of a Lattice Boltzmann Model for Computational Fluid Dynamics. The model was specially coded for a Graphic Processing Unit using NVIDIA Compute Unified Device Architecture. Two-dimensional backwards-facing step flow simulation results were validated against a proved Navier-Stoke Finite Element solver. We obtain very promising speed-up results up to 40 times on a GeForce 8800 GT, compared with a normal "single core" PC-CPU.

Keywords: CUDA, GPU Computing, Lattice Boltzmann Methods.

1 Introduction

Lattice Boltzmann Methods (LBM) basically consists of a Cellular Automata applied to Computational Fluid Dynamics (CFD); it approximates incompressible Navier-Stokes equations in second order with a simple collision – advection scheme. In the last years, LBM has drawn considerably attention as efficient simulation model for complex fluids [1, 2, 3, 4]. Derived from the Lattice Gas Automata [5], discrete velocities act as unions between lattice cells while the union populations are dynamic variables. Even tough LBM applications field has grown considerably, some unresolved items (performance, stability, boundary conditions, grid refinement, etc.) delays a wide acceptance in CFD applications.

One of the mayor benefits of LBM is their easy parallelization. Like any explicit scheme CA, same code is run over entire grid in one time step. This makes LBM especially suitable for efficient implementations over massively parallel platforms like graphic hardware [6, 7, 8].

2 Graphic Processing Unit GPU

GPU is the main graphic card chip that renders pixels on the screen. Modern GPUs are optimized for executing a simple instruction (*kernel*) over each element in a large set simultaneously (i.e. SIMD, Simple Instruction Multiple Data). GPU's performance has overcome Moore's law by more than 2.4 times/year [9], mainly because of the lack of control circuits which uses more space in normal CPUs. In contrast, GPUs

have most of the chip space filled with Arithmetic Logical Units (ALU) raising their processing power. Figure 1 shows the transistors count with different tasks in a normal CPU compared with a GPU

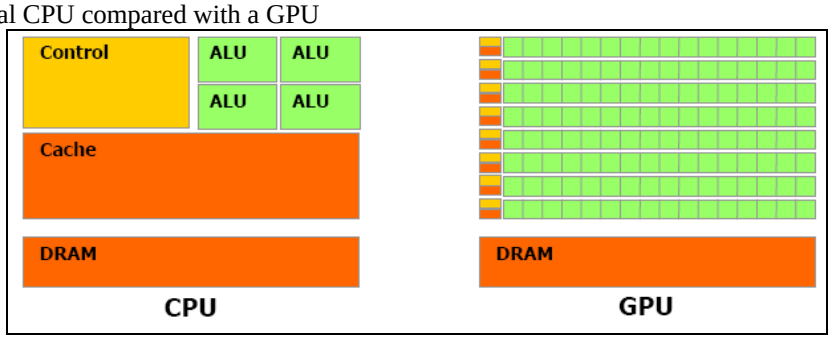

Figure 1. Transistors usage comparison. CPU (left) and GPU(right) [10]

The graphic card used in this work is NVIDIA GeForce 8800 GT from the G80 series GPUS. G80 cards were developed together with the CUDA programming model from the same brand NVIDIA. GPUs from this series are made up by a set of SIMD multiprocessors, like shown in figure 2. This means that each processor inside the multiprocessors, executes the same instruction at each clock cycle simultaneously over different data.

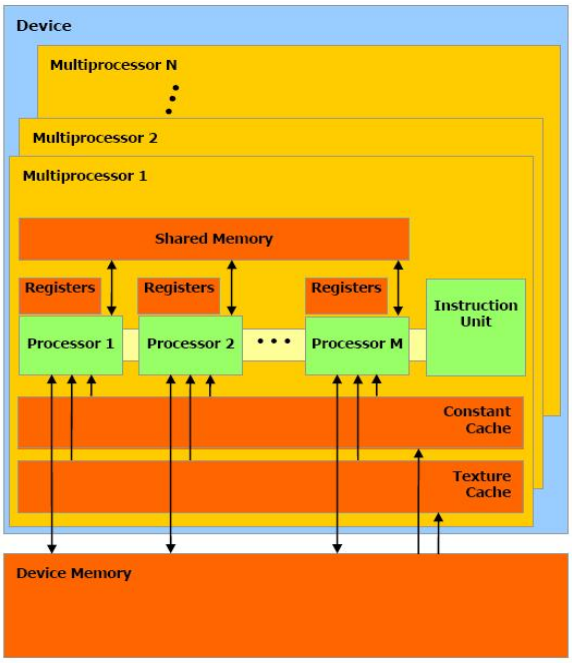

Figure 2. SIMD multiprocessors with shared memory inside G80 NVIDIA GPU. [10]

