Paralelización de simulaciones mesoscópicas de cristales líquidos en GPUs programadas con CUDA

Parallelization of mesoscopic simulations of liquid crystals on GPUs programmed with CUDA

Jorge Fierro[1] y Humberto Híjar[1,2,*]

[1]Centro de Investigación, Universidad La Salle México
Benjamín Franklin 45, Colonia Condesa, Alcaldía Cuauhtémoc, Ciudad de México, 06140, MÉXICO
[2]Facultad de Ciencias, Universidad Nacional Autónoma de México,
Circuito Exterior s/n, Ciudad Universitaria, Alcaldía Coyoacán, Ciudad de México, 04510, MÉXICO
[*]humberto.hijar@lasalle.mx

PALABRAS CLAVE:

Tarjetas Gráficas, Simulaciones de Fluidos, Optimización, Dinámica de Colisión de Múltiples Partículas

RESUMEN

Los cristales líquidos son fluidos que guardan cierta cantidad de orden en la orientación y posición de sus moléculas. Éstos han sido objeto de numerosas investigaciones debido a su relevancia tecnológica. En este trabajo de investigación se propone un método de simulación para la fase de cristal líquido con la simetría más sencilla, conocida como fase nemática. El método está basado en partículas que interactúan en conjuntos independientes, lo que permite plantear programarlo en paralelo. Esto se lleva a cabo en unidades de procesamiento gráfico (GPU) en la arquitectura CUDA de NVIDIA. Se demuestra que el método permite simular la aparición de orden molecular en condiciones reproducibles. También se exhibe claramente que el procedimiento en paralelo tiene un desempeño mucho más alto que el que brinda una versión serial del mismo algoritmo de simulación.

KEYWORDS:

Graphic Processing Units, Fluids simulations, Speedup, Multi-particle Collision Dynamics

ABSTRACT

Liquid crystals that are fluids that preserve certain amount of order in the orientation and position of their molecules. 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.

• Recibido: 15 de noviembre de 2023 • Aceptado: 25 de abril de 2024 • Publicado en línea: 1 de junio de 2025

1. INTRODUCCIÓN

La física introductoria describe tres estados de la materia: sólido, líquido y gas, los cuales se diferencian por la rapidez y libertad con la que las moléculas pueden moverse en ellos [1]. Usualmente, los materiales realizan la transición entre los estados sólido y líquido sin una etapa intermedia. Sin embargo, Freiderich Reinitzer descubrió a finales del siglo XIX que pueden existir fases intermedias entre estos dos

DOI: 10.30973/progmat/2025.17.2/3

estados [2]. Pocos años después, Otto Lehmann nombró a estas fases tal como las conocemos hoy en día: *cristales líquidos* [2].

Las aplicaciones de los cristales líquidos incluyen pantallas de televisiones, teléfonos celulares y computadoras conocidas como LCD, por sus siglas en inglés (*Liquid Crystal Displays*); filtros de longitud de onda ajustables; cavidades resonantes para láseres sintonizables; termómetros y ventanas inteligentes [3]. Investigaciones recientes sugieren su uso en la detección de agentes patógenos, antígenos, tumores cancerosos [4] [5], así como en el control de la trayectoria de microorganismos [6].

Se conoce una enorme variedad de cristales líquidos. Todos ellos están formados por moléculas cuya simetría no es esférica, *e. g*. moléculas alargadas en forma de barra. La fase de cristal líquido con la simetría más sencilla se conoce como fase nemática. El ordenamiento molecular en un cristal líquido nemático (CLN) se ilustra esquemáticamente en la figura 1. Los centros de masa de las moléculas en un CLN pueden moverse arbitrariamente, como en un líquido simple, pero los ejes moleculares permanecen orientados en torno a un eje común conocido como *director*, representado por un vector unitario $\hat{n}$ [7].

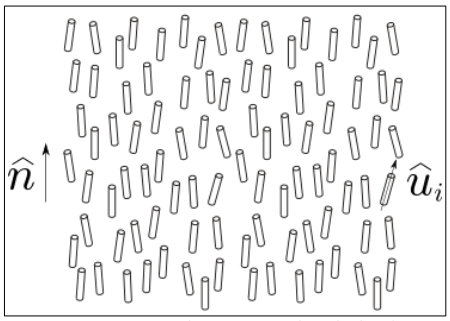

Figura 1. Esquema de un CLN donde las barras representan las moléculas del material. El vector director, $\hat{n}$, indica la orientación promedio. La orientación molecular individual se indica mediante el vector unitario $\hat{u}_i$.

Por las razones mencionadas previamente, los cristales líquidos son de mucho interés en las ciencias aplicadas y la ingeniería de materiales, en donde las simulaciones computacionales han jugado un papel determinante en el entendimiento de sus propiedades [8].

Recientemente, se ha propuesto un algoritmo conocido en inglés como *Nematic Multiparticle Collision Dynamics* (N-MPCD), el cual describe al CLN como un sistema de partículas que acarrean un vector de orientación [9] [10]. Periódicamente, se permite que las partículas interactúen con las que se encuentren en su vecindad. Para ello, el espacio de simulación se discretiza en celdas cúbicas contiguas dentro de las cuales se llevan a cabo operaciones independientes, lo que sugiere que el método podría ser paralelizado.

