Contents lists available at SciVerse ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

A Lattice-Boltzmann solver for 3D fluid simulation on GPU

P.R. Rinaldi [a,*], E.A. Dari [b], M.J. Vénere [a], A. Clausse [a]

[a] CNEA-CONICET and Universidad Nacional del Centro, 7000 Tandil, Argentina
[b] CONICET-CNEA and Instituto Balseiro, 8400 Bariloche, Argentina

ARTICLE INFO

Article history:
Received 24 October 2011
Received in revised form 26 February 2012
Accepted 13 March 2012
Available online 9 April 2012

Keywords:
GPGPU
3D Lattice-Boltzmann Methods
CUDA

ABSTRACT

A three-dimensional Lattice-Boltzmann fluid model with nineteen discrete velocities was implemented using NVIDIA Graphic Processing Unit (GPU) programing language "Compute Unified Device Architecture" (CUDA). Previous LBM GPU implementations required two steps to maximize memory bandwidth due to memory access restrictions of earlier versions of CUDA toolkit and hardware capabilities. In this work, a new approach based on single-step algorithm with a reversed collision–propagation scheme is developed to maximize GPU memory bandwidth, taking advantage of the newer versions of CUDA programming model and newer NVIDIA Graphic Cards. The code was tested on the numerical calculation of lid driven cubic cavity flow at Reynolds number 100 and 1000 showing great precision and stability. Simulations running on low cost GPU cards can calculate 400 cell updates per second with more than 65% hardware bandwidth.

1. Introduction

The Lattice-Boltzmann Method (LBM) is a class of cellular automata (CA) that approximates the Navier–Stokes equations to second order with an explicit collision–advection scheme [4]. Derived from the Lattice Gas Automata [2], discrete velocities act as arcs between lattice cells whose populations are state variables. Like any explicit CA scheme the same simple code is run over the entire grid in every time step, making LBM especially suitable for parallel implementations [5,15,18].

An interesting economic alternative for parallelizing LBM are the Graphic Processing Units (GPUs) [6,16,20]. A GPU is the chip used by graphic cards to render pixels on the screen. Modern GPUs are optimized for executing single-instruction multiple threads (SIMT) that is a simple kernel over each element of a large set simultaneously operating as a co-processor of the host CPU [12]. Several researchers have shown that the combination of GPU and Parallel CA is a valid tool to simulate fluids [6,8,16,20]. Although there has been interesting advances in this type of implementations, inter-processor data communication continues limiting the calculation performance, thus memory access optimizations is always necessary to take good advantage of GPU processing power. Particularly, in "Compute Unified Device Architecture" (CUDA) language, parallel global memory calls should be coalesced to reduce latency [13].

In this article, a novel efficient CUDA implementation of the LBM based in a single-loop "pull" scheme is presented. The new implementation makes use of a new version of the CUDA language, with less memory alignment restrictions than the used in previous works [8]. The implementation is tested in 3D flow simulations over a GPU-NVIDIA GeForce card achieving high performances and near 65% of the hardware theoretical bandwidth.

* Corresponding author. Address: Instituto PLADEMA, UNCPBA, Pinto 399, Tandil 7000, Argentina. Tel./fax: +54 249 4439690.
E-mail address: prinaldipablo@gmail.com (P.R. Rinaldi).

1569-190X/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.simpat.2012.03.004

2. NVIDIA CUDA programming model

NVIDIA GeForce GTX 260 is a graphic card of the GTX200 series, composed of a set of multiprocessors with SIMD architecture. Each processor within a multiprocessor executes the same instruction in every clock cycle simultaneously over different data. Table 1 details the main characteristics of the GPU.

NVIDIA developed the programming language CUDA for computationally intensive applications in GPU [11] based on standard C language, which greatly simplifies the software implementation. The GPU, called device, is seen like a computational unit capable of executing multiple parallel execution threads. It works like a co-processor to the main CPU, which is called host. Computational intensive applications are offloaded from the CPU to the GPU using CUDA function calls. The host and the device use their own RAM memory. Data can be copied between RAMs using optimized Direct Memory Access methods (DMA) provided by CUDA. CUDA uses several memory spaces including global, local, shared, texture, and registers. Global, local, and texture memory have the greatest access latency, followed by constant memory, registers, and shared memory [13].

