On the parallelization of Lattice Boltzmann on GPU’s

Joaquim José da Coste e Simas

December 5, 2010

Abstract

In this work, two numerical schemes of Lattice Boltzmann are implemented in three different parallel machines, each with its own programming model: a multiprocessor computer, a cluster of computers and a GPU. The latter, is a graphics processor with a parallel architecture. This architecture allows it to achieve high computational power with very competitive performances in terms of energetic efficiency.

Lattice Boltzmann methods are known for their efficient and simple numerical schemes. Despite their simplicity, these particle methods top almost any other CFD implementation when dealing with shape-shifting irregular and disordered boundary conditions. This fact alone gives them a very competitive edge. But, it is the combination of these characteristics with its highly parallel nature and the still growing fields of research and applications, that all leads to believe in the present and future importance of this method.

Noticeable results have been achieved, in terms of simulations speed. The more promising of these have been obtained using the GPU: speedups in the order of the hundredths. A cheap, common 0 card, can be used almost as a mini-cluster. This fact will lead to numerous work and investigation to be devoted to Lattice Boltzmann and GPU’s.

Keywords: Lattice Boltzmann, LGBK, MRT, OpenMP, MPI, GPU, CUDA

1 Introduction

Lattice Boltzmann methods hold all the physical validity of the Navier-Stokes while bringing a simpler set of equations that are highly parallelizable. Adding these factor to the exceptionally fast simulations performed with lowcost Graphics Processing Units (GPU’s), has generated high expectatives regarding this method as a future leader in many fields of the CFD world.

The GPU’s are processors that co-work with the CPU’s, relieving them of the intensive work of graphic rendering. Built with that purpose, their architecture allows a much higher computational power and energetic efficiency. These characteristics are making them an attractive and cheaper way of ‘competing’ with Supercomputers and Clusters, as one can pack a bunch of cheap GPU’s together and achieve an exceptional computational power.

Numerous applications are already taking advantage of the GPU’s. In terms of CFD, Euler methods have successfully been ported to the GPU. The achieved speedup rounds 12 [3]. A Burguer’s Equation solver reached 18 [2]. Lattice Boltzmann, with Tolke’s method, obtained a 341 times better performance in a GPU than in a CPU [7].

Since Lattice Boltzmann has it’s strong points in handling with traditionally cumbersome boundary conditions, the combination of these characteristics with that of a highly parallel implementation are very promising.

The next section presents Lattice Botzmann methods and the theoretical background of two numerical implementations.

In section 3 all the developed implementations are described in terms of the modifications performed to adjust the serial code to each parallel machine. The results obtained with different parallel approaches are compared and contrasted in section 4.
2 Numerical method

In general, every LB implementation is composed of a temporal loop where, to a given time and to every node of the lattice, there is a collision between particles, followed by the propagation of these particles to neighbor nodes. The propagation path is well defined by the lattice used. Differences between implementations result from the number of particles present in each node (at any given time), the way the collision step is made, and what type of formulation was used to the boundary conditions [6]. Lattice Botzmann schemes are all explicit second-order accurate (more in the case of turbulence) with the possibility of plummeting to first-order on the boundaries [6].

Generally, the number of particles used per node and the type of space where they exist are represented as $DnQm$. $n$ stands for the number of the dimensions and $m$ the number of particles. It is important to choose properly the number of particles for a given space, as some models may be unable to capture certain physical phenomena. This is due to the stability criterion of the LB: the lattice has to be able to support the fastest speed of physical information. For this reason, LB methods are limited to incompressible regimes (numbers of Mach inferior to 0.33) [10] [4].

To two dimensional simulations the $D2Q9$ model was used. In this model, every node has 9 particles, 8 of which move to 8 neighboring nodes and one particle that remains at rest. To three dimensional lattices, the $D3Q19$ model was used.

![Particles' direction of travel for the D2Q9 (left) and the D3Q19 (right).](image)

The number of particles and their propagation paths are important to the quality of the obtained results. For example, the $D3Q15$ model is very susceptible to short-wave oscillations [4]. The $D3Q19$ is not. That is due, not only to the extra four particles, but also to the way that particles are free to move. Choosing the $D3Q27$ model would not bring much more stability and precision than the $D3Q19$, but it would bring more complexity and computational demand.