NVIDIA CUDA technology is a new architecture that allows the graphic hardware to solve complex computational problems. CUDA programming interface give access

to GPU hardware processing power to computationally intensive applications. It uses standard C language and has software compatibility with the most used operative systems. In this work we used the 1.1 release over Debian GNU/Linux OS.

2.1 Application Programming Interface (API)

The GPU, called *device* is seen like a computational device capable of executing multiple parallel execution threads. It works like a co-processor to the main CPU called *host*. Computational intensive applications are offloaded from the CPU to the GPU using CUDA function calls. The *host* and the *device* keep its own RAM memory. Data can be copied between RAMs using high performance optimized Direct Memory Access (DMA) methods provided.

2.1.1 Threads Blocks:

A thread block is a group of executing threads that communicate efficiently by sharing data through fast access shared memory and coordinating data access by synchronization points. To maximize hardware performance a block must contain at least 64 threads and no more than 512 [7].

2.1.2 Grids of Thread Blocks:

There are a hardware limited number of threads that can be launched in a single execution block: 512, it can be even less depending on registers number or shared memory needed for each thread. However, blocks executing the same kernel code can be grouped in a grid of thread blocks. With this, the number of threads that can be executed simultaneously is much bigger. The only tradeoff is that threads from different blocks cannot communicate in a safe way or synchronize. Different blocks in a grid can run in parallel and at least 16 blocks must be launched to efficiently use the graphic hardware [7].

3 Lattice Boltzmann Methods

LBM are an evolution of Lattice-Gases, initially developed to overcome numerical noise problems [11]. The basic concept behind LBM is building a messoscopic kinetic model with discrete internal variable (i.e. velocity) which macroscopic properties obeys the macroscopic desired equations [12].

3.1 LBM BGK Equation

The Lattice Boltzmann equation named BGK (Bhatnagar-Gross-Krook)[13] over two dimensions nine velocities square grid (d2Q9) is presented as in [14, 15]:

$$f_i(x+\delta e_i, t+\delta) - f_i(x,t) = -\frac{1}{\tau}\left[f_i(x,t) - f_i^{(eq)}(x,t)\right], \quad i = 0,1,...,8 . \quad (1)$$

where equation 1 is in physical units. Both, time and space scales are valued $\boldsymbol{\delta}$ in physical units. Density distribution function along e_i direction for the cell in $\boldsymbol{x}$ position at $\boldsymbol{t}$ time (x, t) is denoted $f_i(x,t)$. The particle velocity $\boldsymbol{e_i}$ is given by:

$$\begin{aligned} e_i &= 0 && i = 0 \\ e_i &= \left(\cos\left(\frac{\pi(i-1)}{2}\right),\ \sin\left(\frac{\pi(i-1)}{2}\right)\right) && i = 1,2,3,4 \\ e_i &= \sqrt{2}\left(\cos\left(\frac{\pi\left(i-4-\frac{1}{2}\right)}{2}\right),\ \sin\left(\frac{\pi\left(i-4-\frac{1}{2}\right)}{2}\right)\right) && i = 5,6,7,8\ . \end{aligned} \tag{2}$$

Figure 3 shows velocity directions in d2Q9 model

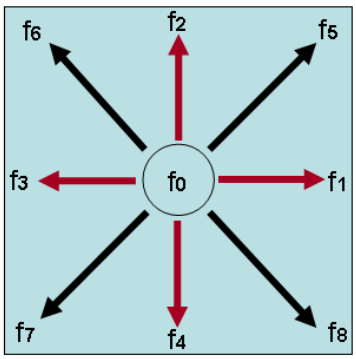

Figure 3. LBM distribution functions in d2Q9 model.

