ISSN 1996-3343

# Asian Journal of Applied Sciences

http://knowledgiascientific.com

Knowledgia SCIENTIFIC
A Place to Publish Outstanding Research

Asian Journal of Applied Sciences 4 (8): 762-770, 2011
ISSN 1996-3343 / DOI: 10.3923/ajaps.2011.762.770


# Lattice-boltzmann Navier-stokes Simulation on Graphic Processing Units

[1]Pablo Rafael Rinaldi, [2]Enzo Alberto Dari, [1]Marcelo Javier Vénere and [1]Alejandro Clausse

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

*Corresponding Author: Pablo Rafael Rinaldi, Instituto PLADEMA, UNCPBA, Pinto 399, Tandil (7000), Argentina Tel: +54(2293)439690 Fax: +54(2293)439690*

## ABSTRACT

Lattice Boltzmann Methods (LBM) was one of the first simulation models which successfully run on Graphic Processing Units. Earlier LBM implementations using NVIDIA Compute Unified Device Architecture programming language required two steps (collision-propagation and exchange) to maximize memory bandwidth. This article presents a parallel single-step implementation of the Lattice Boltzmann method with fully coalesced memory access using shared-memory. The resulting code, running on low cost Personal Computer Graphic Processing Unit was able to process more than 800 million cells updates per second, approaching High Performance Computing clusters' performances. Substantial reductions of the calculation rates were achieved, lowering down to 240 times the time required by a CPU to execute the same model. The code was tested on the numerical calculation of the flow in a two-dimensional channel with a sudden expansion. The precision of the results were validated against a proved finite-element solver.

**Key words:** Graphic processing units, lattice boltzmann methods, navier-stokes simulations, compute unified device architecture, high performance computing

## INTRODUCTION

One of the novelties in recent parallel computing technologies is the Graphic Processing Unit (GPU) (Anderson *et al*., 2008). Essentially, a GPU is the chip used by graphic cards to render pixels on the screen. Modern GPUs are optimized for executing a simple instruction simultaneously over each of the elements within a large set (SIMT, Single Instruction Multiple Thread). A GPU can execute a large number of threads in parallel, operating as a co-processor of the host CPU. In this way, a fraction of the operations requiring intensive computation is streamed to the GPU using functions executed as multiple parallel threads. The GPU has its own RAM memory and data can be copied to and from the CPU memory by means of optimized direct access methods (NVIDIA, 2010). Recently, several researches have shown that the application of GPU on Cellular Automata (CA) is a valid tool to simulate fluids (Goodnight, 2007; Tolke, 2007; Zhao, 2007; Kuznik *et al*., 2010).

The Lattice Boltzmann Method (LBM) consists in two independent separated steps named collision and propagation (or advection). The number of data transfers can be reduced by

executing the two steps in the same loop (Wellein *et al.*, 2006). In most high performance CPU implementations (Wellein *et al.*, 2006; Zhao, 2007; Petkov *et al.*, 2009; Guo *et al.*, 2009), the propagation is realized as last step of the iteration loop resulting in non-local write operations (scatter data). However, most implementations of LBM in GPU based on this scheme does not take full advantage of grouped memory calls provided by NVIDIA Compute Unified Device Architecture (CUDA) (NVIDIA, 2010). More efficient LBM CUDA implementations use one single loop for collision and propagation (Tolke, 2007; Kuznik *et al.*, 2010). But the way memory coalescence is achieved requires a second loop to complete the LBM calculations which in turn reduces the global performance.

In this paper, a single loop, more efficient CUDA implementation is presented. A pull scheme LBM is developed using a newer CUDA version with less memory alignment restrictions. Flow simulations were run over a GPU NVIDIA GeForce achieving high performances and near 60% of hardware theoretical bandwidth.

## MATERIALS AND METHODS

**The NVIDIA GeForce GPU:** For the present study, carried in 2010-2011, 2 different NVIDIA GeForce graphic cards were used: 8800 GT from the G80 series and GTX 260 from the GTX200 series. These GPUs were developed by NVIDIA, together with the programming model CUDA (NVIDIA, 2008). The NVIDIA GPU is composed of a set of multiprocessors with SIMD architecture, as shown in Fig. 1. 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 GPUs.