2.1 LBGK

Generally, the LBGK model is given by

\[ f_i(x + c_i \Delta t, t + \Delta t) - f_i(x, t) = -\omega(f_i(x, t) - f_i^{eq}(x, t)) \]  

(1)

The terms on the left side represent the advection, and the ones on the right the collision. $\omega$ is defined as the inverse of the time scale and is linked with the kinematic viscosity.

\[ \omega = \frac{1}{\tau}, 0 < \omega < 2 \]  

(2)

\[ \nu = \frac{2\tau - 1}{6} \]  

(3)

$f^{eq}$ is the Maxwell-Boltzmann equilibrium distribution approximation, function of the local conserved quantities: $\rho$, $\vec{v}$, etc. It’s inside this member that all the non-linearities of the Navier-Stokes are hidden.

\[ f_i^{eq} \approx \rho W_i \left(1 + \frac{c_i u}{c_s^2} + \frac{(c_i u)^2}{2c_s^2} + \frac{u^2}{2c_s^2} + \frac{(c_i u)^3}{2c_s^2} + \frac{(c_i u)^4}{2c_s^2}\right) + O[u^4] \]  

(4)
Here $c_s$ represents the speed of sound in lattice units, which is $\frac{1}{\sqrt{3}}$ for both the D2Q9 and D3Q19. Finally, conservation of mass (eq. 5) and momentum (eq. 6) have to be reinforced.

$$\rho = \sum_{i=0}^{m-1} f_i$$  \hspace{1cm} (5)

$$\rho u_{\alpha} = \sum_{i=0}^{m-1} f_i e_{i\alpha}$$  \hspace{1cm} (6)

In case there are external forces acting upon the fluid, each particle will receive a momentum input in accordance to the mass of the particle and it’s direction of propagation [6].

![Figure 2: Algorithm of LGBK’s method.](image)

### 2.2 Multi-Relaxation Time method

With the purpose of having a model that could reduce ghost modes, allow high Reynolds numbers in (relatively) small lattices, while maintaining the simplicity of the LBGK, the Multi-Relaxation Time (MRT) method was devised [4]. Numbers of Reynolds as high as $10^6$ have been obtained for a 2D lid driven cavity, although the accuracy of the results could not be verified [5].

The collision term of the MRT model is given by:

$$C_{12} = -[M]^{-1}[S] (|R\rangle - |R^{eq}\rangle)$$  \hspace{1cm} (7)

To be in accordance with the literature, we have used [ ] to represent a matrix and $|$ to represent a column vector.

$S$ is the relaxation matrix. It is diagonal and every element is given by $s_{ii} = \omega_i = \frac{1}{\tau_i}$. The possibility of having multiple relaxations is what offers this model the increased stability. Note that the entries of $S$ that act upon mass and momentum are taken to be zero to reinforce the conservation of these properties.
If all non-zero values of $S$ take the same $\omega$, the MRT model will have the same behaviour as the LGBK (with the same relaxation) [4].

$$S_{2D} = \text{diagonal} (0, s_2, s_3, 0, s_5, 0, s_8, s_9)$$  \hspace{1cm} (8)$$

$$S_{3D} = \text{diagonal} (0, s_1, s_2, 0, s_4, 0, s_4, s_9, s_{10}, s_9, s_{10}, s_{13}, s_{13}, s_{16}, s_{16}, s_{16})$$  \hspace{1cm} (9)$$

The fluid viscosity is obtained by:

$$\nu = \frac{1}{3} \left( \frac{1}{s_9} - \frac{1}{2} \right)$$  \hspace{1cm} (10)$$

$R$ is a column vector that stores the macroscopic properties of the node. The number of elements of the column are equal to the number of particles living in a node. The elements of $R$ are density $\rho$, the part of the kinetic energy independent of the density $\varepsilon$, the momentum $j$, the energy flux independent of the mass flux $q$, viscous stress tensor $p$ and an anti-symmetric third-order moment $m$ [4].