The collision term is represented by the right side of equation 1, and the simple temporal relaxation that controls the equilibrium approximation range is $\boldsymbol{\tau}$. Node density $\boldsymbol{\rho}$, and fluid macroscopic velocity $\boldsymbol{u=(u_x,\ u_y)}$, are defined in terms of the distribution functions in the following way:

$$\sum_{i=0}^{8} f_i = \rho, \qquad \sum_{i=0}^{8} f_i e_i = \rho u\ . \tag{3}$$

Equilibrium distribution function $f_i^{(eq)}(x,t)$ depends only from local density and velocity as shown in equation 4 [15]:

$$\begin{aligned} f_i^{(eq)} &= t_i \rho\left[1 + 3(e_i \cdot u) + \frac{9}{2}(e_i \cdot u)^2 - \frac{3}{2} u \cdot u\right], \\ t_0 &= \frac{4}{9},\ t_i = \frac{1}{9},\ i = 1:4; \quad t_i = \frac{1}{36},\ i = 5:8. \end{aligned} \tag{4}$$

Pressure is given by $p = c_s^2 \rho$, where c_s is the speed of sound been $c_s^2 = \frac{1}{3}$, and the cinematic viscosity is:

$$\nu = \left[\frac{(2\tau - 1)}{6} \right] \delta \tag{5}$$

3.2 Advection Step

The first operation during each iteration is to calculate the particles displacement to the neighbor cells along their corresponding directions. This step, called "advection" (see figure 4) each cell interchanges particles with neighbor cells following this rule:

$$f^a(\vec{e}, \vec{x}, t) = f(\vec{e}, \vec{x} - \vec{e}\,\delta t, t - \delta t) \tag{6}$$

where f^a is de distribution function.

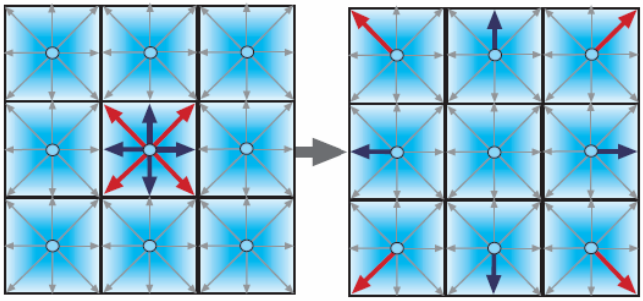

Figure 4. Advection step

3.3 Boundary Conditions

Boundary conditions are one of the more complex issues in LBM implementations. Basically two different kinds of conditions where implemented in this work: Non-slip wall boundary conditions and In-out opened boundary conditions.
No-slip conditions where used in the channel walls implementation, meaning zero velocity in every direction on wall cells. To achieve this on-grid bounce-back scheme [11] was followed (figure 5). It is well known that directly reverting the populations into the fluid gives only first order accuracy, so we implemented a modified version proposed by Zou and He [16]. For example, unknown *f's* in an upper wall cell are defined like this:

$$f_4 = f_2, \quad f_7 = f_5 + \frac{1}{2}(f_1 - f_3), \quad f_8 = f_6 - \frac{1}{2}(f_1 - f_3) \tag{7}$$

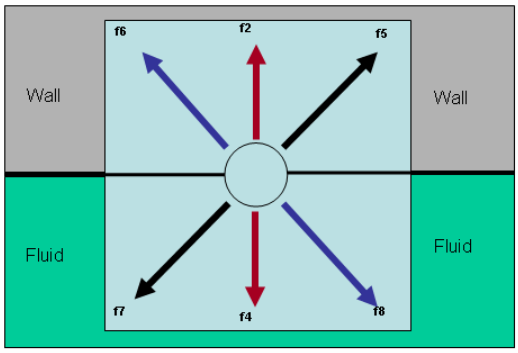

Figure 5. On-grid bounce-back boundary conditions

Any other wall is treated in similar way to equation 5, starting from the fact that to have total velocity zero must be:

$$\begin{aligned} \rho u &= f_1 + f_5 + f_8 - f_3 - f_6 - f_7 = 0, \\ \rho v &= f_2 + f_5 + f_6 - f_4 - f_7 - f_8 = 0 \ . \end{aligned} \tag{8}$$