N-MPCD es una extensión de un método de simulación de fluidos simples, conocido como *Multiparticle Collision Dynamics* (MPCD) para el cual se han planteado diversos algoritmos que operan en paralelo. Uno de los primeros fue desarrollado por Petersen *et al*. quienes adaptaron el método para ser ejecutado en múltiples procesadores de una computadora Cray XT3 [11]. Westphal *et al*. han desarrollado una implementación de MPCD basada en unidades de procesamiento gráfico (GPU, por sus siglas en inglés) obteniendo una ganancia en el rendimiento de hasta dos órdenes de magnitud con respecto a una versión comparable en unidades de procesamiento central (CPU) [12]. Howard *et al*. han presentado una implementación de código abierto del algoritmo que escala para correr en cientos de GPU [13][14]. Más recientemente, Halver *et al*. han utilizado nodos heterogéneos de GPU para paralelizar MPCD en una implementación basada en Cabana [15].

El objetivo del presente trabajo es paralelizar el método N-MPCD aprovechando que trabaja con cantidades que representan a las partículas agrupadas en espacios discretos e independientes. El problema a resolver consiste en llevar a cabo las operaciones

sobre las propiedades de las partículas que conforman al sistema de manera simultánea e independiente cuando sea necesario, así como realizar operaciones convergentes paralelas que requieren a las partículas agrupadas en las celdas cúbicas del sistema. Todo lo anterior se debe realizar respetando las reglas fisicomatemáticas que producen un comportamiento de CLN del sistema. El propósito es que esta simulación se ejecute en GPU. También se busca que el desempeño de esta implementación en paralelo supere al de una versión serial desarrollada previamente [16] [17]. Al desarrollar el método N-MPCD paralelizado en GPU, una de las dificultades más fuertes fue que muchos hilos de procesamiento necesitaban escribir sobre la misma sección de memoria al mismo tiempo. Este problema se resolvió al utilizar un hilo de procesamiento por cada una de las celdas de colisión. Dedicado exclusivamente a promediar las propiedades de las partículas contenidas en él. Esta implementación mejoró el tiempo de cómputo en un orden de magnitud con respecto al requerido por una versión serial.

El contenido de este artículo es el siguiente. En la sección 2 se describirán las características básicas del algoritmo N-MPCD. Posteriormente, en la sección 3, se discutirá cómo se llevó a cabo la paralelización del método en GPU. En la sección 4 se presentarán los resultados principales de nuestra investigación y en la sección 5 se sintetizarán las conclusiones y propondrán posibles trabajos a futuro.

2. METODOLOGÍA

El sistema de simulación está formado por N partículas puntuales que se mueven dentro de una caja cúbica de lado L, el cual se considera un múltiplo entero de la unidad de longitud a. Todas las partículas tienen la misma masa m. Sus posiciones y velocidades se representan por los vectores r_i y v_i, con i=1,2, ..., N. Cada partícula tiene asociado un vector de orientación unitario $\hat{u}_i$, que servirá para generar el orden de orientación característico de los cristales líquidos nemáticos. Los vectores r_i, v_i y $\hat{u}_i$ se consideran funciones continuas del tiempo t, y se actualizarán para generar la dinámica del material. El algoritmo encargado de ello consta de dos pasos conocidos como *paso de propagación* y *paso de colisión*. Ambos se describirán a continuación.

2.1. Propagación

Las partículas se desplazan en movimiento rectilíneo uniforme durante un intervalo de tiempo fijo Δt. Esto actualiza la posición de cada partícula de acuerdo con la ecuación:

$$r_i(t+\Delta t)=r_i(t)+v_i(t)\cdot\Delta t \tag{1}$$

Este desplazamiento implica la salida de algunas partículas de la caja de simulación. Para preservar el número de partículas constante y aproximar un comportamiento a escala macroscópica, se imponen condiciones de frontera periódicas. Así, cada partícula que sale por un extremo de la caja se reemplaza por otra que entra por el extremo opuesto y tiene las mismas velocidad y orientación. Esto se logra mediante la transformación

$$x_i \rightarrow x_i - L\cdot floor\left(\frac{x_i}{L}\right) \tag{2}$$

en donde *floor* es la función de argumento real que devuelve el entero más alto que es menor o igual al argumento y x_i es la primera componente cartesiana de r_i. Una transformación similar aplica para y_i y z_i.

2.2. Colisión

Posterior a la propagación, las partículas se agrupan en celdas cúbicas de volumen a^3, que llenan uniformemente la caja de simulación. Estas celdas son llamadas *celdas de colisión* porque las partículas que resultan estar dentro de la misma celda participan en un intercambio de velocidades y orientaciones

que equivale a una colisión ficticia múltiple entre ellas.

Las nuevas velocidades se asignan con la regla del termostato de Andersen [18],

$$v_i(t+\Delta t)=v^c(t)+\xi_i(t)+\xi^c(t) \tag{3}$$

donde

$$v^c(t)=\frac{1}{N^c}\sum_{i\in c} v_i(t) \tag{4}$$

es la velocidad del centro de masa en la celda, con N^C el número de partículas en la celda de colisión.

