ISSN 1870-4069

Particle-based Simulations of Liquid Crystals Supported by GPU Parallelization in CUDA

Jorge Fierro[1], Humberto Híjar[1,2]

[1] Universidad La Salle México, Centro de Investigación, Ciudad de México, Mexico

[2] Universidad Nacional Autónoma de México, Facultad de Ciencias, Investigación Científica, Ciudad de México, Mexico

jorge.fierro@lasalle.mx, humberto.hijar@lasalle.mx

Abstract. Liquid crystals are fluids that show certain amount of order in the orientation and position of their molecules in contrast with simple fluids where both types of order are absent. They have been subject of numerous studies due to their technological relevance. In this research work, it is proposed a method for simulating the liquid crystal phase with the simplest symmetry, known as the nematic phase. The method is based on particles that interact in independent sets, which allows to propose programming it in parallel. This is done in Graphic Processing Units (GPUs) on NVIDIA's CUDA architecture. It is shown that the method allows to simulate the appearance of molecular order on reproducible conditions. It is also clearly exhibited that the parallel procedure has a much higher performance than that given by the serial version of the same simulation algorithm.

Keywords: Liquid crystal, simulation, GPU parallelization.

1 Introduction

Introductory physics describes three states of matter: solid, liquid, and gas, which are differentiated by the amount of order shown by their molecules [5]. Usually, materials transition between the solid and liquid states without an intermediate stage. However, in the late 19th century, Freiderich Reinitzer discovered that intermediate phases can exist between these two states [4]. A few years later, Otto Lehmann named these phases as we know them today: liquid crystals [4].

The applications of liquid crystals include displays for televisions, cell phones, and computers known as LCDs (Liquid Crystal Displays); tunable wavelength filters; resonant cavities for tunable lasers; thermometers, and smart windows [15]. Recent research suggests their use in detecting pathogens, antigens, cancerous tumors [21, 22], as well as in controlling the trajectory of microorganisms [16].

A vast variety of liquid crystals is known. All of them are formed by molecules whose symmetry is not spherical, e.g., elongated rod-shaped molecules as those

of N–(4–Methoxybenzylidene)–4–butylaniline (commonly referred to as MBBA) illustrated in Figure 1 (a), or 4–Cyano–4′–pentylbiphenyl (customarily known as 5CB) illustrated in Figure 1 (b). When atomistic details are not relevant, these molecules can be modeled as rigid rods as those illustrated in Figure 2.

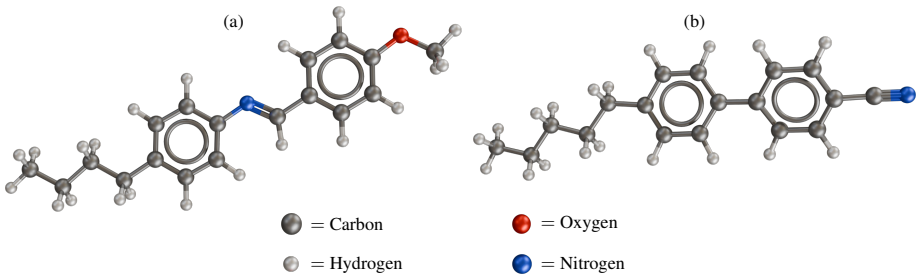

Fig. 1. Two elongated molecules that form liquid crystal phases: MBBA (a) and 5CB (b). Their constituents atoms can be identified by the levels at the bottom. The central part of both molecules contains benzene rings (flat hexagons with six carbon atoms) that give them a rather rigid structure.

The liquid crystal phase with the simplest symmetry is known as the nematic phase. The molecular arrangement in a nematic liquid crystal (NLC) is schematically illustrated in Figure 2, where it is compared against that of the crystal and isotropic liquid phases. In a crystal, molecules are perfectly positioned at the nodes a periodic lattice and all of them point along the same direction. In the completely opposite case, corresponding the a simple or isotropic liquid, molecules move arbitrarily through the sample and they point in every direction with the same probability. In a NLC, the centers of mass of the molecules move arbitrarily, as in a simple liquid, but the molecular axes remain oriented around a common axis known as the *director*, represented by a unit vector, $\hat{\mathbf{n}}$, [11]. Thus, NLCs are states of matter with an intermediate order between that of crystal a that of usual liquid. The reader is referred to reference [9] (in Spanish) where concepts and mathematical aspects of liquid crystal phases are discussed in an introductory manner.

For the previously mentioned reasons, liquid crystals are of great interest in applied sciences and materials engineering, where computational simulations have played a crucial role in understanding their properties due to their ability to test a wide range of system's parameters and to handle conditions that are hard to achieve experimentally [1]. Recently, an algorithm known as Nematic Multiparticle Collision Dynamics (N-MPCD) has been proposed, which describes the NLC as a system of particles that carry an orientation vector [19]. Periodically, the particles are allowed to interact with those in their vicinity. To do this, the simulation space is subdivided into contiguous cubic cells within which independent operations are performed, suggesting that the method could be parallelized. Two alternative variants of N-MPCD are known. One is due to Shendruk and

