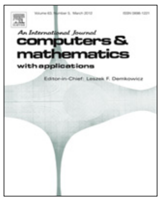
Contents lists available at SciVerse ScienceDirect

Computers and Mathematics with Applications

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



Multi-GPU implementation of the lattice Boltzmann method

Christian Obrecht[a,b,∗], Frédéric Kuznik[b], Bernard Tourancheau[c], Jean-Jacques Roux[b]

[a] *EDF Recherche et Développement, Département EnerBAT, France*
[b] *Centre de Thermique de Lyon, UMR5008, CNRS, INSA-Lyon, Université de Lyon, France*
[c] *Laboratoire de l'Informatique du Parallélisme, UMR 5668, CNRS, ENS de Lyon, INRIA, UCB Lyon 1, France*

ARTICLE INFO

Keywords:
GPU programming
CUDA
Lattice Boltzmann method
TheLMA project

ABSTRACT

The lattice Boltzmann method (LBM) is an increasingly popular approach for solving fluid flows in a wide range of applications. The LBM yields regular, data-parallel computations; hence, it is especially well fitted to massively parallel hardware such as graphics processing units (GPU). Up to now, though, single-GPU implementations of the LBM are of moderate practical interest since the on-board memory of GPU-based computing devices is too scarce for large scale simulations.

In this paper, we present a multi-GPU LBM solver based on the well-known D3Q19 MRT model. Using appropriate hardware, we managed to run our program on six Tesla C1060 computing devices in parallel. We observed up to 2.15×10^9 node updates per second for the lid-driven cubic cavity test case. It is worth mentioning that such a performance is comparable to the one obtained with large high performance clusters or massively parallel supercomputers.

Our solver enabled us to perform high resolution simulations for large Reynolds numbers without facing numerical instabilities. Though, we could observe symmetry breaking effects for long-extended simulations of unsteady flows. We describe the different levels of precision we implemented, showing that these effects are due to round off errors, and we discuss their relative impact on performance.

1. Introduction

Although the original Moore's law [1], i.e. the exponential growth of transistor count on processors is still valid nowadays, the advances in computing performance are less straightforward. During the last decade, graphics processing units (GPU) have gradually outrun CPUs in terms of raw computational power. Using nVidia's CUDA technology [2], GPUs have proven to be effective platforms to implement various high performance computing applications, ranging from linear algebra [3] to CFD [4] and PDE solvers [5].

The lattice Boltzmann method (LBM) is a novel approach in computational fluid dynamics. It appears to be an interesting alternative to the solving of the Navier–Stokes equations for various applications such as multiphase flows or porous media. As other CFD methods, the LBM is very demanding from a computational standpoint. High performance parallel implementations are therefore necessary for the LBM to be of practical interest.

Several successful implementations for the GPU are described in literature [6–8]. Nonetheless, single-GPU implementations are bound by the device memory. The maximum available amount, when using GT200 GPUs, is 4 GB, which enables us to handle at most about 2.83×10^7 nodes in single precision using the three-dimensional D3Q19 stencil.

∗ Corresponding author at: EDF Recherche et Développement, Département EnerBAT, France.
E-mail address: christian.obrecht@insa-lyon.fr (C. Obrecht).

0898-1221/$ – see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.camwa.2011.02.020

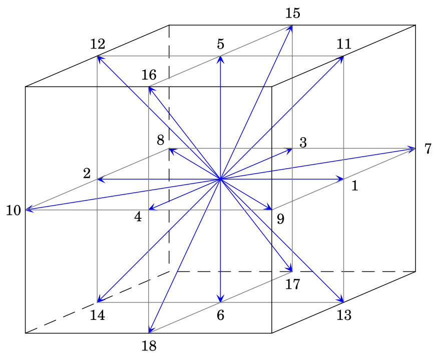


Fig. 1. D3Q19 stencil.

Multi-GPU implementations are therefore mandatory to run large scale LBM simulations, but are still a pioneering field and performance is often below what is expected from such hardware [9].

Recently released motherboards are able to manage up to eight GPU based computing devices. Although an MPI based multi-GPU LBM solver would be of great interest to run on hybrid clusters, we chose as a first step to implement a simpler POSIX thread based solver. The remainder of the paper is organised as follows. We first briefly review the LBM and the CUDA technology. Then we give some general guidelines for implementing the LBM on GPUs and describe our multi-GPU implementation. Last, we discuss numerical issues we could observe running large scale simulations at high Reynolds numbers.

2. Multiple-relaxation-time LBM

Although originating from the lattice–gas automata theory [10], the lattice Boltzmann method is now generally interpreted as a way to solve the linearised Boltzmann equation [11]. In our work, we used the multiple-relaxation-time (MRT) approach [12] instead of the more popular Bhatnagar–Gross–Krook (BGK) version of the LBM [13]. In this section we shall briefly describe the MRT LBM.

With the Boltzmann equation, a fluid is described using a single-particle distribution function f depending on space and particular velocity, i.e. phase space, and on time. In the LBM, space is usually represented by a regular orthogonal mesh of resolution δx and time is split in constant steps δt. The discrete counterpart of the continuous velocity space is a finite set of velocities $\boldsymbol{\xi}_i$, carefully chosen in order to ensure sufficient isotropy. Usually, vectors $\delta t\boldsymbol{\xi}_i$ link nodes to only some of their nearest neighbours. As an example, Fig. 1 shows the D3Q19 stencil we used in our computations.

