XL CILAMCE
IBERO-LATIN AMERICAN CONGRESS ON COMPUTATIONAL METHODS IN ENGINEERING
NOVEMBER 11-14, 2019
Praiamar Natal Hotel & Convention
Natal, RN-BRAZIL

PERFORMANCE ANALYSIS OF THE LATTICE BOLTZMANN METHOD IMPLEMENTATION ON GPU

Waine B. de Oliveira Jr.
Alan Lugarini
Admilson T. Franco
waine@alunos.utfpr.edu.br
alansouza@utfpr.edu.br
admilson@utfpr.edu.br
Research Center for Rheology and non-Newtonian Fluids (CERNN). Universidade Tecnológica Federal do Paraná (UTFPR)
Deputado Heitor Alencar Furtado Street, 5000, 81280-340, PR, Brazil

Abstract. The computational interest on the lattice Boltzmann method (LBM) has grown over the years. The simplicity of its implementation and the local nature of most of its operations allows the use of a parallel computing architecture. In this way, GPUs (Graphics-Processing Units) suits very well for LBM implementation, offering good performance and scalability for a relatively low price. This work presents a LBM implementation on GPUs for D2Q9 and D3Q19. It is written in C/C++ programming language and uses CUDA for the use of its GPU resources. For the code optimization, strategies such as memory layout and merging all operations are presented. Also, designs principles to facilitate alternating boundary conditions are described. The code is validated and its performance is evaluated. Because of the lack of work on the matter, it is analyzed and discussed how some simulation's parameters affects the performance of the code. These are: single and double precision; number of threads per block; ECC (error-correcting code memory) on and off; storing macroscopic variables; domain size; rates of saving and residual. Tests are made using Nvidia's GPUs from Kepler, Pascal and Volta micro-architectures. The results are considered satisfactory, achieving the state-of-the-art performance.

Keywords: CUDA, Graphics processing unit (GPU), Lattice Boltzmann method (LBM), High perfomance computing (HPC)

1 Introduction

The lattice Boltzmann method (LBM) is an alternative to traditional CFD (computational fluid dynamics) methods, like FVM (finite volume method) or FDM (finite difference method), for fluid simulation. It solves the lattice Boltzmann equations (LBE), as opossed to solving Navier-Stokes. It has been receiving a lot of attention over the last years, due to its computational benefits over other methods. One of the main benefits is that most of the LBM's operations are local, which allows the use of parallel architectures for simulation.

The implementation of LBM on GPUs (Graphics Processing Unit) has been studied since early 2000's, as by Li et al. [1], because of the cost effectiveness it achieves. Since then, many works, as Mawson and Revell [2], Habich et al. [3], Herschlag et al. [4], have proposed techniques to LBM optimization on GPUs. The main strategies are about memory layout, due to the performance being mostly limited by memory bandwidth. The release of CUDA by Nvidia [5] has facilitated the development of GPU algorithms, with it being used by most of recent works about LBM on GPUs.

This work presents a LBM implementation on GPU using CUDA API and C/C++ language. The design choices and optimization techniques are based on Januszewski and Kostur [6] and Mawson and Revell [2]. The code is validated using parallel plates flow and turbulent Taylor-Green vortex (TGV). Evaluation of the error for single precision is made using TGV on two dimensions.

It is analyzed the impact of simulations parameters and code changes on the program's performance, such as single and double precision, number of threads per block, ECC (error-correcting code memory) on and off, storing macroscopic variables in global memory, domain's size and rate of savings and residual calculation. State-of-the-art performance is achieved for GPUs Tesla K20Xm and Tesla P100. Tesla V100 performance exceeds the limitation of the GPU bandwidth. Speculations about the reasons for that are made.

The structure of the paper is as follows. In section 2 a mathematical description of LBM is given. Section 3 briefly describes GPU core concepts for the implementation and the motivation for using GPUs to run LBM. Section 4 discusses the algorithm implemented, its design, the motivation for some choices and how LBM's performance is measured. Section 5 presents the code validation, the impact of single and double precision on simulation's error and discusses how parameters impact code's performance. In section 6 are the conclusions.

2 LBM

The lattice Boltzmann method (LBM) is a mesoscopic method for fluid flow simulations. It is usually discretized into a regular Cartesian grid, in which each node represents part of the fluid and has set of populations (f_i). Each one representing the distribution of the fluid for a given direction (c_i), given by a sum of unit vectors (e_i). The directions are defined by the velocity set, represented as DnQm, which also defines the weight (w_i) of each population. Where n is the number of dimensions (usually two or three) and m the number of discrete velocities. In this work, D3Q19 and D2Q9 were used.

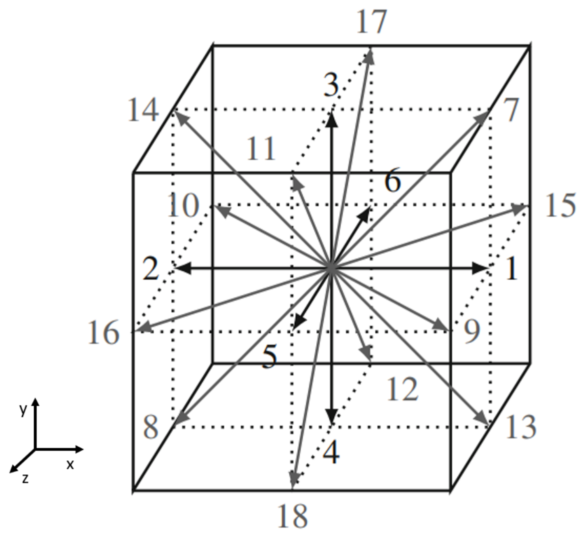
The governing equation for the LBM is the lattice Boltzmann equation (LBE). It is usually divided in two parts for the algorithm. The collision, when the result of the right side of Eq. (1) is computed, and the streaming, when that value is assigned to the left side of Eq. (1).

Table 1. D2Q9 velocity set.