Además, en la ecuación (3), $\xi_i(t)$ una contribución aleatoria tomada con probabilidad

$$P(\xi_i)=\left(\frac{m}{2\pi k_B T}\right)^{\frac{3}{2}} e^{\frac{-m\xi_i^2}{2k_B T}} \tag{5}$$

que es la que corresponde a las moléculas en un fluido con temperatura T, siendo k_B la constante de Boltzmann. Por otra parte,

$$\xi^c(t)=\frac{1}{N^c}\sum_{i\in c} \xi_i(t) \tag{6}$$

es un término que garantiza la conservación del momento lineal después de la transformación de velocidades.

Para actualizar las orientaciones se considera que las partículas dentro de la misma celda de colisión interactúan con el director producido por ellas mismas, $\hat{n}^c$. Para seleccionar las nuevas orientaciones de las partículas en las celdas de colisión se considera la distribución de probabilidades [9].

$$P(\theta_i)=C_0 e^{\frac{3US^c}{2k_B T}\left[|\cos\theta_i|^2-\frac{1}{3}\right]}\sin\theta_i \tag{7}$$

Donde C_0 es una constante de normalización, θ_i es el ángulo entre $\hat{u}_i$ y $\hat{n}^c$, U es una cantidad escalar referida como la *nematicidad*, la cual se encarga definir el orden en la fase simulada y S^c es el llamado parámetro de orden en la celda, el cual se calcula mediante

$$S^c=\frac{1}{2}\left\langle 3|\cos\theta_i|^2-1\right\rangle$$

con los *brakets* indicando un promedio dentro de la celda de colisión. S^c mide la cantidad de orden orientacional y toma dos valores extremos, $S^c=0$, cuando el fluido está completamente desordenado y $S^c=1$, cuando el alineamiento de las partículas es perfecto.

La densidad de probabilidad en la ecuación (7) se conoce como distribución de Dawson y esta se ilustra en la figura 2 para diferentes valores de U. Puede apreciarse que cuando U es pequeña, la probabilidad de los ángulos tiene a ser uniforme, lo que implica una fase desordenada; mientras que al aumentar U los ángulos más probables son 0 y π, lo que es señal de un alineamiento de las partículas alrededor de $\hat{n}^c$.

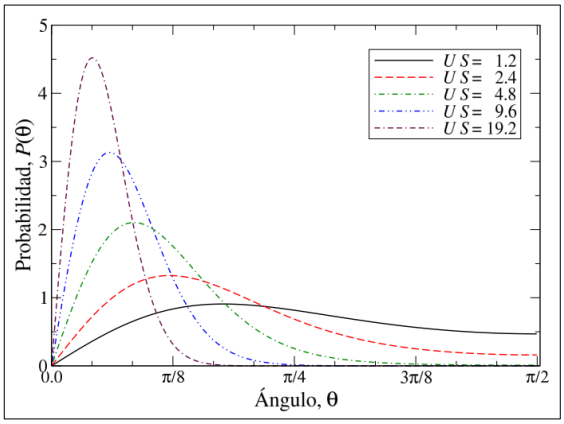

Figura 2. Distribución de Dawson para diferentes valores del parámetro U S.

Para validar la implementación numérica se tomaron las orientaciones de las partículas resultantes en pruebas con diferentes valores de U. A partir de estas orientaciones se construyeron histogramas que ajustaron muy bien con la distribución teórica dada por la ecuación (7), tal como se muestra en la figura 3.

3. PARALELIZACIÓN

3.1. Consideraciones generales

La escritura del programa se llevó a cabo en lenguaje C y la API de CUDA. Se optó por utilizar tarjetas gráficas en lugar del procesador central de la máquina, debido a que las primeras tienen normalmente más hilos de procesamiento. Los recursos de cómputo con los que se contó para el desarrollo de esta investigación y la ejecución de pruebas numéricas fueron: una computadora con sistema operativo Linux, con una tarjeta gráfica Tesla T4 que cuenta con 2,560 hilos de CUDA y con un CPU Intel Xeon E5 2640 con 16 hilos de procesamiento y una arquitectura x86_64.

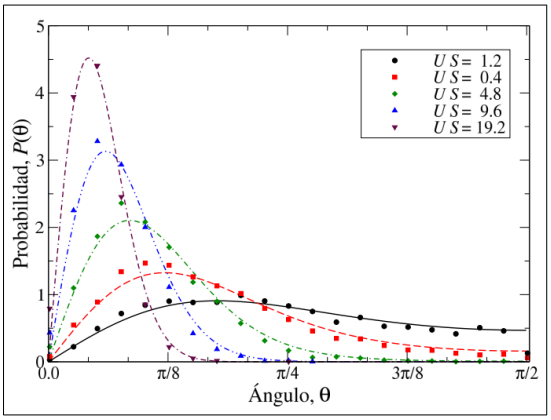

Figura 3. Distribución de Dawson para diferentes valores del parámetro U S. Las curvas se obtienen a partir de la ecuación (7) y los símbolos corresponden con histogramas construidos con base a una muestra de 81,920 ángulos generados con la implementación numérica.