To implement the in-out flow condition, *velocity-pressure boundary-condition* proposed by Zou and He [16] was used. Little modification was done to force quadratic parabolic profile in the channel inlet. The *Velocity-Pressure Boundary* (VPB) scheme, assuming a left-right channel flow, fixes vertical velocity zero and horizontal velocity ***u*** at the inlet. So, for the inlet left boundary we have:

$$\rho = \frac{f_0 + f_3 + f_7 + 2(f_5 + f_4 + f_6)}{1 - u} \tag{9}$$

$$f_1 = f_5 + \frac{2}{3}\rho u \tag{10}$$

$$f_2 = f_6 + \frac{1}{6}\rho u - \frac{1}{2}(f_3 - f_7) \tag{11}$$

$$f_8 = f_4 + \frac{1}{6}\rho u + \frac{1}{2}(f_3 - f_7) \tag{12}$$

Pressure is fixed in the right outlet wall. That is the same that fixing the proportional value ***ρ.*** Velocity component in y direction is also set to zero. For the outlet right boundary we have:

$$u = \frac{f_0 + f_3 + f_7 + 2(f_1 + f_2 + f_8)}{\rho} - 1 \tag{13}$$

$$f_5 = f_1 - \frac{2}{3}\rho u \tag{14}$$

$$f_4 = f_8 - \frac{1}{6}\rho u - \frac{1}{2}(f_3 - f_7) \tag{15}$$

$$f_6 = f_2 - \frac{1}{6}\rho u + \frac{1}{2}(f_3 - f_7) \tag{16}$$

4 CUDA LBM Implementation

For the physical implementation, eighteen one-dimension floating point arrays were used to represent the grid (one for each *f* direction in both time steps *t* and *t+1*). Rectangular grids were implemented with different refinement sizes, this sizes should be multiple of 16 for occupancy and performance issues [17]. Simulated walls are located in the middle of border cells because of *on-grid bounce-back* strategy, so that one more cell is needed for channel flow [16]. Hence, maximal possible scale in backward facing step with ratio $H/h = 2$ is $\delta = 1/(n-2)$, been *n* the vertical size of the grid in cell units. Boundary conditions were defined as cell type. 12 different advection steps were implemented, one for each type of cell:

1. Flow: Internal cells, classical advection.
2. Inlet: Left open channel wall cells, inlet velocity condition is applied.
3. Outlet: Right open channel wall cells, outlet pressure condition is applied.
4. Upper wall: On-grid bounce-back condition. Upper half is solid, lower half is flow.
5. Bottom wall: On-grid bounce-back condition. Upper half is flow, lower half is solid.
6. Left wall: On-grid bounce-back condition. Right half is flow, left half is solid. (used in the expansion)
7. Upper left corner: Inlet and bounce-back condition.
8. Lower left corner of the channel: Inlet and bounce-back condition.
9. Upper right corner: Outlet and bounce-back condition.
10. Lower right corner: Outlet and bounce-back condition.
11. Expansion corner: Double bounce- back. Lower left quarter is solid, rest is flow.
12. Null: Extra domain cells. No advection is needed.

Individual cell type is stored in integer extra array with the same dimensions than LBM grid. This is a flexible tool to change geometry with only changing this parameter and adding some advection rule.

4.1 The Algorithm

The explicit lattice Boltzmann algorithm as outlined above can be easily implemented. A straightforward implementation could consist of two nested loops over the two spatial dimensions and treat the collision step independently from the propagation process and the boundary conditions.

The number of memory data transfers can be reduced by executing the collision process and propagation step in one loop. Using two arrays, holding data of successive time steps t and $t+1$ keeps the implementation simple and avoids data dependencies between the two time steps allowing parallelization. During an update, values are read from one array and written to the other including the propagation step (within the reading or writing step). At the end of each time step the two arrays are switched, i.e. the source becomes the destination and vice-versa. The propagation step is realized as first step of the iteration loop, resulting in a ‘‘pull’’ scheme of the update process [18].