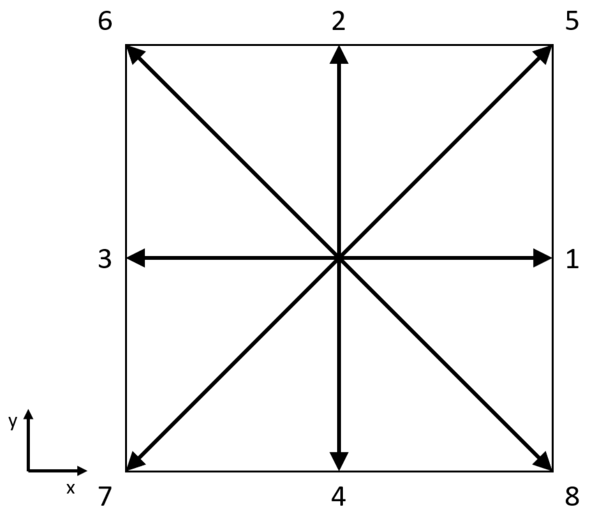
i	0	1	2	3	4	5	6	7	8
w_i	$\frac{4}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$
c_{ix}	0	+1	0	-1	0	+1	-1	-1	+1
c_{iy}	0	0	0	0	-1	+1	+1	-1	-1

Table 2. D3Q19 velocity set.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
w_i	$\frac{1}{3}$	$\frac{1}{18}$	$\frac{1}{18}$	$\frac{1}{18}$	$\frac{1}{18}$	$\frac{1}{18}$	$\frac{1}{18}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$
c_{ix}	0	+1	-1	0	0	0	0	+1	-1	+1	-1	0	0	+1	-1	+1	-1	0	0
c_{iy}	0	0	0	+1	-1	0	0	+1	-1	0	0	+1	-1	-1	+1	0	0	+1	-1
c_{iy}	0	0	0	0	0	+1	-1	0	0	+1	-1	+1	-1	0	0	-1	+1	-1	+1



(a) D3Q19 (from Kruger et al. [7]).



(b) D2Q9.

Figure 1. Velocities for D3Q19 and D2Q9.

On Eq. (1), f_i is the probability density function (population) and Ω_i the collision operator for a direction i, at $\vec{x}$ in space and t in time.

$$f_i(\vec{x} + \vec{c}_i \Delta t, t + \Delta t) = f_i(\vec{x}, t) + \Omega_i(\vec{x}, t). \tag{1}$$

All discretized values are defined in terms of lattice units. For simplicity, $\frac{\Delta x}{\Delta t}$ (lattice speed) is assumed to be one and c_i is the velocity i in the velocity set.

One of the most common and used collision operator is the Bhatnagar-Gross-Krook operator (BGK) by Bhatnagar et al. [8], defined by Eq. (2).

$$\Omega_i(f) = \frac{f_i - f_i^{eq}}{\tau} \Delta t. \tag{2}$$

Where τ is the relaxation time, related to the fluid's viscosity via $\tau = (1 + 6\nu)/2$ and f_{eq} is the equilibrium distribution, given by Eq. (3).

$$f_i^{eq}(\vec{x}, t) = \rho w_i \left(1 + \frac{\vec{u} \cdot \vec{c}_i}{c_s^2} + \frac{(\vec{u} \cdot \vec{c}_i)^2}{2c_s^4} - \frac{\vec{u} \cdot \vec{u}}{2c_s^2}\right). \tag{3}$$

In which w_i is the velocity weight, c_s is the speed of sound ($1/\sqrt{3}$ for D2Q9 and D3Q19), $\vec{u}$ and ρ

are macroscopic variables, velocity and pressure respectively, given in terms of the populations as Eq. (4) shows.

$$\rho = \sum_i f_i \qquad \rho\vec{u} = \sum_i f_i e_i. \tag{4}$$

There are modifications to the LBM, like multiple relaxation times (MRT), regularization, immersed boundary method (IBM), force terms and many others.

The boundary conditions are also a key for LBM simulations. As said in Kruger et al. [7], there are two classes of methods: link-wise and wet-node. In the link-wise family, the boundary nodes are $\Delta x/2$ away from the wall and for wet-node, the nodes are on the wall. This must be take into account when interpreting simulations results.

The bounce-back boundary condition is a link-wise method and one of the most used for stationary walls. It is defined as

$$f_{i^*} = f_i \qquad c_{i^*} = -c_i. \tag{5}$$

Where f_{i^*} is the unknown population. For moving walls or pressure, the Zou-He boundary condition, proposed by Zou and He [9] is very used. It is a wet-node method and can be defined as

$$f_{i^*}^{neq} = f_i^{neq} + \frac{\vec{t}\cdot\vec{c_i}}{|\vec{c_i}|}N_t \qquad c_{i^*} = -c_i. \tag{6}$$

In which f_{i^*} are the unknown populations and $f_i^{neq} = f_i - f_i^{eq}$. $\vec{t}$ is the tangential vector of the wall and N_t is the transverse momentum correction. It can be determined by forcing a velocity or pressure condition on the node and solving the linear system using Eq. (4). Hecht and Harting [10] shows the solution for D3Q19.

3 GPU

GPUs are well suited for parallel operations, mainly when the operations are the same for all domain. Also, the capacity of floating point operations per second (FLOPS) is very high for modern GPUs, with Nvidia [11] getting up to 15.7 TFLOPS for single precision. Nvidia [5] compares the theoretical FLOPS and memory bandwidth between Nvidia's GPUs and Intel CPU's over the years. The graphics cards presents a considerably higher performance for both. This makes LBM fits very well on GPUs, due to the local nature of the method and it being memory and processing (e.g. calculations) intensive.

The development of CUDA by Nvidia [5] has lead to many LBM applications using it, as Habich et al. [3], Januszewski and Kostur [6], Schreiber et al. [12]. One of the reason for that is the low learning curve for it and an easy integration with C/C++ code.

CUDA uses threads to parallelize the program. They are combined in sets of 32 called warps, as reported in Nvidia [5]. All threads in a warp must be in the same block and they always execute the same instruction. If a thread diverges, it is disabled. The number of threads per block must be a multiple of 32 to optimize performance. Otherwise resources would be lost by processing a warp with less than 32 threads.