Al analizar las distintas etapas que comprenden un paso de simulación en el intervalo Δt del método NMPC-D, podemos ver que estas caen en una de dos categorías:

1) Etapas exhaustivas, donde cada partícula se analiza individualmente,
2) Etapas resumidas, donde el sistema es analizado a nivel de celda de colisión.

Armados con esta información, podemos notar que tenemos dos unidades mínimas de cómputo, la partícula y la celda de colisión, dependiendo de la etapa de simulación en la que nos encontremos. Con el fin de maximizar la cantidad de trabajo que se lleva a cabo de manera paralela, la cantidad de hilos de procesamiento durante la ejecución del programa se elige igual a la unidad mínima de cómputo.

Para guardar la información procesada y evitar las colisiones de memoria, se crearon dos grandes arreglos de estructuras que corresponden a las dos unidades mínimas de cómputo. Cabe mencionar que versiones antiguas de CUDA no soportan el uso de variables de punto flotante de doble precisión, sin embargo, en nuestras pruebas notamos que el error de truncamiento producido por usar variables de tipo *float* es demasiado grande para obtener resultados de simulación confiables. Por lo que el código producido no es capaz de ser ejecutado en tarjetas gráficas antiguas [19].

Otra consideración que se debe de tener en cuenta, son las limitaciones de la arquitectura de computadoras x86_64. En nuestro caso, nos encontramos con dos muy importantes, una de carácter técnico y otra de carácter histórico.

La limitación técnica consiste en que, en la arquitectura x86_64, la memoria gráfica es distinta a la memoria principal, por lo que toda información que se desee computar en las GPUs deberá ser transferida entre las mismas. La limitación histórica tiene que ver con las restricciones de hardware que existían cuando se propuso el estándar de la PC, ya que en ciertos equipos la cantidad de direcciones de memoria disponibles para la GPU es inferior a la que sería necesaria para indexar toda la memoria gráfica al *bus* de datos de la computadora.

Para mantener la compatibilidad con x86_64, las tarjetas gráficas de NVIDA no exponen toda su memoria al *bus* de datos al mismo tiempo. En su lugar, sólo exponen una pequeña ventana de memoria, la cual a petición del procesador se puede desplazar para permitir la lectura y escritura de toda la

memoria gráfica. Este procedimiento se ilustra esquemáticamente en le figura 4. Si bien esta solución permite que las tarjetas gráficas tengan grandes cantidades de memoria, convierte la transferencia de datos entre RAM y memoria gráfica en un proceso lento, el cual consume tiempo de CPU y, por lo tanto, se busca evitar lo más posible.

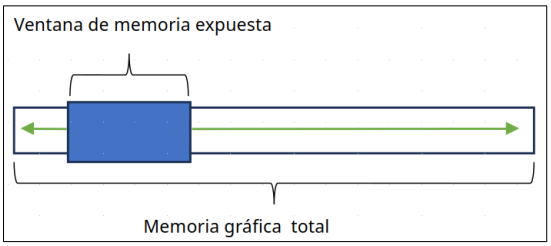

Figura 4. Representación visual de la memoria gráfica del sistema, el rectángulo grande representa toda la memoria de la GPU. El CPU sólo puede leer y escribir información de la ventana de memora representada por el rectángulo azul. El CPU le ordena a la GPU qué segmento de memoria exponer.

Por último, se debe de tener en cuenta que todas las funciones que se ejecutan dentro de la GPU deben de ser de tipo *void*, por lo que la transferencia de datos y condiciones de error entre las funciones de debe de llevar a cabo mediante apuntadores [20].

Al tomar en cuenta las consideraciones generales anteriores se decidió implementar el algoritmo paralelizado N-MPCD de acuerdo con el diagrama de bloques mostrado en la figura 5(a) en donde para propósitos de comparación también se muestra el diagrama correspondiente a una versión serial, figura 5(b). Los pasos principales del método se describen en detalle a continuación.

3.2. Inicialización

El proceso de inicialización es uno de carácter exhaustivo donde a cada partícula dentro el sistema de simulación se le asignan valores iniciales de posición, orientación y velocidad de manera aleatoria. Afortunadamente, CUDA incluye funciones para la generación de números aleatorios utilizando distintas distribuciones.

3.3. Propagación

La etapa de movimiento sigue siendo un proceso exhaustivo donde se actualiza la posición de las partículas dentro del sistema de simulación asumiendo un movimiento rectilíneo uniforme, por lo que simplemente se procede a multiplicar la velocidad por Δt .y sumar el resultado a la posición actual de la partícula por cada uno de los ejes cartesianos, tal como lo indica la ecuación (1).

Condiciones de frontera

El método NMPC-D asume que el sistema de simulación se encuentra rodeado por copias idénticas de sí mismo, por lo que, si una partícula llegase a salir del sistema, esta será reemplazada en automático por una de las características idénticas proveniente de alguno de los sistemas aledaños. En la práctica, debido a la finitud de los recursos de cómputo lo que se hace programáticamente es reintroducir la misma partícula al sistema al implementar la ecuación (7).

3.4. Colisión

La etapa de colisión, a diferencia de las anteriores, ya no es de carácter exhaustivo. Para obtener la información que describe el estado actual de las celdas de colisión, es necesario primero promediar la orientación y la velocidad de todas las partículas que se encuentran dentro de las mismas. Idealmente, mientras el número de hilos de procesamiento sigue siendo igual al número de partículas simuladas, se buscaría que cada partícula sumara sus datos propios al promedio de su celda correspondiente para que después de la reducción de hilos, cada celda de colisión haga la división final.

