The TheLMA project: a thermal lattice Boltzmann solver for the GPU

C. Obrecht, F. Kuznik, Bernard Tourancheau, Jean-Jacques Roux

The TheLMA project: a thermal lattice Boltzmann solver for the GPU

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

[a] *EDF R&D, Département EnerBAT, F-77818 Moret-sur-Loing Cedex, France*
[b] *Université de Lyon, F-69361 Lyon Cedex 07, France*
[c] *INSA-Lyon, CETHIL, UMR5008, F-69621 Villeurbanne Cedex, France*
[d] *INSA-Lyon, CITI, INRIA, F-69621 Villeurbanne Cedex, France*
[e] *Université Lyon 1, F-69622 Villeurbanne Cedex, France*

Abstract

In this paper, we consider the implementation of a thermal flow solver based on the lattice Boltzmann method (LBM) for graphics processing units (GPU). We first describe the hybrid thermal LBM model implemented, and give a concise review of the CUDA technology. The specific issues that arise with LBM on GPUs are outlined. We propose an approach for efficient handling of the thermal part. Performance is close to optimum and is significantly better than the one of comparable CPU solvers. We validate our code by simulating the differentially heated cubic cavity (DHC). The computed results for steady flow patterns are in good agreement with previously published ones. Finally, we use our solver to study the phenomenology of transitional flows in the DHC.

Key words: Thermal lattice Boltzmann method, GPU programming, CUDA

1. Introduction

Originating from the lattice gas automata theory [7], the lattice Boltzmann method (LBM) was first introduced by McNamara and Zanetti in 1988 [14] and developed later on by Qian *et al.* [20]. It has since proved to be an interesting alternative to the solving of the Navier-Stokes equations. Besides other advantages over traditional methods in computational fluid dynamics, the LBM happens to be intrinsically parallel, thus easing high performance implementations.

Graphics processing units (GPU) have nowadays outrun CPUs in terms of raw computational power. Their use in general-purpose computations [24], and more specifically in CFD [5], is promising. Successful attempts were made to implement LBM solvers on the GPU [6]. Nevertheless, the wide range of potential applications of the LBM on the GPU remains mostly unexplored, especially for problems involving heat and fluid flows.

The Compute Unified Device Architecture (CUDA), first released by nVidia in 2007, is today's leading technology for general-purpose computation on graphics processing units (GPGPU). The CUDA technology is based on general hardware specifications and a specific programming model, which allows to state generic optimization principles.

In this contribution, we shall present our CUDA implementation of a thermal flow solver. This program is an extended and improved version of the isothermal solver decribed in [17]. It is part of the TheLMA project [19] which aims at providing a comprehensive framework for implementing LBM solvers on GPUs and other emerging many-core architectures.

* christian.obrecht@insa-lyon.fr

2. Hybrid thermal lattice Boltzmann model

In contrast to isothermal simulations, solving thermal fluid flows using the LBM is still a pioneering field. Up to now, adding energy-conservation constraint to the isothermal lattice Boltzmann equation seems to be a more effective method than the double-population approch. We chose to use a hybrid scheme belonging to this category. More specifically, the flow simulation is accomplished by using a D3Q19 multiple-relaxation-time (MRT) model [4], while the heat equation is solved by using a finite-difference scheme. This model is a simplified version of the one described in [11], where the heat capacity ratio $\gamma = C_P/C_V$ is set to $\gamma = 1$, since we were not interested in acoustic effects.

The LBM can be seen as a threefold discretization of the Boltzmann equation, involving time, space and velocities. The discrete velocities $\{\boldsymbol{\xi}_i \,|\, i = 0, \dots N\}$ where $\boldsymbol{\xi}_0 = \mathbf{0}$, are chosen such as to link each lattice site to some of its neighbors. Figure 1 shows the D3Q19 stencil, where each node is connected to 18 of its nearest neighbors.

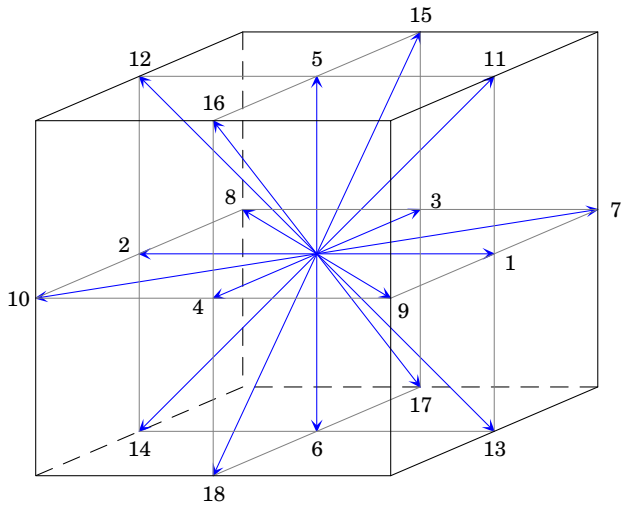


Figure 1: D3Q19 stencil

The equivalent of the single-particle distribution function in the Boltzmann equation is a discrete set of velocity distribution functions $\{f_i \,|\, i = 0, \dots N\}$. Let us denote:

$$\big|f_i(\boldsymbol{x}, t)\big\rangle = (f_0(\boldsymbol{x}, t), \dots f_N(\boldsymbol{x}, t))^{\mathsf{T}}$$

for given lattice node $\boldsymbol{x}$ and time t, T being the transpose operator. The lattice Boltzmann equation (LBE), i.e. the discretized version of the Boltzmann equation, thus writes:

$$\big|f_i(\boldsymbol{x} + \delta t \boldsymbol{\xi}_i, t + \delta t)\big\rangle - \big|f_i(\boldsymbol{x}, t)\big\rangle = \Omega\left(\big|f_i(\boldsymbol{x}, t)\big\rangle\right) \tag{1}$$

where Ω is the collision operator. This equation naturally breaks into the elementary steps of the LBM: the right-hand side describing *collision*, the left-hand side describing *propagation*.

In the MRT approach, the velocity distribution is mapped to a set of moments $\{m_i \,|\, i = 0, \dots N\}$ by an orthogonal matrix M:

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

where $\big|m_i(\boldsymbol{x}, t)\big\rangle$ is the moment vector. Matrix M for the D3Q19 stencil is given in table 1.

