VNIVERSIDAD NACIONAL
AVTÓNOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"Simulación de dinámicas moleculares usando unidades de procesamiento gráficas (GPUs)"

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS (COMPUTACIÓN)

P R E S E N T A:

José María Zamora Fuentes

DIRECTORES DE TESIS:

Luis Miguel de la Cruz Salas
Ismael Herrera Revilla

México, D.F. **2012.**

Índice general

Agradecimientos

Este trabajo fue apoyado por el proyecto IX 101110 de la convocatoria 2010 del Observatorio de Visualización Ixtli.

Capítulo 1

Introducción

El avance de las tecnologías de cómputo de alto desempeño, tanto en hardware como software, han permitido mejorar la aproximación de soluciones a problemas complejos de la ciencia y la ingeniería. Por ejemplo, en [SCC+09] se demuestra que el uso de las GPUs para resolver problemas de optimización usando algoritmos genéticos, reduce drásticamente el tiempo de cálculo de las soluciones. En dicho trabajo, las implementaciones están basadas en algoritmos muy sencillos y adaptables para diferentes arquitecturas de cómputo. Sin embargo, estas soluciones no son generalizables y en cada área de investigación se tienen que desarrollar nuevos algoritmos y nuevas técnicas de programación, para obtener beneficio de la aplicación de estas y otras tecnologías.

En este trabajo se abordará en particular la arquitectura GPU por sus siglas en inglés (Graphic Processor Unit). Esta arquitectura en un principio fue diseñada y desarrollada para solucionar problemas en el cómputo gráfico, sobre todo en el área de los videojuegos. Sin embargo en los últimos años los GPUs han encontrado una aplicación fundamental en el cómputo científico.

En este trabajo se revisan algunos algoritmos para dinámica molecular, los cuales se basan en la simulación de N-cuerpos. Desde hace varias décadas, esta área de estudio ha tenido mucho avance tanto en los algoritmos como en las implementaciones. Uno de los primeros trabajos en esta dirección es el de Allen y Tildesley [AT89]. Existen 3 componentes muy importantes que se destacan en esta área:

1. La solución del problema de N-Cuerpos requiere de un cálculo numérico intensivo, por lo que es necesario optimizar los algoritmos reduciendo lo más posible el número de operaciones.

2. Al mismo tiempo que se reduce el número de operaciones, se requieren generar soluciones cada vez más precisas, con el objetivo de que los resultados puedan mapearse de manera directa a lo que se observa en el mundo real. Por lo tanto, se necesitan muchas variables de medición para describir fielmente los procesos que suceden en una dinámica molecular.

3. La aplicación correcta de arquitecturas de cómputo de alto desempeño, como los

GPUs en esta área, es muy importante pues es a través de éstas que se pueden generar soluciones en tiempos razonablemente cortos, véase por ejemplo [vMAF+08].

Tanto en el punto 1 como en el 2, el impacto de las ciencias de la computación son vitales para el desarrollo de la química computacional. El estudio exahustivo de nuevos algoritmos, así como de arquitecturas de cómputo recientes, permiten generar herramientas muy útiles para obtener soluciones precisas y en tiempos cortos, véase [AAS+07]. No obstante, se sigue teniendo el desafío de aplicar de manera óptima dichas arquitecturas de cómputo, de tal manera que se tengan las implementaciones más óptimas posibles de los algoritmos.

1.1. Las simulaciones moleculares

Una simulación molecular se basa en la interacción de N partículas, las cuales siguen ciertas leyes de la física. En este tipo de simulaciones, como es obvio, el número N de partículas es finito. El número finito de partículas tiene un efecto importante en las simulaciones, por lo que ha sido un tema de estudio por parte de los expertos.

Por ejemplo, en [WJ57, AW57] se hace un estudio de la transición entre fases usando un modelo molecular de partículas duras. En este estudio se notó que los cúmulos de partículas simulados, se "redondeaban", es decir, se acumulaban en formas "esféricas". Por otro lado, Alder y Wainwright encontraron que las transiciones entre la fase líquida y la sólida, sobre la ecuación de estado, están conectados por un ciclo (del inglés "loop") continuo [AW57]. La figura 1.1 muestra el cambio de la presión con respecto a la densidad para un sistema en estado líquido. Se observa que la presión toma los mismos valores para diferentes densidades, a esto se le conoce como loop. En [AW62] se dice que dicho loop es una reminiscencia del "loop" de Van Der Walls y la figura 1.1 muestra uno de estos loops generado por un sistema con $N = 256$, usando el software desarrollado en este trabajo. Además, a la gráfica de la figura 1.1, se le conoce como isoterma, debido a que se calcula a una temperatura constante, y es una curva que describe la presión como función de la densidad. Esta curva también se puede calcular con respecto del volumen, debido a la relación $\rho = N/V$. El significado de este "loop" y su dependencia de N, es decir del tamaño del sistema, fue explicado posteriormente por Mayer y Wood en [MW65]. Estos autores notaron que la apariencia de una porción de la isoterma con presiones negativas, implica que dentro de la caja de simulación las partículas se aglutinan en forma de una gota. Lo anterior sucede sólo para sistemas pequeños (para mayor información de estos resultados véase [AT89]).

La pregunta ¿en que condiciones esta gota es estable?, se volvió de gran interés, particularmente en estudios de simulación molecular con nucleación, véase [RBK78]. Este es un tópico que ha provocado mucha atención en los últimos años. Mientras que la apariencia de estos fenómenos en un "loop" sobre la gráfica del Potencial químico contra la densidad, puede ser considerado un artificio generado por los efectos del tamaño finito, desde un punto de vista práctico esto permite el cálculo de propiedades termodinámicas sobre la gota formada. Además, con el siempre creciente interés en la

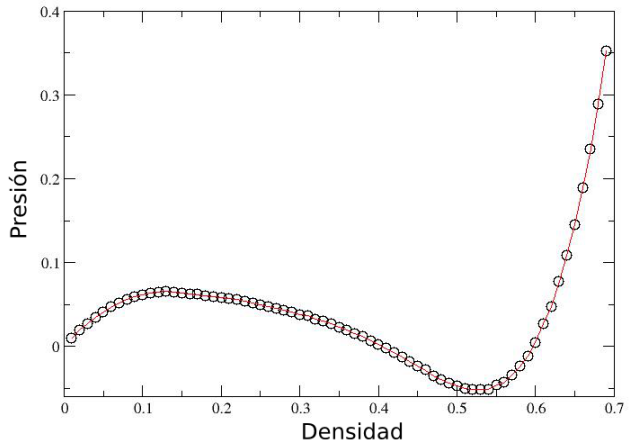


Figura 1.1: *Gráfica de presión vs. densidad a temperatura constante generada mediante una simulación típica usando el software desarrollado en este trabajo. Esta gráfica muestra la forma de un* `loop` *de* `Van Der Waals`.

nanotecnología, el comportamiento de las fases de la materia y la ecuación de estado de sistemas pequeños, se ha convertido en un campo con un gran apoyo dentro de la ciencia multidisciplinaria, permitiendo por ejemplo la caracterización de dimensiones porosas, cavidades o formación de *droplets* (gotas) en algunos materiales. Por estas razones, el estudio de la estabilización de gotas y "loops" relacionados con la ecuación de estado son muy importantes en la compresión de la naturaleza de los materiales. Grandes aportaciones al estudio de este problema se han realizado usando el modelo de Ising (un método de simulación molecular), véase [SJC$^+$00, PH01, NH03], donde la naturaleza discreta de la magnetización hace posible el estudio de sistemas extremadamente grandes, y las propiedades simétricas permiten algunas simplificaciones en su análisis teórico.