Let us denote: $|a_i\rangle = (a_0, \dots, a_N)^\mathsf{T}$, T being the transpose operator. The lattice Boltzmann equation (LBE) writes:

$$|f_i(\boldsymbol{x} + \delta t\boldsymbol{\xi}_i, t + \delta t)\rangle - |f_i(\boldsymbol{x}, t)\rangle = \Omega\Big[|f_i(\boldsymbol{x}, t)\rangle\Big] \tag{1}$$

where $\{f_i \mid i = 0, \dots, N\}$ is the discrete equivalent of f, and Ω is the collision operator.

In the MRT approach, collision is performed in moment space. The particle distribution is mapped to a set of moments $\{m_i \mid i = 0, \dots, N\}$ by an orthogonal matrix M:

$$|f_i(\boldsymbol{x}, t)\rangle = \mathsf{M}^{-1}|m_i(\boldsymbol{x}, t)\rangle \tag{2}$$

where $|m(\boldsymbol{x}, t)\rangle$ is the moment vector. Matrix M for the D3Q19 stencil can be found in Appendix A of [14]. The corresponding moment vector is:

$$|m_i(\boldsymbol{x}, t)\rangle = \big(\rho, e, \varepsilon, j_x, q_x, j_y, q_y, j_z, q_z, 3p_{xx}, 3\pi_{xx}, p_{ww}, \pi_{ww}, p_{xy}, p_{yz}, p_{zx}, m_x, m_y, m_z\big)^\mathsf{T} \tag{3}$$

where ρ is the mass density, e is energy, ε is energy square, $\boldsymbol{j} = (j_x, j_y, j_z)$ is the momentum, $\boldsymbol{q} = (q_x, q_y, q_z)$ is the heat flux, $p_{xx}, p_{xy}, p_{yz}, p_{zx}, p_{ww}$ are related to the components of the stress tensor, π_{xx}, π_{ww} are fourth-order moments and m_x, m_y, m_z are third-order moments with respect to the particle velocities. The mass density and the momentum are the conserved moments.

The LBE may thus be written as:

$$|f_i(\boldsymbol{x} + \delta t\boldsymbol{\xi}_i, t + \delta t)\rangle - |f_i(\boldsymbol{x}, t)\rangle = -\mathsf{M}^{-1}\mathsf{S}\Big[|m_i(\boldsymbol{x}, t)\rangle - |m_i^{(\text{eq})}(\boldsymbol{x}, t)\rangle\Big] \tag{4}$$

where S is a diagonal collision matrix and the $m_i^{(\text{eq})}$ are the equilibrium values of the moments. For the sake of isotropy, S obeys:

$$\mathsf{S} = \operatorname{diag}(0, s_1, s_2, 0, s_4, 0, s_4, 0, s_4, s_9, s_{10}, s_9, s_{10}, s_{13}, s_{13}, s_{13}, s_{16}, s_{16}, s_{16}). \tag{5}$$

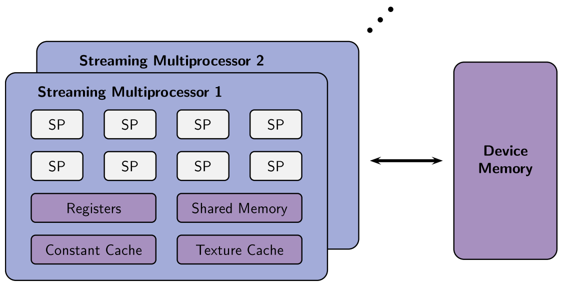


Fig. 2. CUDA hardware.

We additionally set $s_9 = s_{13}$, the initial density $\rho_0 = 1$, and the speed of sound $c_s = 1/\sqrt{3}$, the unit of speed being $\delta x/\delta t$. The equilibrium values of the non-conserved moments are thus given by:

$$e^{(\text{eq})} = -11\rho + 19\boldsymbol{j}^2 \tag{6}$$

$$\varepsilon^{(\text{eq})} = -\frac{475}{63}\boldsymbol{j}^2 \tag{7}$$

$$\boldsymbol{q}^{(\text{eq})} = -\frac{2}{3}\boldsymbol{j} \tag{8}$$

$$3p_{xx}^{(\text{eq})} = 3j_x^2 - \boldsymbol{j}^2 \tag{9}$$

$$p_{ww}^{(\text{eq})} = j_y^2 - j_z^2 \tag{10}$$

$$p_{xy}^{(\text{eq})} = j_x j_y, \qquad p_{yz}^{(\text{eq})} = j_y j_z, \qquad p_{zx}^{(\text{eq})} = j_z j_x \tag{11}$$

$$3\pi_{xx}^{(\text{eq})} = \pi_{ww}^{(\text{eq})} = 0 \tag{12}$$

$$m_x^{(\text{eq})} = m_y^{(\text{eq})} = m_z^{(\text{eq})} = 0. \tag{13}$$

The kinematic viscosity ν of the model is related to relaxation rate s_9 by:

$$\nu = \frac{1}{3}\left(\frac{1}{s_9} - \frac{1}{2}\right). \tag{14}$$

The other rates are set according to [15]. Namely: $s_1 = 1.19$, $s_2 = s_{10} = 1.4$, $s_4 = 1.2$, and $s_{16} = 1.98$.