The corresponding moment vector is:

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

$$
\mathsf{M} = \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-30 & -11 & -11 & -11 & -11 & -11 & -11 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\
12 & -4 & -4 & -4 & -4 & -4 & -4 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\
0 & -4 & 4 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & -4 & 4 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
0 & 0 & 0 & 0 & 0 & -4 & 4 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
0 & 2 & 2 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -2 & -2 & -2 & -2 \\
0 & -4 & -4 & 2 & 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -2 & -2 & -2 & -2 \\
0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -2 & -2 & 2 & 2 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1
\end{pmatrix}
$$

Table 1: Velocity space to moment space mapping

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} are components of the stress tensor and $p_{ww} = p_{yy} - p_{zz}$, π_{xx}, π_{ww} are third order moments, m_x, m_y, m_z are fourth order moments. The mass density and the momentum are the conserved moments.

The LBE becomes:

$$
\left|f_i(\boldsymbol{x} + \delta t \boldsymbol{\xi}_i, t + \delta t)\right\rangle - \left|f_i(\boldsymbol{x}, t)\right\rangle = -\mathsf{M}^{-1}\mathsf{S}\left[\left|m_i(\boldsymbol{x}, t)\right\rangle - \left|m_i^{(\mathrm{eq})}(\boldsymbol{x}, t)\right\rangle\right] \tag{3}
$$

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

$$
\mathsf{S} = \mathrm{diag}(0, s_1, s_2, 0, s_4, 0, s_4, 0, s_4, s_9, s_{10}, s_9, s_{10}, s_{13}, s_{13}, s_{13}, s_{16}, s_{16}, s_{16}) \tag{4}
$$

We additionally set $s_9 = s_{13}$ and the lattice speed of sound $c_s = 1/\sqrt{3}$. With T denoting the temperature, the equilibrium quantities of the non-conserved moments are given by:

$$
e^{(\mathrm{eq})} = -11\rho + 19\boldsymbol{j}^2 + T \tag{5}
$$

$$
\varepsilon^{(\mathrm{eq})} = 3\rho \tag{6}
$$

$$
\boldsymbol{q}^{(\mathrm{eq})} = -\tfrac{2}{3}\boldsymbol{j} \tag{7}
$$

$$
3p_{xx}^{(\mathrm{eq})} = 3j_x^2 - \boldsymbol{j}^2 \tag{8}
$$

$$
p_{ww}^{(\mathrm{eq})} = j_y^2 - j_z^2 \tag{9}
$$

$$
p_{xy}^{(\mathrm{eq})} = j_x j_y \tag{10}
$$

$$
p_{yz}^{(\mathrm{eq})} = j_y j_z \tag{11}
$$

$$
p_{zx}^{(\mathrm{eq})} = j_z j_x \tag{12}
$$

$$
3\pi_{xx}^{(\mathrm{eq})} = \pi_{ww}^{(\mathrm{eq})} = 0 \tag{13}
$$

$$
m_x^{(\mathrm{eq})} = m_y^{(\mathrm{eq})} = m_z^{(\mathrm{eq})} = 0 \tag{14}
$$

Since we have $\gamma = 1$, unlike [11], we chose to use T instead of $57T$ in eq. 5, which improves numerical stability. The kinematic viscosity ν and the bulk viscosity ζ of the model are:

$$\nu = \frac{1}{3}\left(\frac{1}{s_9} - \frac{1}{2}\right) \qquad \zeta = \frac{2}{9}\left(\frac{1}{s_1} - \frac{1}{2}\right) \tag{15}$$

The temperature T evolves according to the following finite-difference equation:

$$T(\boldsymbol{x}, t + \delta t) - T(\boldsymbol{x}, t) = \kappa \Delta^* T - \boldsymbol{j} \cdot \nabla^* T \tag{16}$$

where κ denotes the thermal diffusivity, and the finite-difference operators are given by:

$$\begin{aligned} \partial_x^* f(i,j,k) &= f(i+1,j,k) - f(i-1,j,k) \\ &-\frac{1}{8}\big(f(i+1,j+1,k) - f(i-1,j+1,k) + f(i+1,j-1,k) - f(i-1,j-1,k) \\ &\quad + f(i+1,j,k+1) - f(i-1,j,k+1) + f(i+1,j,k-1) - f(i-1,j,k-1)\big) \end{aligned} \tag{17}$$

$$\begin{aligned} \Delta^* f(i,j,k) = 2\big(&f(i+1,j,k) + f(i-1,j,k) + f(i,j+1,k) \\ &+ f(i,j-1,k) + f(i,j,k+1) + f(i,j,k-1)\big) \\ -\frac{1}{4}\big(&f(i+1,j+1,k) + f(i-1,j+1,k) + f(i+1,j-1,k) \\ &+ f(i-1,j-1,k) + f(i,j+1,k+1) + f(i,j-1,k+1) \\ &+ f(i,j+1,k-1) + f(i,j-1,k-1) + f(i+1,j,k+1) \\ &+ f(i-1,j,k+1) + f(i+1,j,k-1) + f(i-1,j,k-1)\big) \\ -9 f(i,j,k) \end{aligned} \tag{18}$$

It should be noted that these finite difference operators share the same symmetries as the D3Q19 stencil.

3. Review of the CUDA technology

In this section, we shall at first give a brief review of the CUDA programming model [16]. Then, we shall describe the CUDA hardware in general and the GT200 GPU we used for our computations. Last, we shall discuss the induced constraints which should be taken into account to achieve optimal efficiency.

3.1. CUDA programming model

The CUDA programming model is implemented in the CUDA C language which is an extension to C/C++. Functions in a CUDA C program belong to one of the three following categories:

1. Host code, i.e. functions run by the CPU.
2. Kernels, i.e. functions launched by host code and run by the GPU.
3. Device functions, i.e. functions run by the GPU and called by kernels or other device functions.[1]

A kernel is run in parallel on the GPU. The execution pattern is given at launch time by specifying a *grid*. Threads are grouped in identical arrays called *blocks*, which in turn are assembled to form the execution grid as shown in fig. 2. Blocks may have up to three dimensions, a grid is one- or two-dimensional. The `x`, `y`, and `z` fields of the predefined `blockIdx` and `threadIdx` structures identify each individual thread within the execution grid.[2]

[1] Due to hardware limitations, device functions are in general inlined at compile time.
[2] When a dimension is not in use, the corresponding field is always 1.