A block must be processed in only one streaming multiprocessor (SM) as well. But more than one block can be processed by one SM, if resources (as registers) are available. The number of threads per block and the number of blocks are defined in the kernel launch, a special CUDA function.

4 Algorithm

The algorithm's core is the main LBM operations, collision, streaming, boundary conditions and macroscopic variables calculation. There are two main schemes of LBM algorithm: push and pull. Push first make the collision and then stream the populations to the neighbour nodes, pushing them. Pull first stream the population from adjacent nodes and then collides it. Both schemes are shown in Fig. 2.

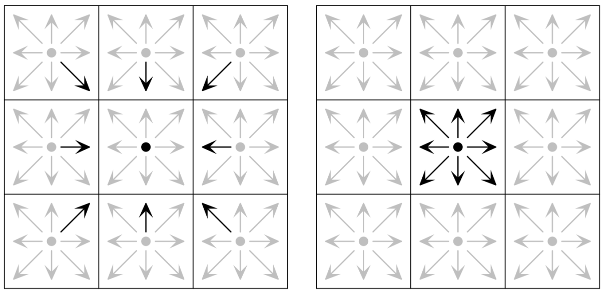


Figure 2. Representation of the streaming populations for the pull scheme on the left and for push scheme on the right. D2Q9 is used. Source is Obrecht et al. [13].

For the pull scheme, there are misaligned reads from memory and aligned writes. On the other hand, push scheme has aligned reads and misaligned writes to memory. Some works, as Mawson and Revell [2], shows that the cost of misaligned writes are more impactful than misaligned writes for Kepler architecture. Despite this, pull presented no significant overcome on performance over the push. So, for simplicity, the push scheme was used.

The code was written using C/C++ and CUDA. For compilation, it was used the `nvcc` compiler. Earlier works about LBM on GPUs using CUDA, as Xian and Takayuki [14] and Calore et al. [15], used one kernel for each operation of the LBM, usually one for streaming, one for collision and another for boundary conditions. But this leads to higher synchronization and kernel launches overheads and diminishes the temporal locality of the algorithm. So the approach taken for the code organization was the same as by Januszewski and Kostur [6], using just one kernel for all LBM operations, as showed in algorithm 1. Another kernel is used for population and macroscopic variables initialization.

Algorithm 1 Kernel strucutre of the LBM push algorithm for one node in the domain

```
Load all node's populations to local variable
if Node has boundary condition then
    Apply boundary condition
end if
Calculate macroscopic variables (velocity and density) using Eq. (4)
if macroscopic variables are required then
    Write macroscopic variables to global array
end if
Collide and stream populations to global array using Eq. (1)
```

All global arrays are one dimensional and its spatial location is converted from 3D to 1D via `index_scalar` for macroscopic variables, and `index_pop` for populations. All kernel launches consists of a block with t threads in the x axis, where t is usually 32, 64 or 128. The grid size is calculated for completing the domain of the lattice in all directions (x, y and z). There are no optimizations using `__launch_bounds__` or `-maxrregcount`, because it is usually handcrafted for each GPU. So this may be a possible improvement on the code performance, as pointed out by Calore et al. [15].

For boundary condition, a similar scheme of the one by Januszewski and Kostur [6] is used. A bitmap of 32 bits is used to classify the node and its boundary condition, whether is has one or not. The bitmap contains the information if the node has or not a boundary condition to apply; the normal direction of the node; the boundary condition scheme to apply (Zou-He, bounce back, free slip, etc.); the index for a global array that contains the macroscopic variables values of the node, as density or velocity, that are used. This allows for easy change between boundary conditions and schemes to use.

4.1 Memory layout

As discussed in Mawson and Revell [2], Herschlag et al. [4], Januszewski and Kostur [6], Kruger et al. [7], Schreiber et al. [12], the performance of LBM is limited by the memory bandwidth. So it is very important that the algorithm has a good spatial and temporal locality. This is achieved mainly by the memory layout.

There are two main kinds of memory layout for LBM, the array of structures (AoS) and the structure of arrays (SoA). The AoS consists of an array containing many structures, such as the population. In this layout, the population zero of one node is followed, in memory, by the population one of the same node and so on. So when loading the population zero to the cache, the next populations of the node is also be loaded.

In the SoA, all the populations zero are contiguous in memory. So the population zero of the node i is followed, in memory, by the population zero of the node $i+1$ and so on. The same for all populations. In this layout, when loading the population of an node i, the same population of the adjacent nodes is also loaded.

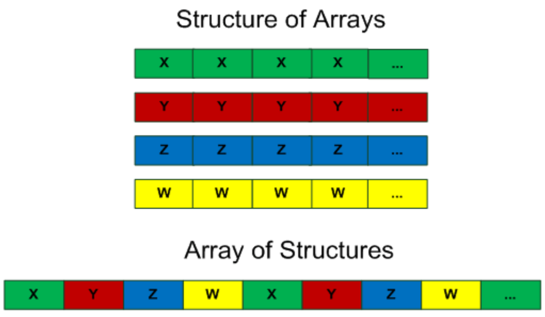


Figure 3. Representation in memory of structures containing the variables X, Y, Z and W with the array of structures layout and with structure of arrays.

Despite the AoS being more common and intuitive, for SIMD (single instruction, multiple data) architectures, as GPUs, it presents low spatial locality. The most common layout on the literature, as observed by Herschlag et al. [4], is the SoA. It takes advantage of how the GPU access memory, allowing for contiguous memory reading or writing, depending on the LBM scheme used (push or pull).

So the chosen layout was the SoA, for populations and macroscopic variables, with the x axis made continuous. This is because of the thread continuity in x for the warps, as said in Nvidia [5].

4.2 Host functions

Some tasks can only be done or are more efficient with host functions. One example is saving macroscopic variables to the hard drive, which can only be done by the operational system, therefore by the host. Another one is calculating the velocity (or any other macroscopic) residual using the Eq. (7).

$$Res = \frac{\sum \mid v_1 - v_0 \mid}{\sum \mid v_1 \mid}. \tag{7}$$

Where v_1 is the present velocity value and v_0 is the reference velocity, from a previous iteration, to be compared to.