3. Overview of the CUDA technology

The *Compute Unified Device Architecture* (CUDA) is nowadays the leading technology for general purpose computations on GPUs. Initiated in late 2007 by the nVidia company, CUDA defines both a programming model and general hardware specifications. CUDA capable GPUs consist of a set of streaming multiprocessors (SM), each containing several scalar processors (SP) as outlined in Fig. 2. The SPs within a SM follow a single-instruction multiple-data (SIMD) execution scheme. Yet, SMs are not globally synchronised, thus the overall execution scheme may be described as single-instruction multiple-thread (SIMT).

CUDA computing devices show a complex memory hierarchy. The main storage consists of a rather large off-chip device memory. This memory is not cached except for specific read-only data (i.e. constants and textures); hence it suffers from high latency which has to be properly hidden. Each SM provides its SPs with non-addressable registers and some addressable shared memory which allows inter-SP communication.

The CUDA programming language is an extension to C/C++ (with some restrictions). A CUDA program basically consists of CPU code and (at least) one kernel, i.e. a void returning function to be executed by the GPU. Kernels are executed in several threads with private local variables. Threads are grouped in identical blocks which may have up to three dimensions. During execution, a block cannot be partitioned and therefore must fit into a single SM. Nonetheless, an SM may execute several blocks concurrently. Threads within a block may be synchronised and have access to a shared memory space. Yet, no protection mechanism, e.g. mutexes, is available: it is up to the programmer to manage this aspect.

Blocks are grouped into a one or two-dimensional execution grid, specified at launch time. Blocks are executed asynchronously and there is no efficient dedicated mechanism to ensure global synchronisation. All threads within a grid have access to a global memory space which is hosted in the device memory and is persistent along the application life cycle (see Fig. 3). Global synchronisation is therefore achieved by performing multiple kernel launches.

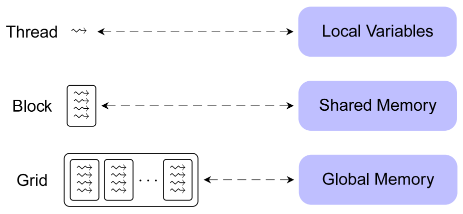


Fig. 3. CUDA programming model.

4. GPU implementation guidelines

From an algorithmic point of view, the LBM breaks into two elementary steps: *collision* in which the collision operator is applied to the particle distribution, and *propagation* in which updated particle populations are propagated to the neighbouring nodes. Eq. (4) may therefore be split in:

$$|\tilde{f}_i(\boldsymbol{x}, t)\rangle = \mathsf{M}^{-1}\left(|m_i(\boldsymbol{x}, t)\rangle - \mathsf{S}\left[|m_i(\boldsymbol{x}, t)\rangle - |m_i^{(\mathrm{eq})}(\boldsymbol{x}, t)\rangle\right]\right) \tag{15}$$

$$|f_i(\boldsymbol{x} + \delta t\boldsymbol{\xi}_i, t + \delta t)\rangle = |\tilde{f}_i(\boldsymbol{x}, t)\rangle \tag{16}$$

where Eq. (15) describes the collision step and Eq. (16) the propagation step. Thus, the LBM described in Section 2, may be summarised by the following pseudo-code:

```
1.  for each time step $t$ do
2.    for each lattice node $\boldsymbol{x}$ do
3.      read velocity distribution $f_i(\boldsymbol{x}, t)$
4.      if node $\boldsymbol{x}$ is on boundaries then
5.        apply boundary conditions
6.      end if
7.      compute moments $m_i(\boldsymbol{x}, t)$
8.      compute equilibrium values $m_i^{(\mathrm{eq})}(\boldsymbol{x}, t)$
9.      compute updated distribution $\tilde{f}_i(\boldsymbol{x}, t)$
10.     propagate to neighbouring nodes $\boldsymbol{x} + \delta t\vec{\xi}_i$
11.   end for
12. end for
```

The most convenient approach to take advantage of the massive parallelism of GPUs is to assign one thread to each node. Threads within a block are executed in groups of 32 threads named *warps*.[1] Global memory transactions are issued by half-warp. Best performance is achieved when these operations may be coalesced into single transactions of 32 B, 64 B, or 128 B. Yet, segment transactions face the important restriction that the segment's offset has to be a multiple of its size.

Optimised CPU implementations of the LBM generally store the particle distribution in an array of structures, which improves data locality. In order to allow coalescing, GPU implementations must adopt a reverse approach. A simple and efficient solution is to use one dimensional blocks corresponding to a given spatial direction and to store the particle distribution in a multi-dimensional array. The minor dimension of the array is chosen such that contiguous threads access contiguous memory locations.

Nonetheless, this approach is not sufficient to ensure optimal memory transactions. For most of the particle populations, the propagation step leads to one unit shifts in addresses as illustrated by Fig. 4.