3. Lattice-Boltzmann Method

LBM is basically a mesoscopic kinetic model with a discrete internal velocity variable, whose average magnitudes obey some macroscopic field equations [4]. In the present work, the BGK instance of LBM is used for solving 3D incompressible Navier–Stokes equations [1,3,14]. Defining $f_i(\vec{x},t)$ as the population of particles in $(\vec{x},t)$ having velocity e_i, the basic rule of LBM is:

$$f_i(\vec{x}+\delta t\cdot e_i, t+\delta t) - f_i(\vec{x},t) = \frac{1}{\tau}\left[f_i^{(eq)}(\vec{x},t) - f_i(\vec{x},t)\right], \quad i = 0,1,\ldots,18 \tag{1}$$

where the equilibrium function $f_i^{(eq)}(\vec{x},t)$ defines the macroscopic equations that the automata simulates. To simulate the Navier–Stokes equations, BKG uses the following function [4]:

$$f_i^{(eq)} = w_i\rho\left[1 + 3(e_i\cdot u) + \frac{9}{2}(e_i\cdot u)^2 - \frac{3}{2}u\cdot u\right] \tag{2}$$

where w_i are the weights corresponding to each e_i lattice velocity [7]. For a 3D grid with 19 internal velocities, d3Q19 (Fig. 1):

$$w_0 = \frac{1}{9}; \qquad w_i = \frac{1}{18}, \quad i = 1:6; \qquad w_i = \frac{1}{36}, \quad i = 7:18 \tag{3}$$

and

$$\begin{aligned}\rho &= \sum_i f_i \\ \rho u &= \sum_i f_i e_i\end{aligned} \tag{4}$$

are the macroscopic density and mass flux respectively.

Eqs. (1) and (2) consists of two steps, advection and collision, combined with a linear relaxation. The fluid viscosity can be controlled via the relaxation parameter τ, as:

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

where e is the lattice velocity, $\Delta x/\Delta t$. In the present study, on-grid bounce-back boundary conditions were implemented, which was developed by Maier et al. [10], extended to second order by [21] and generalized to d3Q19 by Hecht and Harting [7]. This condition allows arbitrary boundary conditions of pressure and/or velocity.

Table 1
GPU hardware specifications [12].

Geforce 260 GTX	
Number of stream processors	192
Global memory size (MB)	896
Memory bus width (bits)	448
Memory bandwidth (GB/s)	111.9
Estimated peak performance (Gflops)	805
Core clock (MHz)	576
Stream processors clock (MHz)	1242
Memory clock (MHz)	999
CUDA compute capability	1.3

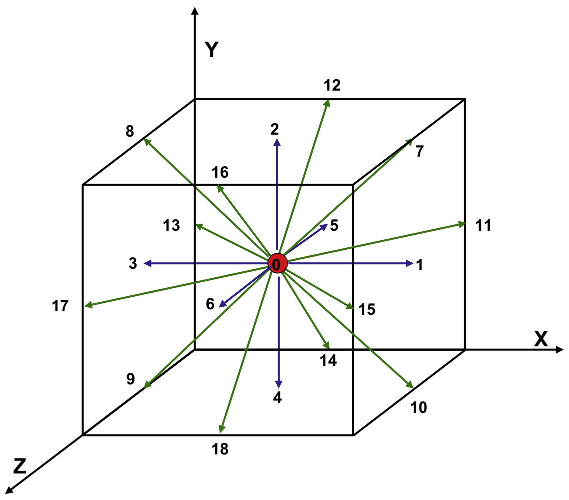

Fig. 1. Space of discrete velocities e_i in LBM- d3Q19, corresponding to the population functions f_i.

4. Implementation

The optimization of data transfer is the most relevant issue regarding performance in LBM implementations in GPU. This assertion is crucial when working in three dimensions. There are some basic strategies, like using a single one-dimensional array and macro functions for the data representation, which were successfully used in distributed CPU programming [17], that are suitable for GPU. However, memory access patterns, execution configurations and memory layout, are aspects that must be rethought due to the particularities of NVIDIA GPUs.