Para modelos continuos, los esfuerzos se han concentrado principalmente en fluidos donde se utiliza el potencial de Lennard-Jones. En este tipo de simulación el consumo de recursos de cómputo es bastante considerable, como se verá en capítulos posteriores.

En la actualidad, tanto la velocidad como la capacidad de almacenamiento de las computadoras ha crecido de tal manera que, es posible probar ciertas predicciones teóricas simulando sistemas muy grandes. Sin embargo, dentro del campo de la simulación molecular, aunque se han hecho grandes avances multidiscplinarios, algunas de las preguntas fundamentales sobre la naturaleza de algunos fenómenos físicos siguen sin respuesta.

1.2. Objetivos

El objetivo general de este trabajo es encontrar estructuras moleculares dentro de una caja de simulación cúbica cambiando la densidad de un sistema en un ensamble NVT. El ensamble NVT es aquel que contiene todas las configuraciones de un sistema encerrado en una caja, en el cual se mantiene el número de partículas (N), el volumen

(V) y la temperatura (T) con valores fijos durante toda la simulación. Las estructuras se encuentran cuando se grafica la traza del tensor de presión del sistema contra la densidad, véase [MSE06] y sección 3.2.6.

Para obtener esta gráfica actualmente se requiere aproximadamente de un mes de simulación como se verá en los resultados. Uno de los objetivos principales de este trabajo es disminuir este tiempo de cálculo, usando unidades de procesamiento gráfico (GPUs).

Los objetivos de esta tesis se pueden resumir como sigue:

Suponemos una caja de simulación, es decir un cubo acotado en sus tres ejes cartesianos x,y,z. Ahora dentro del cubo metemos un conjunto finito de esferas (partículas o moléculas). Cada esfera tiene propiedades particulares como masa, volumen, etc. Además estas esferas se pueden mover libremente dentro de la caja respondiendo a las leyes físicas establecidas por Newton. También el sistema responderá a las leyes de la termodinámica clásica y por lo tanto se podrán medir propiedades como la temperatura, la presión, etc. También definimos el tiempo de simulación, como el tiempo que dejamos a las esferas moverse libremente dentro de la caja de simulación. Las preguntas que se reponden en este trabajo se listan a continuación:

1. *¿Se forman cúmulos de partículas con algún tipo de estructura geométrica?*

2. *¿Se puede reducir el tiempo de cómputo usando GPUs?*

3. *¿Existe alguna relación entre la presión, la densidad y la formación de estructuras en los cúmulos de esferas.?*

1.3. Metas y organización de la tesis

La metas principales de este trabajo son:

1. Analizar los algoritmos de simulación molecular para sistemas basados en el potencial de Lennard-Jones.

2. Implementar estos algoritmos para: (a) Unidades de Procesamiento Central (CPU); y para (b) Unidades de Procesamiento Gráfico (GPU)

3. Analizar el desempeño de los algoritmos cuando se ejecutan en ambas arquitecturas.

4. Determinar las condiciones bajo las cuales se obtienen distintos tipos de estructuras moleculares.

5. Hacer uso de la programación orientada a objetos, para generar una primera versión de un software que pueda ser re-utilizado en otras investigaciones.

Adicionalmente, se busca que este trabajo documente de una manera simple y directa, los procesos y algoritmos requeridos en una simulación molecular. Por esta razón, se hace una revisión de algunos aspectos teóricos de la dinámica molecular, sin profundizar en ellos, de tal manera que esta tesis sea autocontenida. También, se documentan y detallan algunas de las partes más importantes de los algoritmos y programas, de tal manera que se puedan usar en trabajos futuros por otros autores.

La organización de la tesis es como sigue:

- Capítulo 2. Se dan algunos antecedentes de la simulación molecular y del uso de arquitecturas de alto desempeño en esta área.
- Capítulo 3. Se da una descripción teórica de la dinámica molecular basada en el problema de N-Cuerpos.
- Capítulo 4. Se describen los algoritmos requeridos en una simulación molecular.
- Capítulo 5. Se muestra la forma en que se implementaron los algoritmos usados en este trabajo.
- Capítulo 6. Se describe el uso de las GPUs en problemas de cómputo científico y se muestran algunos programas básicos para probar el desempeño de esta arquitectura. Finalmente, se describe en detalle la implementación del cálculo de fuerzas en el GPU, que es el proceso que toma más tiempo durante la simulación.
- Capítulo 7. Se muestran los resultados obtenidos en las simulaciones en donde se encuentran varias estructuras moleculares típicas. También, se describe un análisis de la aceleración obtenida cuando se hacen algunos de los cálculos en el GPU.

Capítulo 2

Antecedentes

Las soluciones a problemas de muchas áreas de la ciencia, por métodos computacionales se han expandido rápidamente en muchos campos de estudio, debido al rápido y estable crecimiento del poder de cómputo en los últimos 50 años, como lo muestra la figura 2.1. La relación de rendimiento-precio ha incrementado en un orden de magnitud aproximadamente cada 5 años, y no hay signos de algún cambio en esta tendencia. Todo lo contrario, con la incursión de nuevas tecnologías es posible predecir que en el año 2019 se puede alcanzar la barrera de los exaflops. Esto significa que podrán ser simulados problemas de gran escala, por ejemplo sistemas moleculares muy complejos durante un periodo de tiempo muy largo. De igual manera será posible manejar interacciones cada vez más complejas en sistemas como el DNA. Se vislumbra que esto se podrá hacer en al menos una década.