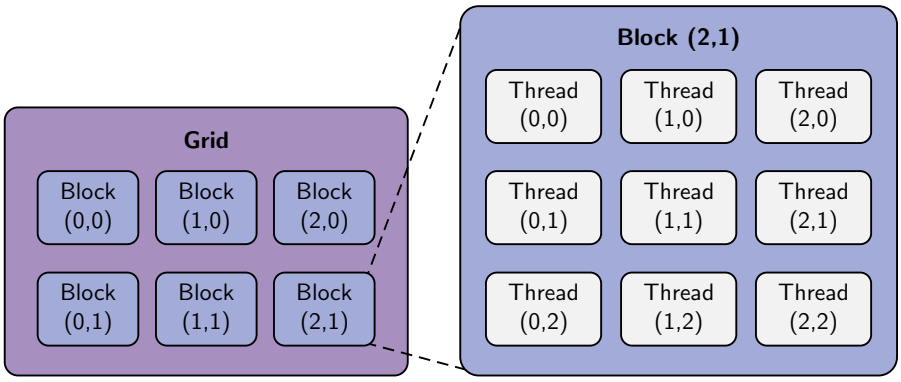


Figure 2: CUDA execution model

Variables local to a kernel are specific to each thread unless declared as *shared*, in which case they are accessible by all threads within a block. No mutual exclusion mechanism is available for shared variables. It is up to the programmer to manage this aspect using block-wise synchronization primitives.

Kernels may also access to *global* memory space, which is visible to each thread of the execution grid. Again, no protection mechanism is available. Nonetheless, global memory is persistent across kernel execution, hence a common way to ensure global synchronization is to perform multiple kernel launches. Global memory, being accessible to host code, is also the usual communication path between CPU and GPU.

Last, it is worth mentioning that threads may also access read-only to *constant* and *texture* memory. Constant memory is a convenient way to store parameters that will stay unchanged all along runtime.[3]

3.2. CUDA hardware

A CUDA capable GPU consists in a set of *streaming multiprocessors* (SMs). Each of these SMs contains several *scalar processors* (SPs), registers, shared memory, and caches for constants and textures, as shown in fig. 3. Furthermore, the GPU is linked to off-chip device memory.

Since being specific to each SM, registers and shared memory are fast, though in rather limited amount. Device memory, in comparison, has limited throughput, suffers from high latency, but is also considerably larger. Table 2 summarizes the technical specifications of the GT200 processor we used for our computations.

Number of SMs	30
Number of SPs per SM	8
Registers per SM	16 384
Shared memory per SM	16 KB
Constant cache per SM	8 KB
Texture cache per SM	8 KB
Device memory	up to 4 GB

Table 2: Technical specifications of the GT200

The recently released Fermi architecture provides in addition L1 and L2 caches for global memory. The L1 cache is local to each SM and has configurable size: either 16 KB with 48 KB shared memory or 48 KB with 16 KB shared memory.

[3]Since textures are mostly relevant in graphics processing, and is of no use in our case, we shall not discuss this feature any further.

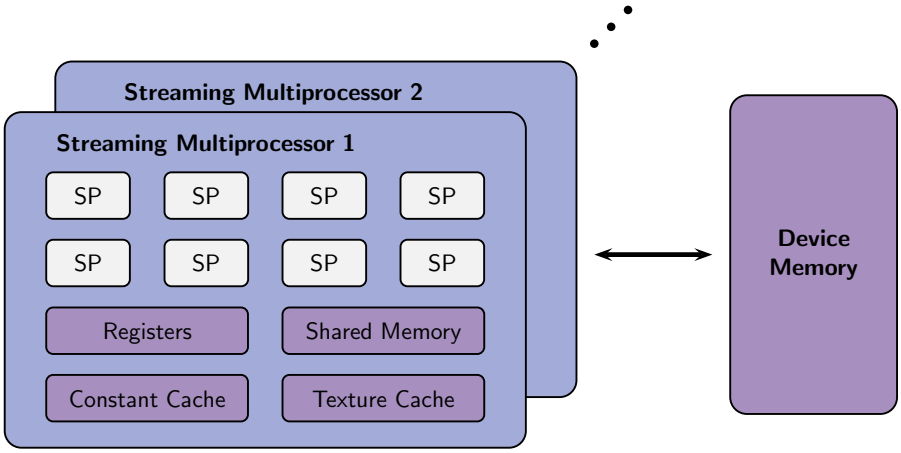


Figure 3: CUDA hardware

Variables local to a kernel are stored in registers except for arrays, since registers are not addressable. Local arrays are stored in the so-called *local* memory, which in fact is hosted in device memory, besides global, constant, and texture memory. Local memory is also used to spill registers if needed.

3.3. Optimization guidelines

A block of the execution grid can only be processed by a single SM. Yet, a SM may handle several blocks concurrently. This leads to several constraints regarding the layout of the grid which are summarized in table 3. To take advantage of the massively parallel architecture of the GPU, the number of concurrently active threads should be as large as possible. It is up to the programmer to define a grid achieving this goal, while avoiding register spilling. Nevertheless, the *occupancy rate*, i.e. the ratio of the number of active threads to the maximum, is usually not a reliable performance indicator. With data intensive applications, for instance, the limiting factor is more likely to be the global memory maximum throughput. Nonetheless, a minimal occupancy rate is required to hide global memory latency.[4]

Max. number of blocks per SM	8
Max. number of threads per SM	1 024
Max. number of threads per block	512
Maximal block dimensions	$512 \times 512 \times 64$
Maximal grid dimensions	$65\,535 \times 65\,535$

Table 3: Grid layout constraints for compute capability 1.3

When run on a SM, a block of threads is sliced into *warps* of 32 threads.[5] To achieve actual parallelism, all threads in a warp must follow the same instruction path. When branching divergence occurs, the execution of the branch paths are serialized, which may dramatically impact performance. Whenever possible, branch granularity should be a multiple of the warp size.

[4] According to nVidia, at least 192 active threads per SM are required to completely hide global memory latency.

[5] The warp size is 32 since the first generation of CUDA capable GPUs. Nevertheless, this value is implementation dependent and might change in the future.

Shared memory is arranged in 32 bits wide memory banks. For the GT200, there are sixteen memory banks.[6] Shared memory transactions are issued by half-warps. Threads in a half-warp accessing to different memory locations lying in the same bank cause a *bank conflict*, which is resolved by serializing the transactions. Yet, shared memory may be as fast as registers, provided care is taken to avoid bank conflicts. The primary purpose of shared memory is to enable block-wise communication. Nonetheless, shared memory is also convenient to prefetch data from global memory, store small local arrays, or avoid register shortage. Such uses contribute to curtail transactions to device memory, therefore may have a major impact on performance.