4.1. Algorithm

The following code shows a CUDA macro function for main data structure access: it receives the lattice position (x, y, z), the velocity index q, and the parameter t for time step switching. The grid and model dimensions are also parameterized. This macro defines memory layout where the population of consecutive x-dimension cells are stored in a consecutive global memory address.

```
(1) #define dirQ(x,y,z,q,t, dimx, dimy, dimz, Q)
 (x+(y)*dimx+(z)*dimx*dimy+(q)*dimx*dimy*dimz+(k)*Q*dimx*dimy*dimz))
```

To minimize the data transferred between iterations, the collision and propagation rules are executed in a single step. Propagation is performed first in each iteration loop, resulting in a *pull* scheme of the update process [17]. Using two data copies to hold data of successive time steps keeps the implementation simple and avoids data coupling between steps, thus allowing parallelization.

4.2. Coalesced memory access

The effective memory bandwidth substantially depends on the access pattern. Since the global memory is slow and does not have cache, accesses to it should be minimized by copying data to local or shared memory for later processing. Provided that threads from the same block access the GPU global memory simultaneously in aligned directions, groups of 16 threads (named half-warp) can share a single access. This procedure is called “coalescence” and can substantially increase memory bandwidth taking into account some restrictions. In this work, as consecutive x cells are assigned to each thread block, each thread within a block accesses consecutive global memory elements. During the advection step, thread x reads base + x address, while thread $x + 1$ reeds base + $x + 1$ memory position, and so on.

There are different restrictions to the possibility to access 16 consecutive threads as a single global access, depending on the CUDA version and hardware compute capabilities. In earlier versions, for example, each thread read or wrote single values and the initial address of the memory block should be a multiple of the data size [13]. With this restriction, only the advection of particles distributions located in the same x-plane (i = 0, 2, 4, 5, 6, 12, 14, 16 and 18 in Fig. 1) can be coalesced. In turn, devices with compute capability of 1.2 or above have fewer requirements for coalesced memory access running code compiled with CUDA toolkit 3.0 or more. In these situations, coalescing is achieved for any accesses pattern fitting into a segment size of 32 bytes for 8-bit words, 64 bytes for 16-bit words, or 128 bytes for 32- and 64-bit words [13]. All simulations presented in this work were made using CUDA 3.2 toolkit combined with Geforce GTX 260 Graphic Board (compute capability 1.3).

Figs. 2 and 3 help explaining the coalescing of 32-bit words (floats) by a half warp in the LBM advection step for shifted x-plane particle distributions (i.e. $i = 1, 3, 7, 8, 9, 10, 11, 13, 15, 17$). Global memory is shown as 64-bytes rows (16 floats). Two rows of the same tone represent a 128-bytes aligned segment. A half-warp of threads that accesses the global memory is indicated at the bottom of the figure. Fig. 2 shows an example where 16 accesses are grouped in a single one. Fig. 3 shows a case where 16 consecutive memory accesses are issued as two.

In the present LBM implementation, each half-warp memory access was issued as a single one or two. In this way forced alignments through extra sweeps are avoided [8,16]. An additional benefit obtained by performing the advection step first is that misaligned memory readings are less time consuming than misaligned memory writings.

4.3. Execution configuration

The execution is parallelized in thread blocks keeping constant the indexes y and z, with a thread for every x index. Thus, every thread executes a single cell code and complete parallelization is achieved (Fig. 4).

For example, in a cube of 64 cells per size, the domain cell located at $x = 12$, $y = 10$, $z = 32$ is processed by the thread number 12 into the thread block with indexes 10 and 32. The following code shows how cell location is calculated from block and thread indexes.

```
(1) int x, y, z;
(2) x = treadidx.x;
(3) y = blockidx.x;
(4) z = blockidx.y;
```

However, the domain cannot grow more than 512 cells in the x direction, due to the upper bound of threads per block. Actually, in cubic domains the global memory size limit is reached before (<180 cells per dimension). Therefore, for simulations in non-cubic domains it is recommended to take x as the smaller dimension.