El problema con esta implementación es que inevitablemente nos lleva a tener condiciones de carrera entre los distintos hilos de procesamiento que se encuentran escribiendo datos a una misma variable. Tradicionalmente esto no sería un problema, ya que se soluciona fácilmente

implementando un semáforo [21]. Desafortunadamente, el API de CUDA en su estado actual no incluye funciones de semáforo.

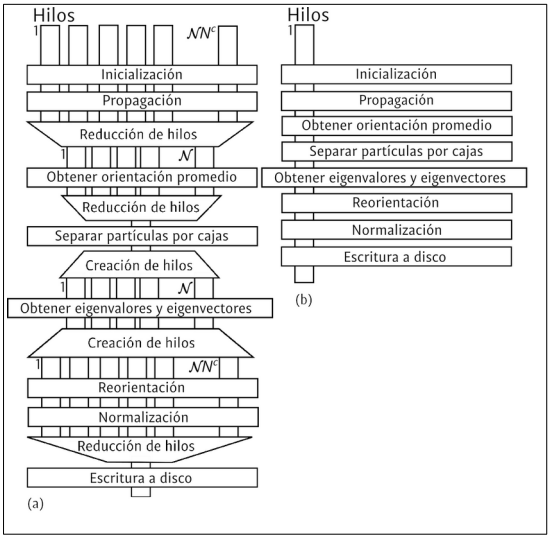

Figura 5. Diagrama de bloques del método N-MPCD paralelizado (a) y serial (b). El paso de colisión en ambos casos se comprende desde obtención de orientaciones promedio hasta la normalización. El algoritmo serial ejecuta las mismas operaciones, pero a través de un solo hilo de procesamiento.

Para evitar condiciones de carrera, se optó por realizar el promedio de los datos de las partículas después de la reducción de hilos y que cada celda de colisión haga la suma de su propio promedio. De esta manera se evita tener múltiples hilos intentando escribir una misma variable. Si bien esta implementación no es tan eficiente como la descrita anteriormente, sigue realizando trabajo en paralelo, por lo que continúa siendo más veloz que una implementación completamente serial.

Cálculo del vector director

Para reorientar las partículas dentro de cada celda de colisión, el método N-MPCD requiere que se calcule el vector director de las mismas, alrededor del cual rotarán las partículas contenidas en estas. Esto lo exhibe claramente la ecuación (7) al depender del ángulo que forma cada partícula con su director local.

El director se calcula obteniendo los valores y vectores propios de una matriz Q^c, conocida como el parámetro de orden tensorial, la cual está definida de la siguiente manera,

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

donde $\otimes$ denota el producto exterior de dos vectores e I es la matriz identidad. El parámetro de orden escalar en la celda, S^c, es equivalente al eigenvalor más alto de Q^c, mientras que el eigenvector correspondiente es el director local $\hat{n}^c$.

Como podemos observar, el cálculo de Q^c nos lleva nuevamente a tener el problema de condición de carrera. En este caso específico, CUDA cuenta con un conjunto de así llamadas funciones atómicas, las cuales están diseñadas para llevar a cabo las cuatro operaciones aritméticas básicas de manera paralela para los distintos tipos de variables numéricas que existen en lenguaje C [20].

Si bien las funciones atómicas hacen un buen trabajo manejando las posibles colisiones entre hilos de procesamiento, en las pruebas numéricas llevada a cabo en este proyecto se notó que cuando el tamaño del sistema simulado supera 8^3, el número de colisiones se vuelve tan grande que la GPU tiene que dedicar una cantidad importante de tiempo resolviéndolas antes de poder llevar a cabo la suma. A la larga, esto ocasiona que la ejecución del programa paralelo sea más lenta que la de la versión serial. Por lo que se optó por realizar esta parte del programa de manera serial, nuevamente optando por una implementación subóptima, pero manteniendo la integridad de los resultados.

Asignación de nuevas orientaciones

Una vez finalizado el cálculo de s^c y $\hat{n}^c$, obtuvimos nuevas orientaciones para las partículas a partir de la distribución de Dawson, ecuación (7). También se asignan

nuevas velocidades para las mismas a partir de las ecuaciones (3) a (6).

3.5. Escritura a disco

Por último, se escriben los resultados obtenidos en el disco duro. Por restricciones físicas en el movimiento de cabezas en la mayoría de los discos duros, este no es realmente un proceso paralelizable. Sin embargo, es posible generar un hilo de procesamiento adicional que escriba los datos a disco mientras la GPU continúa con la ejecución del programa.

4. RESULTADOS

Las pruebas numéricas realizadas persiguen el fin de establecer la validez física del método y su rendimiento con respecto a una versión serial ya existente.

4.1. Comportamiento nemático

El método permite observar una fase orientacionalmente ordenada para ciertos valores de la nematicidad, U. En un primer conjunto de simulaciones se ejecutaron pruebas para un sistema cúbico de lado $L=32a$, en el que mantuvimos 20 partículas promedio por celda de colisión. El parámetro de orden promedio se calculó para diferentes valores de la nematicidad $U=1,2,4,8,16,20$ y $32K_bT$.