For data-intensive applications, since global memory is not cached on the GT200, using a well designed memory access pattern is of crucial importance. As for shared memory, global memory transactions are issued by half-warp. These memory accesses may be coalesced into one single transaction of 32, 64, or 128 bytes, provided the address of the corresponding segment is aligned to its size. When the alignment condition is not met, several transactions are issued. As of compute capability 1.2, the hardware is able to reduce the transaction size if possible. For instance, when the threads of a half-warp read consecutive 32-bit words, it yields a single 64 bytes transaction in case of alignment, two 32 bytes transactions for a 32 bytes offset, a 32 bytes and a 64 bytes transaction otherwise.

4. Implementation

4.1. Algorithmic aspect

As described in section 2, for a given time and lattice node, the LBM breaks up in two elementary steps, namely collision and propagation. The lattice Boltzmann equation as formulated in eq. 3 can therefore be split in:

$$\left|\tilde{f}_i(\boldsymbol{x},t)\right\rangle = \mathsf{M}^{-1}\left(\left|m_i(\boldsymbol{x},t)\right\rangle - \mathsf{S}\left[\left|m_i(\boldsymbol{x},t)\right\rangle - \left|m_i^{(\mathrm{eq})}(\boldsymbol{x},t)\right\rangle\right]\right) \tag{19}$$

$$\left|f_i(\boldsymbol{x}+\delta t\boldsymbol{\xi}_i,t+\delta t)\right\rangle = \left|\tilde{f}_i(\boldsymbol{x},t)\right\rangle \tag{20}$$

where eq. 19 describes the collision step and eq. 20 the propagation step. Thus, the hybrid thermal LBM outlined in section 2 corresponds to 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. read neighboring temperatures $T(\boldsymbol{x}+\delta t\boldsymbol{\xi}_i,\, t)$
5. **if** node $\boldsymbol{x}$ is on boundaries **then**
6. apply boundary conditions
7. **end if**
8. compute moments $m_i(\boldsymbol{x},\, t)$
9. compute equilibrium values $m_i^{(\mathrm{eq})}(\boldsymbol{x},\, t)$
10. compute updated distribution $\tilde{f}_i(\boldsymbol{x},\, t)$
11. propagate to neighboring nodes $\boldsymbol{x}+\delta t\boldsymbol{\xi}_i$
12. compute and store new temperature $T(\boldsymbol{x}, t+\delta t)$
13. **end for**
14. **end for**

[6] As for the warp size, the number of shared memory banks is implementation dependent.

4.2. GPU implementation of the LBM

To take advantage of the massive parallelism of the GPU, CUDA implementations of the LBM usually assign a thread to each lattice node [9, 22]. The layout of the execution grid needs therefore to reflect the geometry of the lattice. Global synchronization is achieved by launching a kernel at each time step.

The velocity distribution functions may be stored in either an array of structures (AoS) or a structure of arrays (SoA). The AoS approach happens to be optimal for sequential CPU implementations since it improves data locality, and thus ensures efficient use of the caches. Conversely, for GPU implementations, threads within a warp should access to consecutive global memory locations in order to enable coalesced memory transactions. Therefore, a SoA data layout (or an equivalent multi-dimensional array) is mandatory to achieve efficiency.

To meet alignment constraints, the least significant dimension of the array should be a multiple of the size of a half-warp. Nonetheless, this does not suffice to avoid misalignments, since propagation leads to one unit shifts for the minor dimension. To face this issue, an effective way is to use shared memory to perform propagation along the minor dimension. Yet, the scope of shared memory being limited to the current block, special care has to be taken of distribution values crossing the block boundaries. As described in [21], outgoing values may temporarily be stored in locations left vacant by incoming values. For large lattices, this approach requires therefore a second kernel to rearrange data.

As stated in [18], misaligned read accesses are far less expensive than misaligned write accesses. Hence, an alternative way to handle misalignment consists in replacing the usual out-of-place propagation by in-place propagation at the next time step. Figures 4 and 5 outline the two propagation schemes (in the two-dimensional case, for the sake of simplicity). It was shown in [17] that the cost of misaligned reads is of the same order of magnitude than the overhead of a rearrange kernel.

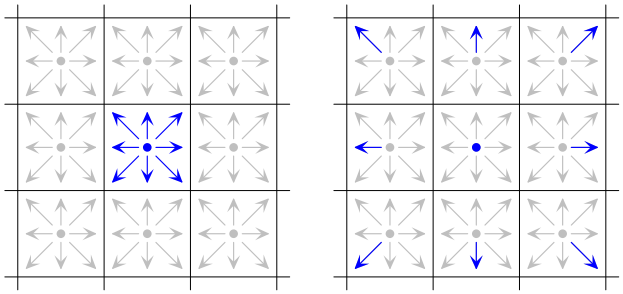


Figure 4: Out-of-place propagation scheme

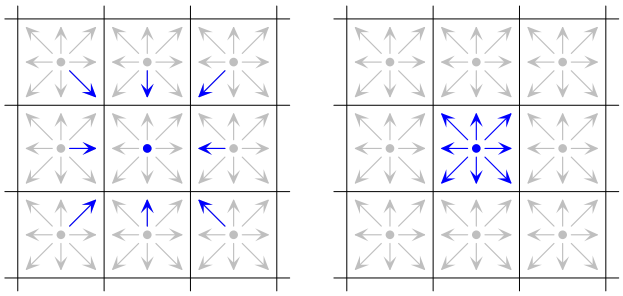


Figure 5: In-place propagation scheme

It should be noted that the in-place propagation approach is simpler and exerts less pressure on hardware than the shared memory approach. Yet, the later has only been used for isothermal LBM implementations

and leaves few room for possible model enhancement. As a matter of fact, using this approach to implement on the GT200 the thermal model we chose would lead to shared memory shortage for blocks greater than 192, since ten floating point numbers per node for particle distribution and nine floating point numbers per node for temperature are required. Handling larger cavities would require the use of a rearrange kernel which has a significant impact on performance.

4.3. Proposed implementation

Our implementation is based on the isothermal flow solver described in [17]. The lattice is a rectangular cuboid of dimensions $N_x \times N_y \times N_z$. We use one-dimensional blocks of size N_x and a two-dimensional grid of dimensions $N_y \times N_z$. For better performance, N_x should be a multiple of the warp size. Additionally, when the three lattice dimensions are different, N_x should be chosen such as to maximize the occupancy rate. Though simple, such a tilling proves to be convenient, since it ensures coalescing of global memory transactions, sufficient occupancy rate and straightforward retrieving of the node coordinates.