4.4. Shared memory usage

Accessing the shared memory is ~100 times faster than accessing the global memory. The former can be used for intra-block thread communication or as a user-managed cache, like in the present implementation [12]. Data is copied from global to shared memory at each iteration during the advection step. Then, the entire algorithm is computed on the shared memory and finally the actualized distribution functions are written back to the global memory. The corresponding algorithm steps are as follow:

1. Advection: Read distribution functions $f_i(x - e\delta t, t - \delta t)$ from global memory and copy them in shared memory (mostly coalesced).
2. Synchronize threads.
3. Apply boundary conditions to obtain missing f_i (shared memory).
4. Calculate the macroscopic averages ρ, u (shared memory).
5. Calculate the collision step $f_i^{(eq)}$ (shared memory).
6. Synchronize.
7. Copy the new values $f_i(x - e\delta t, t - \delta t)$ in the global memory (fully coalesced).

Only steps 1 and 7 access the global memory, all other calculations are made with values stored in fast on-chip shared memory. The drawback of the described procedure is that the maximum number of threads per multiprocessor is limited by the size of the shared memory.

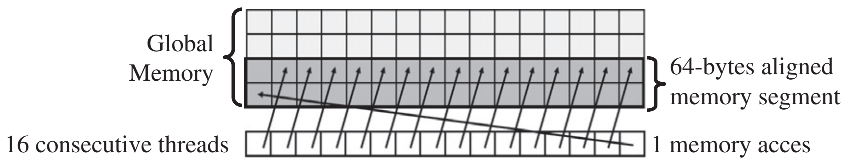

Fig. 2. 16 Threads accessing global memory issued as a single one.

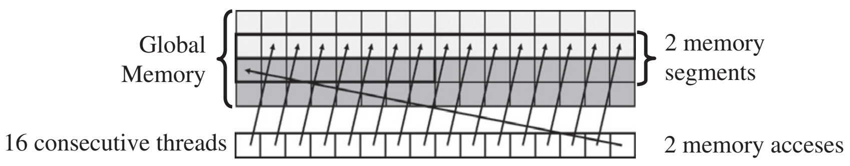

Fig. 3. 16 Threads accessing global memory issued as 2.

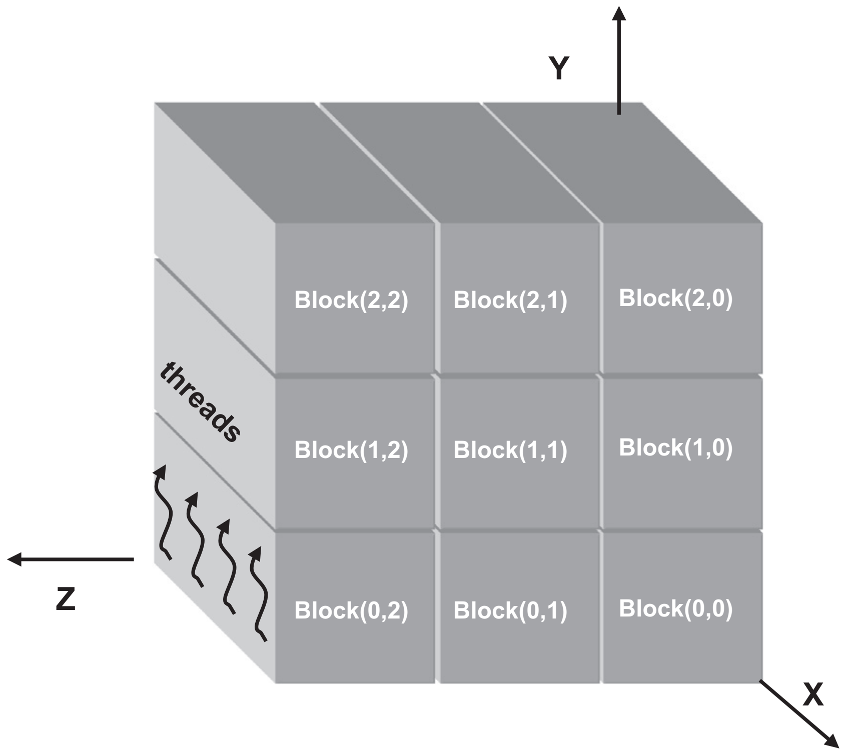

Fig. 4. Domain division in thread blocks.