Los resultados obtenidos se ilustran en la figura 6, en donde se aprecia una transición de estados desordenados en los cuales $S<0.3$, hacia estados ordenados con $S\sim0.8$. Los primeros corresponden a fases de fluido simple y se obtienen para $U<5$. Los estados ordenados se aprecian cuando $U\geq5$ y corresponden a fases nemáticas.

Otra forma de entender este comportamiento es a través de la Figura 7, la cual ilustra el estado del sistema simulado mediante el código paralelizado en GPU, para diferentes valores de U. Se pueden observar dos estados completamente desordenados cuando U=1 y $4k_bT$, casos 7(a) y 7(b), respectivamente. Por otra parte, el sistema adquiere orden orientacional para valores U=8 y $16k_bT$, casos 7(c) y 7(d), respectivamente.

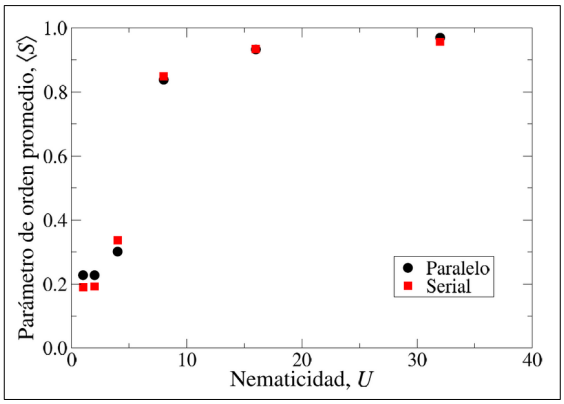

Figura 6. Comportamiento del orden en la orientación de las partículas como función del parámetro U. Los resultados muestran que el material se ordena cuando U crece. Se observa una pequeña diferencia entre los resultados del método serial y el paralelizado debido a la precisión numérica manejada por los mimos.

4.2. Rendimiento

Para estimar el rendimiento de la implementación desarrollada se tomó como propiedad el tiempo de cómputo, debido a que los métodos más tradicionales de comparación de rendimiento dependen de una similitud en la arquitectura de los procesadores donde se esté ejecutando el código.

La primera comparación se ejecutó entre una serie de simulaciones en donde se modificó la nematicidad en $U=1,2,4,8,16,20$ y $32K_bT$, manteniendo fijos los parámetros: $L=32a$, 20 partículas promedio por celda de colisión, $\Delta t=0.1$, $K_bT=1$ y $m=1$. Los valores reportados son el resultado de una muestra de 100 simulaciones consecutivas para cada valor de la nematicidad. El tiempo de cómputo que aparece en la tabla 1 es el promedio sobre dichas muestras. Cabe mencionar que los tiempos de cómputo tienen contribuciones debidas a la escritura a disco la cual tiene el propósito de guardar la

información correspondiente al estado del sistema (valor del parámetro de orden en cada celda de colisión), posterior a cada paso del algoritmo.

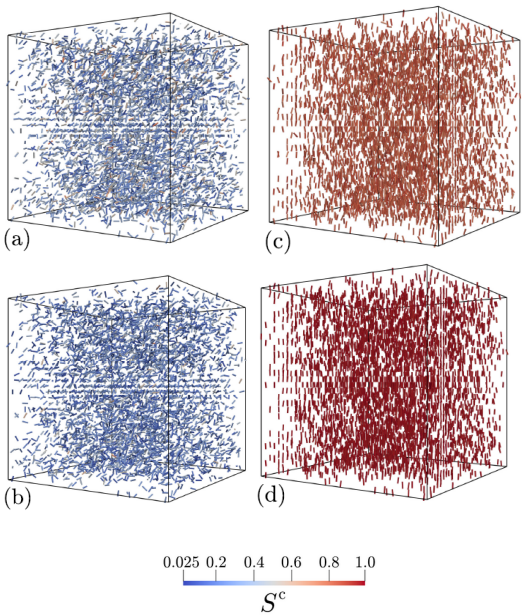

Figura 7. Fases desordenadas y ordenadas simuladas por el método N-MPCD paralelizado en GPU. (a) Estado desordenado a U=1k_bT. (b) Estado desordenado a U=4k_bT. (c) Estado ordenado a U=8k_bT. (d) Estado ordenado a U=16k_bT. Las barras en cada ilustración representan los ejes de alineamiento de las partículas y la escala de colores indica la cantidad de orden el sistema.

Esta escritura de archivos puede considerarse opcional, debido a que su fin era el de permitir una visualización de la configuración del sistema.

Tabla 1. Tiempos de cómputo promedio para distintos valores de U en simulaciones del algoritmo N-MPCD paralelizado con parámetros N^c=20, Δt=0.1, k_BT=1 y m=1.

Nematicidad (U)	Tiempo de cómputo (h:m:s)
1	00:02:15
2	00:02:15
8	00:02:20
16	00:02:17
32	00:02:17

Asimismo, se calcularon los tiempos de ejecución para sistemas de diferentes tamaños con 20 partículas en promedio por celda de colisión y $U=1k_BT$, fijos. Los resultados obtenidos en este caso se muestran en las tablas 2 y 3.