To store the velocity distribution functions, we used a multi-dimensional array. The velocity index may correspond to any of the dimensions but the minor one, in order to preserve coalescence. According to [1, 24], the SMs contain translation look-aside buffers (TLB) for global memory. Using the least significant dimension possible to span the velocity distribution reduces the occurrences of TLB misses. We experimented a 13% performance improvement over the major dimension version which is used in [17].

The main concern when implementing a finite-difference solver for the GPU is to curtail global memory read redundancy [15]. For a given block, the required temperatures form a $N_x \times 3 \times 3$ cuboid. Our approach is to fetch these temperatures in shared memory. To perform aligned and coalesced global memory transactions, the threads read the temperatures of nodes sharing the same abscissa. More precisely, thread of index i within block (j, k) reads the temperatures at nodes:

$$(i, j, k),\ (i, j+1, k),\ (i, j-1, k),\ (i, j, k+1),\ (i, j, k-1),$$
$$(i, j+1, k+1),\ (i, j+1, k-1),\ (i, j-1, k+1),\ (i, j-1, k-1).$$

Figure 6 outlines the read access pattern we propose for temperature (in two dimensions, for the sake of simplicity). It should be noted that, using this approach, no read redundancy occurs at block level.

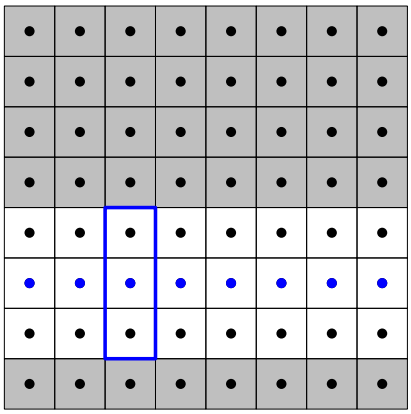


Figure 6: Read access pattern for temperature

4.4. Performance

To evaluate performance, we carried out computations in single precision for a cubic cavity using a Tesla C1060 computing device. As usual for GPU implementations of the LBM, the limiting factor appears to

be the global memory maximum throughput. For the Tesla C1060, the maximum sustained throughput, provided by the CUDA `bandwidthTest` program, is about 73.3 GB/s. Table 4 gives the obtained performance in million lattice node updates per second (MLUPS) for increasing values of N, as well as the corresponding data throughput and the ratio to the maximum throughput.[7]

Size of the cavity	96^3	128^3	160^3	192^3	224^3	256^3
Performance (MLUPS)	319	247	305	335	309	301
Data throughput (GB/s)	61.2	47.4	58.6	64.3	59.3	57.8
Ratio to max. throughput	83.6%	64.7%	79.9%	87.8%	81.0%	78.9%

Table 4: Performance using a Tesla C1060

As shown by the rates, performance is close to optimum for most sizes. It is also worth mentioning that performance is notably higher than with state-of-the-art CPU thermal LBM implementations. For instance, we tested a Palabos [12] based single precision thermal LBM code on a dual Xeon E5560 at 2.8 GHz. We recorded 16.7 MLUPS with 16 OpenMPI processes on a 257^3 cavity.[8]

5. Differentially heated cubic cavity

5.1. Phenomenology of the differentially heated cavity

The hybrid LBM solver implemented on the GPU is used to study the differentially heated cubic cavity outlined in fig. 7. Two opposite vertical walls have imposed temperatures $-T_0$ and $+T_0$, whereas the remaining walls are adiabatic. This configuration has been extensively studied in the two-dimensional configuration (for example [3, 8, 13]) for laminar, transitional and fully turbulent flows.

The three-dimensional configuration has been less studied in the literature than the two-dimensional case because of its computational cost. The first bifurcation is observed for $\mathrm{Ra}_1 \approx 3.3 \times 10^7$ [23] and the consequence is the unsteadiness of the flow pattern. The flow returns to a steady state for higher values of the Rayleigh number. This second transition takes place at a Rayleigh number Ra_2 belonging to the interval $[6.5 \times 10^7;\ 7 \times 10^7]$ [2]. Finally, the flow reverts to unsteadiness for $\mathrm{Ra}_3 \approx 3 \times 10^8$.

5.2. Computational procedure

The fluid is supposed to be incompressible. Applying the Boussinesq approximation, the buoyancy force $\boldsymbol{F}$ is given by:

$$\boldsymbol{F} = -\rho\beta T\boldsymbol{g} \tag{21}$$

where β is the thermal expansion coefficient, and $\boldsymbol{g}$ the gravity vector of magnitude g.

We imposed half-way bounce back no-slip boundary conditions for the LBE part [26]. The temperature field at the adiabatic walls is computed using second order extrapolation.

In order to conserve mass up to the second order, we add $\delta t\boldsymbol{F}$ to the momentum $\boldsymbol{j}$ in two steps: one half before collision, and one half after.

Setting the Prandtl number $\mathrm{Pr} = 0.71$, we use the Rayleigh number Ra and the kinematic viscosity ν as parameters. The thermal diffusivity κ and the value of βg are determined using the dimensionless numbers:

$$\mathrm{Ra} = \frac{2T_0\beta g N^3}{\nu\kappa} \qquad \mathrm{Pr} = \frac{\nu}{\kappa} \tag{22}$$

where $N = N_x = N_y = N_z$ is the size of the cavity. Furthermore, following [10], we set $s_1 = 1.19$, $s_2 = 1.4$, $s_4 = 1.2$, $s_{10} = 1.4$, $s_{16} = 1.98$.

[7] For each node, 48 floating point numbers are transmitted per time step : nineteen numbers are read and written for particle distribution, nine numbers are read and one number written for temperature.

[8] We used $N = 257$ instead of $N = 256$ to avoid cache thrashing.