4.5. Code branching

Any flow control instructions in CUDA can significantly affect the instruction throughput by causing threads of the same warp to diverge; that is, to follow different execution paths. In this case, the different execution paths are serialized increasing the total number of instructions. Hence, code branching should be reduced as much as possible. In the case of LBM, the treatment of boundary conditions interferes with this rule, for each type of boundary requires a separate code. In the implementation presented here the worst case leads to the coexistence of three different conditions in the same half-warp of 16 threads. However, since generally boundary cells are just a small fraction of the total number of cells, the computational cost is very low. Nevertheless, to minimize the number of code lines within branches, the same code is applied to every cell in the advection step, shifting indexes to give special treatment on the matrix boundaries as shown in this code:

```
west = 1;
east = 1;
...
if (x==0)//left border
west = 0;
if (x==DIMX-1) //right border
east = 0;
...
f[base] = F[dirQD(x,y,z,0,k,DIMX,DIMY,DIMZ,Q)];
f[base + 1] = F[dirQD(x – west,y,z,1,k,DIMX,DIMY,DIMZ,Q)];
...
f[base + 18] = F[dirQD(x,y + north,z – back,18,k,DIMX,DIMY,DIMZ,Q)];
```

5. Simulation of a lid-driven cavity

The lid-driven cavity is a classic idealization of recirculation flows, with numerous environmental, geophysical, and industrial applications. It is a typical benchmark problem for solvers of the Navier–Stokes equations in 3D. The problem is described as the steady-state flow in a cubic cavity, with stagnation at all walls but the upper boundary, where a constant

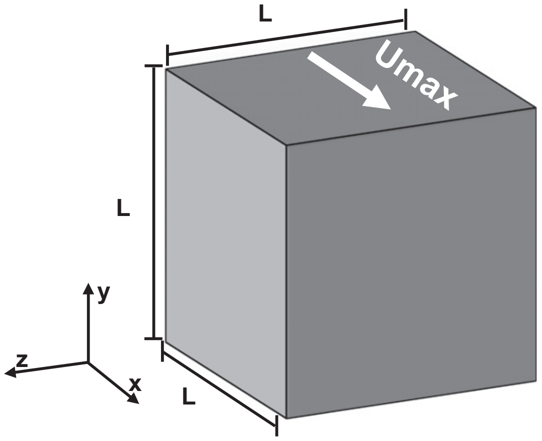

Fig. 5. Lid-driven cavity flow problem.