CUDA is a technology specially developed by NVIDIA to facilitate the access of intensive applications to the processing capabilities of GPU through a program interface. The GPU, called device, works like a co-processor of the main CPU, called host. The host and the device keep their own RAM memory.

**The BGK lattice boltzmann scheme:** LBM is an explicit streaming-collision iterative scheme defined in a regular grid. It is essentially a mesoscopic kinetic model with discrete internal velocities, supported on regular discrete time and space domains. The macroscopic properties approximate to second order a set of transport equations in the continuum limit (Chen and Doolen, 1998). In the present study, the BGK instance of LBM is used which solves the

Table 1: GPU hardware specifications (NVIDIA, 2008)

| NVIDIA GeForce | 8800 GT | GTX 260 |
|---|---|---|
| Number of multiprocessors | 14 | 24 |
| Number of stream processors | 112 | 192 |
| Memory size | 512 MB | 896 MB |
| Memory bus width | 256 bits | 448 bits |
| Memory bandwidth | 57,6 GB/s | 111,9 GB/s |
| Estimated peak performance | 200 Gflops | 805 Gflops |
| Core clock | 600 MHZ | 576 MHZ |
| Stream processors clock | 1500 MHZ | 1242 MHZ |
| Memory clock | 900 MHZ | 999 MHZ |
| CUDA compute capability | 1.1 | 1.3 |

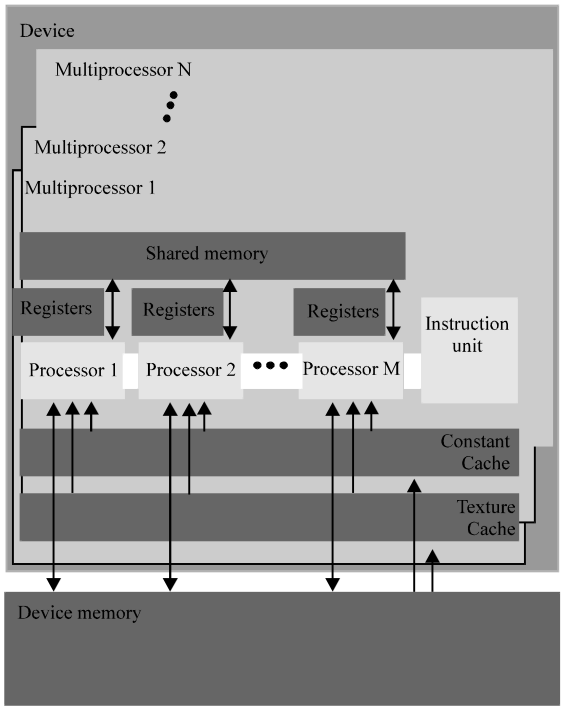


Fig. 1: SIMD multiprocessors with shared memory inside G80 NVIDIA GPU (NVIDIA, 2008)

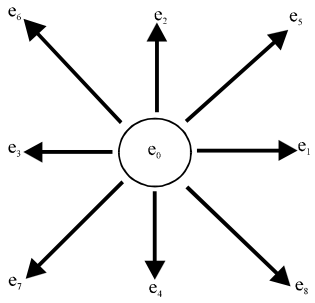


Fig. 2: Space of discrete velocities $e_i$ in LBM-D2Q9, corresponding to the population functions $f_i$

Navier-Stokes equations in two-dimensions (Bhatnagar *et al.*, 1954; Chen *et al.*, 1991; Qian *et al.*, 1992). Defined in a regular grid of square cells, the evolution rule is given by:

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

where, $f_i(x, t)$ is the population function in (x, y, t) of particles having velocity $e_i$. In two dimensions only 9 velocities are allowed (Fig. 2). The following equilibrium function $f_i^{(eq)}(x, t)$ ensures that the Navier-Stokes equations are recovered to second order (Chen and Doolen, 1998):

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

Where:

$$w_0 = \frac{4}{9}; \qquad w_i = \frac{1}{9}, i = 1:4; \qquad w_i = \frac{1}{36}, i = 5:8 \quad (3)$$

and:

$$\rho = \sum_i f_i, \rho u = \sum_i f_i e_i \quad (4)$$

A second order precision bounce-back condition was imposed in the close boundaries of the grid to simulate solid walls and special velocity-pressure conditions were imposed on the inlet and exit boundaries to enforce a parabolic profile (Zou and He, 1997).