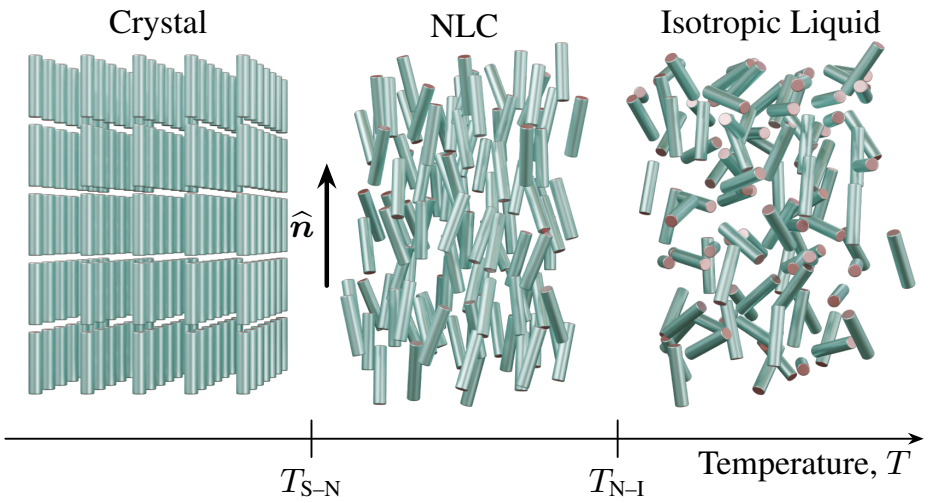

Fig. 2. Schematic of the molecular order in three phases of matter: crystal (left), NLC (center), and isotropic liquid (right). These phase occur as function of temperature where T_{S-N} and T_{N-I} indicate transitions from solid to nematic and from nematic to isotropic liquid, respectively. Vector $\hat{\mathbf{n}}$ represents the average molecular orientation in the liquid crystal phase.

Yeomans [19] and is based on a collision operator that promotes reorientation of particles dictated by a local mean-field potential to achieve nematic order. The other one, due to the Mazza and coworkers [14], simulates nematic order by incorporating explicit hydrodynamic equations of liquid crystals.

N-MPCD is an extension of a simple fluid simulation method known as Multi-particle Collision Dynamics (MPCD), for which various algorithms that operate in parallel have been proposed. One of the first was developed by Petersen et al., who adapted the method to run on multiple processors of a Cray XT3 computer [17]. Westphal et al. have developed an MPCD implementation based on graphics processing units (GPUs), achieving a performance gain of up to two orders of magnitude compared to a comparable version on central processing units (CPUs) [23]. Howard et al. have presented an open-source implementation of the algorithm that scales to run on hundreds of GPUs [13, 12]. Halver et al. have used heterogeneous GPU nodes to parallelize MPCD in an implementation based on Cabana [7]. More recently, Ratan has created parallelized simulations based on an hybrid Molecular Dynamics–MPCD scheme to investigate the behavior of active matter systems [18].

The aim of the present work is to parallelize the N-MPCD method, in the Shendruk and Yeomans version, taking advantage of the fact that it works with quantities representing particles grouped in discrete and independent spaces. The problem to be solved consists of performing operations on the properties of the particles that make up the system simultaneously and independently when nec-

essary, as well as performing parallel convergent operations that require particles grouped in the system's cubic cells. All of this must be done while respecting the physical and mathematical rules that produce nematic behavior in the system. The goal is for this simulation to run on GPUs. Additionally, the performance of this parallel implementation is expected to surpass that of a previously developed serial version [8, 10].

One of the main challenges in developing the GPU-parallelized N-MPCD method was that many processing threads needed to write to the same memory section simultaneously. This problem was solved by using a processing thread for each collision cell, dedicated exclusively to averaging the properties of the particles contained within it. This implementation improved computation time by an order of magnitude compared to the serial version.

The content of this article is as follows. In Section 2, the basic characteristics of the N-MPCD algorithm will be described. Subsequently, in Section 3, the parallelization of the method on GPUs will be discussed. In Section 4, the main results of our research will be presented, and in Section 5, conclusions will be synthesized and possible future work will be proposed.

2 Simulation Method

The simulation system consists of point particles that move within a cubic box with side length L, which is considered an integer multiple of the unit length a. All particles have the same mass m. Their positions and velocities are represented by the vectors $\mathbf{r}_i$ and $\mathbf{v}_i$, with $i = 1, 2, \ldots, N$. Each particle has an associated unit orientation vector, $\hat{\mathbf{u}}_i$, which will serve to generate the characteristic orientation order of NLCs. The vectors $\mathbf{r}_i$, $\mathbf{v}_i$, and $\hat{\mathbf{u}}_i$ are considered continuous functions of time, t, and will be updated to generate system's dynamics. The algorithm responsible for this consists of two steps known as the propagation step and the collision step. Both will be described below.

2.1 Propagation Step

Particles move in uniform rectilinear motion for a fixed time interval Δt. This updates the position of each particle according to the equation

$$\mathbf{r}_i\left(t + \Delta t\right) = \mathbf{r}_i\left(t\right) + \mathbf{v}_i\left(t\right)\Delta t. \tag{1}$$

This displacement implies that some particles will leave the simulation box. To keep the number of particles constant and approximate macroscopic behavior, periodic boundary conditions are imposed. Thus, each particle that exits one side of the box is replaced by another that enters from the opposite side with the same velocity and orientation. This is achieved through the transformation

$$x_i \longrightarrow x_i - L\,\mathrm{floor}\left(\frac{x_i}{L}\right), \tag{2}$$

where the floor function, $\mathrm{floor}\left(x\right)$, that returns the largest integer less than or equal to x, and x_i is the first Cartesian component of $\mathbf{r}_i$. A similar transformation applies to components y_i and z_i.