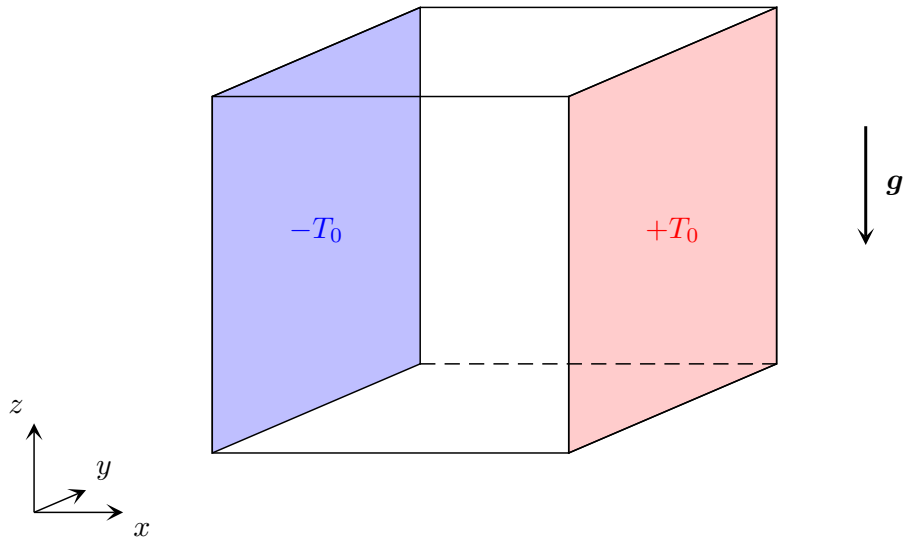


Figure 7: Differentially heated cavity

To check for convergence, the following estimator is computed:

$$\varepsilon_n = \max_{\boldsymbol{x}} |T(\boldsymbol{x}, n\delta t) - T(\boldsymbol{x}, n\delta t - k\delta t)| \tag{23}$$

every $k = 500$ iterations. Convergence to steadiness is declared when the criterion $\varepsilon_n < 10^{-5}$ is satisfied.

The Nusselt number at the isothermal wall is computed using:

$$\mathrm{Nu}_w = \frac{1}{2T_0 N^2} \sum_{y,z} \partial_x T|_{x=wall}\,. \tag{24}$$

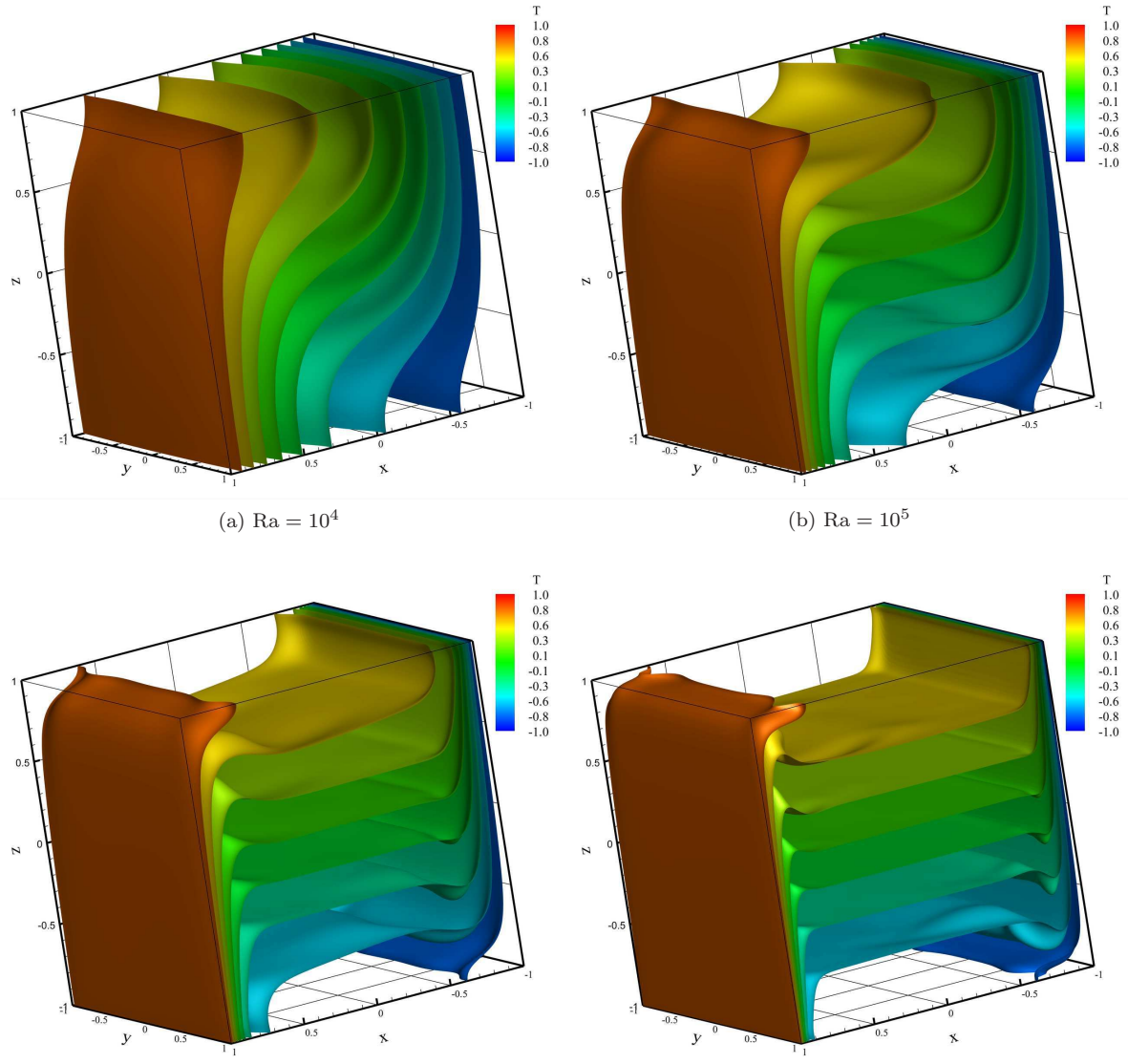
5.3. Numerical results for steady flow patterns

In order to validate our approach, the flow in the differentially heated cavity is computed for Rayleigh numbers equal to 10^4, 10^5, 10^6, and 10^7. The results are compared with data from the literature. Table 5 gives the obtained Nusselt numbers as well as the values published in [25] and [23]. As shown by the relative deviation, our results are in good accordance with the reference values.

Rayleigh number	10^4	10^5	10^6	10^7
Present	2.0560	4.3382	8.6457	16.4202
Wakashima *et al.* [25]	2.0624	4.3665	8.6973	—
Relative deviation	0.3%	0.6%	0.6%	—
Tric *et al.* [23]	2.054	4.337	8.640	16,342
Relative deviation	0.09%	0.03%	0.06%	0.5%

Table 5: Comparison of Nusselt numbers at the isothermal wall ($N = 256$)

In addition, fig. 8 shows the isosurfaces of temperature in the cavity. As the Rayleigh number increases, the temperature in the core of the cavity becomes more stratified. The flow is less influenced by the cavity sidewalls ($y = \pm 1$) and the boundary layers at the active walls become thinner and thinner. It is also possible to confirm the centrosymmetry of the flow and temperature fields.