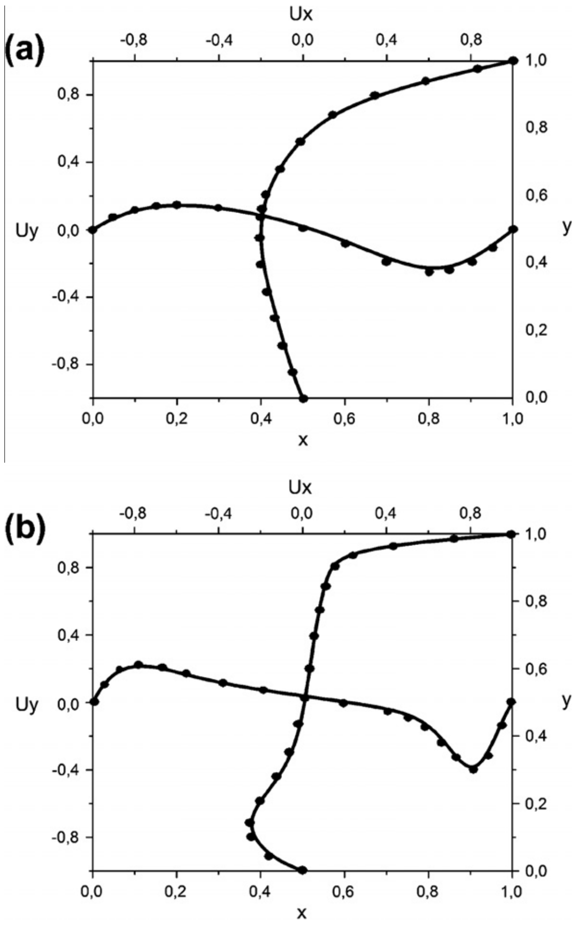

Fig. 6. Profiles of the velocity components u_x and u_y in the cavity midplane $z = 0.5$ for Re = 100 (a) and 1000 (b). The curves are LBM simulation and the dots are the calculations reported by Yang et al. [19].

velocity, U_{max}, in x direction is imposed (Fig. 5). Different flow regimes are encountered depending on the Reynolds number, defined as:

$$\mathrm{Re} = \frac{LU_{\max}}{\nu} \tag{6}$$

where ν is the viscosity and L is the cavity side.

Numerical simulations were performed for Reynolds numbers 100, and 1000, using the described LBM implementation in GPU. Only the grid refinement was changed to obtain different Reynolds numbers. With an imposed velocity $U_{max} = 0.24$ and viscosity $\nu = 0.038$, 16 and 160 cells per side are needed to simulate Re = 100 and 1000 respectively. The average relative difference of velocity fields between two time steps was taken as convergence criterion to determine steady-state conditions, that is:

$$\sum_i \frac{\|U(x_i, t+\delta) - U(x_i, t)\|}{\|U(x_i, t+\delta\|} \leqslant 10^{-6} \tag{7}$$

Fig. 6 show the velocity profiles along the cavity midplane ($z = 0.5$), u_y along the x direction and u_x along the y direction. The LBM results are compared to the solution obtained by Yang et al. [19]. Fig. 7 show the streamlines passing through the central line in direction z at $x = 0.5$ and $y = 0.5$, for different Reynolds numbers. It can be see that a main vortex rotating around an axis in the z direction is formed

6. Performance

In order to assess the performance of the GPU implementation presented in the present work, a comparison was made against the implementation of the same LBM in CPU using C programming language. The most popular metric of LBM codes performance is the number of lattice updates per second, or LUPS [9]. The LBM-CPU code was implemented as a single thread

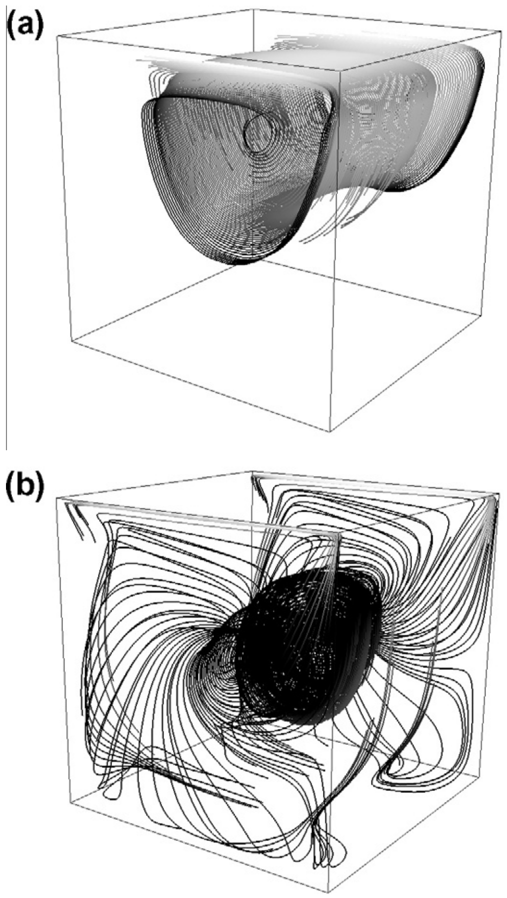

Fig. 7. Streamlines calculated for Re = 100 (a) and 1000 (b).

Table 2
Comparison of performances of LBM implementations in GPU and CPU, obtained over 1000 iterations, bold indicates peak performance and speedup respectively.