With the first generation of CUDA enabled GPUs, i.e. for compute capability up to 1.1, alignment is mandatory for coalescence to occur, hence misalignment has a dramatic impact on performance. To address this problem, propagation within the blocks may be performed using shared memory as described in [6]. As of compute capability 1.2, misaligned memory accesses are issued in as few segment transactions as possible. As thoroughly shown in [16], misaligned reads are far less expensive than misaligned writes, hence a rather efficient way to perform propagation is to use the out-of-place propagation scheme [17], outlined in Fig. 5.

[1] The size of a warp is implementation dependent and may vary in the future.

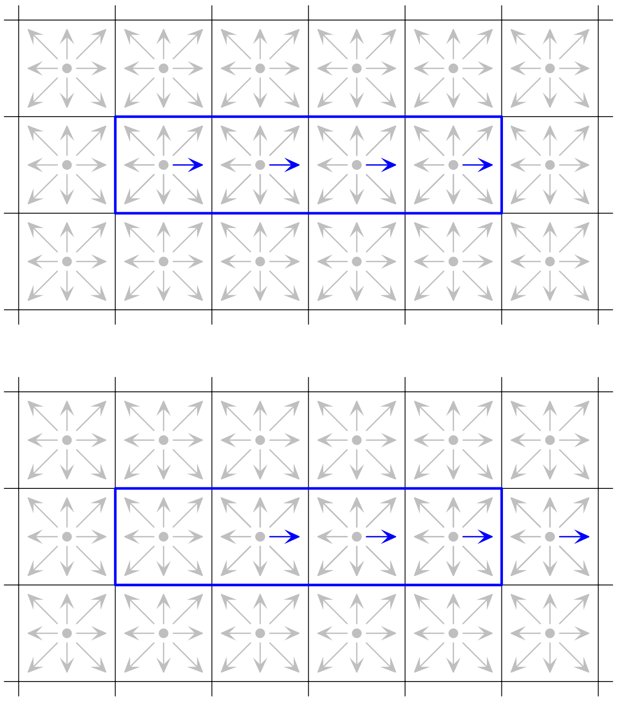


Fig. 4. Misalignment issue.

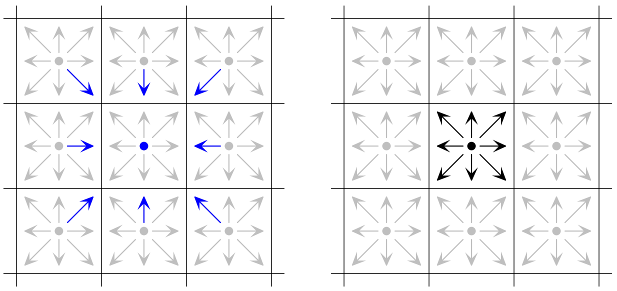


Fig. 5. Out-of-place propagation.

5. Multi-GPU implementation of the LBM

Developing libraries is a common and acknowledged practise in software engineering. The Palabos project [18] for instance, in the case of LBM, provides a wide set of generic functions which allows to efficiently implement a parallel CPU LBM solver with given geometry, boundary conditions, and the lattice Boltzmann model.

Nevertheless, the CUDA technology has some inherent limitations which make it difficult to follow the same path when developing GPU LBM solvers. The compilation tool chain, for instance, being unable to link GPU binaries forbids actual modular programming. Likewise, devices of compute capability up to 1.3 have limited support for functions. The so-called *device* functions, i.e. functions to be executed by the GPU, are mostly inlined at compile time, which restricts their use in practise.

In order to improve code reusability, we designed the TheLMA framework [19]. TheLMA stands for *Thermal LBM on Many-core Architectures*, thermal flow simulation being our main topic of interest. It provides a global template for multi-GPU LBM solvers on which we developed the present implementation. Fig. 6 outlines the structure of the framework.

The `main.c` file contains the main loop of the simulation and may access to a set of commodity functions in order to retrieve parameters, initialise variables, perform statistical calculations, and output simulation results in various formats. The `thelma.cu` file is a hub containing some general macros and including the CUDA components responsible for setting up the geometry, initialising, running the simulation and extracting results. Each of these component contains a launch function which is accessible to the C part of the program and handles the actual kernel invocation.

At initialisation, the program creates one POSIX thread for each requested computing device in order to hold the corresponding CUDA context. A sub-domain of the global lattice is assigned to each device. As for single-GPU

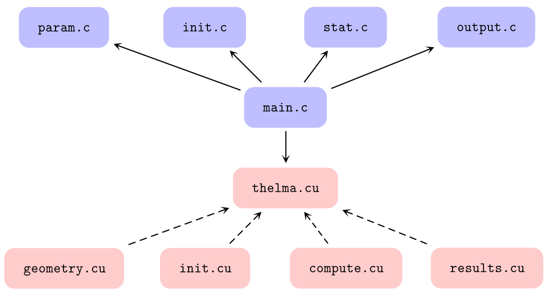


Fig. 6. The TheLMA framework.