2.2 Collision Step

After propagation, particles are grouped into cubic cells of volume a^3, distributed in a cubic lattice which fills the entire simulation box. These cells are called *collision cells* because the particles that end up within the same cell participate in an exchange of velocities and orientations that is equivalent to a fictitious multiple collision among them. At every collision step the number of particles in each collision cell could be different since particles move from one collision cell to another during the propagation step. At any given instant, the physical fields of the system can be calculated using the particles located within each collision cell.

The new velocities are assigned using the Andersen thermostat rule [6],

$$\mathbf{v}_i\left(t+\Delta t\right)=\mathbf{v}^{\mathrm{c}}(t)+\boldsymbol{\xi}_i-\boldsymbol{\xi}^{\mathrm{c}}, \tag{3}$$

where

$$\mathbf{v}^{\mathrm{c}}(t)=\frac{1}{N^{\mathrm{c}}}\sum_{i=1}^{N^{\mathrm{c}}}\mathbf{v}_i, \tag{4}$$

is the center of mass velocity in the cell where the particle is located, with N^{c} being the number of particles in that cell.

In addition, in equation (3), $\boldsymbol{\xi}_i$ is a random contribution taken with probability

$$P\left(\boldsymbol{\xi}_i\right)=\left(\frac{m}{2\pi k_{\mathrm{B}}T}\right)^{\frac{3}{2}}\exp\left(-\frac{m}{2k_{\mathrm{B}}T}\boldsymbol{\xi}_i\cdot\boldsymbol{\xi}_i\right), \tag{5}$$

which corresponds to the velocities of molecules in a fluid at temperature T, with k_{B} being the Boltzmann constant.

In equation (3),

$$\boldsymbol{\xi}^{\mathrm{c}}=\frac{1}{N^{\mathrm{c}}}\sum_{i=1}^{N^{\mathrm{c}}}\boldsymbol{\xi}_i, \tag{6}$$

is a term that guarantees the local conservation of linear momentum after velocity update.

To update orientations, it is considered that the particles within the same collision cell interact with the director produced by themselves, $\hat{\mathbf{n}}^{\mathrm{c}}$. To select the new orientations of the particles in the collision cells, the probability distribution is considered [19]

$$P\left(\theta_i\right)\sin\theta_i\,d\theta_i=C_0\exp\left(\frac{3US^{\mathrm{c}}}{2k_{\mathrm{B}}T}\left(\cos^2\theta_i-\frac{1}{3}\right)\right)\sin\theta_i\,d\theta_i, \tag{7}$$

where C_0 is a normalization constant, θ_i is the angle between $\hat{\mathbf{u}}_i$ and $\hat{\mathbf{n}}^{\mathrm{c}}$, U is a scalar quantity referred to as the *nematicity*, which defines the order in the simulated phase, and S^{c} is the so-called order parameter in the cell, which is the largest eigenvalue of the tensor order parameter at the cell,

$$\mathbf{Q}^{\mathrm{c}}=\frac{1}{2N^{\mathrm{c}}}\sum_{i=1}^{N^{\mathrm{c}}}\left(3\hat{\mathbf{u}}_i\hat{\mathbf{u}}_i-\mathbf{I}\right). \tag{8}$$

S^c measures the amount of orientational order and takes two extreme values, $S^c = 0$, when the fluid is completely disordered, and $S^c = 1$, when the alignment of the particles is perfect. In addition, $\hat{\mathbf{n}}^c$ is the eigenvector of $\mathbf{Q}^c$ corresponding to S^c.

The probability density in equation (7) has the form of a canonical law based on the Maier-Saupe mean-field energy. It is known as the Dawson distribution and is illustrated in Figure 3 for different values of U. It can be seen that when U is small, the probability of the angles tends to be uniform, implying a disordered phase; while as U increases, the most probable angle is close to 0, indicating alignment of the particles around $\hat{\mathbf{n}}^c$.

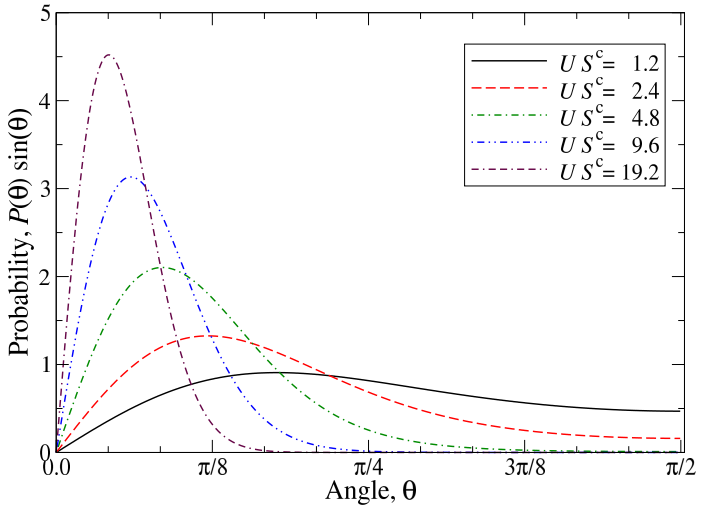

Fig. 3. Dawson distribution, equation (7), for diverse values of the quantity US^c normalized with respect to $k_B T$.

To validate the numerical implementation, the orientations of the resulting particles were taken in tests with different values of U. From these orientations, histograms were constructed that fit very well with the theoretical distribution given by equation (8), as shown in Figure 4.