Domain size	CPU	GPU CUDA		Speedup
	MLUPS	Blocks (threads)	MLUPS	
32^3 (30768)	2.1	32×32 (32)	198	92
64^3 (262144)	2.1	64×64 (64)	245	116
96^3 (884736)	2.06	96×96 (96)	**259**	125
128^3 (2097152)	1.92	128×128 (128)	217	113
144^3 (2985984)	1.88	144×144 (144)	233	123
160^3 (4096000)	1.89	160×160 (160)	248	**131**

with one nested loop for each direction *x*, *y* and *z*. The grid access pattern and the memory layout were optimized for cache-based architectures following [17]. Table 2 compares the performances of each implementation for different platforms, scaling the problem to various domain grid sizes varying from 16^3 to 160^3 cells. The performance achieved with LBM-GPU resulted between 92 and 130 times faster than LBM-CPU, depending on the domain size and execution configuration. Both implementations use single precision floating point representations.

To estimate the memory bandwidth, the number of bytes per cell transferred from/to the device global memory is calculated. In each iteration, 38 floating points for the distribution functions are transferred (19 reads + 19 writes) plus one integer read for cell type, that is 38×4 bytes + 1×2 bytes = 154 bytes, at 259 MLUPS memory bandwidth is about 39.9 GB/s.

Further optimization strategies can be applied resigning some flexibility and accuracy. For example, simulation parameters can be replaced by constants, first order boundary conditions can be applied, and output calculation between iterations can be minimized. With these modifications a performance of 400 MLUPS was achieved, with a maximum memory bandwidth of 61.6 GB/s (>55% of the theoretical bandwidth and >65% of real memory bandwidth calculated with the CUDA test program). Standard compilation options were used in all cases. By contrast, Kuznik et al. [8] reached memory bandwidths of 69.1 GB/s, corresponding to 49% of the theoretical 147.1 GB/s of GTX 280.

7. Conclusions

A novel CUDA implementation in GPU of the Lattice-Boltzmann Method to solve tha Navier–Stokes equations in 3D was presented. The implementation makes use of a reversed advection–collision scheme, applying a simple single loop algorithm in combination with shared memory and recent CUDA sdk versions. Numerical validation was done with lid driven cavity flow simulations for different Reynolds numbers varying the grid size showing good agreement in comparison with pervious works. A fully parameterized code with second order boundary conditions run at 259 Mlups on Geforce GTX 260 graphic board. Tuned implementation actualizes 4×10^8 cells per second running on the same hardware, with near 70% peak memory bandwidth. The speed up over similar CPU implementations is about two orders of magnitude.

It was shown that using the shared memory for local caching can improve performance by reducing the number of accesses to global memory. Nvidia recommends exploiting any opportunity to replace global memory accesses by shared memory accesses [12]. However, this comes at the cost of fewer threads per multiprocessor, as well as the requirement to synchronize threads in a block after each memory load. Nevertheless, the limitations in the number of threads per multiprocessor is compensated by the reduced number of global memory access. Once a multiprocessor occupancy of more than 50% has been reached, it generally does not pay to obtain higher occupancy ratios [13].

A particularly important issue from the point of view of applications is the implementation of boundary conditions, particularly the capability of representing irregular or dynamic frontiers. There are several ways to treat irregular boundaries with Lattice Boltzmann. Each way requires a different implementation in GPU. In the present case a rather rigid treatment was chosen, consisting in classifying every boundary cell according to the boundary condition it represents, and associating the corresponding actualization rule. Boundaries forming an angle respect to the grid directions can be treated in this way using generalized interpolation rules similar to bounce-back or slip [22]. However, the most promising way to treat irregular boundaries in a GPU implementation is the immersed boundaries method [23], which represents boundaries by means of a force term in the streaming equation. This issue will be studied in future works.

Further studies can also pay attention to other optimization opportunities, like allowing each thread to process more than one element or using pitched memory arrays to enable non-hardware-dependent domain sizes.