(a) Ra = 10^4

(b) Ra = 10^5

(c) Ra = 10^6

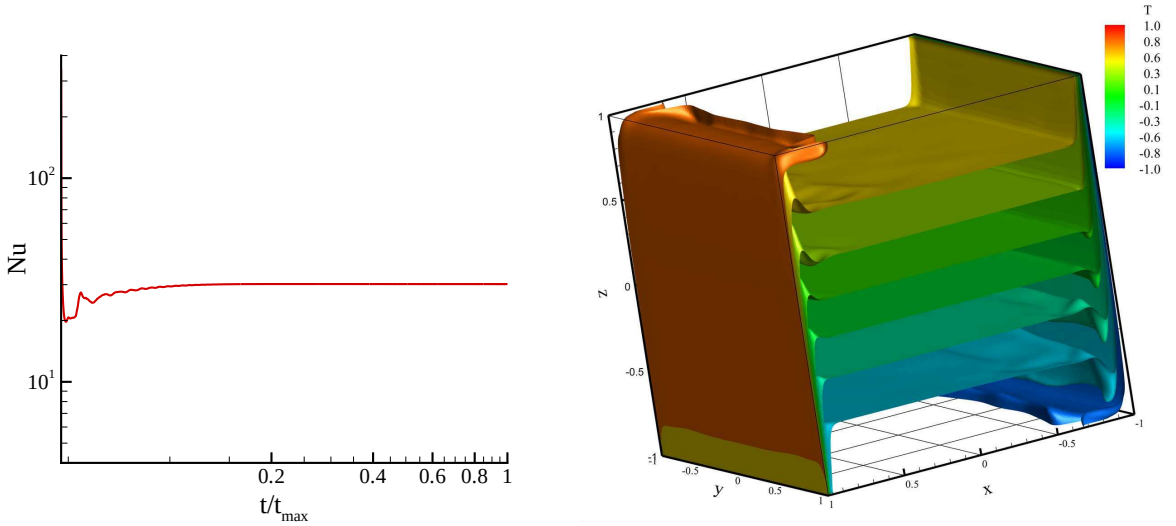
(d) Ra = 10^7

Figure 8: Isosurfaces of temperature

5.4. Numerical results concerning the first and second bifurcation

To locate the bifurcations, we proceeded a to systematic exploration using several GPUs in parallel. According to our computations, the first bifurcation occurs between 3.224×10^7 and 3.225×10^7. The critical Rayleigh number for the first bifurcation is therefore $\mathrm{Ra}_1 = 3.2245 \pm 0.0005 \times 10^7$.

For Rayleigh numbers greater than the second critical Rayleigh number Ra_2, the flow returns to a steady state. The limit is located between 6.401×10^7 and 6.402×10^7. Hence, the second critical Rayleigh number is $\mathrm{Ra}_2 = 6.4015 \pm 0.0005 \times 10^7$.



(a) Nusselt number at the wall

(b) Isosurfaces of temperature

Figure 9: Results concerning the differentially heated cavity at $\mathrm{Ra} = 10^8$

In order to exemplify the flow and temperature fields for the reversion to steadiness, the flow at $\mathrm{Ra} = 10^8$ is exhibited here. Figure 9a shows that the Nusselt number reaches a constant value of $\mathrm{Nu} = 30.2027$, which is in good agreement with the value Nu=30.311 extrapolated in [23].

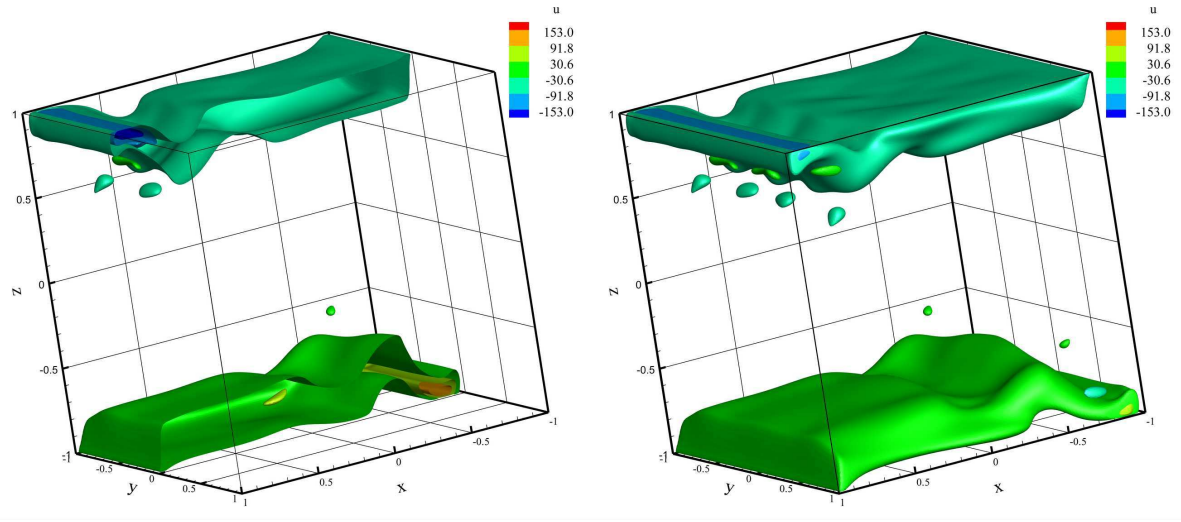
The isosurfaces of temperature, fig. 9b, show that the thermal stratification in the core of the cavity is conserved.

The modifications of the flow field are illustrated by the isosurfaces of the u velocity component in the half-cavity (fig. 10a) and in the entire cavity (fig. 10b). The flow field is no more centrosymmetric. However, there is a symmetry with respect to the plane $y = 0$. The same conclusion holds for the isosurfaces of v and of w.