3 Parallelization

3.1 General Considerations

The program was written in C and the CUDA API. Graphic cards were chosen instead of the machine's central processor because the former typically have more processing threads. The computing resources used for this research and the execution of numerical tests were: a computer with a Linux operating system,

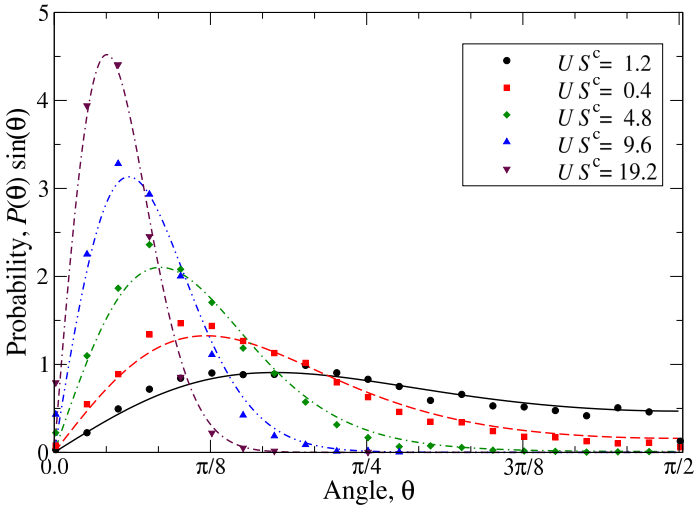

Fig. 4. Dawson distribution for diverse values of the quantity US^c normalized with respect to k_BT. Curves are obtained from equation (7) whereas symbols correspond to normalized frequencies on samples of 81,920 angles generated by the numerical implementation.

equipped with a Tesla T4 graphics card that has 2,560 CUDA threads, and an Intel Xeon E5 2640 CPU with 16 processing threads and an x86_64 architecture.

When analyzing the various stages that comprise a simulation step in the N-MPCD method interval Δt, we can see that these fall into one of two categories:

1. exhaustive stages, where each particle is analyzed individually; and
2. summarized stages, where the system is analyzed at the collision cell level.

Armed with this information, we can see that we have two minimal computing units, the particle and the collision cell, depending on the simulation stage we are in. To maximize the amount of work carried out in parallel, the number of processing threads during the program execution is chosen to be equal to the minimal computing unit.

To store the processed information and avoid memory collisions, two large arrays of structures corresponding to the two minimal computing units were created. It is worth mentioning that older versions of CUDA do not support the use of double-precision floating-point variables. However, in our tests, we noticed that the truncation error produced by using float variables is too large to obtain reliable simulation results. Therefore, the produced code cannot be executed on older GPUs [20].

Another consideration to take into account is the limitations of the x86_64 computer architecture. In our case, we encountered two very important ones, one technical and one historical. The technical limitation is that in the x86_64 architecture, graphics memory is different from main memory, so all information

to be computed on the GPUs must be transferred between them. The historical limitation has to do with the hardware restrictions that existed when the PC standard was proposed, as on certain equipment, the number of memory addresses available for the GPU is less than what would be necessary to index all the graphics memory to the computer's data bus.

To maintain compatibility with x86_64, NVIDIA GPUs do not expose all their memory to the data bus simultaneously. Instead, they expose only a small memory window, which can be shifted at the processor's request to allow reading and writing of the entire graphics memory. This procedure is schematically illustrated in Figure 5. While this solution allows graphics cards to have large amounts of memory, it makes the data transfer between RAM and graphics memory a slow process that consumes CPU time and, therefore, should be avoided as much as possible.

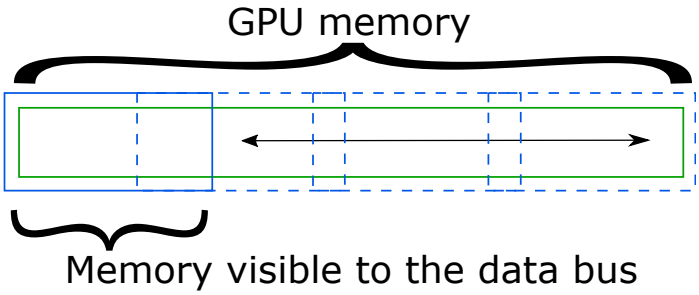

Fig. 5. Visual representation of the graphics memory in modern systems. The green rectangle represents all the graphics memory inside the GPU. The CPU can only read or write information inside the blue window. The CPU can move the window in order to write the whole graphics memory.

Lastly, it should be noted that all functions executed within the GPU must be of the void type, so data transfer and error conditions between functions must be handled via pointers [2]. Taking into account the general considerations above, it was decided to implement the parallelized N-MPCD algorithm according to the block diagram shown in Figure 5(a). For comparison purposes, the corresponding diagram for a serial version is also shown in Figure 5(b). The main steps of the method are detailed below.

3.2 Initialization

The initialization process is exhaustive, where each particle in the simulation system is randomly assigned initial values of position, orientation, and velocity. Fortunately, CUDA includes functions for generating random numbers using various distributions. For each numerical test, random number series were generated from seeds taken from the computer's clock.