References

[1] P.L. Bhatnagar, E.P. Gross, M. Krook, A model for collisional processes in gases. I: Small amplitude processes in charged and neutral one-component system, Phys. Rev. 94 (3) (1954) 511–525.
[2] J.P. Boon, J.P. Rivet, Lattice Gas Hydrodynamics, Cambridge University Press, Cambridge, 2001.
[3] S. Chen, H. Chen, D.O. Martinez, W.H. Matthaeus, Lattice Boltzmann model for simulation of magnetohydrodynamics, Phys. Rev. Lett. 67 (1991) 3776–3779.
[4] S. Chen, G.D. Doolen, Lattice Boltzmann methods for fluid flows, Annu. Rev. Fluid Mech. 30 (1998) 329–364.
[5] G.M. Crisci, R. Rongo, S. Di Gregorio, W. Spataro, The simulation model SCIARA: the 1991 and 2001 lava flows at Mount Etna, J. Volcanol. Geotherm. Res 132 (2–3) (2004) 253–267.
[6] N. Goodnight, CUDA/OpenGL Fluid Simulation, 2011. <http://new.math.uiuc.edu/MA198-2008/schaber2/fluidsGL.pdf>.
[7] M. Hecht, J. Harting, Implementation of on-site velocity boundary conditions for D3Q19 lattice Boltzmann simulations, J. Stat. Mech. (2010) 1018–1021.
[8] F. Kuznik, C. Obrecht, G. Rusaouën, J.J. Roux, LBM based flow simulation using GPU computing processor, Comput. Math. Appl. 59 (7) (2009) 2380–2392.
[9] P. Lammers, U. Küster, Recent performance results of the lattice Boltzmann method, in: M. Resch, T. Bönisch, S. Tiyyagura, T. Furui, Y. Seo, W. Bez (Eds.), High Performance Computing on Vector Systems 2006 Part 2, Springer, Stuttgart, 2007, pp. 51–59.
[10] R.S. Maier, R.S. Bernard, D.W. Grunau, Boundary conditions for the lattice Boltzmann method, Phys. Fluids 8 (7) (1996) 1788–1802.
[11] NVIDIA CUDA Home Page, 2011. <http://developer.nvidia.com/category/zone/cuda-zone>.
[12] NVIDIA CUDA C Programing Guide Version 4.0, 2010. <http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf>.
[13] NVIDIA CUDA C Best Practice Guide Version 4.0, 2010. <http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf>.
[14] Y.H. Qian, D. d'Humie'res, P. Lallemand, Lattice BGK models for Navier–Stokes equation, Europhys. Lett. 17 (1992) 479–484.
[15] P. Rinaldi, D. Dalponte, M. Vénere, A. Clausse, Cellular automata algorithm for simulation of surface flows in large plains, Simul. Model. Practice Theory 15 (2007) 315–327.
[16] J. Tölke, Implementation of a lattice Boltzmann kernel using the compute unified device architecture developed by nVIDIA, Comput. Visual. Sci. 13 (1) (2010) 29–39.
[17] G. Wellein, T. Zeiser, G. Hager, S. Donath, On the single processor performance of simple lattice Boltzmann kernels, Comput. Fluids 35 (8–9) (2006) 910–919.
[18] S. Wolfram, Cellular automata fluids. 1: Basic theory, J. Stat. Phys. 45 (3) (1986) 475–526.
[19] J.Y. Yang, S.C. Yang, Y.N. Chen, C.A. Hsu, Implicit weighted ENO schemes for the three-dimensional incompressible Navier–Stokes equations, J. Comput. Phys. 146 (1) (1998) 464–487.
[20] Y. Zhao, Lattice Boltzmann based PDE Solver on the GPU, Vis. Comput. 24 (5) (2007) 323–333.
[21] Q. Zou, X. He, On pressure and velocity boundary conditions for the lattice Boltzmann BGK model, Phys. Fluids 9 (6) (1997) 1591–1598.
[22] Z. Guo, C. Zheng, An extrapolation method for boundary conditions in lattice Boltzmann method, Phys. Fluids 14 (6) (2002) 2007–2011.
[23] Y. Cheng, H. Zhang, Immersed boundary method and lattice Boltzmann method coupled FSI simulation of mitral leaflet flow, Comput. Fluids 39 (5) (2010) 871–881.