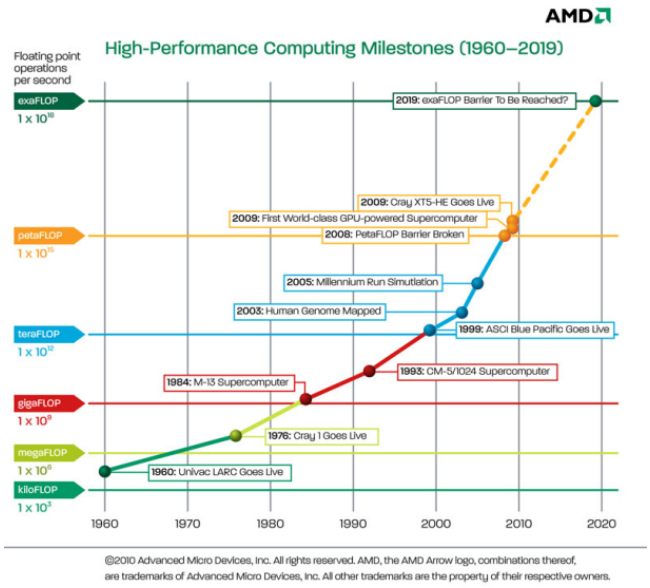


Figura 2.1: *Crecimiento en FLOPS del cómputo de alto rendimiento desde 1960 y la proyección al 2019 [Advanced Micro Device, Inc. Todos los derechos reservados].*

En este trabajo, se desea realizar un estudio de algunos sistemas moleculares, que son de interés en diferentes áreas de estudio. Desde hace décadas, ha habido una gran variedad de técnicas usadas para acelerar las simulaciones en Dinámica Molecular. Se han usado clusters (cúmulos de computadoras) para HPC (High Performance Computing, por sus siglas en inglés) y también arquitecturas de computadoras bastante novedosas. En general es posible clasificar estas estrategias en dos categorías: de grano grueso (*coarse-grained*) y de grano fino (*fine-grained*).

Las arquitecturas en la categoría de grano grueso, incluyen supercomputadoras de propósito general, clusters de PC's o Bewolfs, hasta nubes (*grids*) de computadoras. Usualmente las supercomputadoras como la Blue Gene, que en 2006 alcanzó el primer lugar en el TOP500 con 478.2 teraFLOP/s, en Dinámica Molecular pueden dar un rendimiento asombroso [GZP+06]. Sin embargo, en algunos casos pueden ser bastante caras e inaccesibles para algunos investigadores de países poco desarrollados. El Lawrence Livermore National Laboratory, donde reside la supercomputadora Blue Gene, reportó en 2002 un presupuesto anual de aproximadamente 1.5 billones de dólares y mantiene a 8300 empleados. Sin embargo la construcción de clusters caseros sigue siendo la opción mas viable para que los investigadores tengan acceso a resultados con un cómputo modesto. Y por lo tanto la opción de usar clusters [BCX+06] y nubes [PBC+03, TAKB06] en Dinámica Molecular, sigue siendo utilizada para dar mayor accesibilidad al HPC a un bajo costo. Es importante destacar que el hecho de usar clusters sufre problemas de escalabilidad para crecer el número de procesadores debido a las altas latencias en las comunicaciones entre computadoras. Proyectos basados en nubes como son Folding@home [PBC+03] y Predictor@Home [TAKB06] han atraído a varios cientos de miles de voluntarios alrededor de todo el mundo para colaborar con tiempo de CPU en sus PCs y Playstations. Desafortunadamente, estos recursos de cómputo voluntario solamente permiten comunicación entre servidores maestros y clientes, es por eso que este enfoque es sólo deseable para simulaciones con un número bastante grande de configuraciones de partículas, para pasos de tiempos cortos de tal manera que las escalas de integración son grandes.

Algunas de las arquitecturas en la categoría de grano fino son: arquitecturas de propósito general, arquitecturas reconfigurables. En esta última categoría se pueden mencionar Anton, FASTRUN, MDGRAPE y MD Engine (véase [SCE+07]). Estas arquitecturas pueden proveer de una rapidez bastante significativa, haciendo uso de una alta densidad aritmética, en las cuales es posible ejecutar algoritmos de dinámica molecular para configuraciones muy específicas. Cada unidad aritmética puede ser específicamente diseñada para el cálculo de interacciones y potenciales muy particulares, por ejemplo, un hardware que sólo hace cálculo de potenciales Lennard-Jones y que además sólo mida la presión (véase [SCE+07]). Sin embargo, estas arquitecturas están limitadas a un sólo algoritmo, por lo que su flexibilidad para correr una variedad de algoritmos (que constantemente sigue creciendo) para simulaciones de dinámica molecular es prácticamente inexistente.

Las arquitecturas reconfigurables están basados en programación lógica ó recableado de hardware. Un ejemplo son los arreglos de compuertas programables (FPGAs, por sus

siglas en inglés). Estas son generalmente más rápidas que las arquitecturas de simulación de propósito general [AAS+07]. Son flexibles, pero la configuración debe ser cambiada para cada algoritmo, lo cual es generalmente mucho más complicado que escribir código para una arquitectura programable.

Todos estos enfoques para realizar una simulación molecular, pueden ser vistos como algoritmos que intentan obtener la mayor potencia de cómputo intensivo al menor costo económico. La principal ventaja de los GPUs comparados con las arquitecturas antes mencionadas es que son componentes bastante flexibles. En particular la mayoría de los usuarios con una computadora normal ya tienen acceso a una tarjeta gráfica bastante moderna. Para estos usuarios esta solución tiene cero costo. Además, la instalación de la misma es trivial (la mayoría son *plug & play*). Para escribir software que aproveche las características de una GPU, se requiere conocimiento bastante especializado. Sin embargo, la aparición de nuevos modelos de programación de alto nivel, tales como CUDA, ofrecen un ambiente de programación similar a C, lo que provoca que la curva de aprendizaje sea un poco más plana.

2.1. Dinámica Molecular Computacional

La química computacional es un área de estudio cuyo interés va en ascenso. En esta disciplina los problemas son resueltos por métodos computacionales. Lo que se modela son sistemas de partículas distribuidas en un espacio físico. Cada partícula tiene un conjunto de propiedades y en una dinámica molecular lo que se desea es calcular las interacciones entre partículas y obtener propiedades, como la energía o la presión. Estas propiedades así calculadas se comparan con datos que son determinados experimentalmente. Esta comparación se hace para validar el modelo numérico y computacional. Una vez que el modelo se ha validado, éste puede ser usado para estudiar relaciones entre parámetros del modelo y hacer predicciones desconocidas, en el mejor de los casos medir las propiedades no medibles.

A la química le concierne el estudio de propiedades de substancias o sistemas moleculares en términos de átomos, el desafío básico es encarar la química computacional para describir o predecir lo siguiente:

1. La estructura o estabilidad de un sistema molecular,

2. El comportamiento de la energía en diferentes estados del sistema molecular,

3. Los procesos de reacción dentro de los sistemas moleculares.

Todo lo anterior en términos de la interacción a nivel atómico. De los tres problemas básicos listados anterioremente el más simple es el primero y el objetivo es la predicción de la estructura del sistema en el estado de más baja energía. La estructura es la forma en la que están ordenadas las partículas dentro del espacio que ocupa el sistema.

En este trabajo en particular, se intentan contestar dos preguntas básicas.

- ¿Existe alguna relación entre una estructura molecular y una propiedad como la presión?

- Si uno varía alguna propiedad con respecto a la densidad del sistema, ¿La estructura molecular cambia?

Aquí la densidad se define como $\rho = N/V$, con N el número de partículas y V es el volumen que ocupa el sistema.

