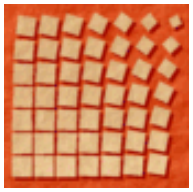



"Politehnica" University of Timişoara
Faculty of Automation and Computers
Departament of Computer and Software Engineering
2, Vasile Pârvan Bv., 300223 – Timişoara, Romania
Tel: +40 256 403261, Fax: +40 256 403214
Web: http://www.cs.upt.ro



FLUID DYNAMICS SIMULATION WITH LATTICE BOLTZMANN MODELS USING CUDA ENABLED GPGPUS

Master Thesis

Adrian HORGA

Supervisors:
Dr. Fiz. Victor SOFONEA
Conf. Dr. Ing. Marius MINEA

Timişoara,
2013

Contents

1 Introduction

The use of graphical processing units for solving non-graphical parallel problems has increased in recent years due to their growing flexibility and high computing power compared to a CPU. The Lattice Boltzmann Models benefit from the increased computing power of such devices and their parallel nature.

In this dissertation we present a solution for implementing a D2Q9 Finite Difference Lattice Boltzmann Model simulation using CUDA C toolkit from NVIDIA. We compare this to an existing multi-CPU implementation developed using PETSc, a scientific toolkit available for C and Fortran programming languages. The results are also compared to similar simulations using CUDA for Lattice Boltzmann Models that use the streaming and collision model.

Our solution starts from the multi-CPU version and transforms the code to run for CUDA. We use CUDA-specific elements for the GPU implementation that on the multi-CPU are already managed by the library functions. So our approach provides functions that are implemented in standard C or CUDA-specific functions.

We first present the theory behind the Lattice Boltzmann Models starting from the simpler streaming collision model and continue to the finite difference one. The formulas needed in the implementation of the model are presented.
We continue by describing the CUDA C toolkit with the hardware model for its General Purpose Graphical Processing Units (GPGPUs) and the programming model for the C language.
The next chapter is a view of the existing CUDA solutions for Lattice Boltzmann Models and their implementations.
After this presentation, chapter 5 contains our proposed solution. We start by presenting the implementation and continue by discussing the CUDA changes and improvements for the implementation.
In chapter 6, we present the results that can be used for scientific purposes (simulation of phase separation in single-component fluids), we compare the running time with the multi-CPU version and with some of the published results from the domain.
The last chapter sums up the dissertation by presenting conclusions and future work ideas.

2 Lattice Boltzmann Theory

2.1 Introduction

A dilute gas system can be described by a distribution function $f^{(N)}(\boldsymbol{x^1}, \boldsymbol{x^2}, \dots, \boldsymbol{x^N}, \boldsymbol{p^1}, \boldsymbol{p^2}, \dots, \boldsymbol{p^N}, t)$ where N represents the number of particles, $\boldsymbol{x^1}, \boldsymbol{x^2}, \dots, \boldsymbol{x^N}$ the position vectors and $\boldsymbol{p^1}, \boldsymbol{p^2}, \dots, \boldsymbol{p^N}$ the momentum of each individual particle at some instant of time [1]. Given the extremely large number of particles (~10^{23}) existing in the system the changes in the system cannot be feasibly deduced by the standard equations used (6N variables). The interest usually falls to the lower order distribution functions (N=1, 2).
Using Statistical Mechanics, a system can be represented statistically by an ensemble of many copies [1] in which $f^{(1)}(\boldsymbol{x}, \boldsymbol{p}, t)$ gives the probability to find a particular molecule with a given position and momentum. Because the experiment cannot distinguish the particles, the distribution functions of the remaining N-1 molecules can be left unspecified. $f^{(1)}$ is the "single particle" distribution function and is the first order distribution function of the described system.
When we know the positions and momenta at a particular time t, $f^{(1)}$ at a future time t+dt can be determined by considering the so-called streaming process. Due to the existence of particles that may arrive from another point than (x,p) we also have collisions. This collisions change the equations of the streaming process [1].

$$f^{(1)}(\boldsymbol{x} + d\boldsymbol{x}, \boldsymbol{p} + d\boldsymbol{p}, t + dt)d\boldsymbol{x}d\boldsymbol{p} = f^{(1)}(\boldsymbol{x}, \boldsymbol{p}, t)d\boldsymbol{x}d\boldsymbol{p} + (\Gamma^{(+)} - \Gamma^{(-)})d\boldsymbol{x}d\boldsymbol{p}dt \qquad (1)$$

Expanding this resulting equation (1) using the Taylor expansion we have the Boltzmann equation:

$$v \cdot \nabla_x f^{(1)} + F \cdot \nabla_p f^{(1)} + \frac{\partial f^{(1)}}{\partial t} = \Gamma^{(+)} + \Gamma^{(-)} \qquad (2)$$

Where $\Gamma^{(-)} dx\, dp\, dt$ is the number of molecules that exit the volume element d**x**d**p** of the phase during the time interval dt, and $\Gamma^{(+)} dx\, dp\, dt$ is the number of particles that enter the same volume element during the time interval dt. The velocity is: $v = \frac{p}{m}$.
According to [1, 4], an approximate solution for this equation can be found using the lattice Boltzmann models.

2.2 Lattice Boltzmann Models (LBM)

The LBM reduce the complexity of the Boltzmann original concept. In these models we have discretized time steps, and only a handful of possible particle positions and momenta. The positions of the particles are confined to the nodes of a two-dimensional (square) or a three-dimensional (cubic) lattice. In a 2D model, the variations of momenta are reduced to a handful of directions (8 in the case of D2Q9 model). The DnQm model , proposed in [3], consists in describing a model in "n" dimensions using "m" discretized velocities.
In figure 1 we have the 9 depicted velocities e_a, where a=0,1,..,8. The velocity e_0=0 is assigned to the particle at rest.

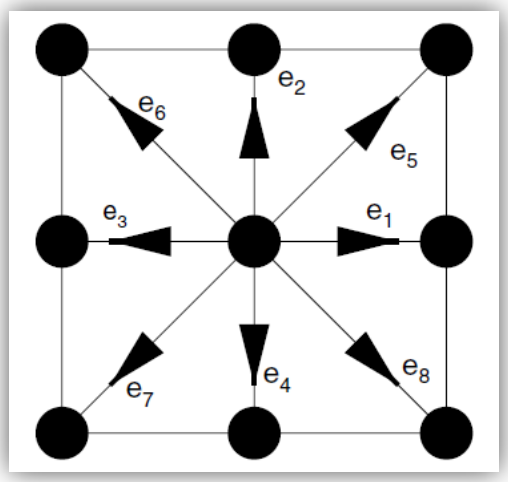


Figure 1. Particle velocities for D2Q9 model [14]

In the LB models the fundamental measure for length is the lattice unit ($lu\ or\ \delta s$) i.e. the distance that separates two adjacent nodes along the Cartesian axes. The time step ($ls\ or\ \delta t$) is considered to be the unit time [1]. In figure 1 the velocities for north, east, south and west have a velocity e_a=1 ($\delta s/\ \delta t$) respectively, for north-east, north-west, south-west and south-east the velocity magnitude is $e_a = \sqrt{2}$ ($\delta s/\ \delta t$). Even if this is the most common used scheme for the 2D LB model, other schemes can be used as well.
By using equation (1) we can further incorporate the single particle distribution function with the property that now it is not a continuous function but a discrete one with m bins (9 in D2Q9 model) that represent the frequency of occurrence. Accordingly, these frequencies can be considered to be "direction-specific fluid densities" [1] and thus the macroscopic fluid density is:

$$\rho = \sum_{a=0}^{m-1} f_a \qquad (3)$$

The macroscopic velocity "u" is the average of the microscopic velocities e_a weighted by the directional densities f_a:

$$u = \frac{1}{\rho} \sum_{a=0}^{m-1} f_a\, e_a \qquad (4)$$

Equations (3, 4) allow us to retrieve the macroscopic quantities (ρ, u) from the discrete microscopic velocities of the LB model.

2.3 Single Relaxation Time Bhatnagar-Gross-Krook

The Bhatnagar-Gross-Krook (BGK) approximation [4, 5] originates from the streaming and collision concept and allows one to simplify the Boltzmann equation (1). According to statistical physics, the interparticle collisions relax the fluid system towards local equilibrium. The distribution function at the next time step t+ δt will be composed of the streaming part and the collision part, as seen in equation (5).

$$f_a(\boldsymbol{x} + e_a\Delta t, t + \Delta t) = f_a(\boldsymbol{x}, t) - \frac{f_a(\boldsymbol{x},t) - f_a^{eq}(\boldsymbol{x},t)}{\tau} \quad (5)$$

The streaming part is represented by $f_a(\boldsymbol{x} + e_a\Delta t, t + \Delta t) = f_a(\boldsymbol{x}, t)$ and the collision by the $\frac{f_a(x,t) - f_a^{eq}(x,t)}{\tau}$ term.