**CUDA LBM implementation:** In order to maximize performance, it is convenient that the size of the data structures is multiple of 16 (NVIDIA, 2010). Accordingly, a rectangular 32×96 grid of square cells was represented by means of 18 floating point arrays. Nine arrays represent each $f_i$ at t and the other nine represent each $f_i$ at t+1. These arrays are created in the CPU and afterwards copied to the GPU. The geometry, including the boundary conditions, is specified by means of an integer array containing the cell types which are detailed in Table 2.

In every update, values are read from one array and written to the other including the propagation step (within the reading step). The propagation is performed first in each iteration loop, resulting in a pull scheme of the update process (Wellein *et al*., 2006).

CUDA enables grouping threads in blocks that communicate efficiently by sharing data through fast shared memory and coordinating data access by synchronization points.

Table 2: Cell types

| Type | Description |
|---|---|
| Void | Cell outside the simulation domain |
| Flow | Cell carrying fluid |
| Inlet | Cell of the inlet boundary (parabolic velocity profile) |
| Outlet | Cell of the outlet boundary |
| Close up | Cell adjacent to a wall by the top |
| Close down | Cell adjacent to a wall by the bottom |
| Close left | Cell adjacent to a wall by the left |
| Close right | Cell adjacent to a wall by the right |
| Upper inlet vertex | Closed at the top and inlet to the left |
| Lower inlet vertex | Closed at the bottom and inlet to the left |
| Upper outlet vertex | Closed at the top and outlet to the right |
| Lower outlet vertex | Closed at the bottom and outlet to the right |
| Inner vertex | Vertex located inside the channel (in our case defining the channel expansion) |

To maximize the hardware potential each block should contain at least 64 and no more than 512 threads (Tolke, 2007). Threads from different blocks cannot communicate synchronically and safely between themselves. Nevertheless, all blocks executing the same code can be grouped in grids, maximizing the number of threads triggered by a single call. To take full advantage of the hardware the number of blocks of a single grid should exceed 16 (Tolke, 2007). Each thread has its local memory, each block has its memory shared by its own threads and all threads can access the GPU memory. Accessing a data from a shared memory takes approximately 4 clock cycles, whereas accessing the global GPU memory takes from 400 to 600 cycles. The thread scheduler can execute independent arithmetic instructions during the waiting time. Therefore, it is convenient to trigger several thread blocks simultaneously, such that when a block is waiting for data retrieval, another block takes its place on the multiprocessor.

Shared memory can be used for thread communication or, like in this implementation, as a user-managed cache to improve global memory bandwidth (NVIDIA, 2010). The effective memory bandwidth also depends substantially 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 memory simultaneously in aligned directions, groups of 16 threads can share a single access. This procedure is called coalescence and can substantially increase memory bandwidth. In grids ordered by lines, coalescence is ensured if neighbour threads sweep the grid by columns. For GPU devices with computing capability 1.0 and 1.1 (G80), threads need to access words in sequence and coalescing only happens if the half-warp addresses a single segment (NVIDIA, 2010). In turn, for GPU devices of computing capability 2.0 (GTX200) threads can access any words in any order, including the same words, issuing a single memory transaction for each segment. Tolke (2007) proposed to execute the advection step on shared memory avoiding unaligned access at the cost of an additional sweep. However, this procedure is no longer needed in the later versions of CUDA. In the present study we show a single step implementation that runs fully coalesced on GTX280 devices and also has good performance in G80 devices, allowing certain unaligned accesses during the advection step. The update of a cell is the following:

- Advection: Copy the states of the neighbour cells from the global memory to the shared memory, i.e., $f_i(\vec{x}-\vec{e}\delta t, t-\delta t)$ (not fully coalesced in G80)
- Synchronize
- Apply boundary conditions to complete the set of states if needed. (shared memory)
- Calculate the macroscopic averages $\rho, \vec{u}$ . (shared memory)
- Calculate the collision step $f_i^{(eq)}$. (shared memory)
- Synchronize
- Copy the new values, i.e., $f_i(\vec{x},t)$ , in the global memory. (fully coalesced)

Since the initial grid used in this study was small (approximately 3000 cells), all the cells are executed in parallel, such that each thread updates a single cell, generating 32 blocks of 64, 96 or 128 threads.

## RESULTS

The LBM implemented in the GPU was applied to simulate the flow in a two-dimensional rectangular channel with a sudden expansion (Fig. 3). The channel length is 3 length units

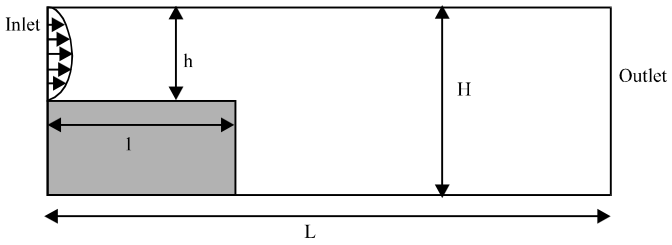


Fig. 3: Channel flow with a sudden expansion

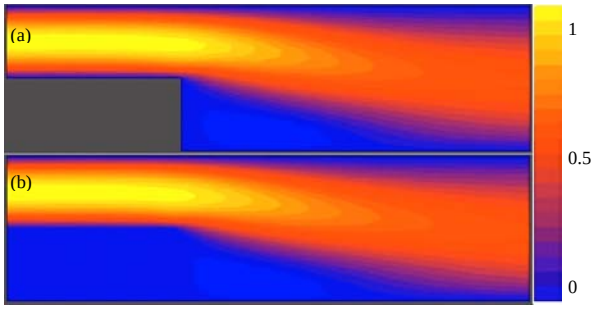


Fig. 4(a-b): Contour map of the x-component of the velocity calculated with (a) Finite Elements and (b) CUDA LBM

and the inlet and outlet are 0.5 and 1, respectively. The expansion is located at position 1 from the inlet. The inlet velocity profile is parabolic with maximum central velocity 0.3. At the exit the pressure is uniform and the velocity profile is forced parallel to the inlet velocity (i.e., $u_y = 0$). The exit density is p = 1 and the viscosity is v = 1. The corresponding Reynolds number is 100.

In order to check accuracy, the results were compared against the solution produced by a Finite Element simulator running in CPU. This fluid solver is an equal-order FE model with subgrid scale stabilization. The channel was supported in a regular mesh of 1821 nodes. Figure 4 and 5 showed the contour map of the velocity component obtained with FEM and LBM CUDA. It can be seen that the velocity profile of the x component expands smoothly and the maximum velocity decreases as the flow develops. There is a transition region after the expansion where the *y* component of the velocity peaks (blue region in Fig. 4) corresponding to the flow development after the expansion. The yellow region in Fig. 5 indicates a recirculation pattern. It can be seen that the general trend is the same in both calculations.

Figure 6 compares the normalized horizontal velocity profiles at two different axial positions downstream of the expansion. It can be seen that there is good agreement between both solutions.

The measured execution time for FE application was about 13.28 sec for 100 sec simulation (7.5 times CPU LBM and 100 times CUDA LBM). It should be mentioned that the FE solver used is a software simulation tool with much larger application areas and was not optimized to this particular problem. This execution time is only reported here to show that it is not exceeded by LBM simulations.