$$R_{2D} = (\rho, \varepsilon, j_x, q_x, j_y, q_y, p_{xx}, p_{xy})$$  \hspace{1cm} (11)$$

$$R_{3D} = (\rho, \varepsilon, j_x, q_x, j_y, q_y, j_z, q_z, p_{xx}, p_{yy}, p_{zz}, \pi_{xx}, \pi_{yy}, \pi_{zz}, m_x, m_y, m_z)$$  \hspace{1cm} (12)$$

$R$ is obtained by performing a matrix transformation of $f$ through the matrix $M$. This matrix depends upon the numeration given to the directions a particle can travel (fig. 1). A use of a different numeration makes it necessary to re-order the columns of $M$, and is common in the literature to spot different versions of these matrices. But keep in mind it’s function is to summarize the conservation laws, as it is the matricial version of eq. 5 and eq. 6, among others.

$$| R | = [ M ] | f |$$  \hspace{1cm} (13)$$

More concisely, this method consists in working in the momentum space for the collision and in the velocity space for the propagation. That means that in the collision phase we star by calculating the macroscopic properties, $R$ and $R_{eq}$, then calculating a post-collision macroscopic property, $R^*$ (ver eq 14) and then calculate the post-collision PDF as:

$$| f | = [ M ]^{-1} ( | R | - [ S ] ( | R | - | R_{eq} | )) = [ M ]^{-1} | R^* |$$  \hspace{1cm} (14)$$

Since the streaming step is equal to the one of the LGBK, the algorithm is very similar to the one showed to the LGBK, differing only in the collision term. Overall, this method is only 15 % slower than the SRT [4] [9]. All this extra time takes place in the collision step.

## 3 Implementations

All the implementations were made using the C language, its extensions and libraries. OpenMP and OpenMPI were used, respectively, for the threads and the message-passing parallel programming model. All codes were compiled with gcc 4.3.2 except for the CUDA application, compiled with the nvcc 2.2.

In terms of the machines, the OpenMP and the CUDA implementation used a computer with four Intel Xeon processors and four GPU’s (three Tesla C1060 and one Quadro FX1800). For the OpenMPI application, a cluster of four AMD’s Opteron 252 processor was used.

To measure and compare performances we will basically use the speedup and the MLUPS. The speedup, defined as the ratio between the elapsed time of the serial program and that of the parallel implementation, should only be used to compare among implementations on the same computer. To compare implementations across different architectures we will use the MLUPS ‘units’. MLUPS stands for Mega Lattice Updates per Second. It is obtained by inverting the elapsed time of the temporal loop divided by the number or iterations and the number of nodes.
3.1 Serial implementation

The serial implementation is a straightforward transcription of the fluxogram (fig. 2). As will be seen later, the collision step consumes about three quarters of the total time of the temporal loop.

3.2 Shared memory implementation: OpenMP

The best approach of parallelizing LB with OMP is to create multiple threads in the begging of the temporal loop (fig.2). Each thread executes the collision and propagation of a certain number of nodes and, at the end of the iteration, they synchronize with each other through a barrier. After the temporal loop is over, all the threads join into a single thread, responsible for finishing the program. This way we reduce the overhead of creating and destroying threads, as there is only one creation and one destruction. Unfortunately, this approach could not be implemented due to restrictions of the OpenMP’s parallel construct: only one if clause is supported.

The second approach consists in creating the team of threads at the beggining of both spatial loops (fig. 2). Threads are joined at the end of each loop, meaning that, per iteration, there is overhead due to two creations and destructions of threads. The impact of the total time of the overheads can become neglectable if the number of nodes in the mesh is high and if the number of threads is not too high (they normally are in the order of the tenths). This holds true for most cases.

3.3 Distributed memory implementation: MPI

In LB, the collision step is totally local, indepent of the neighboring nodes. It is in the stream step that interaction between nodes occur. It is during this step that one needs to use the message-passing standards when using a distributed memory system.