The equilibrium distribution function is defined as in equation (6) according to [1]:

$$f_a^{eq}(\boldsymbol{x}) = w_a\rho(\boldsymbol{x})\left[1 + 3\,\frac{e_a \cdot u}{c^2} + \frac{9}{2}\,\frac{(e_a \cdot u)^2}{c^4} - \frac{3}{2}\,\frac{u^2}{c^2}\right] \quad (6)$$

where the weights are:

$$w_a = \begin{cases} \frac{4}{9} & a = 0 \\ \frac{1}{9} & a = 1,2,3,4 \\ \frac{1}{36} & a = 5,6,7,8 \end{cases} \quad (7)$$

and *c* is the basic speed on the lattice (1 $\delta s/\,\delta t$ in the simplest implementation).
Equations (6, 7) are discussed in [1,4].

2.4 Boundary conditions

The boundary conditions are necessary in order to obtain meaningful results for a simulated model.
The periodic boundary condition is important when wanting to simulate an infinite domain occupied by a multiphase fluid. This periodic condition is one of the simplest and represents the system's edges as connected to the opposite edges.
When simulating a fluid flow in a micro channel we could use the bounceback boundary conditions [1] for the walls and periodic boundaries for the open ends of the channel.
The bounceback condition implies that a node can be considered as a solid and not taken into consideration when applying these conditions.
In [4] there is a survey on various boundary conditions.

2.5 Multiphase LBM

The most important aspect of the LBM is that it can simulate a single- or multicomponent fluids that exhibit phase separation. A component of the fluid refers to a chemical substance (like H_2O) and a single component multiphase fluid would involve multiple phase systems made from a single chemical substance (e.g. liquid and vapor phases of water). A multicomponent fluid could contain for example water and oil. Studies for these have been made in [6, 7].
The "ideal" gas law characterizes the behavior of the gases at low density and is known as the equation on state (EOS). The ideal gas law is:

$$PV = nRT \qquad (8)$$

where P-pressure(atm), V-volume(L), n-number of moles, R-gas constant (0,0821 (L*atm)/(mol*K)) and T-temperature(K).
The van der Waals EOS was developed to account for the behavior of real gases:

$$P = \frac{nRT}{V-nb} - a(\frac{n}{V})^2 \qquad (9)$$

where the second term represents the attractive forces between molecules and the nb term represents the non-negligibile volume of molecules [1].
For liquid-vapor systems, the use of the van der Waals EOS and of the finite difference lattice Boltzmann (FDLB) [9-13] provides a better solution [8] compared to the normal DnQm solution presented above.
The FDLB model starts from the Boltzmann equation but as stated in [8] it has a better numerical stability and is more flexible when simulation multiphase fluids that produce different lattice speeds for different phase masses, compared to the solution presented before.
The new discretization of the Boltzmann equation will be as follows:

$$\delta_t f_a + e_{a\beta}\partial_\beta f_a = \frac{1}{\chi c^2} f_a^{eq}\left(e_{a\beta} - u_\beta\right)F_\beta - \frac{1}{\tau}(f_a - f_a^{eq}) \quad (10)$$

where the constant χ is equal to 1/3 as presented in [8], and the force term F is computed according to:

$$F_\beta = \frac{1}{\rho}\partial_\beta\left(p^i - p^w\right) + k\partial_\beta(\nabla^2\rho) \qquad (11)$$

where p^i is the ideal fluid pressure and p^w is the van der Waals pressure and the parameter k controls the surface tension.
When discretizing equation (9), two finite difference schemes may be used [8]: the first-order upwind scheme and the flux limiter schemes. Even though the computer simulations show that the flux limiter schemes produce more accurate results, the first order upwind scheme is easier to be implemented. For this reason we have chosen, in our work, to implement the last type of scheme given the complicated manner and complexity of operations that are required when using flux limiter schemes.

3 NVIDIA CUDA C

3.1 General Purpose Graphical Processing Units' architecture

Graphical processing units (GPU) are developed for use in problems that can be highly parallelizable. Image processing is one of those problems.
The GPU could be used to solve other problems that can harness their computational power. Their initial usage in image rendering and processing have made them inaccessible to general programmers because they would have to learn image programming languages like openGL to solve problems that were not related to image processing.

NVIDIA has introduced the Compute Unified Device Architecture (CUDA) in 2006 as a general purpose parallel computing platform and programming model that harness the parallel computing power of NVIDIA GPUs and uses them to develop programs that are faster than those on the CPU [5].

The CPU hides memory latency by using large caches. In contrast, the GPU tries to hide the memory latency by high parallelization of the algorithm (running as many threads as possible at once) and by high operation throughput.

Figure 2. shows the architecture of a CPU and an NVIDIA GPGPU:

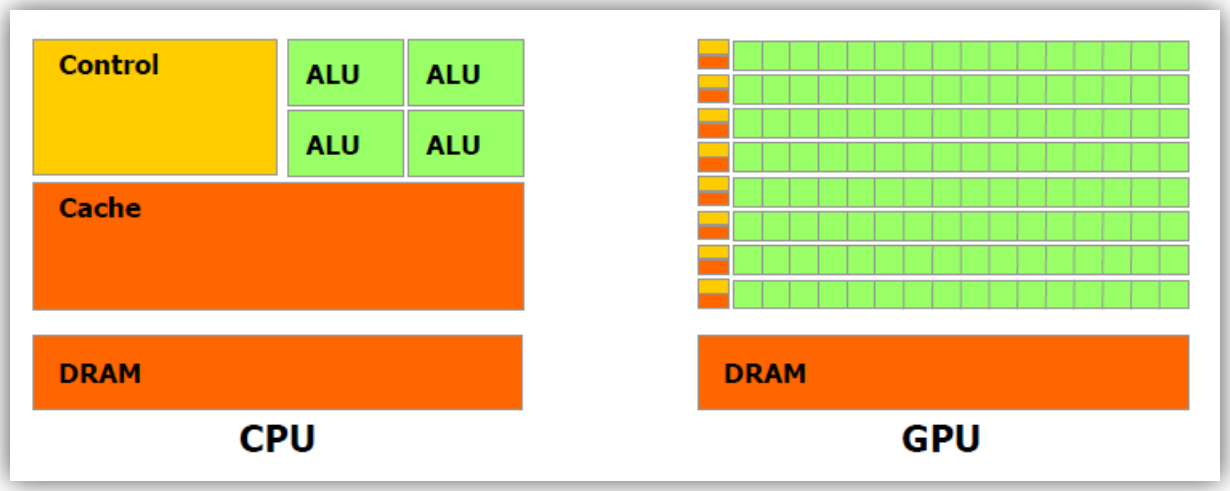


Figure 2. The hardware difference between CPU and GPU [15]

Given that the GPUs are more compute-intensive, the floating point operations per second (FLOPS) are very high compared to even the most powerful CPUs. A comparison between the CPU and the GPU regarding theoretical computational power (measured in GFLOPS or

Giga FLOPS) can be seen in Figure 3. For GPUs the use of double precision halves the performance compared to single precision [15].