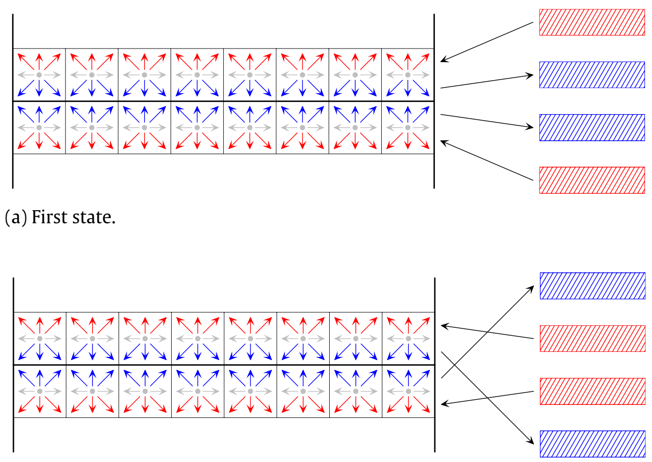


Fig. 7. Inter-GPU communication scheme.

implementation, synchronisation within the sub-domains is achieved by launching a kernel for each time step. Global synchronisation uses standard POSIX barriers. Inter-GPU communication is performed using page-locked CPU memory and zero-copy memory transactions.

As for global memory accesses, zero-copy transactions require coalescing to achieve optimal performance. This implies that the interfaces between sub-domains should be parallel to the direction associated with the minor dimension of the particle distribution array. For the sake of simplicity, we chose to split the lattice in rectangular cuboids along the direction corresponding to the major dimension. Fig. 7 outlines the inter-GPU communication scheme; incoming populations are drawn in red, outgoing populations are drawn in blue.

Each interface between sub-domains is associated to four buffers: two for incoming populations and two for outgoing. Pointers are switched after each time step. Maximal parallelisation efficiency requires perfect overlapping of computations and communication. The zero-copy feature enables such overlapping, but the overlapping ratio depends on the scheduling of memory transactions at warp level. The execution grid set-up is therefore an important optimisation target.

Another problem arise when considering the configuration of the execution grid, since it may only have up to two dimensions. The simple solution of using one-dimensional blocks and a two-dimensional grid to span the three spatial dimensions does not apply to large lattices. On GT200 hardware, for instance, the resource requirements of an LBM kernel are likely to forbid the use of blocks greater than 256.

We therefore chose to use a two-dimensional grid of size $(\ell_x \times \ell_y \times \ell_z/2^m) \times (2^{m-n})$ with one-dimensional blocks of size 2^n; ℓ_x, ℓ_y, ℓ_z being the dimensions of the lattice, m and n being free parameters. Retrieval of coordinates is done using the following code:

```
w = blockIdx.x<<m |  blockIdx.y<<n
      | threadIdx.x;
x = w % lX;
y = (w/lX) % lY;
z = w/(lX*lY);
```

The optimal values for m and n are $m = 15$ and $n = 7$, which were determined empirically. To validate our code, we implemented the well-known lid-driven cubic cavity test case in which five walls have null velocity boundary conditions

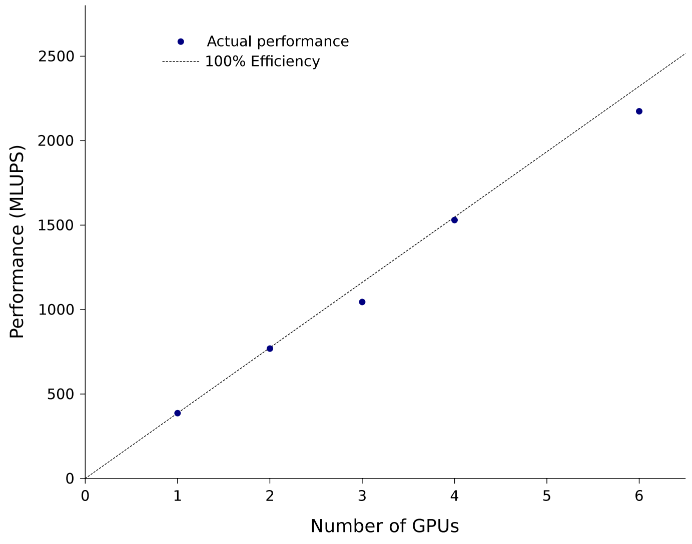


Fig. 8. Performance on a 192^3 lattice.

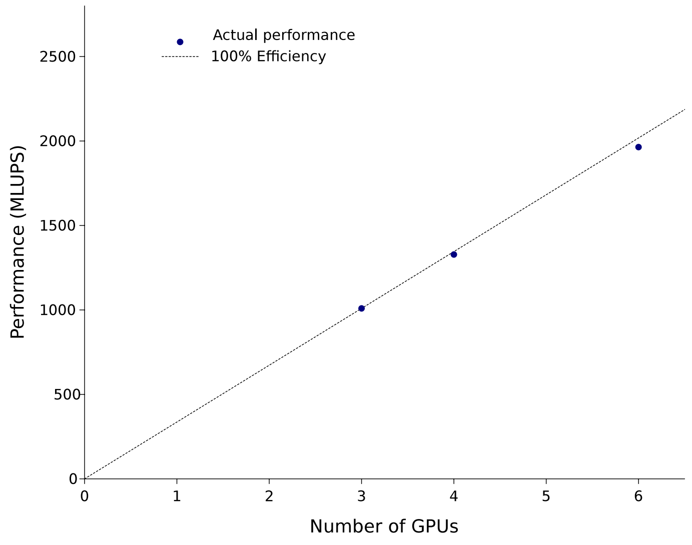


Fig. 9. Performance on a 384^3 lattice.

and the top lid has imposed constant velocity. In order to study the scalability of the program, we chose to run performance tests on a 192^3 lattice which may be handled by one single GPU or split in two, three, four, or six identical sub-domains as well. In addition, we also ran performance tests on a 384^3 lattice using three, four, and six computing devices. Figs. 8 and 9 show the obtained performance in million lattice node updates per second (MLUPS) for single precision with Tesla C1060 computing devices on a Tyan B7015 server. It is worth mentioning that maximum performance is of the same order of magnitude than the one obtained with optimised double precision code on supercomputers (see [20], for instance).