This requires the sum of values in a matrix, which can be quite slow on GPU. But the main reason for using a host function is that the GPU resources are not used. So the GPU capacity is not diminished and the host calculates the residual concurrently with the simulation.

The function for saving macroscopic variables is very simple, writing the binary values of an array to a file. The one for calculating the residual just loops over the domain, calculating one element of the summation and, after the loop, adds it all together.

4.3 Performance measurement

The performance for LBM algorithms is measured usually by two parameters. One is the number of MLUPS (million lattice updates per second), which allows to predict the time of a simulation. The other one is the bandwidth, that is calculated using Eq. (8).

$$Bandwidth = \frac{2 \cdot sizeof(\texttt{float}) \cdot Number\ of\ nodes \cdot Q \cdot Number\ of\ iterations}{Simulation\ Time}. \tag{8}$$

Q represents the number of populations of the velocity set and `float` can be single or double precision. The multiplication by two is because all populations are load and then stored, so there are two memory operations for each population every iteration. The result is given in B/s (bytes per second) and can be converted to Mb/s or Gb/s.

Because the LBM is limited mostly to the bandwidth, as discussed in Mawson and Revell [2], the maximum theoretical MLUPS is bounded to it as well. It can be calculated using the maximum bandwidth of the GPU and then converting it to MLUPS using Eq. (9).

$$MLUPS = \frac{Bandwidth}{Q \cdot 2 \cdot sizeof(\texttt{float})} \cdot 10^{-6}. \tag{9}$$

5 Results

For simulation, three different machines were used, each with GPU models from three Nvidia micro-architectures: Kepler (Tesla K20Xm PCIE 6Gb with a GK110), Pascal (Tesla P100 PCIE 16Gb with a GP100) and Volta (Tesla V100 SXM2 16Gb with a GV100). The processor was an Intel i7-7700k 4.20GHz for the Kepler, and a Intel Xeon 2.30GHz for both Pascal and Volta. The ECC was enabled for Pascal and Volta and for Kepler there are tests with ECC enabled and disabled. For all GPUs the boost clock was disabled.

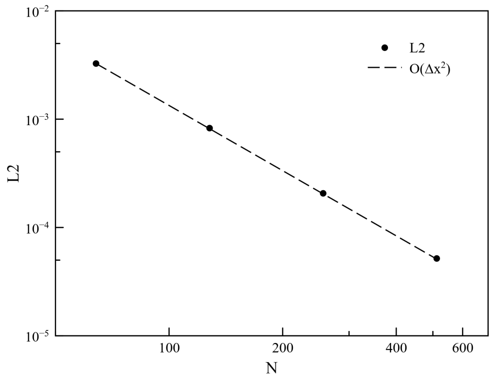
All machines used SSD for recording. The tests were run using CUDA 10.1 Toolkit on Windows 10 64-bit system for Kepler and on Ubuntu 18.04 64-bit system for Pascal and Volta. All the validations and precision tests were made using the Kepler machine.

5.1 Validation

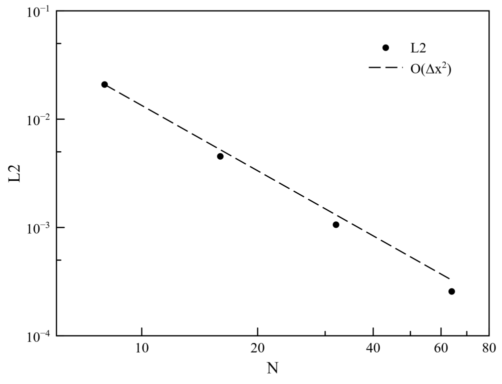
Parallel plates flow was used for validation of D2Q9 and D3Q19. Equation 10 are the pressure boundary conditions imposed and the normalized analytical solution for the flow velocity. For all simulations the domains were a cube with N^3 lattices for D3Q19 and N^2 for D2Q9. Double precision is used. The $L2$ norm, given by Eq. (11), was calculated for the velocities in both cases, using a perpendicular plane to the flow for parallel plates on $x = 0.5$.

$$
\begin{aligned}
\rho(x=0) &= \rho_0. \\
\rho(x=1) &= \rho_0 + \frac{12\rho_0 u_0(\tau - 0.5)}{N}. \\
u_x(y) &= \frac{6(y-y^2)}{u_0}. \\
x, y &\in [0,1].
\end{aligned}
\tag{10}
$$

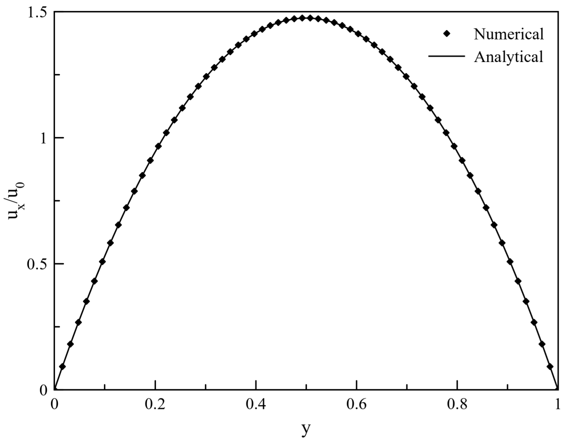
For the lattice test, the N was doubled, u_0 reduced in a half and the number of iterations multiplied by four. This is to keep physical time and τ constant. The $L2$ norm is $O(\Delta x^2)$, as said in Kruger et al. [7]. Figure 4 shows the validations results.



(a) $L2$ for D2Q9.



(b) $L2$ for D3Q19.



(c) Velocity profile for D3Q19 with $N = 64$.