Respecto a los otros dos problemas mencionados antes, se puede decir que son más complejos. En este trabajo se llega a obtener la energía mediante los algoritmos que se han implantado. Sin embargo, esta energía no da una conclusión macroscópica, es decir, no se puede saber que efecto tienen los resultados atómicos en la materia que vemos con el ojo humano. Al resolver el problema de la energía seguimos a nivel de partículas, es por eso que tenemos que extrapolar nuestros resultados a un punto macroscópico, donde los procesos dinámicos de la materia se perciben, esto es resuelto en el tercer problema. El tercer problema no se trata en esta tesis.

Los sistemas químicos son generalmente bastante heterogéneos y complejos para ser tratados por métodos de análisis teórico. Como sabemos la materia tiene 3 estados clásicos (gas, líquido y sólido) y actualmente la ciencia sólo posee dos maneras de abordar el estudio de dichos estados a nivel molecular o atómico:

1. Una es la mecánica estadística clásica, que es una rama de la física la cual deriva propiedades de la materia que emergen a nivel macroscópico a partir de la estructura atómica y de la dinámica microscópica de la misma. Algunas de las propiedades que emergen a nivel macroscópico en la materia son: la temperatura, la presión, los flujos, las constantes magnéticas y deléctricas, etc. Estas propiedades son esencialmente determinadas por la interacción de muchas partículas, átomos o moléculas. El punto clave de la mecánica estadística es la introducción de probabilidades dentro de la fisica intrínseca al problema y conectar ello con la cantidad de de entropía fundamental[Gar08].

2. La otra alternativa es la mecánica cuántica, la rama que puede predecir el comportamiento de la materia a partir de sus propiedades electrónicas y sus energías atómicas, esta rama de la física trata las partículas y la energía como elementos discretos, fuera del contínuo y la estructura molecular que forman varias partículas depende de las propiedades de carga y masa de las partículas elementales (protón, electrón, etc.) [Mor00].

De acuerdo con [AT89], el tratamiento de sistemas moleculares en la fase de gas por métodos de la mecánica cuántica es sencillo, e inclusive si existe una aproximación de la mecánica estadística, el problema se vuelve trivial. Esto es debido a la posibilidad de reducir el número de partículas con base en la baja densidad de un sistema en la fase de gas. En el estado sólido cristalino, el enfrentar problemas con mecánica cuántica o estadística es posible por la reducción de un sistema de muchas partículas a uno de pocas

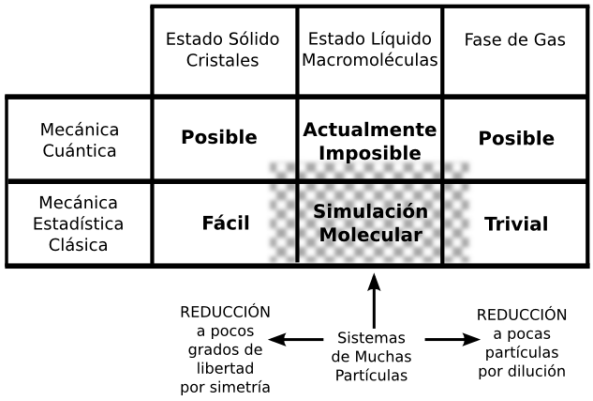


Figura 2.2: *Clasificación de los sistemas moleculares. Los sistemas ubicados en el área sombreada son los mas adecuados de tratar con simulaciones por computadora.*

partículas por propiedades de simetría del estado sólido. Entre estos dos extremos existen los líquidos, como las macromoléculas, soluciones, sólidos amorfos, etc. Este estado de la materia sólo se puede abordar como un sistema de muchas partículas. No hay una reducción simple, debido a que todos los grados de libertad, como traslación y rotación, aportan información para describir las propiedades macroscópicas del sistema. Estos hechos tienen dos consecuencias en el estudio de sistemas tipo líquido:

1. El estudio analítico de estos sistemas es prácticamente imposible. Como alternativa se tiene la *simulación numérica* del comportamiento de un sistema molecular sobre una computadora, lo cual,

2. Se producirá un *ensamble estadístico* de configuraciones que representan el estado del sistema.

Un ensamble estadístico se refiere a una colección de configuraciones del sistema obtenidas en diferentes pasos de tiempo. En este trabajo se usa configuración como sinónimo de estructura del sistema.

La figura 2.2 muestra la amplia aplicación de los métodos de simulación por computadora en la química.

2.2. Simulación de sistemas moleculares

Los dos problemas básicos en una simulación de un sistemas tipo líquido son:

1. El número de configuraciones del sistema molecular debe ser muy grande para tener mayor información del mismo.

2. La precisión del modelo molecular y sus interacciones atómicas.

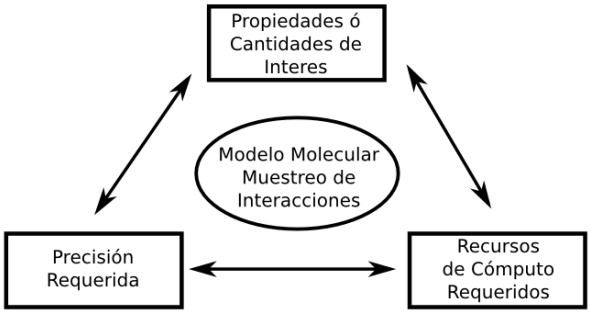


Figura 2.3: *La elección del modelo molecular, las interacciones moleculares ó campo de fuerza y el tamaño de muestreo dependen de 1) las propiedades que uno está interesado en medir (espacio por buscar), 2) precisión requerida acerca de la predicción, y 3) el poder de cómputo disponible para generar el ensamble.*

La simulación de sistemas moleculares a temperaturas diferentes de cero, requieren la generación de un conjunto de configuraciones estadísticamente representativas, esto es conocido como *ensamble*. Las propiedades de un sistema están definidas como promedios en el ensamble ó integrales en el espacio de configuraciones. Este espacio de configuraciones se conoce también como espacio fase. Para un sistema de muchas partículas el promediar o integrar involucrará varios grados de libertad de cada partícula. La traslación está definida por la ubicación espacial de las partículas, mientras que la rotación está definida cuando la partícula tiene una forma diferente a una esfera. La precisión de un modelo molecular depende completamente del campo de fuerza ó interacción definido para las partículas, en secciones posteriores se dará una definición mas formal, mientras tanto es importante saber que existe una gran variedad de estos. Algunas interacciones son muy complejas de implementar, pero muy precisas, otras muy sencillas pero poco precisas, algunas que ocupan muchos recursos computacionales y son muy complejas de implementar, etc.

Cuando estudiamos un sistema molecular por simulaciones computacionales existen tres factores que deben ser considerados (véase figura 2.3).

1. Las propiedades de un sistema molecular que uno está interesado en medir (presión, temperatura, densidad, energía, etc.) y el espacio de configuraciones. Se deben definir las configuraciones relevantes que deben ser estimadas.

2. La precisión requerida por las propiedades especificadas en el punto 1.

3. El tiempo de cómputo también debe ser estimado. La estimación incorrecta de este punto puede acabar en simulaciones suficientemente largas como la edad del universo.