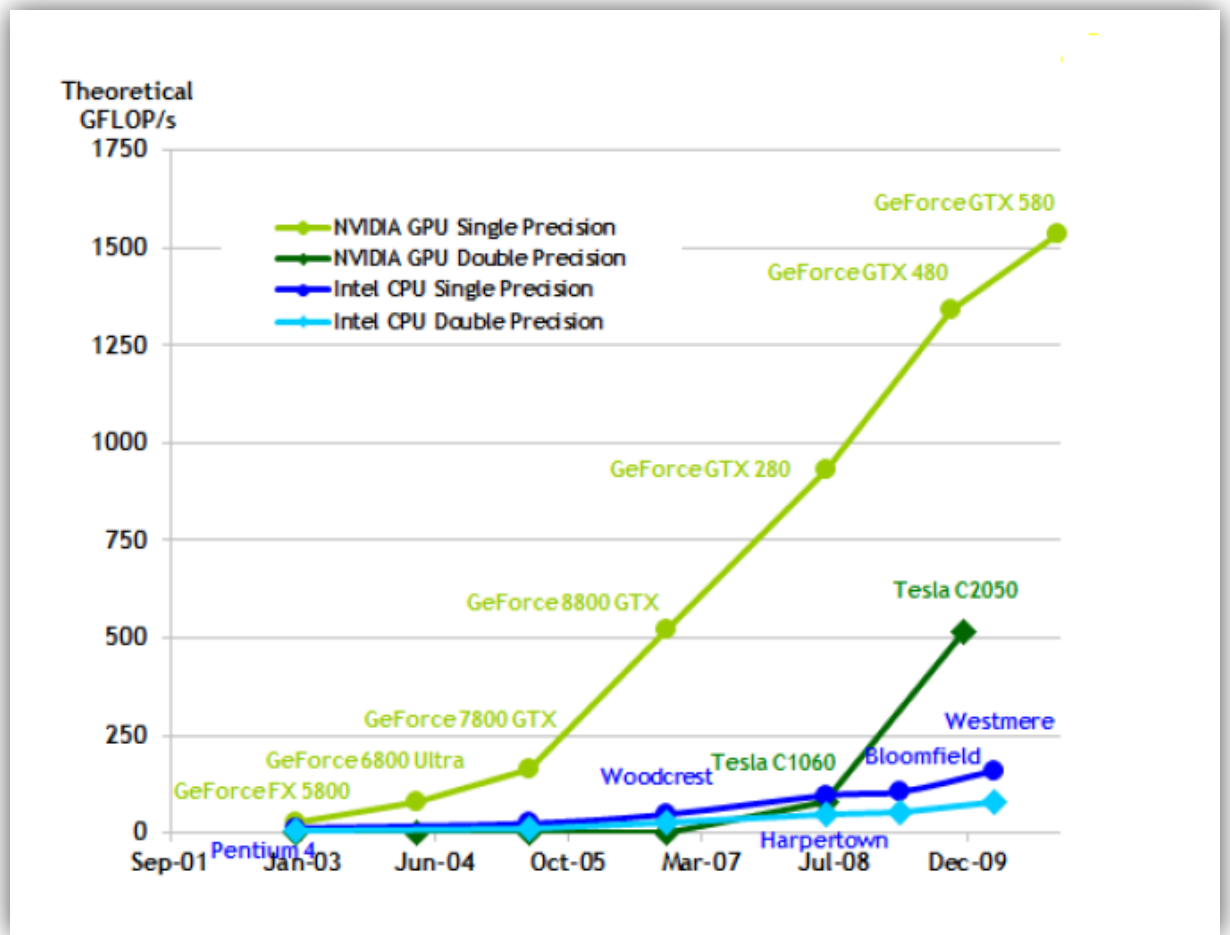


Figure 3. Floating point operations per second for CPU and GPU [15]

3.2 CUDA programming model

The CUDA programming model was developed to be as simple as possible for people with knowledge of standard programming languages like C, C++ or Fortran. It introduces library functions for different operations needed to work with the GPU. It has a special compiler called “nvcc” which is used to compile the files. Files that contain code for the GPU will have the extension “.cu” for C/C++ files and “.cuh” for C/C++ header files.

The important elements of CUDA are the blocks, threads and the memory type. Functions that run on the GPU (device) are started with a number of blocks and a number of threads allocated for each block. The maximum possible numbers of threads per block on devices with CUDA compute capability 2.x is 1024.
The GPU has a number of streaming multiprocessors (SM) on which the blocks run. Multiple blocks can run on a single SM if there are enough resources. If not, after one block finishes

the run, another block is scheduled until no more blocks remain. CUDA is developed as to permit the run of a function with a specified number of blocks and threads on devices with a different number of SMs, as can be seen in Figure 4.

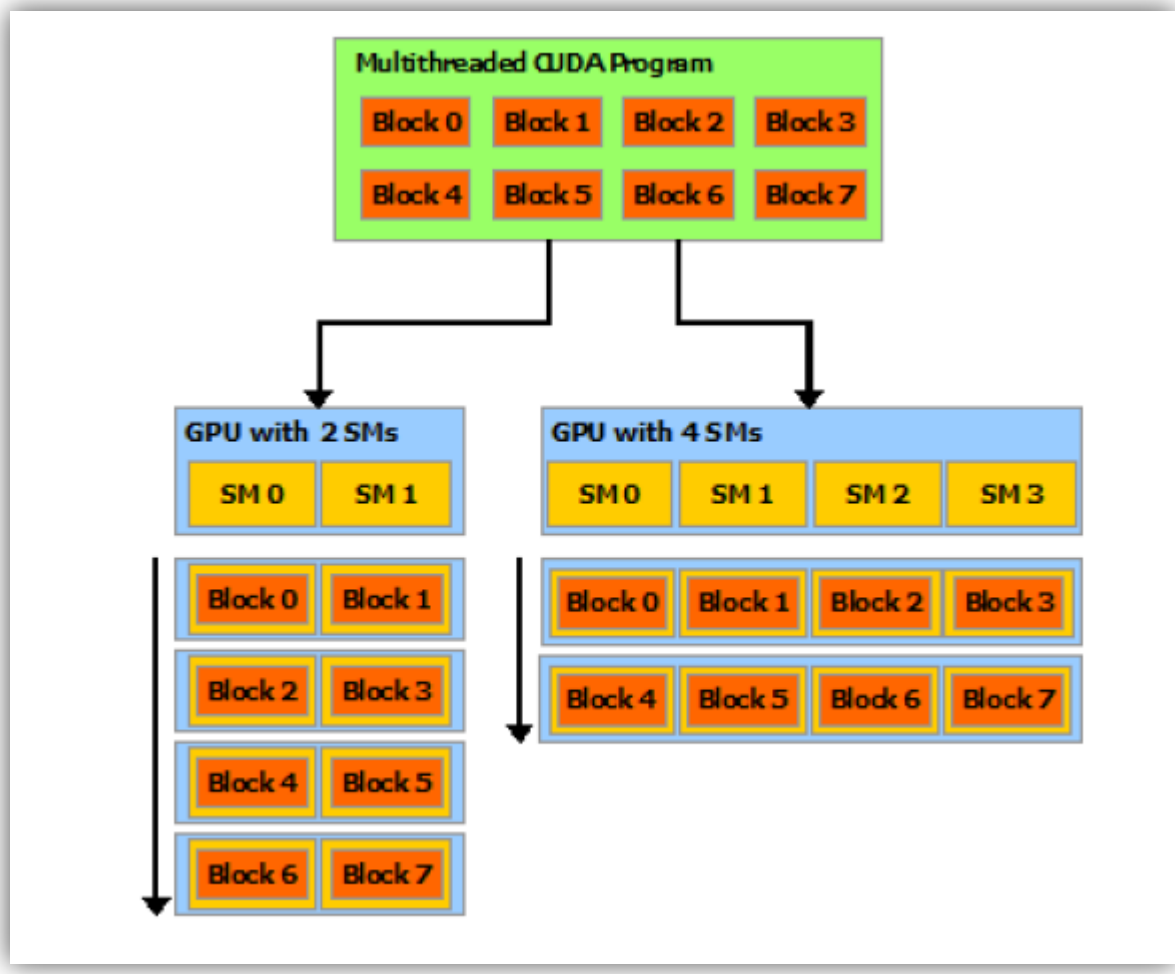


Figure 4. Automatic scalability [15]

Functions that execute on the GPU for multiple threads are called ***kernels*** and in CUDA C, these are C functions that are defined on the CPU and have the **__global__** identifier. The return value of a kernel is of type "void". They are started on the host (CPU) by calling the name of the kernel, followed by the "<<<blocks, threads>>>" and then the arguments of the function. An example of a kernel for adding each element of two vectors and storing the result in a third can be as follows [15]:

```
__global__ void VecAdd(float* A, float* B, float* C)
{
        int i = threadIdx.x;
        C[i] = A[i] + B[i];
}
```

```
int main()
{
        ...
        // Kernel invocation with N threads and 1 block
        VecAdd<<<1, N>>>(A, B, C);
        ...
}
```

Each thread on a block has an unique identifier for the directions x,y,z which specify the position of that thread. The example above calls only the identifier for the x direction because we start a block than has threads only in one dimension. To have threads distributed on multiple axes in the block we would have to define the number N as follows:

```
dim3 N(threadsx, threadsy, threadsz)
```

with the specification that also the product of the three values must not be higher than the value of threads per block allowed by the device's compute capability (1024 for capability 2.x).

Functions that run only on the device and can be called only from code running on the device have the **__device__** function identifier:

```
//example of a device function that computes the global thread index considering
//multiple blocks are started
__device__ int getIndex(){
        return blockIdx.x * blockDim.x + threadIdx.x;
}
```

Each block will run independently of other blocks, there are only synchronization functions available between threads that run on the same block.