1. Read distribution functions from adjacent cells, i.e. $f_i\left(\vec{x}-\vec{e}\,\delta t, t-\delta t\right)$
2. Apply boundary conditions to obtain missing fs.
3. Calculate ρ, $\vec{u}$ and $f_i^{(eq)}$
4. Write updated values to current cell, i.e. $f_i\left(\vec{x}, t\right)$

On cache-based machines, main memory bandwidth and latencies are serious bottlenecks for memory intensive applications like LBM [18]. In shared memory architectures, with no cache like G80 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 otherwise memory bandwidth performance breaks down to about 10 GB/sec [7]. Also to minimize global memory access each thread of a block must stage data coming from device memory into shared memory.

4.2 Execution Configuration.

Initial grid dimensions 32 x 96 allowed to parallelize implementation to cell level i.e. each thread of a block executes the code of only one cell. This could be the reason why performance declines in for the biggest grids tested. See table 1.

5 Results

5.1 Backwards Facing Step

The two-dimensional backwards-facing step flow as seen in figure 6 was used to validate the LBM implementation.

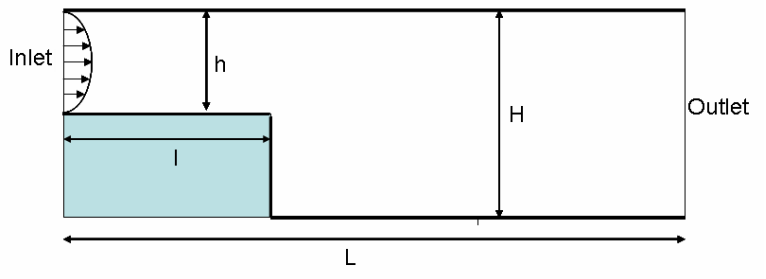

Figure 6. Backwards-facing step channel

Geometry of the setup was as follows: The channel length (L) was 3, the channel width was 0.5 at the inlet (h) and 1.0 at the outlet (H), so that 0.5 is the backwards facing step height. The inflow was a quadratic velocity profile in a duct flow while the outflow condition was a fixed pressure value and vertical zero velocity.

5.2 Validation

The same backward-facing step simulation was done using a Navier-Stokes “equal-order” finite-element (FE) solver for result validation. This FE solver uses equal velocity pressure interpolation and sub-grid scale (SGS) stabilization, it solves stationary state by pseudo-transitory. A regular grid with 1821 triangular elements was used. Boundary conditions are sets over 137 elements. Both simulation, FE and CUDA-LBM were done measuring 100 seconds starting from steady state. Only 100 simulation steps were needed for FE, but 3000 iterations were used for the explicit LBM version. Contours velocity fields and velocity profiles of both simulations corresponding to a Reynolds number equal to 100 are shown in the following figures.

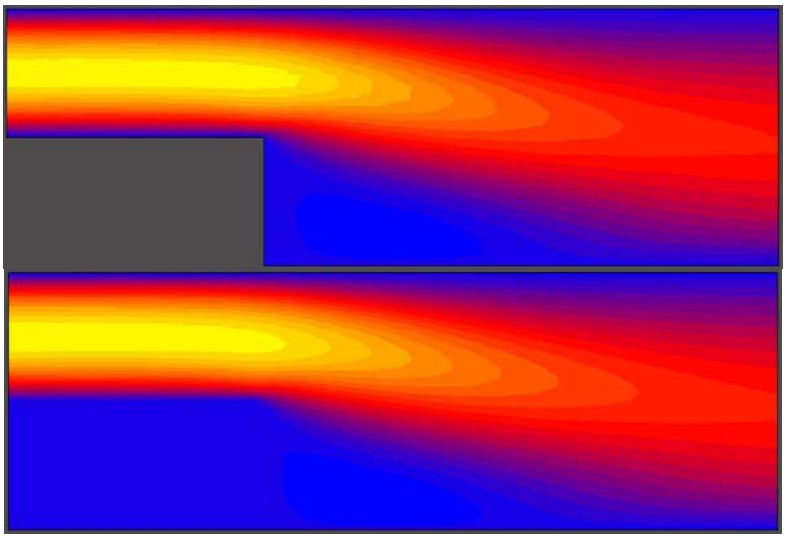

Figure 7. Velocity zones graphics showing *Ux*. Finite Elements (above), Lattice Boltzmann over GPU (below)