La tabla 2.1 muestra el desarrollo de las aplicaciones de simulaciones de dinámica molecular en química, *Alder* y *Wainwright* pioneros del método usaron discos duros en 2 dimensiones. *Rahman* quien puede realmente ser considerado el padre del campo, simuló el líquido de Argon en 1964, el líquido Agua en 1971, y los conductores superiónicos en 1978. *Rahman* también hizo contribuciones a la metodología y a estimular la diseminación de las técnicas de simulación en la academia. En los setentas, se realizó la transición de los líquidos atómicos a los moleculares, moléculas rígidas como el agua en 1971, alcanos flexibles en 1975, y pequeñas proteínas como el inhibidor trypsin en 1977.

Los métodos de simulación de sales fundidas (1971) requirieron el desarrollo de métodos para manejar interacciones de Coulomb de larga escala. En los ochentas se realizaron simulaciones de biomoléculas con un tamaño mayor de solución acuosa [AT89].

Año	**Sistema**	**Tiempo de Simulación (seg.)**	**Tiempo de CPU (horas)**
1957	Discos Duros en dos dimensiones		
1964	Líquido Monoatómico	10^-11	0.05
1971	Líquido Molecular	$5x10^-12$	1
1971	Sal fundida	10^-11	1
1975	Polímero pequeño simple	10^-11	1
1977	Proteína en vacío	2^-11	4
1982	Membrana Simple	2^-10	4
1983	Proteína en un cristal acuoso	2^-11	30
1986	DNA en solución acuosa	10^-10	60
1989	Proteína-DNA compleja en solución	10^-10	300
1989	Polímeros largos	10^-8	10^3
1989	Reacciones	10^-4	10^7
1989	Interacciones Macromoleculares	10^-3	10^8
1989	Plegado de Proteínas	10^-1	10^9

Cuadro 2.1: *Historia de las aplicaciones de la simulación de dinámicas moleculares. [AT89].*

Capítulo 3

Principios de Dinámica Molecular

Las definiciones de un sistema de *N-Cuerpos*, como veremos en este capítulo nos ayudan a simular sistemas muy importantes en la ciencia como son sistemas planetarios, sistemas moleculares, sistemas de partículas, entre otros. En este capítulo se describirá la teoría en la que se basa el estudio de un sistema de N-Cuerpos, particularmente para problemas de Dinámica Molecular

3.1. Descripción Matemática del Problema de N-Cuerpos

Una simulación de N-Cuerpos aproxima la evolución de un sistema de cuerpos a través del tiempo, es decir, cada cuerpo o elemento del sistema interactúa con todos los demás en un tiempo dado. Esto resulta en un cambio del sistema total y entonces sus elementos modificarán sus propiedades en cada paso de tiempo. Para describir esto formalmente, a continuación realizaremos algunas definiciones.

Definición 1. *Un cuerpo* $\boldsymbol{B}$ *es un objeto abstracto el cual tiene* $\boldsymbol{m}$ *propiedades estáticas y* $\boldsymbol{k}$ *propiedades dinámicas, tal que existen los conjuntos* $P_{estáticas} = \{p_1, p_2, ..., p_m\}$ *y* $P_{dinámicas} = \{p_1, p_2, ..., p_k\}$.

Definición 2. *Las propiedades estáticas* ($P_{estáticas}$) ***no*** *cambian al evolucionar el sistema.*

Definición 3. *Las propiedades dinámicas* ($P_{dinámicas}$) ***siempre*** *cambian al evolucionar el sistema.*

Definición 4. *El conjunto total de las propiedades del i-ésimo cuerpo* (B_i)*, se denota como* $P_i = P^i_{estáticas} \cup P^i_{dinámicas}$. *Además* $P_i \neq P_j$ *para toda* $i \neq j$.

Corolario 1. *De la definición 1 se deduce que no pueden existir dos cuerpos iguales y ésta es, precisamente, la restricción que permite diferenciar cada uno de los cuerpos en cada paso de tiempo.*

Definición 5. *Definiremos un universo o sistema U, el cual es una colección de N-cuerpos denotado como $U = \{B_1, B_2, ..., B_N\} = \{B_i\}, 1 \leq i \leq N$, donde $N \in \mathbb{N} \mid N \geq 2$.*

Definición 6. *Denotaremos t_i como el tiempo actual y a la operación $t_i + \Delta t$ como un cambio en el tiempo. Además utilizaremos el termino **paso de tiempo** como Δt para referirnos textualmente a esta operación.*

Las definiciones anteriores colocan cada uno de los elementos de nuestro sistema con sus características y restricciones bien establecidas, ahora definiremos como interaccionan los elementos dentro de nuestro sistema.

Definición 7. *Las interacciones entre los N-Cuerpos están regidas por una función $T(B_i, B_j) : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$ para $1 \leq i \leq N$ y $1 \leq j \leq N$. T es conocida como **función de transición** o **relación de interacción**. Esta relación cambia las propiedades del sistema respecto al tiempo, ya que se debe calcular T en cada Δt.*

Corolario 2. *De la Definición 6 y la definicón 7 deducimos que después de calcular T para todos los cuerpos en un Δt dado, el conjunto de propiedades dinámicas $P_{dinámicas}$ de cada cuerpo cambiará en al menos uno de sus elementos. A esta consecuencia es lo que en física se conoce como evolucionar un sistema en el tiempo.*

Como ejemplo construiremos un modelo para un sistema planetario de N-Cuerpos En este caso, cada cuerpo B_i tiene una sola propiedad estática la cual es su masa denotada como m_i. En este modelo, esta propiedad no cambia al evolucionar el sistema porque los cuerpos no cambian de tamaño. Así, formalmente tenemos que $P^i_{estáticas} = m_i$.

La propiedades dinámicas que cambiarán en cada cuerpo respecto al tiempo serán sus posiciones en coordenadas Cartesianas (x_i,y_i,z_i), y sus velocidades $\mathbf{v}_i$. Este cambio es debido a las aceleraciones, y más propiamente, a las fuerzas que son ejercidas sobre cada uno de los cuerpos. Esto genera el movimiento físico de los cuerpos en el espacio. Por lo tanto, $P^i_{dinámicas} = \{x_i, y_i, z_i, \mathbf{v}_i\}$.

La relación de interacción entre los planetas esta definida como:

$$T(B_i, B_j) = f_{ij} = G\frac{m_i m_j}{\parallel \mathbf{r}_{ij} \parallel^2}\frac{\mathbf{r}_{ij}}{\parallel \mathbf{r}_{ij} \parallel} \quad , \tag{3.1.1}$$

donde G es la constante gravitacional, m_i es la masa del cuerpo B_i, m_j es la masa del cuerpo B_j y $\mathbf{r}_{ij}$ es la distancia entre los cuerpos B_i y B_j, véase [EP05].

Pero esta función por sí misma no hace evolucionar el sistema. Faltará usar T para obtener las nuevas propiedades dinámicas del sistema y para esto necesitamos integrar la ecuación 3.1.1. La definición de un integrador se dará en la sección 3.2.2, pero por el momento nos basta saber que la función T de evolución en el tiempo está compuesta de dos pasos: 1) calcular f_{ij} y 2) integrar T para obtener las nuevas propiedades dinámicas.