Tabla 2. Tiempos de cómputo promedio obtenidos a partir de una versión serial de N-MPCD para distintos valores de N con parámetros U=1, N^c=20, Δt=0.1, k_bT=1 y m=1.

Tamaño del sistema (N)	Tiempo de Cómputo (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

Tabla 3. Tiempos de cómputo promedio obtenidos a partir la implementación de N-MPCD paralelizada en GPU para distintos valores de N con parámetros U=1, N^c=20, Δt=0.1, k_bT=1 y m=1.

Tamaño del sistema (N)	Tiempo de Cómputo (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

Se observó que el rendimiento del algoritmo paralelo es muy superior al de su equivalente serial. En los casos más significativos los sistemas tuvieron tamaño L=48a y L=64a, siendo la reducción en el tiempo de cómputo obtenida mediante el método paralelizado cercana al 80% y al 85%, respectivamente. En este sentido, puede concluirse que nuestra implementación constituye un primer paso sólido en el desarrollo de un método de simulación de cristales líquidos que en el futuro cercano podrá servir como una herramienta robusta para el análisis de dichos sistemas.

Dado que el método desarrollado en el presente trabajo trata con programación no secuencial, resulta conveniente mostrar el escalamiento del algoritmo, también conocido como *speedup*, así como la eficiencia del mismo. Por una parte, el *speedup* se define como el cociente de los tiempos de cómputo serial y paralelo. Por otra, se define la eficiencia como la razón del *speedup* entre la cantidad de hilos de procesamiento. Los valores precisos del *speedup* y la eficiencia del método desarrollado en este trabajo de investigación se muestran en la tabla 4, la cual confirma el

alto desempeño logrado por la implementación paralelizada en GPU.

Tabla 4. speedup y eficiencia de la implementación paralelizada en GPU de N-MPCD.

Tamaño del sistema (N)	Speedup	Eficiencia
8^3	7.5	0.00293
16^3	5.75	0.00225
24^3	7.37	0.00288
32^3	3.47	0.00136
48^3	4.66	0.00182
64^3	13.51	0.00528

5. CONCLUSIONES

En este trabajo de investigación se implementó un algoritmo paralelizado en GPU para simular cristales líquidos nemáticos (CLN). El enfoque utilizado para reproducir las características físicas de la fase nemática ha sido inspirada en el método de *Nematic Multiparticle Collision Dynamics* (N-MPCD), el cual combina movimientos de partículas con reglas de colisión para satisfacer condiciones de equilibrio y controlar el orden orientacional característico de los CLN. Las colisiones entre las partículas ocurren en regiones espaciales limitadas e independientes. Esta propiedad permitió distribuir las operaciones en forma paralela. El desarrollo paralelizado se fundamentó en tecnología de NVIDIA. Las simulaciones se ejecutaron en tarjetas Tesla T4, con 2,560 hilos de procesamiento, en una computadora con un CPU Xeon E5 y arquitectura x86_64.

Las ventajas del código desarrollado son una reducción significativa del tiempo de cómputo y la capacidad de simular sistemas con más partículas, en comparación con una versión serial del algoritmo. Concretamente, las pruebas numéricas realizadas resultaron en una reducción de tiempo de cómputo de un orden de magnitud, cuando se comparan con pruebas de una implementación serial. Cabe enfatizar que si se utilizara una GPU con más hilos de procesamiento podría esperarse una reducción de tiempo aún mayor. Notablemente, aún en los sistemas de mayor tamaño, la memoria de la GPU no fue un problema durante la ejecución del método.

Por razones de la complejidad de la dinámica de los cristales líquidos, el código no incorpora algunos de los pasos propuestos en el algoritmo original de N-MPCD propuesto en la referencia [9]. Nuestra implementación no incluye los pasos de acoplamiento entre las variables de flujo y orientación. Dicho acoplamiento se refiere a que en un cristal líquido real el flujo puede inducir reorientación del director y un cambio en la orientación molecular puede inducir flujo.

La reorientación por flujo puede incorporarse en términos de las derivadas espaciales de la velocidad del fluido, las cuales se estiman utilizando diferencias finitas entre las velocidades de las diferentes celdas de colisión. Estos cambios espaciales de la velocidad imponen torcas sobre el director en cada celda causando así la reorientación. Por otro lado, el flujo inducido por la reorientación se incorpora al transformar la ganancia de momento angular generada por la reorientación, en momento angular orbital. Ambos mecanismos requieren operaciones resumidas y extendidas cuya implementación en paralelo está en proceso de desarrollo.

Como trabajo a futuro se propone complementar el método con fuerzas externas, por ejemplo, campo eléctricos o flujos, con el fin de explorar la capacidad del algoritmo de reproducir situaciones más complejas y que lo vuelvan una herramienta de predicción confiable sobre el comportamiento de los cristales líquidos. Asimismo, se propone desarrollar una interfaz gráfica que permita una manipulación amigable de los parámetros de simulación y desarrollar versiones que puedan ejecutarse en otras plataformas de hardware no limitadas a la tecnología de NVIDIA.

REFERENCIAS

[1] Feynman R, Sands M, Leighton R. *The Feynman Lectures on Physics*. Volume II. New York: Basic Books; 2011.