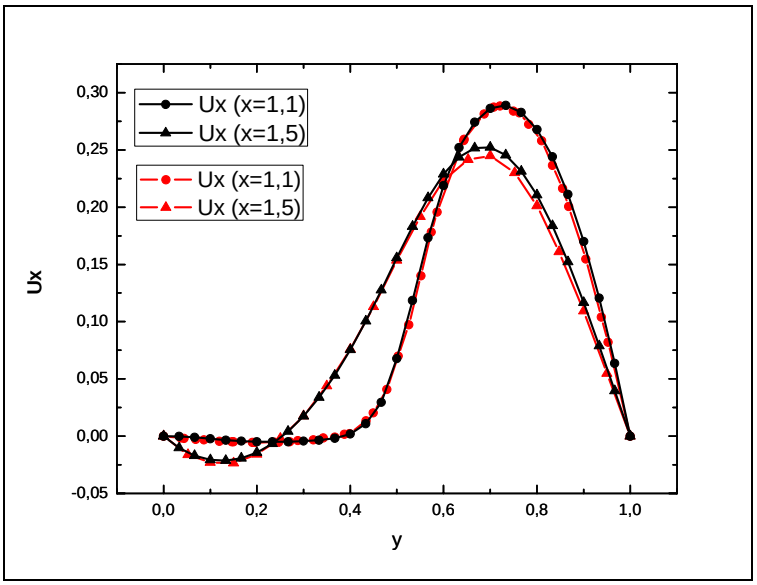

Figure 8. Velocity profiles in two sections of the channel passing the sudden expansion. Red curves stands for FE and black ones for LBM.

The measured execution time for FE application running on Intel Core2 Quad 2.4 GHz and 8 Gb. of RAM over Debian GNU/Linux Operative System was about 13,28 seconds for the entire 100 seconds simulation.

We report this execution time only to show that it is not exceeded by LBM simulations. LBM code presented was specially tuned for 2D flow simulations unlike the FE solver, which is a software simulation tool with much larger application areas. Also a FEM formulation should be much more accurate for the same grid size than LBM.

5.3 Performance

The best unit for performance measuring in LBM is MLUPS (Mega Lattice-Site Updates per Second) and accounts for the time to calculate one time step for a given grid [19]. In table 1 different performance results are shown for the simulation made in this work. Columns list in this order the total grid dimension, the best execution configuration tested, the time consumed by one iteration (averaged from 1000), calculated MLUPS, the time consumed by the similar CPU implementation and the calculated speedup ratio.

Table 1. Performance results averaged from 1000 iterations.

Grid Size	Blocks/Thread Configuration	GPU Time per Iteration (ms)	Average MLUPS	CPU Time per Iteration (ms)	Speedup
(96 x 32) 3072	1 x 32 blocks 96 threads/block	0.041	75	0.59	14.4
(192 x 64) 12288	3 x 64 blocks 64 threads/block	0.085	144	2.1	24.7
(384 x 128) 49.152	6 x 128 blocks 64 threads/block	0.31	158	13.47	43.45
(768 x 256) 196608	6 x 256 blocks 128 threads/block	1.29	152	54.87	42.53
(1.536 x 512) 786432	6 x 512 blocks 256 threads/block	18.23	43	215.52	11.82

As we mentioned in section 4.2, different parallelizations schemes should be used for larger grids in order to keep the number of total threads under 200000 for the same GPU.

6 Conclusions

Recent work on fluid simulation with LBM over High performance CPUs like [20] reaches less than 6 MLUPS for a simple CPU (Power4 1,7 GHz) for 3D flows with complex boundary conditions but very low Reynolds numbers. Only multi-core supercomputers like Hitachi SR8000-F1 (8 cores) with COMPAS intra-node parallelization implementations can do more than 150 MLUPS [21]. LBM

implementations using CUDA like [7] don't improve 1 MLUPS while simulating complex phenomena with proved results validation.

In this work we simulated complex phenomena validating its result against other models with more than 150 MLUPS of average performance over 1000 steps. This performance may be enhanced by tuning the application for the special case input data, for example: replacing variable passing by constant declarations. Also we achieve a speedup of more than 40 times over an equal CPU implementation.