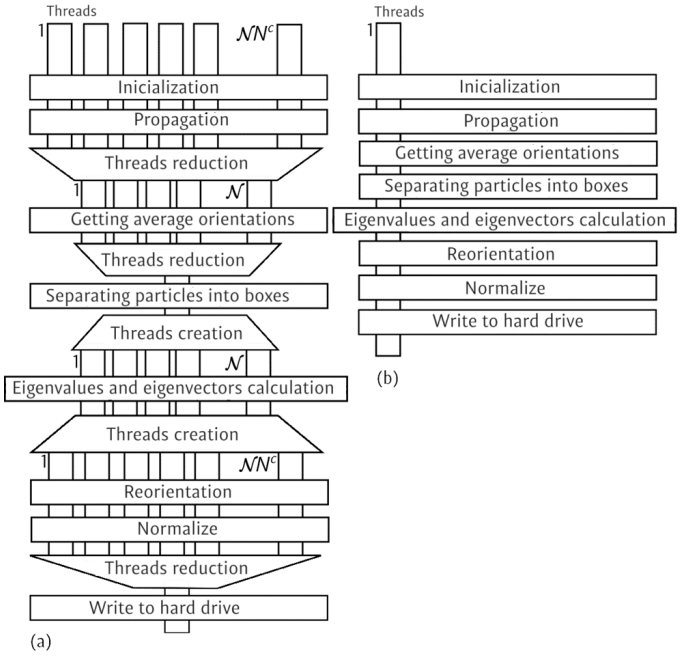

Fig. 6. Block diagram of the N-MPCD method, parallelized (a) and serial (b). The collision step in both cases goes from the calculation of the particles average orientation to the normalization. The serial algorithm runs this operations on a single processing thread.

3.3 Propagation

The movement stage remains an exhaustive process where the position of the particles within the simulation system is updated assuming uniform rectilinear motion. This is done by simply multiplying the velocity by Δt and adding the result to the current position of the particle for each of the Cartesian axes, as indicated by equation (1).

Boundary Conditions The N-MPCD method assumes that the simulation system is surrounded by identical copies of itself. Therefore, if a particle exits the system, it will be automatically replaced by an identical one from one of the adjacent systems. In practice, due to the finite nature of computing resources, the same particle is reintroduced into the system programmatically by implementing equation(2).

3.4 Collision

The collision stage, unlike the previous ones, is no longer exhaustive. To obtain the information that describes the current state of the collision cells, it is neces-

sary to first average the orientation and velocity of all the particles within them. Ideally, while the number of processing threads remains equal to the number of simulated particles, each particle would add its own data to the average of its corresponding cell so that after thread reduction, each collision cell performs the final division. The problem with this implementation is that it inevitably leads to race conditions between the different processing threads writing data to the same variable. Traditionally, this would not be an issue as it is easily solved by implementing a semaphore [3]. Unfortunately, the current state of the CUDA API does not include semaphore functions.

To avoid race conditions, we chose to average the particle data after thread reduction, ensuring that each collision cell sums its own average. This way, we avoid having multiple threads attempting to write to the same variable. Although this implementation is not as efficient as the one previously described, it still performs parallel work, making it faster than a completely serial implementation.

Calculation of the Director Vector To reorient the particles within each collision cell, the N-MPCD method requires calculating the director vector around which the particles will rotate. This is clearly demonstrated by equation (7), which depends on the angle each particle forms with its local director.

As mentioned, the director is calculated by obtaining the eigenvalues and eigenvectors of the order parameter tensor, $\mathbf{Q}^c$, which is defined by equation (8). The scalar order parameter in the cell, S^c, is the highest eigenvalue of $\mathbf{Q}^c$, while the corresponding eigenvector is the local director $\hat{\mathbf{n}}^c$.

As we can see, calculating $\mathbf{Q}^c$ leads us once again to a race condition problem. In this specific case, CUDA provides a set of so-called atomic functions, which are designed to perform the four basic arithmetic operations in parallel for the different types of numerical variables that exist in the C language [2].

Although atomic functions do a good job of handling possible collisions between processing threads, in the numerical tests carried out in this project, it was noted that when the size of the simulated system exceeds 8^3 particles, the number of collisions becomes so large that the GPU has to dedicate a significant amount of time resolving them before it can perform the summation. In the long run, this causes the execution of the parallel program to be slower than the serial version. Therefore, we opted to perform this part of the program serially, once again choosing a suboptimal implementation but maintaining the integrity of the results.

Assignment of New Orientations Once the calculation of S^c and $\hat{\mathbf{n}}^c$ is completed, we obtain new orientations for the particles from the Dawson distribution, equation (8). New velocities are also assigned to them from equations (4) to (6).

3.5 Writing to Disk

Finally, the obtained results are written to the hard disk. Due to physical constraints in the movement of read/write heads in most hard disks, this is not

really a parallelizable process. However, it is possible to generate an additional processing thread that writes data to disk while the GPU continues executing the program.

4 Results

The numerical tests performed aim to establish the physical validity of the method and its performance compared to an existing serial version.

4.1 Nematic Behavior

The method allows observing an orientationally ordered phase for certain values of the nematicity, U. In a first set of simulations, tests were performed for a cubic system with side length $L = 32\,a$, where we maintained an average of 20 particles per collision cell. The average order parameter was calculated for different values of $U = 1, 2, 4, 8, 16, 20$, and 32 $k_{\mathrm{B}}T$.

The results obtained are illustrated in Figure 7, where a transition from disordered states, where $S^{\mathrm{c}} \sim 0$, to ordered states with $S \sim 0.8$ is observed. The former correspond to simple fluid phases and are obtained for $U \lesssim 5\,k_{\mathrm{B}}T$. The ordered states are observed when $U > 5\,k_{\mathrm{B}}T$ and correspond to nematic phases.