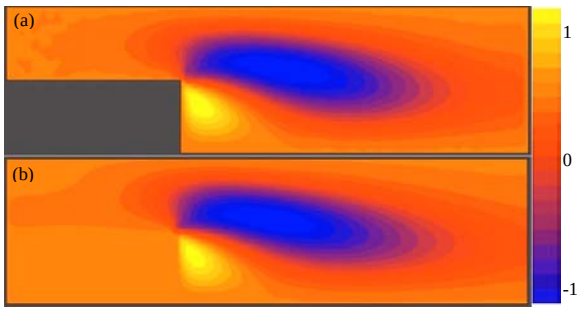


Fig. 5(a-b): Contour map of the y-component of the velocity calculated with (a) Finite Elements and (b) CUDA LBM

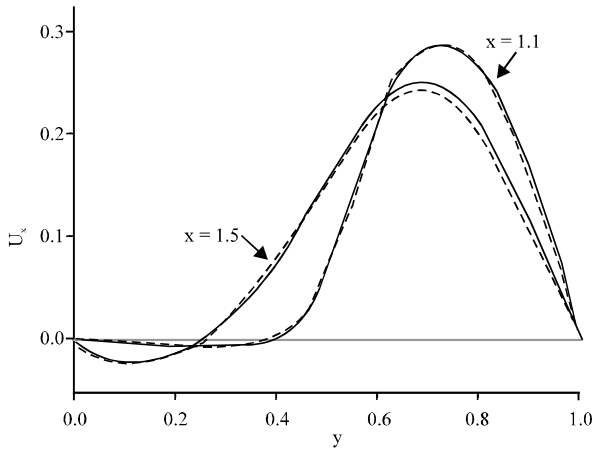


Fig. 6: Normalized profile of the horizontal component of the velocity at positions x = 1.1 (immediately after the expansion) and x = 1.5. Lattice Boltzmann over GPU (solid), Finite Elements (dashed)

In order to test the performance of the GPU implementation, a comparison was made against the implementation of the same LBM in CPU using C programming language. The same CPU Intel Core2 Quad 2.4 GHz with 8 Gb RAM was used in both cases. The performance of a LBM can be measured in terms of the mean time per iteration over the whole grid or the lattice-site updates per microsecond (MLUPS) (Lammers and Kuster, 2007). Table 3 compares the performances of each implementation for different platforms scaling the problem to various domain grid sizes. Depending on the grid size and the parallelization scheme, GPU calculations can be 240 times faster than the CPU.

The peak performance achieved was 158 MLUPS using GeForce 8800 GT and 891 MLUPS with the GeForce GTX 260. In contrast, Tolke (2007) implementation of LBM on GPU achieved less than 1 MLUPS running a Multi-Relaxation-Time LBM. On the other hand, Mazzeo and Coveney (2008) achieved less than 6 MLUPS on a 1.7-GHz CPU. Only multicore supercomputers (e.g., Hitachi SR8000-F1 8 cores) with intra-node parallelization implementations could reach 150 MLUPS (Pohl *et al*., 2004).

Table 3: Performance results averaged over 1000 iterations

| | | MLUPS | | | |
|---|---|---|---|---|---|
| Domain size | Threads per block | CPU | GPU 8800 GT | GPU GTX 260 | Max speedup |
| 96×32 | 96 | 5.2 | 75 | 14 | 14.4 |
| 192×64 | 64 | 5.8 | 144 | 57 | 24.8 |
| 384×128 | 64 | 3.6 | 158 | 188 | 52.2 |
| 768×256 | 128 | 3.6 | 152 | 467 | 129.7 |
| 1536×512 | 256 | 3.6 | 43 | 763 | 211.9 |
| 3072×1024 | 256 | 3.6 | - | 891 | 247.5 |

The maximum memory bandwidth used in the simulations was calculated as the number of bytes per cell and time step interchanged between GPU and CPU multiplied by maximum MLUPS. At 891 MLUPS this gives 65934 GB/s for the GTX 260 which is about 60% of the theoretical memory bandwidth. In comparison, Kuznik *et al.* (2010) implemented LBM with two steps approach in the more powerful Nvidia GTX 280 GPU reaching 947 MLUPS. This is less than 50% of the memory bandwidth.