We also realized that the key to speed-up LBM simulations is improving memory management strategy for a better access pattern.

7 References

1. Higuera F. and Succi S., Simulating the flow around a circular cylinder with a lattice Boltzmann equation, Europhys, Lett. 8, 517 (1989)
2. Karlin I. V., Ferrante A. and Öttinger H. C. Perfect entropy functions of the Lattice Boltzmann method, Europhys. Lett. 47, 182 (1999)
3. Shan X. and He X., Discretization of the Velocity Space in the Solution of the Boltzmann Equation Phys. Rev. Lett. 80**,** 65 **(**1998)
4. Ansumali S., Karlin I. V. and Öttinger H. C. Consistent Lattice Boltzmann Method, Europhys. Lett. 63, 798 (2003)
5. Boon J.-P. and Rivet J.-P., Lattice Gas Hydrodynamics. Cambridge University Press, Cambridge (2001)
6. Goodnight, N. CUDA/OpenGL Fluid Simulation. http://new.math.uiuc.edu/MA198-2008/schaber2/fluidsGL.pdf
7. Tölke, J. Implementation of a Lattice Boltzmann kernel using the Compute Unified Architecture Developed by nVIDIA. http://www.irmb.tu-bs.de/UPLOADS/toelke/Publication/toelked2q9.pdf
8. Zhao, Ye. Lattice Boltzmann based PDE Solver on the GPU. Visual Comput 2007. DOI 10.1007/s00371-007-0191-y. Springer-Verlag (2007)
9. Moreland, K. and Angel, E. The FFT on a GPU. SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware, M. Doggett, W. Heidrich, W. Mark, and A. Schilling, Eds., Eurographics Association. San Diego, California. Eurographics Association. pp. 112–119 (2003)
10. NVIDIA, NVIDIA CUDA Home Page. http://www.nvidia.com/object/cuda_home.html
11. Succi S. The Lattice Boltzmann Equation for Fluid Dynamics and Beyond. Numerical Mathematics and Scientific Computation. Clarendon Press, Oxford (2001)
12. Chen S. and Doolen G. D. Lattice Boltzmann Methods for Fluid Flous. Annu. Rev. Fluid Mech. 30, 329, (1998)
13. Bhatnagar, P., Gross, E. and Krook, M. A model for collisional processes in gases I: small amplitude processes in charged and neutral one-component system. Phys. Rev. 94, 511 (1954)
14. Chen, S., Chen, H., Martinez, D. O. and Matthaeus, W. H. Lattice Boltzmann model for simulation of magnetohydrodynamics. Phys. Rev. Lett. 67, 3776 (1991)
15. Qian, Y., d'Humie`res, D., and Lallemand, P. Recovery of Navier–Stokes equations using a lattice-gas Boltzmann method. Europhys. Lett. 17, 479 (1992)
16. Zou, Q. and He, X., On pressure and velocity boundary conditions for the lattice Boltzmann BGK model, Phys. Fluids, vol.9 (6), 1591-1598 (1997)

17. NVIDIA. NVIDIA CUDA Compute Unified Device Architecture – Programing Guide. http://developer.download.nvidia.com/
18. Wellein, G., Zeiser, T., Hager, G., Donath, S. On the single processor performance of simple lattice Boltzmann kernels, Computers & Fluids, vol. 35, 910-919 (2006)
19. Lammers, P. and Küster U. Recent Performance Results of the Lattice Boltzmann Method, High Performance Computing on Vector Systems 2006. Part 2. 51-59 (2007)
20. Mazzeo, M.D., Coveney, P.V. HemeLB: A high performance parallel lattice-Boltzmann code for large scale fluid flow in complex geometries, Computer Physics Communications Vol. 178, Issue 12, 894-914 (2008)
21. Pohl, T., Deserno, F., Th¨urey, N., R¨ude1, U., Lammers, P., Wellein, G., Zeiser, T. Performance Evaluation of Parallel Large-Scale Lattice Boltzmann Applications on Three Supercomputing Architectures, Proceedings of the 2004 ACM/IEEE conference on Supercomputing, 21 (2004)