Scalability is excellent with no less than 90% parallelisation efficiency. Table 1 displays the required throughput for incoming and outgoing data at 100% efficiency on the 192^3 lattice. With the Tyan S7015 motherboard of our server, the `bandwidthTest` program that comes with the CUDA development kit gives 2.78 GB/s host to device and 1.80 GB/s device to host maximum sustained throughput. The data exchange being symmetric and the PCI-E interface being full-duplex, we may use the lower of these values as a rough estimate of the available throughput for one PCI-E 16 $\times$ slot. A comprehensive study of communications between computing devices and main memory is beyond the scope of this work and shall be given in future reference.

Table 1 shows that even with six GPUs, i.e. five sub-domain interfaces, the required throughput is comparable to the one achievable with a single PCI-E 16 $\times$ slot, therefore data exchange is not likely to overflow the capacity of the PCI-E links. Furthermore, we see that the execution grid configuration we propose enables very satisfactory communication/computation overlapping.

Table 1
Required throughput for data exchange at 100% efficiency.

Number of GPUs	1	2	3	4	6
Kernel duration (ms)	18.29	9.14	6.10	4.57	3.05
Data amount (MB)	0.00	1.47	2.95	4.42	7.37
Required throughput (GB/s)	0.00	0.16	0.48	0.97	2.42

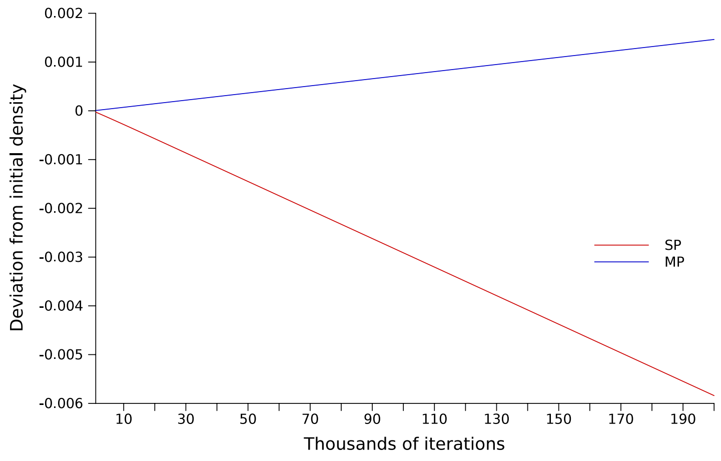


Fig. 10. Deviation for SP and MP.

According to the `bandwidthTest` program, the GPU to device memory maximum sustained throughput is 73.3 GB/s for the Tesla C1060. Performance in single precision using one GPU is 387 MLUPS on the 192^3 lattice which correspond to a data throughput of 80.4% of the maximum. We may therefore conclude that our single precision solver is memory bound and that performance is nearly optimal.

Performance for the double precision version of our solver on the 192^3 lattice ranges from 117 MLUPS using one GPU to 683 MLUPS using six, with similar scalability than for the single precision version. Considering one GPU, the corresponding data throughput is only 48.5% of the maximum, which implies that the double precision version is not memory bound but computation bound.

6. Numerical issues

Although the lid-driven cubic cavity test case is well documented at low Reynolds numbers, there are–to the best of our knowledge–very few references for Re $\geq$ 12 000 [21,22]. Using the six available Tesla C1060 cards, our solver is able to handle cubic lattices containing as much as 480^3 nodes for single precision D3Q19 and 384^3 nodes for double precision D3Q19. We could therefore run simulations for Reynolds numbers up to 30 000 without facing numerical instabilities.

According to nVidia, peak performance for the Tesla C1060 is 933 GFlops in single precision and 78 GFlops in double precision. As a matter of fact, GT200 GPUs are usually less efficient with double precision computations than with single precision. In our case, the performance ratio is about 3.2 to one. Nevertheless, the GT200 implementation of single precision is not fully IEEE 754 compliant. When first running our solver at Re = 30 000 in single precision, we could see some numerical issue arise: the flow loses symmetry at a very early stage of simulation. Further investigation showed us that the average deviation from initial density decreases at a constant pace instead of fluctuating around zero.

To evaluate the impact of machine accuracy on our simulations, we experimented with three levels of precision: single precision (SP), mixed precision (MP), i.e. double precision computations with single precision storage, and double precision (DP). It has been reported that using $\delta\rho = \rho - \rho_0$ instead of ρ in moment space improves accuracy [14]. Thus we also experimented this approach for the three levels of precision: SP*, MP*, DP*.