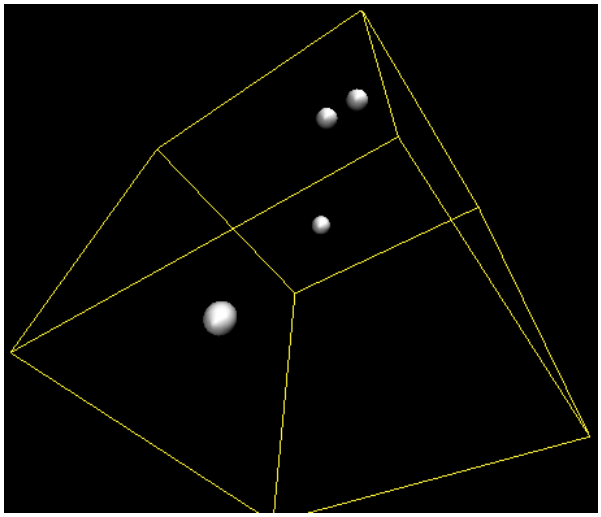


Figura 3.1: *Modelación de un sistema de N-Cuerpos encerrados en una caja.*

Después de implementar el modelo, lo vemos visualizado por una computadora en la figura 3.1. En este caso los planetas ya han evolucionado algunos pasos. Note que el sistema modelo debe estar encerrado por una caja la cual tiene ciertas propiedades y además es necesario definir algunas condiciones de frontera, que dependen del problema particular que se esté abordando. En este trabajo se usarán condiciones de frontera periódicas para Dinámica Molecular, y serán descritas más adelante.

Usando las definiciones y el ejemplo de esta sección es posible bosquejar un algoritmo para resolver numéricamente el problema de N-Cuerpos. El algoritmo 1 describe este proceso.

Algorithm 1 Algoritmo para N-Cuerpos

```
1: N = Número de Cuerpos
2: T_max = Total de pasos
3: for t = 1 to T_max do
4:    for i = 1 to N  do
5:       for j = 1 to N  do
6:          Calcular f(B_i, B_j)
7:       end for
8:    end for
9:    for i = 1 to N  do
10:      Integrar alguna ecuación de movimiento de B_i
11:   end for
12:   for i = 1 to N  do
13:      Actualizar las Propiedades Dinámicas de B_i
14:   end for
15: end for
```

Vale la pena destacar que en el algoritmo 1, el costo computacional de los ciclos ubicados en las líneas 4 y 5 provocan que calcular f para todos los cuerpos sea $O(N^2)$.

Además, integrar las ecuaciones de movimiento y actualizar las propiedades dinámicas, líneas 10 y 13 respectivamente, es $\Theta(N)$, ya que esto requerirá realizar algunas operaciones para cada cuerpo B_i.

Con esto tenemos descrito el caso general del problema de N-Cuerpos. En la siguiente sección abordaremos a fondo el caso particular de las dinámicas moleculares.

3.2. Problema de N-Cuerpos en Dinámica Molecular

3.2.1. Teoría sobre Dinámica Molecular

Las dinámicas moleculares están basadas en la física clásica y, por consecuencia en las ecuaciones de Newton para el movimiento de N cuerpos, véase [Sim]. En este caso un cuerpo es interpretado como una partícula. Así pues, cada partícula en el sistema es visto como un objeto con diferentes características intrínsecas a su comportamiento, las cuales son *posición*, *velocidad* y *aceleración*, definidas dentro del conjunto de *propiedades dinámicas*, y a la masa como una *propiedad estática*.

La ecuación de movimiento del i-ésimo objeto está definida por la segunda ley de Newton que se escribe como:

$$\mathbf{f_i} = m_i \ddot{\mathbf{r}}_i = m_i \mathbf{a}_i \quad , \tag{3.2.1}$$

donde $\mathbf{a}_i$ representa la aceleración de la *i-ésima* partícula. Aquí $\ddot{\mathbf{r}}_i$ representa la segunda derivada con respecto al tiempo de la posición $\mathbf{r}_i$ de la partícula.

La posición de cada partícula está ubicada en un espacio tridimensional de coordenadas Cartesianas, y está denotada por $\mathbf{r}_i$. Así, la distancia entre dos partículas la definimos en la siguiente ecuación:

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} \quad . \tag{3.2.2}$$

En este trabajo, para definir una relación de interacción entre cada una de las partículas, implementaremos un modelo molecular basado en el potencial de Lenard-Jones (véase [Jon24]), el cual se escribe como:

$$U_{ij} = 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \quad . \tag{3.2.3}$$

donde ϵ es la profundidad del potencial, σ es la distancia (finita) a la cual el potencial interpartícula es cero y r_{ij} es la distancia entre dos partículas. En la figura 3.2 se muestra como ϵ modifica la profundidad del pozo del potencial, mientras que σ mueve la gráfica a través del eje horizontal.

El desarrollo de las ecuaciones mostrado a continuación esta basado en [Jon24], y nos parece importante mostrarlo para entender los algoritmos del capítulo 4.

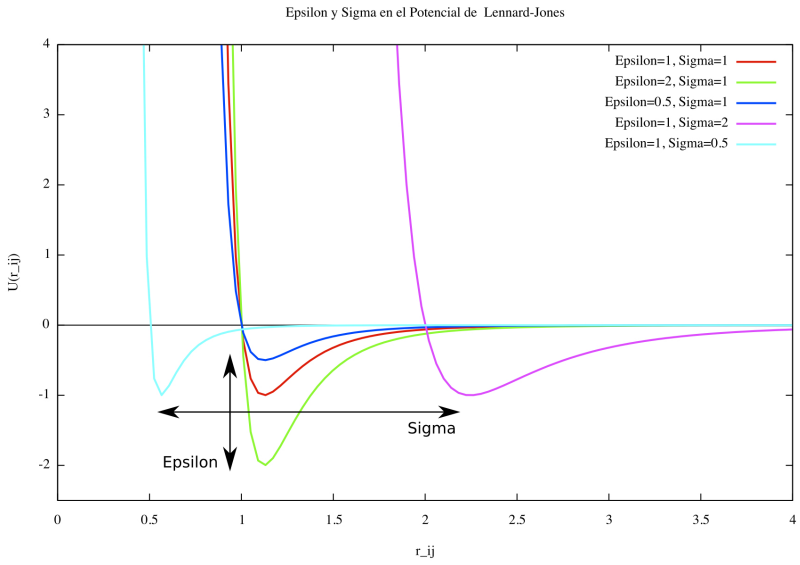


Figura 3.2: *Influencia de los parametros ϵ y σ en el pozo y la distancia entre las partículas*

Usando unidades reducidas (vease Apéndice A), tenemos que $r_{ij}^* = \frac{r_{ij}}{\sigma_0}$ y $U_{ij}^*(r_{ij}) = \frac{U_{ij}(r_{ij})}{\epsilon_0}$, donde σ_0 y ϵ_0 son los parámetros moleculares de referencia de un fluido. Si lo anterior lo sustituimos en 3.2.3, entonces obtenemos:

$$U_{ij}^*(r_{ij}) = \frac{4\epsilon}{\epsilon}\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] = 4\left[\left(\frac{1}{r_{ij}^*}\right)^{12} - \left(\frac{1}{r_{ij}^*}\right)^{6}\right] \tag{3.2.4}$$

Por comodidad en la notación, en adelante usaremos r_{ij}^* como r_{ij} y U_{ij}^* como U_{ij}.