The properties of the SMs on the GPUs are that they can run hundreds of threads concurrently. To achieve this they use a specific architecture name SIMT (Single-Instruction Multiple-Threads) [15] with no branch prediction or speculative execution. This means that branching in kernels can lead to an important downgrade in performance.
Each SM creates, schedules and runs threads in groups of 32 (called a *warp*). The instructions are issued in order to a warp. Even though each thread in a warp has its own instruction address stack and variable values, a warp will receive a single instruction at one time, so if there is any divergent branching in a warp, those instructions for each divergent warp will be issued in a serial order to each thread.

Memory types

The threads can have access to multiple types of memory. Some of it is on-chip, for each streaming multiprocessor, but some is global (visible for all threads).

The **global memory** is visible on all the SMs and can be accessed by all threads from all the blocks. This has the lowest bandwidth, and has increased latency (300 cycles).
A special type of global memory is the **constant memory (64 kb)**, this is specified by using the **__constant__** variable qualifier: __constant__ int speed[16]; . This type of memory has an increased bandwidth (10 times faster than global) if the threads from a half-warp (16 threads) access the same element of the memory on one instruction call [15].

Memory transfers from GPU to CPU and back are done via global memory. Constant memory should be statically defined. The global memory is allocated, copied and destroyed on the CPU using the following functions:

cudaMalloc() – allocating a device pointer
cudaMemcpy() – copying to/from device memory
cudaFree() – free the allocated pointer

For constant memory we use cudaMemcpyToSymbol()/cudaMemcpyFromSymbol().

An example for allocating, copying (to device and back) and freeing the memory can be added to the example above for the vector addition.

```
__constant__ float speed[16];
__global__ void VecAdd(float* A, float* B, float* C)
{
        int i = threadIdx.x;
        C[i] = A[i] + B[i] + speed[i % 16];
}

int main()
{
        float hspeed[16];
        //device pointers
        float *A, *B, *C;

        //host pointers
        float *HA, *HB, *HC;

        //allocate host pointers
        HA = (float*)malloc(N * sizeof(float));
        …

        //allocate device pointers
        cudaMalloc((void**)&A, N * sizeof(float));
        ...


        //copy the two host vectors HA, HB to A and B
        cudaMemcpy(A, HA, N * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(B, HB, N * sizeof(float), cudaMemcpyHostToDevice);

        //copy to constant memory
        cudaMemcpyToSymbol(speed , &hspeed, 16 * sizeof(double), 0,
        cudaMemcpyHostToDevice );
```

```
        // Only device pointers can be sent as kernel parameters
        VecAdd<<<1, N>>>(A, B, C);
        ...

        //copy only the result back
        cudaMemcpy(C, HC, N * sizeof(float), cudaMemcpyDeviceToHost):

        //free host pointers
        free(HA);
        …

        //free device pointers
        cudaFree(A);
        …
}
```

In [15] it is stated that the difference between global memory and on-chip memory is that on-chip memory is not visible to threads from another block. The advantage of local memory is that it is very fast compared to global memory but it is in limited resources.

The two important memory types that we are discussing are the shared memory (declared by **__shared__** and visible to each thread in a block) and local memory which is represented by the automatic variables defined in the kernel function.
The shared memory is at least one order of magnitude faster than global memory but is a limited resource (64 kb per SM). Synchronization between threads when using shared memory in order to avoid race conditions is done by the call of **__syncthreads()** which waits for all the threads in the block to reach this point, and then continue.

Each SM has a limited number of registers to use for local memory. If these are all used, then the rest of the variables will be stored in global memory (register spilling) and therefore reduces the speed capability (27 cycles) of this type of memory. So a well-managed memory can avoid the spilling and make us of the speed of local memory.

Memory coalescing is an aspect than need to be taken into consideration when working with global memory. This is related to global memory access patterns. Each thread in a warp must access a memory location within the same 128 bytes as the rest in order for the call to access the memory to be issued in a single transaction. Each misaligned call will increase the number of transactions needed for a warp to read/write the global memory.

4 State of the art

4.1 Previous work

The use of CUDA for developing flow solvers for Lattice Boltzmann Models has increased in recent years. Some of the most important work has been done for the streaming and collision models for LBM.

One of the most cited work in this domain has been that of Jonas Tolke. One relevant article is [16] in which Tolke studies the different approaches for two LBM types of methods: multiple relaxation and BGK for a D2Q9 implementation. In this article he presents a memory access pattern for improving bandwidth. In this article, he uses the solid wall boundary conditions with a simple bounce back rule. This allows him to obtain, with the use of single precision, a performance of over 500 MLUPS (million lattice updates per second) for all of the implementations on a GeForce 8800 Ultra (410 GFLOPS).

Tolke et al. [20] present a D3Q13 implementation in which they use periodic boundary conditions. They conclude that the use of ghost nodes around the domain layer can reduce the divergent branching statements in the CUDA code and improves performance.
In this article the authors also discuss the difference between the use of single and double precision. They believe that, even though the single precision use is less stable, this could be avoided by the careful implementation of the collision operator.

Based on the work of Tolke et al. [20], J. Habich [21] expands the implementation to a D3Q19 model. He developed, based on Tolke's article, the shared memory implementation for the new model. He discusses the periodic boundary conditions for the model, and references [20] the "ghost" nodes idea for divergent branch reduction. He presents the no-slip wall (solid wall) boundary condition which, as for Tolke's solutions, is simpler and proves to add to the performance of the algorithm.
Also, he discusses the improvement of accessing memory by using a **structure of arrays (SoA)**. This is opposite to using an **array of structures (AoS)**. For a given matrix, a SoA means that the probability distribution functions (PDF) for each velocity are stored in a continuous matrix. The structure will contain a list of **m** matrices for each probability. On the opposite side, an array of structures will store for each node a structure with all the elements for a node. Given the way CUDA warps access memory, the SoA approach is needed.
We can see in the Table 1. the difference between the two:

Structure of Arrays (SoA)	Array of Structures (AoS)
struct domain{ double **pdf[m] … }Domain; Domain layout;	struct domain_node{ double pdf[m] … }Domain; Domain **layout;

Table 1. Difference between Structure of Arrays and Array of Structures

J.E. McClure et al. [17] present a solution for a single phase single relaxation BGK model and a multiple relaxation time model (MRT). They implemented a D3Q19 model and tested both solutions using single precision and the streaming collision model.
They have tested both implementations using a NVIDIA Quadro FX 5600 GPU with 345 GFLOPS in single precision. The results have shown that the GPU implementations were faster than a solution than ran on a 3.0 GHz Intel Clovertown CPU.
The results showed that the BGK model is faster than the MRT model because it does not use as much memory and has less complex terms to be computed.

L. Biferale et al. [18] present the solution for a D2Q37 LBM using a streaming collision approach. This solution is done using double precision arithmetic on Tesla C2050 that has 515 GFLOPS in double precision.
Their multi-CPU implementation is done using an AoS on a 3.3 GHz six-core Westmere CPU platform. The algorithm is composed of three steps : stream(), bc() and collide(). The "stream" step gathers the distribution functions that are used on the next steps. This has irregular memory access patterns.
The "bc" step adjusts the values of the top and bottom nodes corresponding to the used boundary condition. They use the periodic boundary conditions for the left and right edges. They use the elements from a 3 lattice distance, so this is done in the bc step by allocating additional storage for the appropriate edges. This is the same as the ghost nodes approach presented in the above articles.
The "collide" step computes the data according to the equations used. This step is the most compute intensive regarding floating point operations as this will calculate data for the next iteration.
The GPU solution uses the SoA approach. The algorithm for the GPU has 4 steps: comm, move, bc, collide. As presented in [18] the 4 steps do the following:

step 1:
exchanges the Y frames through a cudaMemCopy operation;
step 2:
executes the stream kernel for the three topmost and lowermost rows of the grid;
step 3:
adjusts the boundary conditions for the cells located at the top and bottom rows of the grid. It then runs collide for the same cells;
step 4:
executes a kernel that jointly computes stream and collide for the bulk cells of the lattice.

In the results they state that using the shared memory as L1-cache gives a 15% improvement. They have achieved 33% kernel occupancy and showed that a single GPU approach is 2x better than the optimized multi-CPU version but only delivers 30% of the peak performance allowed by the device. The optimized multi-CPU code however reaches 45% of the peak performance.
Even though the GPU code is faster than the GPU, an optimized multi-CPU code has a higher performance regarding peak performance. The problem is that the multi-CPU approach was

very hard to optimize in order to reach the presented results because of the fact that the use of the C programming language does not have a natural tendency toward the parallel development of programs.

The solutions that exist tend to go towards the development of CUDA implementations that use simpler equations and boundary conditions that would not affect the improvements that could be done. The use of single precision improves the performance of the algorithms, but impacts the accuracy and stability of the results.