[2] Dunmur D, Sluckin T. *Soap, Science, and Flat-Screen TVs: A History of Liquid Crystals*. Oxford: Oxford University Press; 2011.

[3] Palffy P. The diverse world of liquid crystals. *Physics Today*. 2007;60(9):54-59. doi: doi.org/10.1063/1.2784685.

[4] Tomilin MG, Povzun SA, Kurmashev AF, Gribanova EV, Efimova TA. The application of nematic liquid crystals for objective microscopic diagnosis of cancer. *Liquid Crystals Today*. 2001;10(2):3-5. doi: 10.1080/14645180110074819.

[5] Wang H, Xu T, Fu Y, Wang Z, Leeson M, Jiang J, Liu T. Liquid Crystal Biosensors: Principles, Structure and Applications. *Biosensors*. 2022;12(8):639-666. doi: 10.3390/bios12080639.

[6] Peng C, Turiv T, Guo Y, Wei QH, Lavrentovich OD. Command of active matter by topological defects and patterns. *Science*. 2016;354(6314):882-886. doi: 10.1126/science.aah6936.

[7] Hirst L. *Fundamentals of Soft Matter Science*. Boca Raton: CRC Press; 2013.

[8] Care CM, Cleaver DJ. Computer simulation of liquid crystals. *Reports on Progress in Physics*. 2005;68(11):2665-2700. doi: 10.1088/0034-4885/68/11/R04.

[9] Shendruk T, Yeomans J. Multi-particle collision dynamics algorithm for nematic fluids. *Soft Matter*. 2015;11(1):5101-5110. doi: 10.1039/C5SM00839E.

[10] Lee KW, Mazza MG. Stochastic rotation dynamics for nematic liquid crystals. *The Journal of Chemical Physics*. 2015;142(16):164110-164117. doi: 10.1063/1.4919310.

[11] Petersen M, Lechman J, Plimpton S, Grest G, Veld P, Schunk P. Mesoscale hydrodynamics via stochastic rotation dynamics: Comparison with Lennard-Jones fluid. *The Journal of Chemical Physics*. 2010;132(17):174106. doi: 10.1063/1.3419070.

[12] Westphal E, Singh S, Huang C, Gompper G, Winkler R. Multiparticle collision dynamics: GPU accelerated particle-based mesoscale hydrodynamic simulations. *Computer Physics Communications*. 2014;185(2):495-503. doi: 10.1016/j.cpc.2013.10.004.

[13] Howard M, Panagiotopoulos A, Nikoubashman A. Efficient mesoscale hydrodynamics: Multiparticle collision dynamics with massively parallel GPU acceleration. *Computer Physics Communications*. 2018;230:10-20. doi: 10.1016/j.cpc.2018.04.009.

[14] Howard M, Nikoubashman A, Palmer J. Modeling hydrodynamic interactions in soft materials with multiparticle collision dynamics. *Current Opinion in Chemical Engineering*. 2019;23:34-43. doi: 10.1016/j.coche.2019.02.007.

[15] Halver R, Junghans C, Sutmann G. Using heterogeneous GPU nodes with a Cabana-based implementation of MPCD. *Parallel Computing*. 2023;117:103033. doi: 10.1016/j.parco.2023.103033.

[16] Híjar H. Hydrodynamic correlations in isotropic fluids and liquid crystals simulated by multi-particle collision. *Condensed Matter Physics*. 2019;22(1):13601. doi: 10.5488/CMP.22.13601.

[17] Híjar H, Halver R, Sutmann G. Spontaneous Fluctuations in Mesoscopic Simulations. *Fluctuation and Noise Letters*. 2019;18(3):1950011. doi: 10.1142/S0219477519500111.

[18] Gompper G, Ihle T, Kroll DM, Winkler RG. *Multi-Particle Collision Dynamics - A Particle-Based Mesoscale Simulation Approach to the Hydrodynamics of Complex Fluids*. Berlin: Springer; 2009.

[19] Soyata T. *GPU Parallel Program Development Using CUDA*. Boca Raton: CRC Press; 2018.

[20] Cook S. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Waltham: Morgan Kaufmann; 2012.

[21] Downey A. *The Little Book of Semaphores*. Massachusetts: Green Tea Press; 2016.

ACERCA DE LOS AUTORES

Jorge Fierro. Es Ingeniero en Cibernética y Sistemas Computacionales por la Universidad La Salle México. Su trabajo de tesis consistió en desarrollar programas en paralelo para la simulación de fases intermedias de la materia. Ha trabajado como programador de *front end*, *back end*, minería de datos y optimización de procesos en la coordinación de Relaciones Públicas y Comunicación de la Universidad La Salle México desde 2018.

Humberto Híjar. Es Físico y Doctor en Ciencias (Física) por la U.N.A.M. Ha impartido clases de licenciatura y posgrado en la Facultad de Ciencias de la U.N.A.M. y en la Facultad de Ingeniería de la Universidad La Salle México. Es miembro del Sistema Nacional de Investigadores desde 2009. Sus tópicos de interés son el modelado teórico y computacional de la materia condensada suave (cristales líquidos y coloides) alejados del equilibrio.