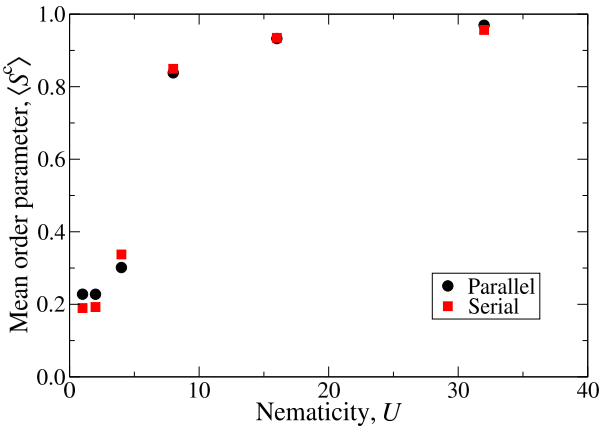

Fig. 7. Behavior of the orientational order of N-MPCD systemas as function of U. Results show that systems achieve order as U increases. A slight difference between results from the serial and parallel implementations can be observed, which is attributable to the numerical precision used in each case.

Another way to understand this behavior is through Figure 8, which illustrates the state of the simulated system using the GPU-parallelized code for different values of U. Two completely disordered states can be seen when $U = 1\,k_{\mathrm{B}}T$

and $U = 4\,k_{\mathrm{B}}T$, cases 7(a) and 7(b), respectively. On the other hand, the system acquires orientational order for values $U = 8\,k_{\mathrm{B}}T$ and $U = 16\,k_{\mathrm{B}}T$, cases 7(c) and 7(d), respectively.

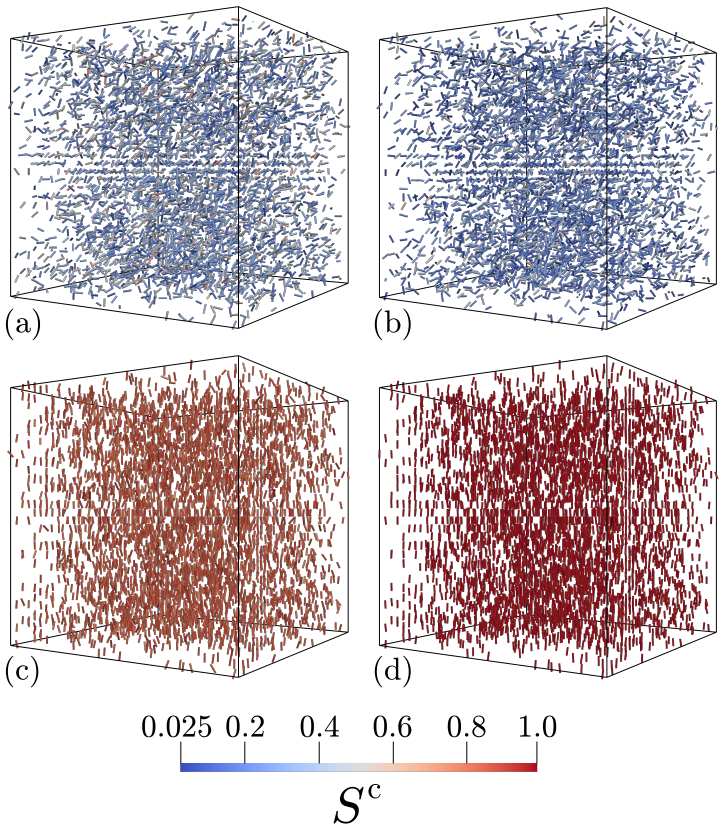

Fig. 8. Ordered and disordered phases simulated by N-MPCD parallelized in GPUs. (a) Disordered sate at $U = 1\,k_{\mathrm{B}}T$. (b) Disordered sate at $U = 4\,k_{\mathrm{B}}T$. (c) Nematic state at $U = 8\,k_{\mathrm{B}}T$. (d) Nematic state at $U = 16\,k_{\mathrm{B}}T$. Small bars indicate the local director field whereas the color scale at the bottom represents the local order parameter. Notice that orientation vectors correspond to local averages in the collision cells. They are not orientations of the individual particles of the method. This is why they are distributed over the same positions.

4.2 Performance

To estimate the performance of the developed implementation, we considered the computation time, as traditional performance comparison methods depend on the similarity in the architecture of the processors where the code is executed.

The first comparison was made between a series of simulations where the nematicity was modified, keeping the parameters fixed: $L = 32\,a$, 20 particles on average per collision cell, $\Delta t = 0.1$, and $k_{\mathrm{B}}T = 1.0$, and $m = 1$. The reported values are the result of a sample of 100 consecutive simulations for each nematicity value. The computation time shown in Table 1 is the average over these samples. It is worth mentioning that the computation times include contributions from writing to disk, which aims to save the information corresponding to the state of the system (value of the order parameter in each collision cell) after each step of the algorithm. This file writing can be considered optional as its purpose was to allow visualization of the system configuration.

Table 1. Average computing times of the parallel N-MPCD method, using different values of U whit parameters $N^c = 20$, $\Delta t = 0.1$, $k_{\mathrm{B}}T = 1$ y $m = 1$.

Nematicity (U)	**Computing time (h:m:s)**
1	00:02:15
2	00:02:15
8	00:02:20
16	00:02:17
32	00:02:17

Execution times were also calculated for systems of different sizes with 20 particles on average per collision cell and fixed $U = 1\,k_{\mathrm{B}}T$. The results obtained in this case are shown in Table 2 for the serial implementation of the N-MPCD method, whereas Table 3 reports on simulation times obtained with the parallel implementation. Notice that the number of collision cells used to assess the performance of the method varied from $8^3 = 512$ to $64^3 = 262\,144$, whereas the total number of simulated particles varied from 10 240 to 5 242 880, respectively.