The streaming collision LBM is mainly used to develop simulations using CUDA. This is much simpler compared to the finite difference LBM which uses more complex equations.
From the results of the articles, when using periodic boundary conditions, the use of ghost nodes seems to improve the performance of the CUDA implementations by reducing the divergent branching statements and improving memory coalescing by allowing the neighbors to be accessed on continuous memory addresses.

We will compare the results from articles [16], [17] and [18] with our solution in the results chapter and we will analyze the reasons for the different performance results.
We have chosen these articles because of the different approaches they employ: different dimensions (2D, 3D), different number of velocities (9, 19, 37), different precisions (single, double), multiple memory use approaches (shared memory, cache).

4.2 Related work

In this part of the chapter we present an overview of the existing solution from which we developed our CUDA implementation. The existing solution [14] was developed in PETSc [19] which is a library for the C programming language that is used for developing of multi-CPU solutions.

The solution is for a D2Q9 finite difference LBM .

The multi-CPU algorithm uses specific PETSc library functions in order to represent a general matrix layout for the fluid. It then partitions the matrix across the X and Y axis (direction can be specified) for the specified number of cores that it is going to run on.
The matrix here is represented as an array of structures, using a three dimensional matrix (double ***global) in which the third dimension represents the values for each term we want to compute (probability distribution functions, temperature, pressure, force).

The matrix is tiles with “ghost” nodes from the start, but these are done by the PETSc function when creating a two dimensional array that will be distributed over multiple processors. The function is DACreate2d, and the management of this global array created is managed through specific PETSc functions.
For a simpler decomposition of the matrix over multiple processors, the implementation is done over only one axis (the vertical one – Y). In the PETSc case the axis are inverted compared to the representation of a matrix in C which goes a[x][y] not a[y][x] like in PETSc.

For each processor there will be a local matrix that is represented only by the elements that will be computed by it. This local matrix is created by calling the two consecutive PETSc functions:

```
DAGlobalToLocalBegin(…);
DAGlobalToLocalEnd(…);
```

These functions create the local matrix for the processor with the specified ghost nodes (on all sides) from the global matrix.

The pseudo code for the algorithm is presented in Algorithm 1.

The specific code for initializing the PETSc environment and several time counters are not presented for the simplicity of the algorithm. We only present and discuss the elements that are relevant in the transition to the CUDA implementation.

Algorithm 1 – PETSc version

```
1.  initialize_global_matrix();
2.  profile();
3.  i=0;
4.  while (i < cycles){
5.          j=0;
6.          while (j < iterations){
7.                  compute_local_quantities();
8.                  lb_time_step();
9.                  j++;
10.         }
11.         profile();
12.         j++;
13. }

14. free_memory();
```

The first line contains the function for generating the data for the 9 probability distribution functions and creating the global matrix that is visible to all processors.

The “profile()” function calculates certain values from the global matrix and writes data used in graphical representation of the results.

At each iteration step the algorithm profiles the result from the global matrix (line 11).
The simulation of the fluid will run for a total number of “cycles * iterations”.

In the “compute_local_quantities()” function at line 7 the data is loaded in each processors’ local matrix with the ghost nodes already existing in each local matrix. The needed data will be computed here using the loaded elements from the local matrix. All the computations will be done locally for each processor.

After each processor has computed the necessary data, it will move to the second function, where “lb_time_step()”, for the next step of the simulated time, in the global matrix the algorithm computes the new values for the probability distribution functions based on the corresponding values from the local matrix. So, for example, element aGlobal[i][j][k] will be computed using the value existing in the element aLocal[i][j][k]. So basically the local matrix works as an auxiliary matrix for the global one.

5 Proposed solution

Our goal is to develop a CUDA implementation using the same formulas like in the PETSc version, transform them for CUDA and measure the speed-up achieved compared to the multi-CPU implementation.

In the next parts of the chapter we will present the changes done for transforming the algorithm to the CUDA version, we will specify different improvements that we have done to the CUDA solution and in the last part we will present the problems that occurred and some observations for the implementation.

5.1 CUDA implementation

In order to transform the CPU based implementation to a GPU based one, we must consider the changes that need to be done.

The first problem that occurs is that on the CPU the memory is visible for all the processors. On the GPU however we must consider that there are two types of memory: one for the host (CPU) and one for the device (GPU). The memory transfers are managed by the CPU.
So, we must allocate the same matrix memory for the GPU as we allocate for the CPU.

The distribution of the computation for the matrix elements was done on the multi-CPU by splitting the global matrix by the number of processors. For each processor is allocated a continuous part of the global matrix. The processor computes the values of the matrix by using local matrices (that are also auxiliaries for the second part of the algorithm). This was done by using an AoS (array of structures).
On the GPU we have blocks but we cannot assign a part of the matrix for each block we run because this will result in noncoalescent memory access patterns by different blocks. We therefore use the SoA (structure of arrays) approach and assign each node of the matrix a thread that calculates the needed data. This is possible because of the way the SMs are structured on the GPUs by allowing millions of threads to be started.
Because we do not have local matrices as for the multi-CPU version to use as auxiliaries, we will allocate a single global auxiliary matrix that is visible from all the threads.

Following the results from Tolke et al. [20] we will also use ghost nodes. The difference between the ghost nodes created for each local matrix in PETSc, we only create the ghost nodes for the global auxiliary matrix. This will reduce the memory usage and also the need to create redundant ghost nodes for each local matrix.

The two functions that compute the data in the two steps presented in chapter 4.2 will now run on the GPU as kernels. The pseudocode of the algorithm in CUDA is presented in Algorithm 2. As with the PETSc presudocode, we kept this at a high level implementation for simplicity of understanding. The compute_local_quantities() function has been renamed to periodic_boundary() because it does no longer compute local data.

Algorithm 1. Pseudocode for the high level implementation

```
1.  generate_data();
2.  copy_to_device();
3.  i = 0;
4.          while(i < cycles)
5.          {
6.          j = 0;
7.          while(j < iterations)
8.          {
9.                  periodic_boundary<<< >>>(…)
10.                 lb_time_step<<< >>>(…);
11.                  j++;
12.         }
13.         copy_to_host();
14.         profile();
15.         i++;
16.         }
17. free_device_memory();
18. free_host_memory();
```

The "generate_data()" function creates the global matrix on the CPU and generates the data for it and for the additional arrays(some of which are have constant values) that will be used during the simulation.
We consider here that the function also allocates a global matrix for the GPU with the same dimensions as for the CPU (dimx * dimy) and another auxiliary matrix that is tilled with the ghost nodes and slightly larger (dimlx * dimly = (dimx + 2 *ghostx) * (dimy + 2 * ghosty)). All other arrays and variables that are used have allocated memory also on the GPU. Because we use a two lattice distance stencil (17 node stencil), we need ghostx=ghosty=2. We also multiply them by two in the calculation of the auxiliary matrix's dimensions because we have ghost nodes on all sides: north, south, east and west.
We call the two GPU matrices dgf_one (global matrix) and dgf_two (auxiliary matrix). It is important to note that the maximum dimension of dgf_one can be defined dgf_one[dimy][dimx], because we use the Y axis as primary axis .

The "copy_to_device()" function will copy all the necessary data from the CPU to the GPU memory. The auxiliary matrix on the GPU has no associated memory on the CPU so at this step it is only created on the GPU.

The "periodic_boundary()" kernel is started with a certain number of blocks and threads per block and will compute the values that are necessary and store them in the auxiliary matrix. The values are computed by using the data from the global matrix on the GPU.
At this step each thread that is on the edge will copy the corresponding value to the appropriate elements on the ghost nodes.
An example is as follows in Table 2. for the auxiliary matrix with 16 elements (from the global matrix) and with two tiles of ghost nodes on each part:

10	**11**	**8**	**9**	**10**	**11**	**8**	**9**
14	**15**	**12**	**13**	**14**	**15**	**12**	**13**
2	**3**	**0**	**1**	**2**	**3**	**0**	**1**
6	**7**	**4**	**5**	**6**	**7**	**4**	**5**
10	**11**	**8**	**9**	**10**	**11**	**8**	**9**
14	**15**	**12**	**13**	**14**	**15**	**12**	**13**
2	**3**	**0**	**1**	**2**	**3**	**0**	**1**
6	**7**	**4**	**5**	**6**	**7**	**4**	**5**

Table 2. Example of creating an auxiliary matrix with 16 elements and 2 lattice distance ghost nodes

We can see in the example how the auxiliary method ghost nodes are created, with the highlighted example that the node zero (green) must be copied to the south ghost node, east ghost node and southeast ghost node.