Figure 4. Validation results for parallel plates. For both velocity sets it was used $u_0 = 0.1$, $\rho_0 = 1$ and $\tau = 0.74$ for the minimum N. For subsequent N, u_0 was divided by tw and the number of steps multiplied by four, such that physical time and τ were kept constant.

$$
L2 = \frac{\sum \left(q_{numerical} - q_{analytical}\right)^2}{\sum q^2_{analytical}}.
\tag{11}
$$

The error reduces as expected for both velocity sets, being reduced by a factor of 0.25 when N is doubled. It was also made a simulation for turbulent Taylor-Green vortex. The initialization was made using f^{neq}, with ρ and u given by Eq. (12).

$$
\begin{gathered}
u_x(x,y,z) = u_0 sin(x)cos(y)cos(z). \\
u_y(x,y,z) = -u_0 cos(x)sin(y)cos(z). \\
u_z(x,y,z) = 0. \\
\rho(x,y,z) = \rho_0 - \frac{3\rho_0 u_0^2}{16}[cos(2x) + sin(2y)][cos(2z) + 2]. \\
x,y,z \in [0,2\pi].
\end{gathered} \tag{12}
$$

The dissipation rate, which is the time derivative of integral kinetic energy, was observed over time. Figure 5 shows the results in comparison with Nathen et al. [16] and Brachet et al. [17]. The present work presents a better approximation to Brachet et al. [17] than Nathen et al. [16] presents. This may be due to different initialization scheme.

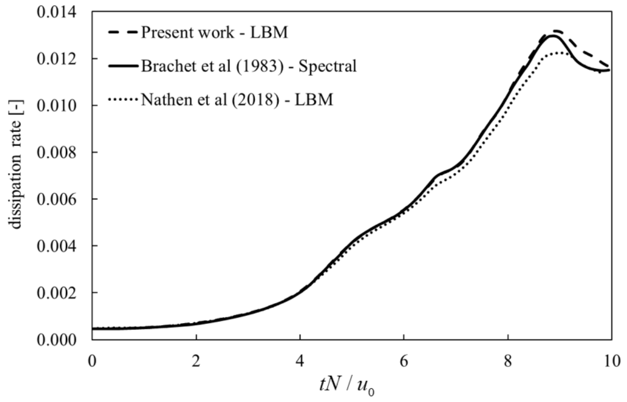


Figure 5. Dissipation rate for Taylor-Green vortex over time. $N = 256$ and $u_0 = 0.05$ were used.

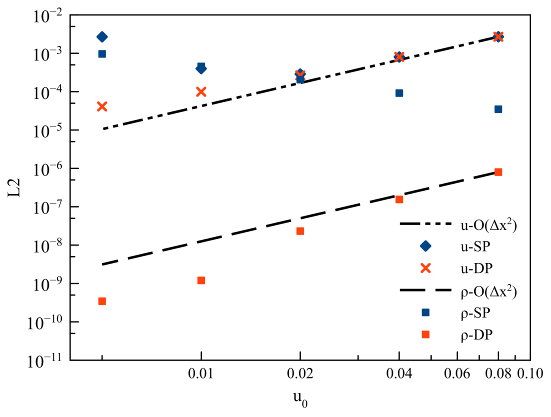
5.2 Single and double precision

Due to the higher performance of single over double, the impact of the numerical error on the simulation results was evaluated. For that, tests were made using Taylor-Green vortex flow and D2Q9. Analytical solution is given by Eq. (13). For initialization it was used f_{eq} and regularization. $L2$ norm is calculated over all the domain. Figure 6 shows the results.

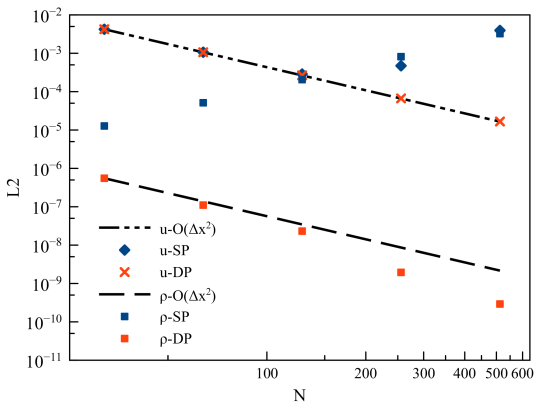
$$
\begin{gathered}
u_x(x,y,t) = -u_0 cos(x)sin(y)e^{-2t\nu}. \\
u_y(x,y,t) = u_0 sin(x)cos(y)e^{-2t\nu}. \\
\rho(x,y,t) = \rho_0 - 0.75\rho_0 u_0^2[cos(2x) + sin(2y)]e^{-4t\nu}. \\
x,y \in [0,2\pi].
\end{gathered} \tag{13}
$$

For the velocities, both when varying N and when varying τ, for $u_0 \leq 0.01$ the single precision error is higher than expected. For $u_0 \geq 0.02$, there is no difference between single and double precision. So for values of $u_0 \geq 0.02$ single precision does not have significant numerical error on the velocities, compared to double. A similar result is obtained by Januszewski and Kostur [6]. But for cases with complex geometries, a very large domain or that heavy differs from the case simulated, this may not be extendable.

For density, single precision shows a significant higher error than double. The error for single is between 10^1 up to 10^7 times the error for double. The expected behaviour of $O(\Delta x^2)$ is also not encountered for single, with the $L2$ norm getting higher as u_0 is decreased. This found both for N constant and for τ constant. For cases where the density field is important, it is highly recommended to use double precision, due to the density's error for single precision.



(a) $N = 128$



(b) $\tau = 0.581$

Figure 6. $L2$ errors for single and double precision for Taylor-Green vortex. A curve showing the expected error behavior is also presented. On the left are the errors for N constant, varying u_0 and τ. On the right are the errors for τ constant, varying u_0 and N. Both were made such that physical time is preserved.

5.3 Benchmark

The performance of the LBM algorithm is measured by the MLUPS and bandwidth of the simulation. The case used was the Taylor Green vortex, with periodic conditions on all surfaces, $\tau = 0.9$ and $u_0 = 0.05$ for D3Q19. Table 3 shows the performance achieved by previous works.

The present work achieves the state of the art performance for both Kepler and Pascal. No work using the Tesla V100 was found. Table 5 shows the performance achieved by the present work. It is important to mention that only one test was made for single precision for each GPU, so it is very likely that the code can achieve greater performance for that configuration. This is true mainly to Volta, because the only test with single precision made was for $N = 128$, when Volta presented a considerable higher performance for $N = 32$.