6. Summary

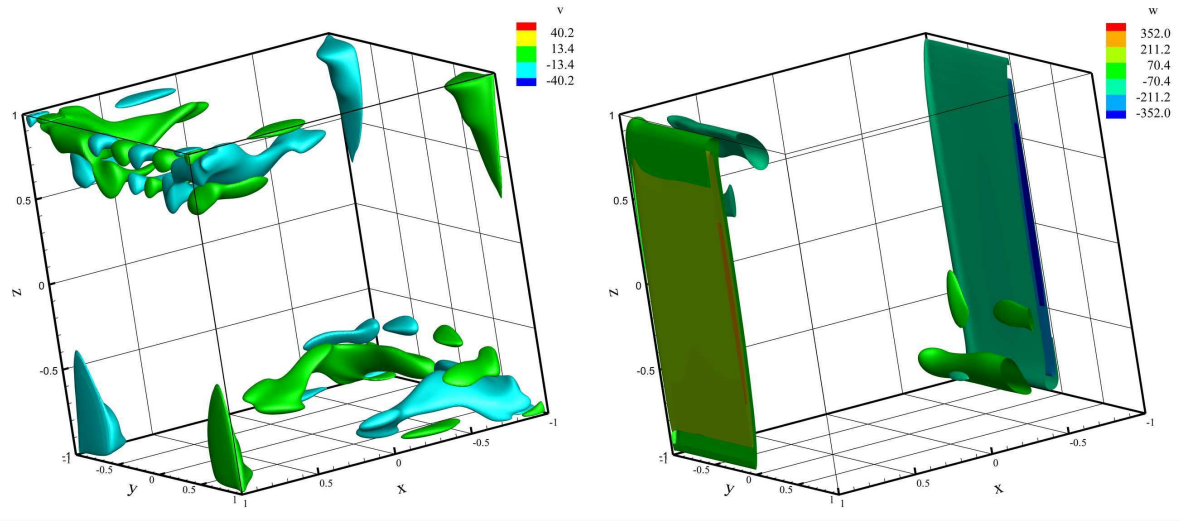
In this work, we devise general optimization strategies for programming data-parallel applications on CUDA enabled GPUs. We describe an effective implementation of a thermal LBM solver for the GPU. The proposed approach for dealing with the thermal part is likely to apply to other multiphysics coupling. Simulation results are in good agreement with available data. Performance is nearly optimal and appears to be significantly higher than for equivalent CPU implementations.

We used a dichotomous procedure to accurately study the different flow patterns in the differentially heated cubic cavity. The flow pattern is laminar up to the first bifurcation at $\mathrm{Ra}_1 = 3.2245 \pm 0.0005 \times 10^7$. The flow becomes unsteady until $\mathrm{Ra}_2 = 6.4015 \pm 0.0005 \times 10^7$ for which it returns to a steady state. The present contribution is the first accurate determination of these critical Rayleigh numbers in the differentially heated cubic cavity. The next step of our work will be the study of the transition to turbulence around $\mathrm{Ra} = 3 \times 10^8$.



(a) Half-cavity (b) Entire cavity

Figure 10: Isosurfaces of u for $\mathrm{Ra} = 10^8$



(a) (b)

Figure 11: Isosurfaces of v (a) and w (b) for $\mathrm{Ra} = 10^8$

References

[1] S. Collange, D. Defour, and A. Tisserand. Power Consumption of GPUs from a Software Perspective. In *Proceedings of the 9th International Conference on Computational Science: Part I*, page 923. Springer, 2009.

[2] G. de Gassowski, S. Xin, and O. Daube. Bifurcations et solutions multiples en cavité 3d différentiellement chauffée – bifurcations and multiple solutions in a differentially heated cubic cavity. *Comptes Rendus Mécanique*, 331(10):705 – 711, 2003.

[3] G. De Vahl Davis. Natural convection of air in a square cavity: A bench mark numerical solution. *International Journal for Numerical Methods in Fluids*, 3:249–264, 1983.

[4] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.S. Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, pages 437–451, 2002.

[5] J. Dongarra, S. Moore, G. Peterson, S. Tomov, J. Allred, V. Natoli, and D. Richie. Exploring new architectures in accelerating CFD for Air Force applications. In *Proceedings of HPCMP Users Group Conference*, pages 14–17. Citeseer, 2008.

[6] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47. IEEE Computer Society, 2004.

[7] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the navier-stokes equation. *Phys. Rev. Lett.*, 56(14): 1505–1508, 1986.

[8] F. Kuznik, Vareilles J., G. Rusaouën, and G. Krauss. A double-population lattice boltzmann method with non-uniform mesh for the simulation of natural convection in a square cavity. *International Journal of Heat and Fluid Flow*, 28(5):862 – 870, 2007.

[9] F. Kuznik, C. Obrecht, G. Rusaouën, and J.-J. Roux. LBM Based Flow Simulation Using GPU Computing Processor. *Computers and Mathematics with Applications*, (27), June 2009.

[10] P. Lallemand and L.S. Luo. Theory of the lattice Boltzmann method: Dispersion, dissipation, isotropy, Galilean invariance, and stability. *Physical Review E*, 61(6):6546–6562, 2000.

[11] P. Lallemand and L.S. Luo. Theory of the lattice Boltzmann method: Acoustic and thermal properties in two and three dimensions. *Physical review E*, 68(3):36706, 2003.

[12] J. Latt, O. Malaspinas, and D. Lagrava. Parallel Lattice Boltzmann Solver. www.lbmethod.org/palabos, 2010.

[13] P. Le Quéré. Accurate solutions to the square thermally driven cavity at high Rayleigh number. *Computers & Fluids*, 20 (1):29 – 41, 1991.

[14] G.R. McNamara and G. Zanetti. Use of the Boltzmann Equation to Simulate Lattice-Gas Automata. *Phys. Rev. Lett.*, 61:2332–2335, 1988.

[15] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

[16] nVidia. *Compute Unified Device Architecture Programming Guide version 2.3.1*, August 2009.

[17] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. A new approach to the lattice Boltzmann method for graphics processing units. *Computers and Mathematics with Applications*, (in press), 2010.

[18] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Global Memory Access Modelling for Efficient Implementation of the LBM on GPUs. VECPAR, 2010.

[19] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. Thermal LBM on Manycore Architectures. www.thelma-project.info, 2010.

[20] Y.H. Qian, D. d'Humières, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *Europhys. Lett*, 17(6): 479–484, 1992.

[21] J. Tölke. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science*, pages 1–11, 2008.

[22] J. Tölke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, 2008.

[23] E. Tric, G. Labrosse, and M. Betrouni. A first incursion into the 3d structure of natural convection of air in a differentially heated cubic cavity, from accurate numerical solutions. *International Journal of Heat and Mass Transfer*, 43(21):4043 – 4056, 2000.

[24] V. Volkov and J.W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press Piscataway, NJ, USA, 2008.

[25] S. Wakashima and T.S. Saitoh. Benchmark solutions for natural convection in a cubic cavity using the high-order time-space method. *International Journal of Heat and Mass Transfer*, 47(4):853–864, 2004.

[26] Q. Zou and X. He. On pressure and velocity flow boundary conditions and bounceback for the lattice Boltzmann BGK model. *Arxiv preprint comp-gas/9611001*, 1996.