The "lbm_time_step()" kernel will compute the new values for the global matrix by using the ones from the auxiliary one that were calculated in the previous kernel.
These are similar to the formulas that are used in the PETSc version in the function with the same name. The difference now is that each thread computes a node from the matrix compared to a specified number in the multi-CPU implementation.
An example for this difference is the use of "for" statements in the multi-CPU version, that are used to compare the associated global matrix nodes to the local matrix nodes:
The PETSc version would use:

```
for (y=localystart, localyend)
        for(x=localxstart, localxend)
                for(element=0, elements)
                global[y][x][element] = compute(
                        local[getLocalY(y)][getLocalX(x)][element]
                                );
```

The GPU version would be:

```
                for(element=0, elements)
                 global[getThreadY][getThreadY][element] = compute(
                        local[getThreadY][getThreadY][element]
                                );
```

We specify that this only an example and that there are certain improvements that were set for the GPU implementation in order to simplify and increase performance. But these are presented in the chapter 5.2 .

Another important element at this step is the use of a 17 node stencil. This means that a node will need to use data from 16 neighbors when computing the data for the global matrix, therefore a 2 lattice distance. This is the reason for using two tile ghost nodes on each side.
In the next example we show the neighbors that are used for a node. The node is notes with 0 and highlighted with red and the unused nodes with X . L. Biferale et al. [18] use 3 lattice

distance which, as shown in the results, greatly impacts performance. The example of neighbors is presented in Figure 6.

14	X	**10**	X	**13**
X	**6**	**2**	**5**	X
11	**3**	0	**1**	**9**
X	**7**	**4**	**8**	X
15	X	**12**	X	**16**

Figure 5. Neighbors for a 17 stencil approach

The "copy_to_host()" function copies the results from the GPU global matrix to the CPU global matrix. These will be used to compute statistics by the "profile()" function at line 14.

We have the two "while" statements at lines 4 and 7 because the first one determines how many profiles we want to make in the simulation and the second one determines how many steps we want to take between each profiling session.

The last steps will clean the device memory (line 17) and the global memory (line 18).

The two kernels have the following definitions:

```
__global__ void periodic_boundary(double *dgf_one, double *dgf_two,
                                  double *cspeed_x, double *cspeed_y)

__global__ void lb_time_step(double *dgf_one, double *dgf_two,
                             double *cspeed_x,  double *cspeed_y, double *cweight)
```

All of the function's parameters are allocated on the host using the "cudaMalloc()" function. The pointers can be passed on the host as well but cannot be used to read the values, because the data is on the device. We explain the reason for using a one dimensional array for the global matrix and the auxiliary one in chapter 5.2. The other arrays are simply one dimensional with the size of the number of velocities used (9 in our case).
The two kernels are called using the following code:

```
periodic_boundary<<<blocks, threads>>>(dgf_one, dgf_two, cspeed_x, cspeed_y);

lb_time_step<<<blocks, threads>>>(dgf_one, dgf_two, cspeed_x, cspeed_y, cweight);
```

5.2 Implementation improvements

In this chapter we explain the improvements we used for the CUDA implementation.
Some are based on the indications from [15], some are from the published solutions presented before. Not all the presented improvements could be used in our solution because of the specific problem and equations we used for the LBM.
We discuss these in this chapter along with the explanation of why we could or could no use them.

As specified before, the use of single precision computations on the GPU is twice as fast as the use of double precision. Even though we wanted a very fast implementation, because we wanted similar results to the PETSc version, we had to use the double precision.

The ghost nodes approach was presented in chapter "4.1 Previous work" as well as in chapter "5.1 CUDA implementation" . This improves the solution because it reduces the divergent branching that results from the use of periodic boundary conditions on edge nodes. This also improves the memory access patterns because each thread in a warp will access a continuous memory location. If we would not use the ghost nodes, the threads for the edge nodes would have had to access the memory from the other side of the matrix.

Because the global memory access for an element is around 300 cycles compared to 27-30 cycles on registers we load the memory we use more than once in registers and reuse them. For example if we have:

```
dgf_two[ lindex ]     = dgf_one[ index ] + dgf_one[ index  - 1];
dgf_two[ lindex + 1]  = dgf_one[ index ] + dgf_one[ index  - 2];
```

we improve the speed by storing dgf_one[index] in a register:

```
double reg = dgf_one[ index ];
dgf_two[ lindex ]     = reg + dgf_one[ index  - 1];
dgf_two[ lindex + 1]  = reg + dgf_one[ index  - 2];
```

Another idea for improving the run time is by storing and computing the intermediate values in registers and only after that storing them in the respective global memory.
This is done because there are complex formulas that compute, use and store different elements from a node (pressure, density, temperature etc.).
The problem with the use of many registers is that each SM has only a limited amount of them and has a limited number of registers per kernel (64 in compute capability 2.x) which will result in register spilling to global memory (the excess registers will be stored in global memory) which will negate the improvement effects specified before.
A solution for this is to reduce the number of register usage. The fewer registers we use per kernel, the more warps could run at the same time on the GPU (increased occupancy).
We do this by reusing certain registers. This unfortunately results in a code that is optimized but is very hard to understand and reuse in further more complex solutions.
We also recalculate the position indexes for the two matrices (gpos – for the global matrix, lpos – for the auxiliary matrix) whenever they are used.

In order to improve memory coalescing when calling global memory elements, we use the SoA (structure of arrays) to store the matrix, but we do this in a one dimensional array where all the matrices (matrix for each element of the node) are stored in a continuous manner. This results in having two arrays corresponding to the global and auxiliary matrix (*dgf_one and

*dgf_two) for which we have to compute the two indices (gpos and lpos) in order to determine which node corresponds to the thread that is running. The two position indexes are calculated based on the global thread index for each node.
The problem here is that having a 17 node stencil, not all the memory calls can be coalescent. This is true mostly for the calls to the north and south neighbors (ne, nw, se,sw calls are also noncoalescent calls). But this is to be expected, as other published results stated.

Even though we specified that each thread could compute a node, this is not a good approach, because of the availability of registers, the number of possible threads that can run on a GPU (sustained by the SMs). So, in order to further reuse the registers and increase the occupancy we reuse the created threads to compute other nodes when finishing with a certain one. This allows us to create a limited amount of threads and blocks and reuse them. This is preferable to creating threads for all the nodes of the matrix (1024 * 1024).

Using shared memory as proposed by Tolke [16] is not viable in our implementation because of the use of the 17 node stencil and the large number of nodes that need to be computed. The shared memory is a limited resource for each SM (64 kb) which is too few to load all the needed elements for computing purposes. This would reduce the occupancy on the GPU to a point that only a few threads (maybe 32) could run and use the shared memory for each SM. This would not be a viable solution.

Another improvement we have used is that of the constant memory. Because we have arrays or coefficients that are not modified during the kernel's run, we store them in constant memory. Even though this is also limited to 64kb per SM, it is sufficient to store all the arrays and coefficients we use in the formulas.

We have developed one version (V1) with the improvements specified above, and a second one (V2) in which we reuse the registers in an even more aggressive manner and recalculate indexes and certain sums. In the Results chapter there is a running time comparison between the two versions.

5.3 CUDA-specific problems for the implementation

The implementation of the CUDA starting from the PETSc version has shown a few specific problems that a developer must take into consideration when implementing such a program.
One of the problems is the different approach to store the data which in a typical CPU program is done using AoS (array of structures). In contrast, the GPU solution needs to have the SoA (structure of arrays) approach in order to provide the indicated coalescent memory access patterns.

Regarding the coalescent memory access pattern, the LBMs have the problem that using neighbors from distant lattices will impact the performance by addressing sparse elements and not continuous ones.

A typical CUDA problem is that new versions introduce new ways to access and modify the memory, to run threads in a warp etc. This results in new ways to improve the solution when switching to another version of CUDA. Even between GPUs the different architectures (Fermi, Kepler etc.) provide different amounts of memory available.

6 Results

Profiling results

In order to have a similar results regarding precision compared to the PETSc[14] version, we must first study if the multiphase fluid we simulate shows a separation after a certain number of iterations. We initialize the fluid in order to have only minor density variations and set the velocities values and the other parameters. Depending of the starting parameters the fluid would show a phase separation after a few number of iterations (a few thousand) or a large number of iterations (hundreds of thousands).

Having the initial parameters as follows:

delta_time	= 0.0001
tau	= 1.000e-03
rho_zero	= 1.000e+00
kappa	= 0.0001
RTzero	= 1.0
RRTzero	= 0.9
temperature	= 0.9

we profile the results of a simulation for a 256x256 matrix for iterations 0, 5000, 10000, 15000 and 20000 and write them in an image file in order to see the results.