Table 3. Performance results from previous works for D3Q19. The MLUPS are the peak ones with the described configuration.

GPU	Reference	Precision	ECC	MLUPS
Tesla K20Xm	Januszewski and Kostur [6]	Double	Off	649
Tesla K20Xm	Januszewski and Kostur [6]	Single	Off	1247
Tesla P100	Herschlag et al. [4]	Double	On	1659
Tesla P100	Schreiber et al. [12]	Double	Off	$\sim$1580
Tesla P100	Schreiber et al. [12]	Single	Off	$\sim$2960

For bandwidth measurement, Table 4 shows the results from the `bandwidthTest` from Nvidia [5]. The values are less than the maximum reported by Nvidia, but reflects a lot better the GPU memory transfer capability.

The performance gain for single precision (i.e. `float`) over double precision (i.e. `double`) is very significant, ranging from 60% to 90%. This is due both to memory and calculations. The size of single precision variables is one half the size of double, so the number of memory reads is reduced by a half.

Table 4. Measured bandwidth taken by the average of five `bandwidthTest` for each GPU and the maximum MLUPS possible for D3Q19, calculated using Eq. (9) and considering sizeof(`float`) 8 bytes (double precision). The test program transfers 100 times 32Mb of data using the option `cudaMemcpyDeviceToDevice`.

GPU	ECC	Avg. bandwidth (Gb/s)	Max. MLUPS
Tesla K20Xm	Off	194.7	687.7
Tesla K20Xm	On	173.8	613.9
Tesla P100	On	501.0	1769.6
Tesla V100	On	738.8	2584.7

Table 5. Performance results from present work for D3Q19. The MLUPS are the peak ones with the described configuration. Performance ratio is calculated by $\frac{Bandwidth}{Avg.\ bandwidth}$, where $Avg.\ bandwidth$ is from Table 4.

GPU	Precision	ECC	MLUPS	Bandwidth (Gb/s)	Performance ratio (%)
Tesla K20Xm	Double	Off	635.6	179.9	92.4%
Tesla K20Xm	Single	Off	1111.1	157.3	90.5%
Tesla P100	Double	On	1643.6	465.3	92.9%
Tesla P100	Single	On	3051.9	432.0	86.2%
Tesla V100	Double	On	3420.8	968.5	131.1%
Tesla V100	Single	On	3959.8	560.6	75.9%

For calculations, Kepler (Nvidia [18]) has a relation 1:3 for single/double operations (i.e. it takes three times longer to do an operation with a `double` than a `float`) while Pascal and Volta have a relation 1:2 (Nvidia [11, 19]). So the calculations are also faster for single precision.

These are the main reasons for the great difference on MLUPS. On the other hand, single precision presents higher numerical error and for cases such as DNS this may impact have a great impact on the simulation. Also, the algorithm's bandwidth for all architectures, specially Kepler and Volta, is lower for single precision than for double.

The Volta achieves a greater bandwidth than the theoretical by $\sim$30% for $N = 32$, the smallest N used for simulation. The probable reason for that is the size of the L1 cache for the Tesla V100, which has 80 streaming multiprocessors (SM) and 128Kb for the L1 of each, achieving the maximum of 10Mb of memory in the L1. The Tesla P100, on the other hand, has 24Kb in the L1 and 56 SM, totalizing 1.3Mb of memory in the L1, 13% the total of the Tesla V100. The total GPU memory used is 9.5Mb for $N = 32$.

This great L1 capacity must minimize the access of global memory for small simulations so that the bandwidth is not limited by global accesses. Another reason may be architectural differences from Volta to Pascal and Kepler, with memory accesses optimized on Volta. Further investigation is required for LBM performance using Volta devices.

Figure 7 shows the performance for the number of threads per block for each GPU. Increasing the size of the block, it is observed a minor increase of the MLUPS for Kepler and Volta. Pascal did not show significant variation. The difference from 32 threads to 128 is 5.5% for Kepler, 0.2% for Pascal and 1.1% for Volta. For blocks with more than one dimension the performance was reduced for all tests. So as long as the number of threads is a multiple of 32 (warp size Nvidia [5]), the impact of it on simulations

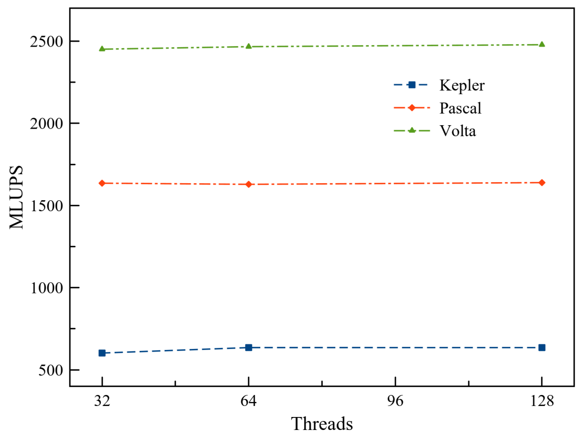


Figure 7. MLUPS for Kepler, Pascal and Tesla with $N = 128$ and varying the number of threads in one block. The blocks are one dimensional, with threads only on the x axis. ECC is off for Kepler and double precision is used.

Table 6. MLUPS for storing macroscopic variables (density and velocity in each direction) in global memory on no iteration and every iteration. The difference is calculated by $1 - \frac{MLUPS\ no\ storing}{MLUPS\ storing}$. For $N = 128$ it was used 64 threads per block and 32 threads per block for $N = 32$. ECC for Kepler is off and double precision is used.

GPU	N	MLUPS no storing	MLUPS storing	Difference (%)
Kepler	32	516.1	494.4	4.2%
Kepler	128	635.54	568.33	10.6%
Pascal	32	711.33	647.0	9.0%
Pascal	128	1628.8	1463.5	10.2%
Volta	32	3420.8	3145.6	8.0%
Volta	128	2466.41	2193.1	11.1%

is low, only considerable for Kepler architecture.