Figs. 10 and 11 show the average deviation from initial density when running a simulation at Re = 30 000 on a 384^3 lattice for the six levels of precision. We can see that, regarding conservation of mass, mixed precision does not provide significant improvement over single precision, and that SP*, MP*, and DP* perform better than DP by an order of magnitude. Furthermore, we may conclude that SP and MP should not be used when simulating unsteady flows.

In order to study the loss of symmetry from a quantitative standpoint, we used the following estimator:

$$\mathcal{L} = \max_{\boldsymbol{x}} \|\boldsymbol{u}(\boldsymbol{x}, t) - \bar{\boldsymbol{u}}(\bar{\boldsymbol{x}}, t)\| \tag{17}$$

where $\boldsymbol{x}$ and $\bar{\boldsymbol{x}}$, and $\boldsymbol{u}$ and $\bar{\boldsymbol{u}}$ are symmetric with respect of the symmetry plane of the cavity. Fig. 12 shows the evolution of $\mathcal{L}$ for the different precision levels running the same simulation than for mass conservation, i.e. Re = 30 000 on a 384^3

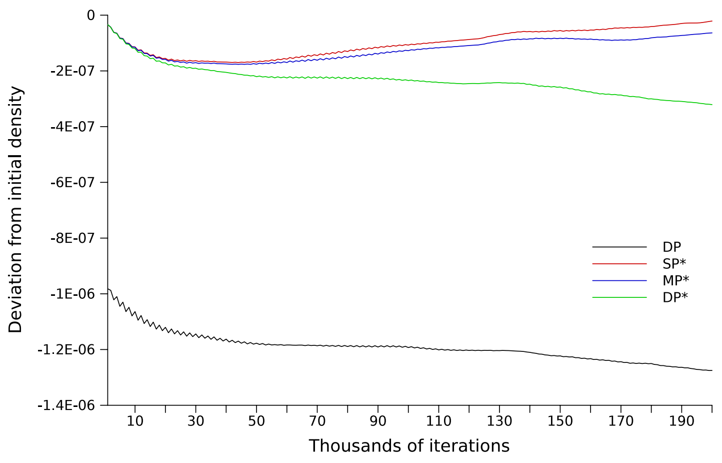


Fig. 11. Deviation for DP, SP*, MP*, DP*.

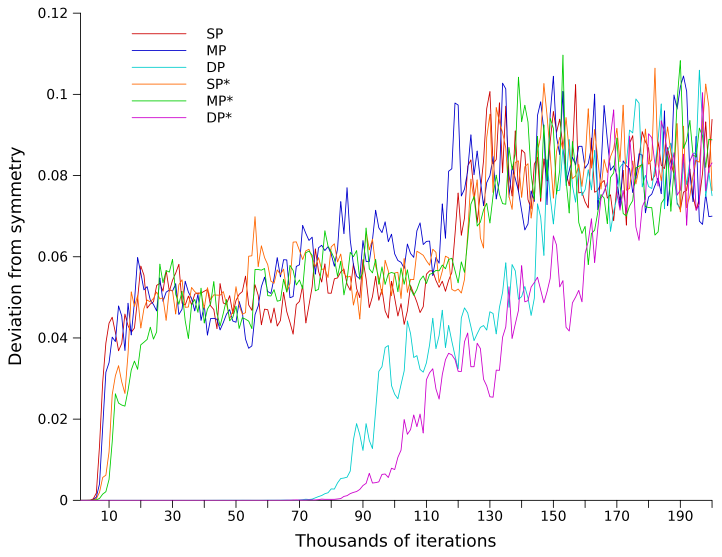


Fig. 12. Evolution of $\mathcal{L}$ for the six precision levels.

lattice. One can deduce from this diagram that the accumulation of round-off errors is the cause for the loss of symmetry. Past a certain threshold, due to the turbulent nature of the flow, the numerical perturbations are steeply amplified.

Fig. 13 displays the evolution of $\mathcal{L}$ at different Reynolds numbers for the DP* precision level on a 384^3 lattice. This diagram shows that the more turbulent the flow pattern is, the sooner the symmetry breaking occurs, which corroborates the former point of view.

From a performance standpoint, SP, MP, and DP behave similarly than their stared counterparts, since the difference in implementation only affects the initialisation section. Using $\delta\rho$ instead of ρ is therefore an advisable improvement. Mixed precision has almost identical performance than double precision, e.g. 602 MLUPS using six GPUs on a 192^3 lattice. In this case, the gain in accuracy is not worth the performance trade-off.

7. Conclusion

In this contribution, we describe a multi-GPU implementation of the LBM, based on rather simple technical choices, i.e. POSIX threads and basic domain tilling. Nevertheless, performance is nearly optimal, rivalling the one of supercomputer or large cluster implementations. Further investigations are needed to improve understanding of the inter-GPU communication potential. Moreover, work has to be done to design some execution grid layout and domain decomposition compatible with MPI parallelisation.

Our multi-GPU LBM solvers enables the use of large lattices, thus allowing direct numerical simulation of unsteady flows. We describe some numerical issues that arise at high Reynolds numbers and investigate the impact of different precision levels both on accuracy and performance.