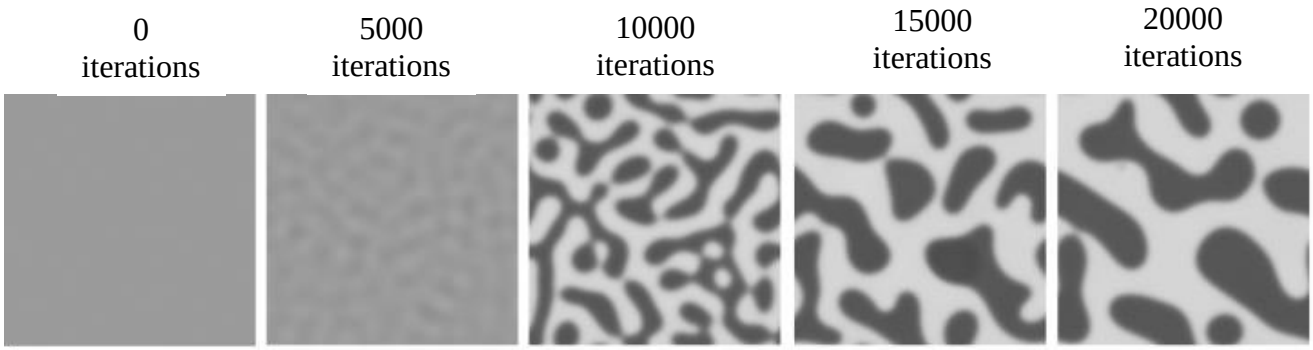


Figure 6. Liquid(black) - vapor(white) phase separation within a 256x256 matrix

As we can see from the results in the initial state there are only small density variations. As the time (i.e., the number ofiterations) increases, these variations intensify start to increase and is shows phase separation. The sum of all probability distribution functions, performed over the nodes of the lattice gives total number of particles in the system. This quantity is quite well conserved during our simulations (up to the 6th digit of the sum, after 5000 simulations)

Performance results and analysis

We have tested the two versions of the algorithm using two different devices: a laptop GPU – NVIDIA GTX460M (168 GLOPS double precision) and a powerful scientific GPU – NVIDIA Tesla M2090 (665 GFLOPS double precision).

As we can see, the M2090 has 4 times more computing power. CUDA allows scalability when increasing the resources (more Streaming Multiprocessors with more computing power) but a poorly implemented parallel algorithm will not make use of this performance increase.
The result of running a simulation for a 256x256 matrix shown in Figure 7 shows us that the performance ratio (4:1) is maintained when using the more powerful device.

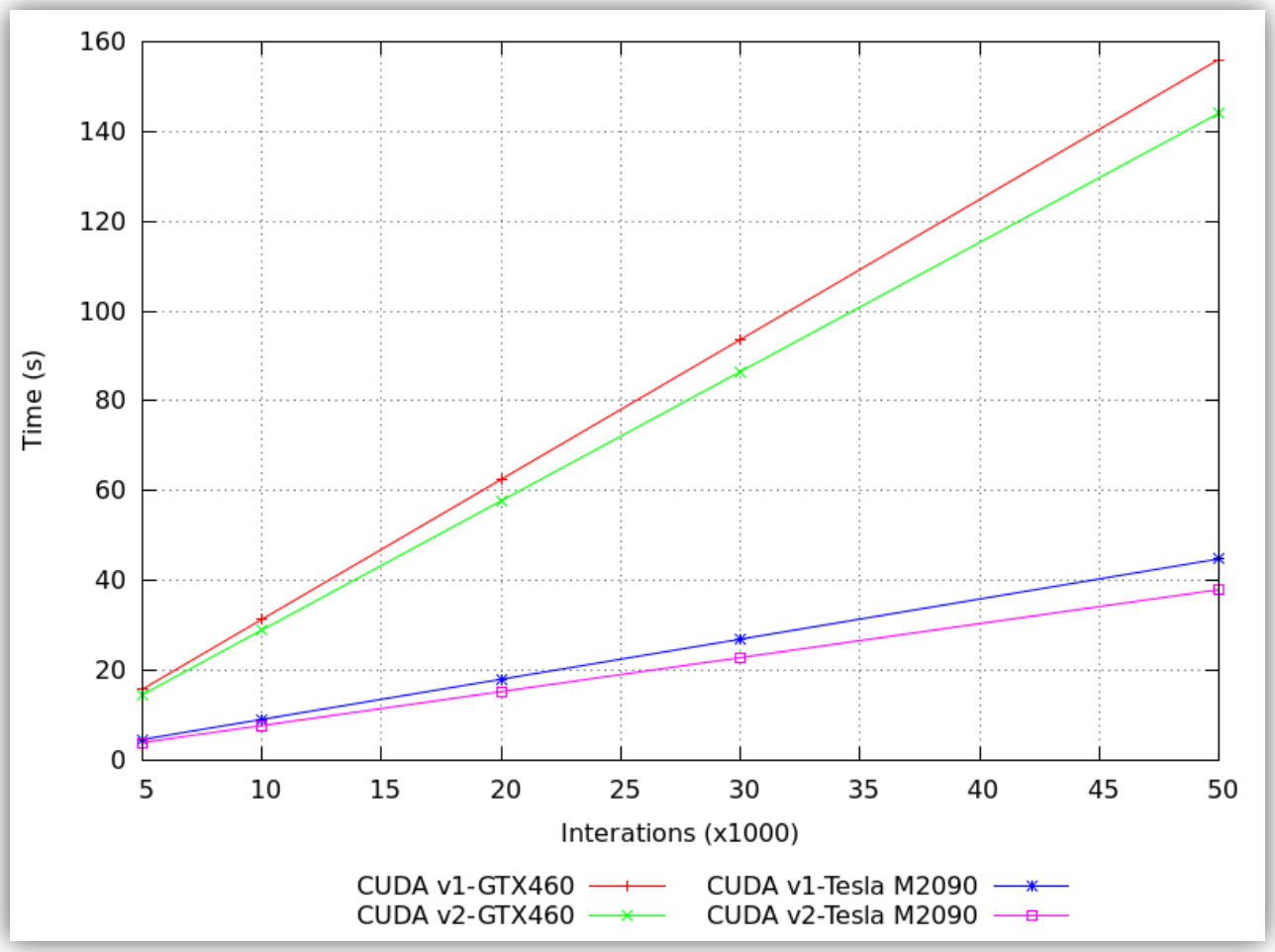


Figure 7. Runtime for V1 and V2 on GTX460 and Tesla M2090

We also must compare the running time results with the PETSc version in order to see the performance increase when compared to a multi-CPU implementation.

We have used 16 cores of the BlueGene[1] for testing the running time for a 256x256 matrix. The results can be seen in Table 3.

[1] Supercomputer at "West" University in Timișoara, http://hpc.uvt.ro/infrastructure/bluegenep/

256x256	Time (s)				
Iterations	PETSc – 16 cores	CUDA v1 – GTX460	CUDA v2 – GTX460	CUDA v1 – Tesla M2090	CUDA v2 – Tesla M2090
5000	199.092	15.653	14.461	4.47	3.789
10000	397.398	31.249	28.869	8.944	7.574
20000	793.755	62.419	57.659	17.878	15.145
30000	1189.43	93.592	86.453	26.81	22.718
50000	1982.489	156.039	144.06	44.722	37.861

Table 3. Results for a 256x256 matrix

For a 512x512 we have increased the number of cores used on the BlueGene by 8 times, giving 128 cores. The running time results can be seen in Table 4.

512x512	Time (sec)				
Iterations	PETSc – 128 cores	CUDA v1 – GTX460	CUDA v2 – GTX460	CUDA v1 – Tesla M2090	CUDA v2 – Tesla M2090
5000	203.573	68.065	58.323	17.795	15.664
10000	417.739	136.2	116.662	35.579	31.319
20000	828.849	272.329	233.188	71.134	62.618
30000	1249.349	408.333	349.978	106.701	93.911
50000	2062.281	680.45	583.007	177.849	156.516

Table 4. Results for a 512x512 matrix

As we can see from the two tables, the CUDA versions are faster than the multi-CPU versions. Even with 128 cores, the PETSc version is 13 times slower than the version that runs on the Tesla GPU. Even the laptop GPU performs 4 times faster than the multi-CPU implementation.
Even more, increasing the number of cores by 8x has only improved performance by 4x. This is mainly due to the fact that the cores need to communicate between the two steps in order to have the updated results. This is very time consuming. The CUDA version does not need to communicate the results between threads as it uses a global memory in which every thread can see the updated values after the synchronization (between the two kernels).