The difference between storing macroscopic variables in global memory values every iteration and in no iteration is also evaluated. Table 6 shows the results for each GPU. The loss for storing is very high, up to 10%, so the macroscopic variables must be written to global memory only when necessary, for performance improvement.

The impact of the ECC for Kepler was measured, MLUPS decreased by 18.8%, for $N = 128$ and 64 threads per block. ECC has a great impact on the LBM performance for the Kepler architecture. The loss on performance is higher than the bandwidth loss, with this being 10.8%. For Pascal and Volta, with the ECC turned on, the bandwidth rate reaches 92.2% and 94.5% for the same case. While on Kepler it gets to only 84.2% (considering the maximum bandwidth as the one from Table 4 with the ECC on).

This may be due to ECC optimizations on Pascal and Volta architectures or the software not being optimized. So if a higher performance is needed, the ECC is a critical point for Kepler and it should be turned off, despite not recommended.

Lattice resolution and the impact of it on the performance was evaluated. Fig. 8 shows the results. For both Kepler and Pascal, the performance increased with the size of the domain. The gain from $N = 32$ to $N = 128$ was a lot higher for Pascal than Kepler, 129.9% for the first one and 16.7% for the second. Volta has a greater MLUPS for $N = 32$ for motives already discussed and does not have significant difference from $N = 64$ to $N = 128$.

This is a great feature, because as long as this relation continues, the simulation becomes faster (or at least does not slow down) for higher resolutions.

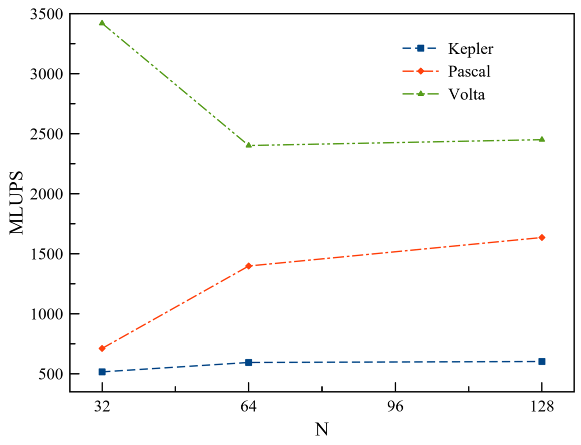
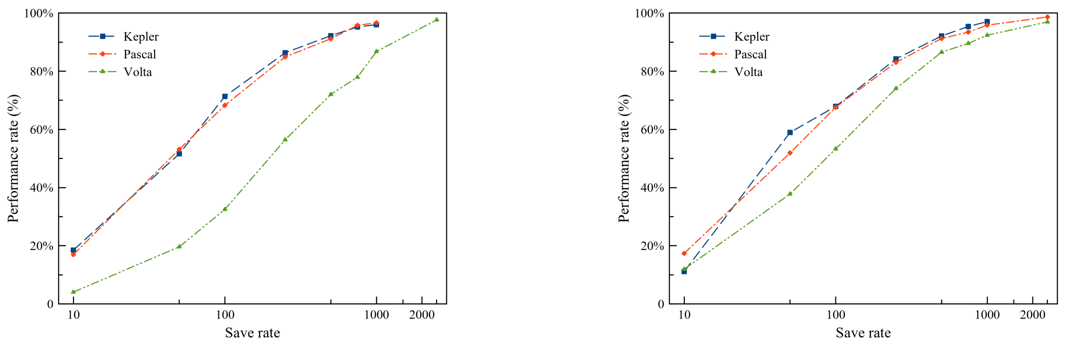


Figure 8. MLUPS for Kepler, Pascal and Tesla with 32 threads per block and varying N. The blocks are one dimensional, with threads only on the x axis. ECC is off for Kepler and double precision is used.

It is very common to save macroscopic variables during the simulation. Like for having a condition to start in case of some system failure, to see transient phenomena as in DNS and many other reasons. The rate of savings impacts on the performance, because a device synchronization is necessary as the transference of device memory to host memory. Figure 9 shows the results of the savings tests.



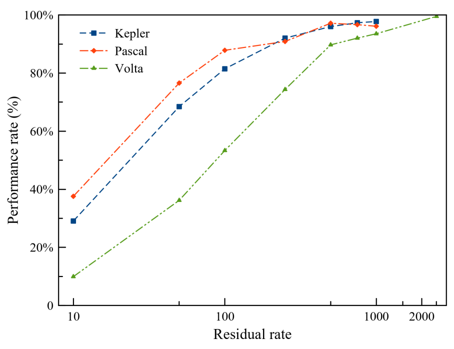
(a) $N = 32$ (b) $N = 128$

Figure 9. Tests performances for each rate of savings, which is the number of iterations between savings. 100% represents the performance with no saving. $N = 32$ uses 32 threads per block and 64 threads per block is used by $N = 128$. ECC was disabled for Kepler and double precision is used.

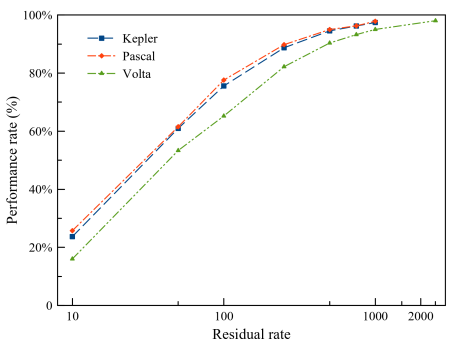
For all GPUs the impact for a rate of savings lower than 500 is quite significant, with more than 10% of loss in performance. For low rates of savings, the simulation gets very slow, losing all advantage of using GPU. Kepler and Pascal presents similar behavior for both $N = 32$ and $N = 128$. On Volta the impact is higher, mostly for $N = 32$. This is due to the Volta performance being higher, so the overheads of saving are a lot more significant than to the other ones. For both N values, with rate of savings higher than 1000, the impact on performance is less than 5% for Kepler and Pascal. The same happens to Volta for a rate of savings of 2500.

Another common thing to do is keep track of some macroscopic variables and its values over the time. One example of that is calculating the velocity residual with the Eq. (7). Figure 10 shows the results of the residual calculation tests.