The assignment of work to the nodes has to be done in a way to minimize the amount of communication needed and, at the same time, maintain similar amounts of work to every processor (load-balancing).

Minimizing communication is the same as minimizing the number of nodes that have to share data across processors. So, for example, if we had a rectangular domain with height H and length L = 4*H, dividing accross the domain would imply that L nodes need to exchange data. A transverse division meant only H. Since both ways hold the same number of nodes for each sub-domain, the latter would provide a better solution, as there are less nodes to exchange data.

Load-balancing is not as easy as assigning the same number of nodes to each processors, as some nodes need different cares. Example of this are boundary or obstacle nodes. This leads to divergencies in the work load. This problem is even worse in irregular meshes. The tipical solution found for this cases is assigning a weight to each node, in accordance to the work it gives to the processor. Then divide the domain in such a way that summing all the weights of the nodes of each sub-domain holds identical values [1].

Independently on how the division is processed, each sub-domain will have to have three sections: the bulk, the frame and the frame-buffer.

Figure 3: The domain division: in white are the lattice sites of the sub-domain 1. The processor responsible for these sites will keep an updated copy of the frontier cells of the sub-domain 2 (first grey cells) so that he can update all the white zone.

In the bulk everything processes normally, as all needed data for propagation is in the sub-domain where the these nodes lives.
The frame consists on the lattice nodes in contact with the border of the domain division. These are the last nodes of the sub-domain to be updated but, in order to do so, they have to use data from nodes living in another sub-domain. This is why there is the frame-buffer. Sometimes called ghost cells, these serve to hold data from other domain and make it available to the frame nodes. At the end of every time iteration the ghost cells must be updated and the loops synchronized. This is where message-passing is used.

3.4 Massively Parallel Processors: CUDA

Since a GPU can have hundredths of thousands of threads running 'simultaneously', we will make a unique association between each thread and a node. This is done with the threads ID. Then each thread will perform the collision and streaming phases of that node. The spatial loops to sweep across all the nodes are no longer needed. Although one still might use them to loop over all the directions of a node.

An important fact valid to all approaches is that the temporal loop must be held by the CPU, as the GPU cannot synchronize all the threads at the end of each iteration.

Our first approach consisted of a direct conversion from the serial version: we used the same data structures as in the CPU, with the difference of replacing the indices used to access the arrays with the threads ID’s.

First, and before entering in the temporal loop, we copy all the needed data to the GPU’s global memory. Then we perform the temporal loop, calling the kernel in every iteration and using a barrier to impede the CPU of proceeding to the following time step. This is done because the GPU will return control to the CPU as soon as the grid is generated, meaning that the CPU will proceed with the program. So we have to guarantee that the GPU has finished it’s task in the same timestep as it was called. After the temporal loop is over, we copy data back to the host.

Subsequently we delve into more careful approaches, taking into account the "rules of thumb for higher speedups": coalescent memory accesses and low global memory usage.

First we reshaped the way the PDF’s were stored in the array. To use coalescence we started by putting the values, of a specific direction, sweeping all the domain nodes. Then sweep the domain again for the following direction, and so on. That way, when threads were accessing the global memory, they would be accessing consecutive data in global memory.

We copy the PDF’s from the global memory to the shared memory. Then perform the collision step. After that, we have a situation similar to a sub-domain of MPI. As threads of different blocks cannot communicate through shared memory, the propagation step needed ghost cells in shared memory, to store values of neighboring PDF’s. But, since there is no global synchronism between threads, we need to break the GPU code into two kernel, just after the collision.

Although it was a good improvement over the first implementation, it hardly reached the solution proposed by [8]. His approach is highly efficient.

Following in his footsteps we separate each direction of the lattice into one-dimensional arrays. Each of these arrays holds the values of the PDF’s of the particles that move in a certain direction, of all the nodes.

![Diagram](image-url)

Figure 4: For a given direction, all the values of the PDF of the particles traveling in that direction are stored sequentially, according to the numeration given.