Por otro lado, el vector de fuerzas entre la partícula i y la partícula j, se puede descomponer como:

$$\mathbf{F}_{ij} = F_{ij}^x \hat{e_x} + F_{ij}^y \hat{e_y} + F_{ij}^z \hat{e_z} \tag{3.2.5}$$

donde $\hat{e_x}$, $\hat{e_y}$ y $\hat{e_z}$ son los vectores unitarios en la dirección de los ejes coordenados. Ahora podemos calcular la fuerza a partir del potencial con la siguiente ecuación:

$$\mathbf{F}_{ij} = -\vec{\nabla} U_{ij}(r_{ij}) = -\frac{\partial U_{ij}(r_{ij})}{\partial(x_i - x_j)}\hat{e_x} - \frac{\partial U_{ij}(r_{ij})}{\partial(y_i - y_j)}\hat{e_y} - \frac{\partial U_{ij}(r_{ij})}{\partial(z_i - z_j)}\hat{e_z} \tag{3.2.6}$$

renombrando $x_{ij} = x_i - x_j$, $y_{ij} = y_i - y_j$, $z_{ij} = z_i - z_j$, entonces podemos escribir 3.2.2 y 3.2.6 como sigue:

$$r_{ij} = \sqrt{x_{ij}^2 + y_{ij}^2 + z_{ij}^2} \tag{3.2.7}$$

$$\mathbf{F}_{ij} = -\frac{\partial U_{ij}(r_{ij})}{\partial x_{ij}}\hat{e_x} - \frac{\partial U_{ij}(r_{ij})}{\partial y_{ij}}\hat{e_y} - \frac{\partial U_{ij}(r_{ij})}{\partial z_{ij}}\hat{e_z} \tag{3.2.8}$$

Ahora, si escribimos la componente en x de la siguiente manera:

$$\frac{\partial U_{ij}(r_{ij})}{\partial x_{ij}} = \frac{\partial U_{ij}(r_{ij})}{\partial r_{ij}} \frac{\partial\, r_{ij}}{\partial x_{ij}} \tag{3.2.9}$$

y usando la ecuación 3.2.4 y 3.2.7 se obtiene:

$$\frac{\partial U_{ij}(r_{ij})}{\partial\, r_{ij}} = 4\left(-12 r_{ij}^{-13} + 6 r_{ij}^{-7}\right) \tag{3.2.10}$$

$$\frac{\partial\, r_{ij}}{\partial x_{ij}} = \frac{x_{ij}}{\sqrt{x_{ij}^2 + y_{ij}^2 + z_{ij}^2}} = \frac{x_{ij}}{r_{ij}} \tag{3.2.11}$$

Por lo tanto, el potencial sobre la componente x es :

$$\frac{\partial U_{ij}(r_{ij})}{\partial x_{ij}} = -48\left[\frac{1}{r_{ij}^{13}} - \frac{1}{2}\frac{1}{r_{ij}^{7}}\right]\frac{x_{ij}}{r_{ij}} = -48\left[\frac{1}{r_{ij}^{14}} - \frac{1}{2r_{ij}^{8}}\right] x_{ij} \tag{3.2.12}$$

Extendiendo la ecuación 3.2.12 sobre las componentes y y z de la fuerza tenemos:

$$F_{ij}^{x} = \frac{\partial U_{ij}(r_{ij})}{\partial x_{ij}} = -48\left[\frac{1}{r_{ij}^{14}} - \frac{1}{2}\frac{1}{r_{ij}^{8}}\right] x_{ij} \tag{3.2.13}$$

$$F_{ij}^{y} = \frac{\partial U_{ij}(r_{ij})}{\partial y_{ij}} = -48\left[\frac{1}{r_{ij}^{14}} - \frac{1}{2}\frac{1}{r_{ij}^{8}}\right] y_{ij} \tag{3.2.14}$$

$$F_{ij}^{z} = \frac{\partial U_{ij}(r_{ij})}{\partial z_{ij}} = -48\left[\frac{1}{r_{ij}^{14}} - \frac{1}{2}\frac{1}{r_{ij}^{8}}\right] z_{ij} \tag{3.2.15}$$

La fuerza total sobre cada una de las partículas, se puede obtener de la siguiente manera:

$$\begin{aligned}
\vec{F_1} &= \vec{F_{11}} + \vec{F_{12}} + \vec{F_{13}} + ... + \vec{F_{1N}} \\
\vec{F_2} &= \vec{F_{21}} + \vec{F_{22}} + \vec{F_{23}} + ... + \vec{F_{2N}} \\
\vec{F_3} &= \vec{F_{31}} + \vec{F_{32}} + \vec{F_{33}} + ... + \vec{F_{1N}} \\
&\vdots \\
\vec{F_N} &= \vec{F_{N1}} + \vec{F_{N2}} + \vec{F_{N3}} + ... + \vec{F_{NN}}
\end{aligned} \tag{3.2.16}$$

La fuerza que siente la partícula *i-ésima* a partir de ella misma, es decir, F_{ii} es la llamada **autofuerza**, la cual no es relevante para este problema. Además la fuerza que siente la partícula *i-ésima*, a partir de la *j-ésima* es igual a la que siente la *j-ésima* a partir de la *i-ésima*, es decir $F_{ij} = -F_{ji}$

Simplificando con las consideraciones anteriores vemos que los únicos elementos que se necesitan calcular para obtener todas las fuerzas son:

$$\begin{array}{ccccc} \vec{F_{12}} & \vec{F_{13}} & \vec{F_{14}} & ... & \vec{F_{1N}} \\ & \vec{F_{23}} & \vec{F_{24}} & ... & \vec{F_{2N}} \\ & & \vec{F_{34}} & ... & \vec{F_{3N}} \\ & & & \ddots & \vdots \\ & & & & \vec{F_{NN}} \end{array} \tag{3.2.17}$$

Obsérvese que en las ecuaciones 3.2.13, 3.2.14 y 3.2.15 sólo se requieren las posiciones para obtener la fuerza en cada componente. Estas posiciones serán arreglos espacialmente ordenados como se muestra en la tabla 3.1 y son parte de las entradas a los algoritmos que se describirán en el capítulo 4.

partículas	1	2	3	...	N
r_x	x_1	x_2	x_3	...	x_N
r_y	y_1	y_2	y_3	...	y_N
r_z	z_1	z_2	z_3	...	z_N