## CONCLUSIONS

The performance of the GPU NVIDIA running a CUDA implementation of a LBM of the Navier-Stokes equations was studied. The results show that the use of GPU can speed up the calculation more than 240 times compared to the same model running in CPU. The GPU was able to achieve 890 MLUP which is comparable with the performance of a multicore supercomputer. The numerical results were compared against a finite element solution, showing good agreement. It should be stressed that the key to speed-up LBM simulations in GPU is the memory management strategy to optimize the access pattern. The reverse advection-collision LBM scheme implemented in this work with utilization of shared memory for local calculations proved to be a progress in this area. Although, the case studied in the present work is relatively simple, the results are encouraging, showing that GPU is a powerful economic tool for fluid simulation.

## REFERENCES

Anderson, J.A., C.D. Lorenz and A. Travesset, 2008. General purpose molecular dynamics simulations fully implemented on graphics processing units. J. Comput. Phys., 227: 5342-5359.

Bhatnagar, P., E. Gross and M.A. Krook, 1954. Model for collisional processes in gases I: Small amplitude processes in charged and neutral one-component system. Phys. Rev., 94: 511-525.

Chen, S. and G.D. Doolen, 1998. Lattice Boltzmann methods for fluid flows. Annu. Rev. Fluid Mech., 30: 329-364.

Chen, S., H. Chen, D.O. Martinez and W.H. Matthaeus, 1991. Lattice Boltzmann model for simulation of magnetohydrodynamics. Phys. Rev. Lett., 67: 3776-3779.

Goodnight, N., 2007. CUDA/OpenGL fluid simulation. NVIDIA Corporation, April 2007. http://new.math.uiuc.edu/MA198-2008/schaber2/fluidsGL.pdf

Guo, W., C. Jin, J. Li and G. He, 2009. Parallel lattice boltzmann simulation for fluid flow on multicore platform. WASE Int. Conf. Inform. Eng., 1: 107-110.

Kuznik, F., C. Obrecht, G. Rusaouen and J.J. Roux, 2010. LBM based flow simulation using GPU computing processor. Comput. Mathe. Appl., 59: 2380-2392.

Lammers, P. and U. Kuster, 2007. Recent performance results of the lattice Boltzmann method. High Perform. Comput. Vector Syst., 2006 2: 51-59.

Mazzeo, M.D. and P.V. Coveney, 2008. HemeLB: A high performance parallel lattice Bolt-mann code for large scale fluid flow in complex geometries. Comput. Phys. Commu., 178: 894-914.

NVIDIA., 2008. NVIDIA CUDA home page. http://www.nvidia.com/object/cuda_home.html

NVIDIA., 2010. NVIDIA CUDA compute unified device architecture-programing guide version 3.2. http://developer.download.nvidia.com

Petkov, K., F. Qiu, Z. Fan, A.E. Kaufman and K. Mueller, 2009. Efficient LBM visual simulation on face-centered cubic lattices. IEEE Trans. Visual. Comput. Graphics, 15: 802-814.

Pohl, T., F. Deserno, N. Thurey, U. Rude1, P. Lammers, G. Wellein and T. Zeiser, 2004. Performance evaluation of parallel large-Scale lattice Boltzmann applications on three supercomputing architectures. Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, (SC'04), IEEE Computer Society, Washington, DC. USA., pp: 1-21.

Qian, Y., D. d'Humieres and P. Lallemand, 1992. Navier-Stokes equations using a lattice-gas Boltzmann method. Europhys. Lett., 17: 479-484.

Tolke, J., 2007. Implementation of a lattice Boltzmann kernel using the compute unified architecture developer by nVIDIA. http://www.irmb.tu-bs.de/UPLOADS/toelke/Publication/toelked2q9.pdf

Wellein, G., T. Zeiser, G. Hager and S. Donath, 2006. On the single processor performance of simple lattice Boltzmann kernels. Comput. Fluids, 35: 910-919.

Zhao, Y., 2007. Lattice Boltzmann based PDE solver on the GPU. Visual Comput., 24: 323-333.

Zou, Q. and X. He, 1997. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. Phys. Fluids, 9: 1591-1598.