We make use of three kernels. One to make the collision step and part of the propagation, other to complete the propagation and a third responsible for the boundary conditions.
The first kernel uses one dimensional blocks. We start by copying all the PDF’s of each node to the registers of the thread associated with the node. We guarantee that this copy takes advantage of coalesced accesses as, for example, two threads of the same block, consecutive in their ID number, will access, at the same time, the two PDF’s of a certain direction that are consecutive in memory. The hardware will notice this and coalesce the access. Then each thread will perform the collision step. Then part of the streaming step. This part is done with the aid of shared memory. We allocate arrays of shared memory, one for each direction and with the size of the block. We will pass all the values of the registers to this shared memory taking the opportunity to write them into the place the PDF would propagate.

For example, the PDF of a particle whose direction of travel is to east will move to the following node. So, when writing to the shared memory, we will write to the following node. But the last node of the block cannot write on the first node of the following block, as threads cannot communicate with shared memory of different blocks. So we store the particles leaving the block through the right frontier (the last node of the block) in the first node of the block, as this node couldn’t receive the new PDF’s and had this directions empty. This is done to all particles with horizontal components of propagation. So a node that moves in the north-east direction will only move to the right node, remaining in the same vertical position. The vertical movement is done in the following step. We are gonna copy the values from shared back to global memory. When doing so, we will take the opportunity to make the vertical propagation. The particles that move in the north-east direction and that have previously moved to the right node, will now move to the node above, completing propagation of vertical and diagonal PDF’s.

This ends the first kernel. We still have to exchange the PDF’s that horizontally couldn’t get out of the block and were ‘imprisoned’ in it. The second kernel does this. It moves the PDF’s stored in the begging of and of the end of each of the previously used blocks into the correct positions. It only uses global memory, as it is an unavoidable one-read-one-write.

The third kernel is a direct conversion of the serial boundary conditons.

4 Test cases

In order to benchmark both the LGBK and the MRT codes, two cases were used: the poiseulle flow and the lid-driven cavity.

4.1 Poiseulle Flow

The Poiseulle flow consists in having a fluid between two infinite plates, at rest, subjected to a pressure gradient between the inlet and the outlet. This makes the fluid move, reaching a steady state with a parabolic velocity profile. To this flow there is an exact solution of the Navier-Stokes (eq. 15).

\[ u(y) = \frac{G H^2}{8 \nu} \left( 1 - \left( \frac{2y}{H} \right)^2 \right), \quad \frac{H}{2} < y < \frac{H}{2} \]  (15)

Here \( \frac{G \Delta P}{\rho L} \), \( H \), \( L \) and \( \Delta P \) are, respectively, the height and length of the channel and the pressure gradient.

This flow is mainly used to 'calibrate the flow viscosity' [6]: we select a viscosity and compare the attained profile with that given by the analytical solution.

The two top plates are simulated with first order bounce-back boundary condition, while the other two frontiers are considered periodic (infinite domain). The effect of the pressure is translated into a bias, \( g \), applied to the PDF’s, during the collision step [6].

\[ f'_i = f_i - \omega (f_i - f_{eq}) + g_i \]  (16)

The bias is calculated by considering the momentum that is necessary to be equivalent to a given pressure drop. We are replacing the pressure at the in and outlet’s by a forcing term in every node. Calculating the moment of each node according to the expression given before, we can see that there is only one horizontal component, pulling to the direction of the lesser pressure.
\[ g = \frac{U_0\nu}{8H^2} \]  

(17)

We used [6]'s suggestion and used a mesh of \(128 \times 32\) with \(\nu = 0.1\) and expected to obtain the parabolic profile with the maximum equal to 0.1.

In can be seen that near the walls, the profile drops abruptly, instead of smoothly as before. This is the result of the first order accurate boundary condition, as it is independent of the mesh size.

4.1.1 Lid-driven cavity Flow

The lid-driven cavity is a rich testing environment with simple boundary conditions: three no-slip and a sliding wall at the top.

We used the cavity to benchmark the values accuracy by comparing the velocity profiles to the ones Ghia obtained with DNS methods. We compare for Reynolds numbers of 100 and 1000. We will use Reynolds of 100 for the LGBK method (fig. 6) and the 1000 for the MRT (fig. 7). This is done only to avoid repeating the same graphic’s twice.