Table 2. Average computing times of a serial implementation of the N-MPCD method for different values of N, with parameters $U = 1$, $N^c = 20$,$\Delta t = 0.1$, $k_{\mathrm{B}}T = 1$ and $m = 1$.

System size (N)	**Computing time (h:m:s)**
8^3	00:00:15
16^3	00:01:32
24^3	00:05:39
32^3	00:07:49
48^3	00:27:05
64^3	01:25:01

It was observed that the performance of the parallel algorithm is much superior to its serial equivalent. This behavior is emphasized when simulation times

Table 3. Average computing times of a parallel implementation of the N-MPCD method for different values of N, with parameters $U = 1$, $N^c = 20$,$\Delta t = 0.1$, $k_{\mathrm{B}}T = 1$ and $m = 1$.

System size (N)	Computing time (h:m:s)
8^3	00:00:02
16^3	00:00:16
24^3	00:00:46
32^3	00:02:15
48^3	00:05:49
64^3	00:12:53

are compared graphically as in Figure 4.2. In the most significant cases, the system sizes were $L = 48\,a$ and $L = 64\,a$, with the reduction in computation time achieved by the parallelized method being approximately 80% and 85%, respectively. In this regard, it can be concluded that our implementation constitutes a solid first step in the development of a liquid crystal simulation method that, in the near future, could serve as a robust tool for the analysis of such systems.

Since the method developed in this work deals with non-sequential programming, it is convenient to show the algorithm's scaling, also known as speedup, as well as its efficiency. On the one hand, speedup is defined as the ratio of serial to parallel computation times. On the other hand, efficiency is defined as the ratio of speedup to the number of processing threads. The precise values of speedup and efficiency of the method developed in this research are shown in Table 4, which confirms the high performance achieved by the GPU-parallelized implementation.

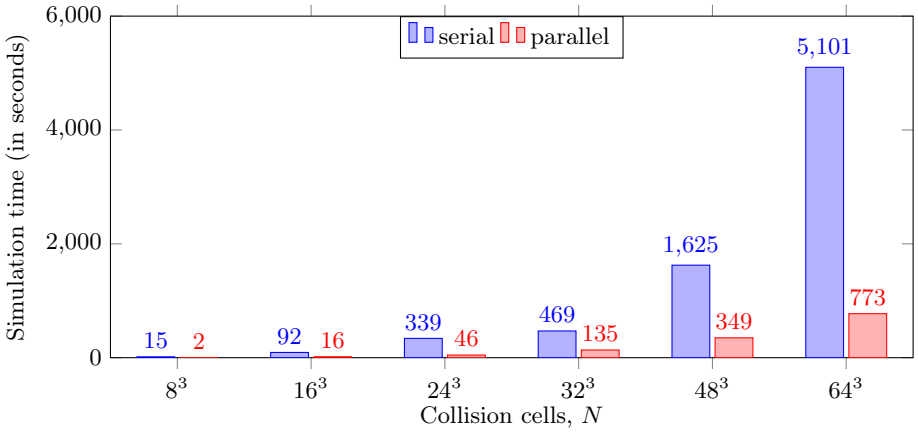

Fig. 9. Comparison between simulation times of the parallel and serial programs (blue and red bars, respectively) for different system sizes quantified by the total number of collision cells.

Table 4. Speedup and efficiency of the parallel implementation of the N-MPCD method

System size (N)	Speedup	Efficiency
8^3	2.17	4.0×10^{-3}
16^3	1.49	3.6×10^{-4}
24^3	2.30	1.6×10^{-4}
32^3	3.12	9.5×10^{-5}
48^3	4.76	4.3×10^{-5}
64^3	41.67	1.5×10^{-4}

5 Conclusions

In this research work, a GPU-parallelized algorithm was implemented to simulate nematic liquid crystals (NLC). The approach used to reproduce the physical characteristics of the nematic phase was inspired by the Nematic Multiparticle Collision Dynamics (N-MPCD) method, which combines particle movements with collision rules to satisfy equilibrium conditions and control the orientational order characteristic of NLCs. Collisions between particles occur in limited and independent spatial regions, allowing operations to be distributed in parallel. The parallelized development was based on NVIDIA technology. The simulations were executed on Tesla T4 cards with 2,560 processing threads, on a computer with a Xeon E5 CPU and x86_64 architecture.

The advantages of the developed code include a significant reduction in computation time and the ability to simulate systems with more particles compared to a serial version of the algorithm. Specifically, the numerical tests performed resulted in a reduction of computation time by an order of magnitude when compared to tests of a serial implementation. Though, in principles, this reduction has to be compared with the two orders of magnitude gain reported for other MPCD implementations based on GPUs [23], it has to be stressed that such implementations do not simulate fluids with nematic features. In addition, it is worth emphasizing that using a GPU with more processing threads could be expected to result in even greater time reductions. Notably, even in the largest systems, GPU memory was not an issue during the method's execution.

Due to the complexity of the dynamics of liquid crystals, the code does not incorporate some of the steps proposed in the original N-MPCD algorithm referenced in [9]. Our implementation does not include the coupling steps between flow and orientation variables. This coupling refers to the fact that in a real liquid crystal, the flow can induce director reorientation and a change in molecular orientation can induce flow.