The second version (V2) is at least 8-10% faster than the first version (V1) but due to the aggressive reuse of registers the code has lost much of its meaning, in the sense that it is very hard to debug or even use other more complex functions in place of those already existing. This is important to consider as there are many other functions that could be used to compute the term for each lattice, and we must choose whether the 10% speed improvement is worth to be compared against the capability of easily extending the algorithm's solutions in the future

Comparison to published results

Our results must be compared to those presented in the State of the art chapter in order to see the impact that using finite difference Lattice Boltzmann Models has on the performance of the algorithm.

We have chosen three published results that we want to compare with: J. Tolke [16], J. E. McClure et al. [17], L. Biferale et al. [18].

In each of the published results, the performance of the algorithms is compared in million lattice updates per second (MLUPS). This metric is calculated by determining how may lattice nodes are updated (e.g. 256x256) during the entire simulation run (e.g. 20000 iterations) and dividing the result by the running time.

In Table 5 we have the comparison of our result with the other 3. Given the fact that each solution was run on a different device we have placed the computing performance in GFLOPS of device in single or double precision depending on the one used in the solution. We have also calculated the ration of GFLOPS needed to compute the MLUPS for each implementation.

	Device name	Device theoretical GFLOPS	Precision	LBM DnQm	MLUPS achieved	MLUPS / GFLOPS
Our solution	**Tesla M2090**	**665**	**Double precision**	**D2Q9 – FDLBM**	**84**	**0.126**
J. Tolke	GeForce 8800 Ultra	410	Single precision	D2Q9 – SCLBM	568	1.385
J. E. McClure et al.	Quadro FX 5600	345	Single precision	D3Q19 – SCLBM	250	0.724
L. Biferale et al.	Tesla C2050	515	Double precision	D2Q37 - SCLBM	20	0.038

Table 5. Performance comparison to published work

Compared to J. Tolke's results, we have an order of magnitude less performance than his results. We can see that a similar model (D2Q9) that uses a finite difference LBM approach (FDLBM) can severely impact performance compared to a simpler streaming collision LBM (SCLBM) implementation.

Our results show 6x times less performance compared to the results from J.E. McClure et al. Being a three dimensional solution, this is half as fast as the solution from J. Tolke. This could be an indicative for a future three dimensional solution of FDLBM, that the performance should decrease by half.

Compared to our solution, the last result from L. Biferale et al. for a two dimensional solution with 4 times as many probability distribution functions to compute shows we achieved 4 times more performance. This also uses double precision as our solution compared to the single precision solutions presented before.
This result shows that the use of double precision, even it increases accuracy of the simulation, proves to impact the performance in a drastic manner

The performance decrease in our solution implies that new memory access patterns should be studied compared to those already used. The FDLBM solution greatly affects the performance by adding more complex terms that need to be calculated compared to the SCLBM solutions.

Another important factor for these types of models is the number of velocities used. Increasing the number increases the model's accuracy, but as shown by the results of L. Biferale et al. this greatly diminishes the performance of the algorithm and also increases the memory needed to store the results. This means that only small matrices can be simulated, because the memory is needed for computing each node's terms. Taken this into consideration, if we want to expand our solution to a variable number of velocities, we would have to take into consideration the limitations of a single-GPU implementation (like limited memory) and consider a scalable multi-GPU approach.

7 Conclusions and future work

Conclusions

We have developed a solution for simulating fluids using the finite difference Lattice Boltzmann Models. These types of models are more complex than those used in other published implementations of CUDA LBM. We have proven that a GPU implementation can compare and be faster than a multi-CPU implementation of the same algorithm using fewer resources and at a lower cost.
Even though our solution is not as fast as other published solutions, the usage of double precision, complex formulas and higher accuracy provide the reasons that other implementations are faster.
Furthermore, our implementation has proven that the algorithm has been correctly parallelized and the usage of a more powerful GPU will preserve the ratio when increasing the GFLOPS of the device used.

The use of ghost nodes has proven to increase the memory coalescing when we uses periodic boundary conditions are used.
The continuous improvements and different approaches that come with each CUDA version and each compute capability of new devices affects the choices for developing a program using CUDA. The results we have achieved have shown us that a portable code may not be optimized for different versions. On the other hand, an optimized code is portable for newer versions but may not be optimized because newer versions could change the way the algorithms must optimized regarding memory accessing patterns, register availability or new types of memory.

Future work

The next steps in improving the current algorithm would be to research different memory access patterns that would reduce the latency even more.
Another important step in the future would be to develop a scalable multi-GPU implementation. This is crucial in allowing us to work with matrices of greater dimensions as the single-GPU solutions are limited by the resources of the used devices. The scalable multi-GPU implementation would be very useful for the developing of a three dimensional model which requires an even larger amount of memory than the two dimensional models.

References

[1] M. C. Sukop, D.T. Thorne Jr, "***Lattice Boltzmann Modeling*** *An Introduction for Geoscientists and Engineers",* Springer-Verlang Berlin Heidelberg, 2006

[2] S. Harris, "An introduction to the theory of the Boltzmann equation", Holt, Rienhart and Winston, Inc., New York, 1971

[3] Y.H. Qian, S. Succi, S.A. Orszag, "Recent advances in lattice Boltzmann computing", Ann Rev Comp Phys 30:195-242, 1992

[4] S. Succi, "The Lattice Boltzmann Equation for Fluid Dynamics and Beyond", Oxford University Press, ISBN 0-19-850398-9, 2001

[5] P.L. Bhatnagar, E.P. Gross, M. Krook (1954), "A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems", Physical Review 94 (3): 511–525. Bibcode 1954PhRv...94..511B. DOI:10.1103/PhysRev.94.511.

[6] J.J. Buckles, R.D. Hazlett, S. Chen, K.G. Eggert, D.W. Grunau, W.E. Soll, "Toward improved prediction of reservoir flow performance", Los Alamos Science 22:112-121, 1994

[7] K. Langaas, P. Papazacos, "Numerical investigations of the steady state relative permeability of a simplified porous medium", Transport in Porous Media 45:241-266, 2001

[8] V. Sofonea, A. Lamura, G. Gonnela, A. Cristea, "Finite-difference lattice Boltzmann model with flux limiters for liquid-vapor systems", DOI: 10.1103/PhysRevE.70.046702, 2004

[9] N. Cao, S. Chen, S. Jin, and D. Martinez, "Physical symmetry and lattice symmetry in the lattice Boltzmann method", Phys. Rev. E 55,R21, 1997

[10] R. Mei and W. Shyy, "On the finite difference-based lattice Boltzmann method in curvilinear coordinates", J. Comput. Phys. 143, 426, 1998

[11] T. Seta, K. Kono, D. Martinez, and S. Chen, ""Lattice Boltzmann Scheme for Simulating Two-Phase Flows", JSME Int. J., Ser.,B 43, 305, 2000

[12] T. H. Lee and C. L. Lin, "A characteristic Galerkin method for discrete Boltzmann equation", J. Comput. Phys. 171, 336, 2001

[13] V. Sofonea and R. F. Sekerka, "Viscosity of finite difference Lattice Boltzmann models", J. Comput. Phys. 184, 422, 2003

[14] V. Sofonea, R. F. Sekerka, "Boundary conditions for the upwind finite difference Lattice Boltzmann model: Evidence of slip velocity in micro-channel flow", Journal of Computational Physics 207, 639–659, 2005

[15] NVIDIA. NVIDIA CUDA C Programming Guide, v. 5.0, 2012

[16] J. Tolke. “Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by NVIDIA”. Comput. Vis. Sci., 13(1):29–39, 2009.

[17] J. E. McClure, J. F. Prins, C. T. Miller, “*Comparison of CPU and GPU* 9 *Implementations of the Lattice Boltzmann Method”*, XVIII International Conference on Water Resources, CMWR 2010, Barcelona

[18] L. Biferale, F. Mantovani, M. Pivanti, F. Pozzati, M. Sbragaglia, A. Scagliarini, S. F. Schifanoc, F. Toschi, R. Tripiccione, “*An optimized D2Q37 Lattice Boltzmann code on GP-GPUs”, http://dx.doi.org/10.1016/j.compfluid.2012.06.003*

[19] PETSc documentation, www.mcs.anl.gov/petsc/documentation

[20] J. Tolke and M. Krafczyk, “Towards three-dimensional teraop CFD computing on a desktop pc using graphics hardware”, In Proceedings of International Conference for Mesoscopic Methods in Engineering and Science ICMMES07, Munich, 2007

[21] Johannes Habich, “Performance Evaluation of Numeric Compute Kernels on nVIDIA GPUs”, Master’s thesis, Friedrich-Alexander-Universitat Erlangen-Nurnberg, 2008