Both boundary conditions were simulated with first order approximations. The lid velocity, in lattice units, is set to be 0.1, the lattice is \(128 \times 128\) and \(\tau\) is set to 0.5384. To the MRT model we used the following relaxations

\[ s_1 = s_4 = s_6 = 0, s_2 = 1.1, s_3 = 1.0, s_5 = s_7 = 1.2, s_8 = s_9 = 1 \quad \frac{1}{\tau} \]  

(18)

Figure 6: Horizontal and vertical velocity profiles for, respectively, a longitudinal and a transversal section. Obtained with SRT for a \(Re=100\).

In bidimensional serial code, the collision step takes about 77 % of the time. The streaming step and boundary conditions take about 21 %. Since both steps can be parallelized, Lattice Boltzmann methods
Figure 7: Horizontal and vertical velocity profiles for, respectively, a longitudinal and a transversal section. Obtained with MRT for a Re=1000.

Table 1: Speedups obtained for the OpenMP implementation for up to four processors.

<table>
<thead>
<tr>
<th>Lattice</th>
<th>Serial (sec.)</th>
<th>2 procs</th>
<th>3 procs</th>
<th>4 procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>32$^2$</td>
<td>3.13</td>
<td>1.84</td>
<td>2.85</td>
<td>3.48</td>
</tr>
<tr>
<td>64$^2$</td>
<td>12.47</td>
<td>1.95</td>
<td>2.97</td>
<td>4.0</td>
</tr>
<tr>
<td>128$^2$</td>
<td>50.0</td>
<td>2.02</td>
<td>2.82</td>
<td>3.89</td>
</tr>
<tr>
<td>256$^2$</td>
<td>200.04</td>
<td>2.02</td>
<td>2.87</td>
<td>3.77</td>
</tr>
<tr>
<td>512$^2$</td>
<td>798.99</td>
<td>2.02</td>
<td>3.02</td>
<td>3.98</td>
</tr>
<tr>
<td>1024$^2$</td>
<td>3188.77</td>
<td>2.08</td>
<td>3.03</td>
<td>3.91</td>
</tr>
</tbody>
</table>

offer a parallelizable fraction of 98%. What this means is that, according to Amdhal’s law, if we maintain the problem size constant, the maximum theoretical speedup that can be achieved is 50. But, to obtain it, more than 10000 processors were needed. To a speedup of 47, ‘only’ 767 (a saturation curve).

Our MRT implementation was about 30% slower than the SRT. This means that some optimization can be performed in order to obtain the 15% referred by [4].

For the three dimensional case (3D lid-cavity), the collision and propagation took, respectively, 84 and 16 % of the time. Meaning that, not only the overhead of initializing and ending the variables of the program are negligible, but also that there is a slightly edge of 2% in parallelization of 3D cases over 2D. This is in accordance with [6]. This little improvement translates, for the same problem size and according to Amdahl, in a linear speedup with the number of processors.

Depending on the processor used, a serial implementation typically achieves values of MLUPs ranging from around 0.5 to 2 MFLUPs. A 3D case offers lower MFLUPs, as each node has more particles in which to take action.

5 Results

5.1 Shared Memory parallelization: OpenMP

We will compare the elapsed time of the program for different mesh sizes and for different number of processors. Each processor is only given, at most, one thread, so, we generate up to four threads. All the values were obtained for the same parameters and the same number of iterations. See table 1.

Fixing a mesh size we can see a linear evolution with the number of processors (fig. 8).

Increasing the number of cores reduces the amount of time needed to complete the program. The linear behaviour, predicted by both Amdahl and Gustafson’s laws, is verified. Bigger meshes are closer to the speedups’ trendline because the overhead of generating/destroying threads is hidden by the amount of work each thread has to perform.
Figure 8: Code scaling with the number of cores.

Figure 9: The increase in performance, up to a certain point, with the increase of the lattice dimension.

Code scaling with the number of cores.

Table 2: Speedups for different mesh sizes and cluster nodes.