Flow-induced reorientation can be incorporated in terms of the spatial derivatives of fluid velocity, which are estimated using finite differences between the velocities of different collision cells. These spatial changes in velocity impose torques on the director in each cell, thus causing reorientation. On the other

hand, flow induced by reorientation is incorporated by transforming the angular momentum gain generated by reorientation into orbital angular momentum. Both mechanisms require summarized and extended operations whose parallel implementation is under development.

For future work, it is proposed to complement the method with external forces, such as electric fields or flows, to explore the algorithm's capability to reproduce more complex situations, making it a reliable predictive tool for the behavior of liquid crystals. Additionally, it is recommended to develop a graphical interface that allows for user-friendly manipulation of simulation parameters and to create versions of the algorithm that can run on other hardware platforms not limited to NVIDIA technology.

References

1. Care, C., Cleaver, D.: Computer simulation of liquid crystals. Reports on progress in physics 68(11), 2665 (2005)
2. Cook, S.: CUDA programming: a developer's guide to parallel computing with GPUs. Morgan Kaufmann, Waltham (2012)
3. Downey, A.: The Little Book of Semaphores. Green Tea Press, Massachusetts (2016)
4. Dunmur, D., Sluckin, T.: Soap, science, and flat-screen TVs: a history of liquid crystals. Oxford University Press, Oxford (2014)
5. Feynman, R., Sands, M., Leighton, R.: The Feynman Lectures on Physics, vol. II. Basic Books, New York (2011)
6. Gompper, G., Ihle, T., Kroll, D.M., Winkler, R.G.: Multi-particle collision dynamics – a particle-based mesoscale simulation approach to the hydrodynamics of complex fluids. In: Holm, C., Kremer, K. (eds.) Advanced Computer Simulation Approaches for Soft Matter Sciences III, vol. 221, p. 1–87. Springer, Berlin, Heidelberg (2009)
7. Halver, R., Junghans, C., Sutmann, G.: Using heterogeneous gpu nodes with a cabana-based implementation of mpcd. Parallel Computing 117, 103033 (2023), https://www.sciencedirect.com/science/article/pii/S016781912300039X
8. Híjar, H.: Hydrodynamic correlations in isotropic fluids and liquid crystals simulated by multi-particle collision dynamics. Condens. Matter Phys. 22(1), 13601 (2019)
9. Híjar, H.: Curso introductorio de cristales líquidos i: fases y propiedades estructurales. Revista Mexicana de Física E (2024), in press, also at https://doi.org/10.48550/arXiv.2403.03366
10. Híjar, H., Halver, R., Sutmann, G.: Spontaneous fluctuations in mesoscopic simulations of nematic liquid crystals. Fluct. Noise Lett. 18(3), 1950011 (2019)
11. Hirst, L.: Fundamentals of Soft Matter Science. CRC Press, Boca Raton (2012)
12. Howard, M.P., Nikoubashman, A., Palmer, J.C.: Modeling hydrodynamic interactions in soft materials with multiparticle collision dynamics. Current Opinion in Chemical Engineering 23, 34–43 (2019), https://www.sciencedirect.com/science/article/pii/S2211339819300024, frontiers of Chemical Engineering: Molecular Modeling
13. Howard, M.P., Panagiotopoulos, A.Z., Nikoubashman, A.: Efficient mesoscale hydrodynamics: Multiparticle collision dynamics with massively parallel

gpu acceleration. Computer Physics Communications 230, 10–20 (2018), https://www.sciencedirect.com/science/article/pii/S0010465518301218
14. Lee, K.W., Mazza, M.G.: Stochastic rotation dynamics for nematic liquid crystals. J. Chem. Phys. 142(16), 164110 (2015)
15. Palffy-Muhoray, P.: The diverse world of liquid crystals. Phys. Today 60, 54–60 (2007)
16. Peng, C., Turiv, T., Guo, Y., Wei, Q.H., Lavrentovich, O.D.: Command of active matter by topological defects and patterns. Science 354, 882–885 (2016)
17. Petersen, M., Lechman, J., Plimpton, S., Grest, G., Veld, P., Schunk, P.: Mesoscale hydrodynamics via stochastic rotation dynamics: Comparison with lennard-jones fluid. J.Chem.Phys 132, 174106 (2010)
18. Ratan, S.S.: Gpu-based multiscale simulation to model active matter hydrodynamics in fluid medium (2023), bS-MS Thesis. Indian Institute of Science Education and Research Pune
19. Shendruk, T.N., Yeomans, J.M.: Multi-particle collision dynamics algorithm for nematic fluids. Soft Matter 11, 5101–5110 (2015)
20. Soyata, T.: GPU Parallel Program Development Using CUDA. CRC Press, Boca raton (2018)
21. Tomilin, M.G., Povzun, S.A., Kurmashev, A.F., Gribanova, E.V., Efimova, T.A.: The application of nematic liquid crystals for objective microscopic diagnosis of cancer. Liq. Cryst. Today 10, 3–5 (2001)
22. Wang, H., Xu, T., Fu, Y., Wang, Z., Leeson, M., Jiang, J., Liu, T.: Liquid crystal biosensors: Principles, structure and applications. Biosensors 12, 639–666 (2022)
23. Westphal, E., Singh, S., Huang, C., Gompper, G., Winkler, R.: Multiparticle collision dynamics: Gpu accelerated particle-based mesoscale hydrodynamic simulations. Comput.Phys.Commun 185, 495–503 (2014)