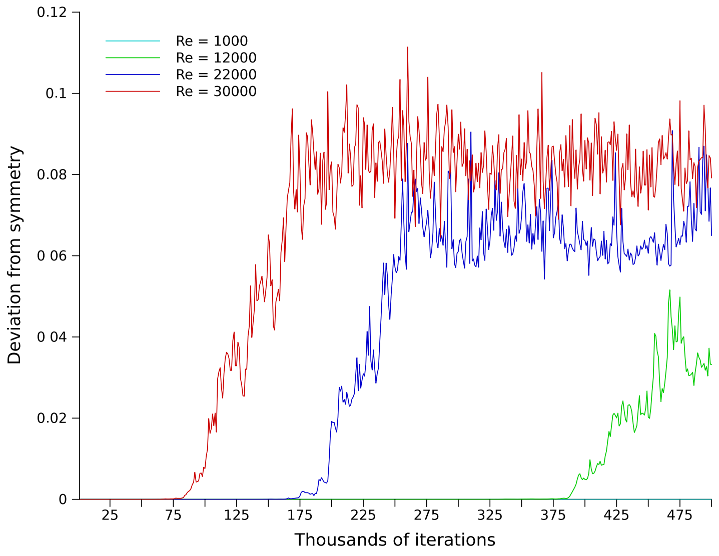


Fig. 13. Evolution of $\mathcal{L}$ at different Reynolds numbers.

The TheLMA framework we designed to implement our flow solver is meant to improve code reusability. We are currently developing several applications based on TheLMA, including a hybrid thermal solver and a LES solver. In the near future, we plan to extend this framework to generic multi-GPU parallelisation.

References

[1] G.E. Moore, Cramming more components onto integrated circuits, Electronics Magazines 38 (8) (1965).
[2] NVidia. Compute Unified Device Architecture Programming Guide version 3.0, February 2010.
[3] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, S. Tomov, Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects, Journal of Physics: Conference Series 180 (2009) 012037. IOP Publishing.
[4] J. Dongarra, S. Moore, G. Peterson, S. Tomov, J. Allred, V. Natoli, D. Richie, Exploring new architectures in accelerating CFD for air force applications, in: Proceedings of HPCMP Users Group Conference, Citeseer, 2008, pp. 14–17.
[5] P. Micikevicius, 3D finite difference computation on GPUs using CUDA, in: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, ACM, 2009, pp. 79–84.
[6] J. Tölke, Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA, Computing and Visualization in Science (2008) 1–11.
[7] J. Tölke, M. Krafczyk, TeraFLOP computing on a desktop PC with GPUs for 3D CFD, International Journal of Computational Fluid Dynamics 22 (7) (2008) 443–456.
[8] F. Kuznik, C. Obrecht, G. Rusaouën, J.-J. Roux, LBM Based Flow Simulation Using GPU Computing Processor, Computers and Mathematics with Applications 27 (2009).
[9] E. Riegel, T. Indinger, N.A. Adams, Implementation of a Lattice–Boltzmann method for numerical fluid mechanics using the nVIDIA CUDA technology, Computer Science—Research and Development 23 (3) (2009) 241–247.
[10] U. Frisch, B. Hasslacher, Y. Pomeau, Lattice-gas automata for the Navier–Stokes equation, Physical Review Letters 56 (14) (1986) 1505–1508.
[11] G.R. McNamara, G. Zanetti, Use of the Boltzmann equation to simulate Lattice-Gas automata, Physical Review Letters 61 (1988) 2332–2335.
[12] D. d'Humières, Generalized lattice-Boltzmann equations, Rarefied Gas Dynamics: Theory and Simulations (1994) 450–458.
[13] Y.H. Qian, D. d'Humières, P. Lallemand, Lattice BGK models for Navier–Stokes equation, Europhysics Letters 17 (6) (1992) 479–484.
[14] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, L.S. Luo, Multiple-relaxation-time lattice Boltzmann models in three dimensions, Philosophical Transactions: Mathematical, Physical and Engineering Sciences (2002) 437–451.
[15] P. Lallemand, L.S. Luo, Theory of the lattice Boltzmann method: dispersion, dissipation, isotropy, Galilean invariance, and stability, Physical Review E 61 (6) (2000) 6546–6562.
[16] C. Obrecht, F. Kuznik, B. Tourancheau, J.-J. Roux, Global memory access modelling for efficient implementation of the LBM on GPUs, in: High Performance Computing for Computational Science- –VECPAR2010, in: Lecture Notes in Computer Science, Springer, 2010.
[17] C. Obrecht, F. Kuznik, B. Tourancheau, J.-J. Roux, A new approach to the lattice Boltzmann method for graphics processing units, Computers and Mathematics with Applications (2010) doi:10.1016/j.camwa.2010.01.054.
[18] J. Latt, O. Malaspinas, D. Lagrava, www.lbmethod.org/palabos.
[19] Thermal LBM on Many-core Architectures. www.thelma-project.info.
[20] J. Latt, Palabos Benchmarks (3D Lid-driven Cavity) www.lbmethod.org/plb_wiki:benchmark:cavity_n1000.
[21] E. Leriche, S. Gavrilakis, Direct numerical simulation of the flow in a Lid-driven cubical cavity, Physics of Fluids 12 (2000) 1363.
[22] R. Bouffanais, M.O. Deville, E. Leriche, Large-eddy simulation of the flow in a Lid-driven cubical cavity, Physics of Fluids 19 (2007) 055108.