<table>
<thead>
<tr>
<th>Lattice</th>
<th>Speedups [s]</th>
<th>2 nodes</th>
<th>3 nodes</th>
<th>4 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$32^2$</td>
<td>11.7</td>
<td>1.80</td>
<td>1.83</td>
<td>2.13</td>
</tr>
<tr>
<td>$64^2$</td>
<td>42.7</td>
<td>1.92</td>
<td>2.79</td>
<td>3.14</td>
</tr>
<tr>
<td>$128^2$</td>
<td>187.2</td>
<td>2.08</td>
<td>3.28</td>
<td>4.28</td>
</tr>
<tr>
<td>$512^2$</td>
<td>2995.2</td>
<td>2.074</td>
<td>3.28</td>
<td>4.14</td>
</tr>
<tr>
<td>$1024^2$</td>
<td>11980.8</td>
<td>2.02</td>
<td>3.1</td>
<td>4.3</td>
</tr>
</tbody>
</table>

In terms of MLUPs, we can say that the total MLUPs achieved by the system is proportional to the MLUPs of a single processor scaled up by the total number of processors used. So, the top performance achieved when using the four CPU’s, is 6.56 MLUPS.

5.2 Distributed memory: MPI

The time to run a simulation, for different meshes is found on table 2.

With small meshes we can verify that, even with 4 nodes, the improvements only occurs by a factor of two. That is due to the fine granularity present in small meshes. To achieve coarse granularity we have to use bigger meshes. It’s only then that the speedup is close to the number of processors used.

With the four node cluster we achieved a maximum of 3.88 MLUPS. It is lower than the value obtained with OpenMP, but that’s because each individual processor of the cluster is ‘slower’ than those of Tesla. The same serial code has 1.64 MLUPS for a single Tesla processor and 0.88 MLUPS for a single processor of the Cluster. Roughly half. Once again one can approximate the number of MLUPs achieved by the cluster by multiplying the number of processors in a cluster by the MLUPs speed of single processor. The velocity fields predicted with serial, OpenMP and MPI are virtually the same.

5.3 Massively parallel processors: CUDA

The maximum speedup achieved with a LGBK implementation of a GPU was 110 or 180MFLUPs.

Independently of the implementation, the GPU achieves it’s peak performance for bigger lattices. We also see that up from a certain point, it saturates, as it was expected (fig. 9).

Overall, the maximum MLUPs achieved with the three parallel methods are resumed in the following table.

As can be seen, the GPU offers a great edge when it comes to Lattice Boltzmann methods and alikes. A single GPU could outperform all the other methods. To achieve the same performance as the GPU, we needed 110 CPU’s or 205 cluster nodes.
Table 3: Comparison of the peak values in the three implementations.

<table>
<thead>
<tr>
<th></th>
<th>OpenMP</th>
<th>MPI</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>4.0</td>
<td>4.28</td>
<td>110</td>
</tr>
<tr>
<td>MFLUPs</td>
<td>6.56</td>
<td>3.88</td>
<td>180</td>
</tr>
</tbody>
</table>

6 Conclusion

In this work we have made two implementations of the Lattice Boltzmann numerical schemes: the LGBK and the MRT. These implementations were initially made in serial and then ported to a shared memory architecture, to a distributed memory and to a GPU, with, respectively, OpenMP, MPI and CUDA.

All the codes were validated with two classical benchmarks: the poiseulle flow and the lid-driven cavity.

In summary, Lattice Boltzmann methods are mainly competitive in cases where there are scattered obstacles across the fluid, where the boundary conditions are irregular and/or change with time. They have a very parallel nature and can be implemented efficiently in shared and distributed memory architectures and in GPU’s. This last one revealed, by far, the best performances. Taking into account that the GPU is fairly cheap, the ratio between performance obtained and price tips the scale, undoubtably, to the GPU’s.

In terms of methods, the LGBK is the fastest of the two, but should only be used in flows with low Reynolds. MRT is a bit slower but compensates this fact by presenting a much more stable nature. Future work should be related with three dimensional applications in GPU’s clusters.

References