Like the rate of savings test, for a rate of residual calculation lower than 500 the performance loss



(a) $N = 32$



(b) $N = 128$

Figure 10. Tests performances for each rate of residual calculation, which is the number of iterations between residual calculation. 100% represents the performance with no residual calculation. $N = 32$ uses 32 threads per block and 64 threads per block are used by $N = 128$. ECC was disabled for Kepler and double precision is used.

is very significant. Also, Kepler and Pascal presented a similar behavior as well. But the comparison of Pascal and Volta to Kepler must be careful. Because different processors are used, so this may have influenced the results. The impact on Volta is higher than on the others, the reason is the same as for saving. The performance loss is negligible (less than 5%) for a rate of residual calculation higher than 500 on Kepler and Pascal and 1000 on Volta.

6 Conclusion

A implementation of the LBM for D2Q9 and D3Q19 using CUDA was presented. The algorithm makes use of the push scheme and combine all operations of the method (collision, streaming, boundary conditions and macroscopics calculation) in one kernel. The memory layout used and its motivation is discussed. Also, host functions and the performance measurement are presented. The two velocity sets are validated using parallel plates flow.

The impact of single precision on the simulation error is analyzed. The results showed that it is not recommended for $u_0 \leq 0.01$ or when density precision is crucial. Code's performance achieved the state-of-the-art, with over 90% bandwidth for all GPUs. It surpassed the theoretical maximum MLUPS for Tesla V100, with a performance 30% over the maximum. It is shown that the impact for rates of savings and residual calculation are negligible for rates higher than 1000 steps.

Future works may analyze the impact of saving and residual in multiple nodes, since there's need of data transactions between nodes in this case. Also code support for more velocity sets and features, as multiple-relaxation-time (MRT), immersed boundary method (IBM), force terms and others. Due to performance higher than theoretical for Volta, a further investigation on the architecture and the perks of its optimizations for LBM is also required.

Acknowledgements

This paper used the resources of CERNN - Research Center for Rheology and non-Newtonian Fluids. The authors would like to thank Luiz Gustavo Ricardo for the technical help on the GPUs and Marco Ferrari for the insights and tips given.

References

[1] Li, W., Wei, X., & Kaufman, A., 2003. Implementing lattice boltzmann computation on graphics hardware. *Visual Computer*, vol. 19, pp. 444–456.

[2] Mawson, M. J. & Revell, A. J., 2014. Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs. *Computer Physics Communications*, vol. 185, n. 10, pp. 2566 – 2574.

[3] Habich, J., Zeiser, T., Hager, G., & Wellein, G., 2011. Performance analysis and optimization strategies for a D3Q19 lattice Boltzmann kernel on nVIDIA GPUs using CUDA. *Advances in Engineering Software*, vol. 42, n. 5, pp. 266 – 272. PARENG 2009.

[4] Herschlag, G., Lee, S., S. Vetter, J., & Randles, A., 2018. GPU Data Access on Complex Geometries for D3Q19 Lattice Boltzmann Method. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 825–834.

[5] Nvidia, 2019. *CUDA 10.1 Toolkit Documentation*.

[6] Januszewski, M. & Kostur, M., 2014. Sailfish: A flexible multi-GPU implementation of the lattice Boltzmann method. *Computer Physics Communications*, vol. 185, n. 9, pp. 2350 – 2368.

[7] Kruger, T., Kusumaatmaja, H., Kuzmin, A., Shardt, O., Silva, G., & Viggen, E. M., 2017. *The Lattice Boltzmann method: Principles and practice*. Springer, 1 edition.

[8] Bhatnagar, P. L., Gross, E. P., & Krook, M., 1954. A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems. *Phys. Rev.*, vol. 94, pp. 511–525.

[9] Zou, Q. & He, X., 1997. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. *Physics of Fluids*, vol. 9, n. 6, pp. 1591–1598.

[10] Hecht, M. & Harting, J., 2010. Implementation of on-site velocity boundary conditions for D3Q19 lattice Boltzmann simulations. *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2010, n. 1, pp. 10–18.

[11] Nvidia, 2017. NVIDIA Tesla V100 GPU Architecture. Technical report, Nvidia Corporation.

[12] Schreiber, M., Riesinger, C., Bakhtiari, A., Neumann, P., & Bungartz, H.-J., 2017. A Holistic Scalable Implementation Approach of the Lattice Boltzmann Method for CPU/GPU Heterogeneous Clusters. *Computation*, vol. 5.

[13] Obrecht, C., Kuznik, F., Tourancheau, B., & Roux, J.-J., 2011. A new approach to the lattice Boltzmann method for graphics processing units. *Computers and Mathematics with Applications*, vol. 61, n. 12, pp. 3628 – 3638. Mesoscopic Methods for Engineering and Science — Proceedings of ICMMES-09.

[14] Xian, W. & Takayuki, A., 2011. Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster. *Parallel Computing*, vol. 37, n. 9, pp. 521 – 535. Emerging Programming Paradigms for Large-Scale Scientific Computing.

[15] Calore, E., Gabbana, A., Kraus, J., Pellegrini, E., Schifano, S., & Tripiccione, R., 2016. Massively parallel lattice–Boltzmann codes on large GPU clusters. *Parallel Computing*, vol. 58, pp. 1 – 24.

[16] Nathen, P., Gaudlitz, D., Krause, M., & Adams, N., 2018. On the Stability and Accuracy of the BGK, MRT and RLB Boltzmann Schemes for the Simulation of Turbulent Flows. *Communications in Computational Physics*, vol. 23, pp. 846–876.

[17] Brachet, M. E., Meiron, D. I., Orszag, S. A., Nickel, B. G., Morf, R. H., & Frisch, U., 1983. Small-scale structure of the Taylor–Green vortex. *Journal of Fluid Mechanics*, vol. 130, pp. 411–452.

[18] Nvidia, 2014. NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110/210. Technical report, Nvidia Corporation.

[19] Nvidia, 2016. NVIDIA Tesla P100. Technical report, Nvidia Corporation.