Cuadro 3.1: *Arreglos que contienen las propiedades dinámicas de las partículas*

3.2.2. Integrando con Velocity Verlet

Quizá el método de integración de la ecuación de movimiento (eq. 3.2.1) más ampliamente usado, es el adoptado por Verlet [Ver67]. Este método es una solución directa a la ecuación de segundo orden 3.2.1. Este método está basado en las posiciones $\mathbf{r}(t)$, las aceleraciones $\ddot{\mathbf{r}}(t) = \mathbf{a}(t)$ y las posiciones en el paso anterior $\mathbf{r}(t - \Delta t)$. La ecuación para avanzar las posiciones se escribe a continuación:

$$\mathbf{r}(t + \Delta t) = 2\mathbf{r}(t) - \mathbf{r}(t - \Delta t) + \Delta t^2 \mathbf{a}(t) \tag{3.2.18}$$

Hay varios puntos a notar sobre la ecuación 3.2.18. Las velocidades no están involucradas en el cálculo de las posiciones siguientes. Éstas han sido eliminadas al realizar la suma en series de Taylor sobre $\mathbf{r}(t)$ y truncar hasta los términos de segundo orden de la siguientes expansiones:

$$\begin{aligned} \mathbf{r}(t + \Delta t) &= \mathbf{r}(t) + \Delta t \mathbf{v}(t) + \frac{1}{2}\Delta t^2 \mathbf{a}(t) + \frac{1}{3!}\Delta t^3 \dddot{\mathbf{r}}(t) + O\left(\Delta t^4\right) \\ \mathbf{r}(t - \Delta t) &= \mathbf{r}(t) - \Delta t \mathbf{v}(t) + \frac{1}{2}\Delta t^2 \mathbf{a}(t) - \frac{1}{3!}\Delta t^3 \dddot{\mathbf{r}}(t) + O\left(\Delta t^4\right) \end{aligned} \tag{3.2.19}$$

Las velocidades no son necesarias para computar las trayectorias, pero son útiles para estimar la energía cinética (e incluso la energía total). Estas pueden ser obtenidas truncando 3.2.19 hasta los términos de segundo orden y restando $\mathbf{r}(t+\Delta t)$ de $\mathbf{r}(t-\Delta t)$, así pues, obtenemos:

$$\mathbf{v}(t) = \frac{\mathbf{r}(t+\Delta t) - \mathbf{r}(t-\Delta t)}{2\Delta t} \tag{3.2.20}$$

La ecuación 3.2.18 introduce un error de orden Δt^4 (el error local) y las velocidades de la ecuación 3.2.20 son sujetas a errores de orden Δt^2. Estimaciones más precisas pueden ser hechas, si más términos de $\mathbf{r}$ son tomados en cuenta en la ecuación 3.2.20. Uno de los inconvenientes implícitos en la ecuación 3.2.20 es que $\mathbf{v}(t)$ puede ser solo computada una vez que $\mathbf{r}(t+\Delta t)$ es conocida. Una segunda observación que considera esta primera versión del método de *Verlet* es que está propiamente centrado entre $\mathbf{r}(t-\Delta t)$ y $\mathbf{r}(t+\Delta t)$, y esto representa simetría lo cual hace a este algoritmo reversible en el tiempo.

Asumiendo que tenemos disponibles las posiciones pasadas y actuales del sistema. Las aceleraciones pueden ser evaluadas como $\mathbf{a}(t) = \frac{\mathbf{F}(t)}{m}$, donde $\mathbf{F}(t)$, se calcula con la ecuación 3.2.5. Y entonces las coordenadas son avanzadas como lo muestra el algoritmo 2.

Como podemos ver este método requiere esencialmente $\theta\,(15N)$ espacio en memoria para guardar cada una de las propiedades de todas las partículas. El algoritmo es exactamente reversible en el tiempo y, dadas las fuerzas conservativas, es garantizado que se conserva el momento lineal de cada partícula. Este método ha mostrado que tiene excelentes propiedades de conservación de la energía inclusive para cantidades de δt bastante largos (véase [Ver67]). En contra del algoritmo de Verlet, podemos decir que el manejo de las velocidades no es adecuado, y que la forma del algoritmo puede introducir alguna impresición numérica [DA74]. Esto surge porque en la ecuación 3.2.18 un pequeño término $O((\Delta t)^2)$ es sumado a una diferencia de valores grandes con el fin de generar la trayectoria.

Modificaciones del algoritmo de Verlet han sido propuestas para atacar estas deficiencias. Una de estas es conocida como método de medio paso *leap-frog* [Hoc70] [Pot73]. El origen viene de cuando se hace una expansión en series de Taylor de la velocidad alrededor de $t + \frac{\Delta t}{2}$ y de $t - \frac{\Delta t}{2}$, restando y truncando los términos de segundo orden y mayores; y rearreglando

$$\mathbf{v}\left(t + \frac{\Delta t}{2}\right) = \mathbf{v}\left(t - \frac{\Delta t}{2}\right) + \Delta t\, \mathbf{a}(t) \tag{3.2.21}$$

de esta manera se obtiene la velocidad en un medio paso de tiempo con $t + \Delta t$. Luego, la posición a un paso de tiempo completo se obtiene mediante:

$$\mathbf{r}(t+\Delta t) = \mathbf{r}(t) + \Delta t\, \mathbf{v}\left(t + \frac{\Delta t}{2}\right) \tag{3.2.22}$$

Como vemos en las ecuaciones 3.2.22 y 3.2.21, las cantidades que se requieren son las posiciones $\mathbf{r}(t)$, las aceleraciones $\mathbf{a}(t)$ y las velocidades de medio paso $\mathbf{v}\left(t - \frac{\Delta t}{2}\right)$. Las ecuaciones deben ser implementadas en el orden mostrado. Durante el paso actual las velocidades se calculan con la siguiente ecuación:

$$\mathbf{v}(t) = \frac{1}{2}\left(\mathbf{v}\left(t + \frac{\Delta t}{2}\right) + \mathbf{v}\left(t - \frac{\Delta t}{2}\right)\right) \tag{3.2.23}$$

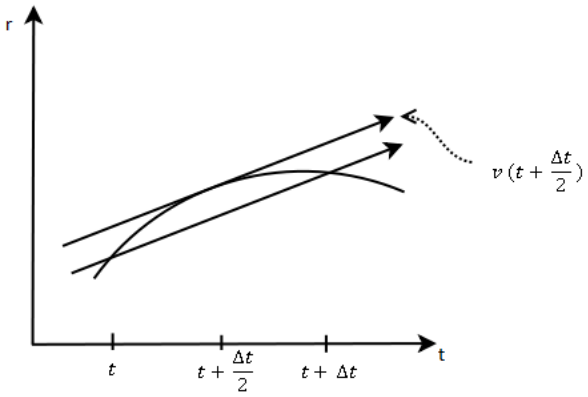


Figura 3.3: *Gráfica de la posición contra el tiempo para el hamiltoniano de una partícula.*

La justificación de esta última ecuación se da en la figura 3.3.

Esto es necesario para calcular la energía cinética al tiempo t, así como cualquier otra cantidad que requiera las posiciones y las velocidades en el tiempo actual. Una vez que hemos resuelto la ecuación 3.2.21 es posible evaluar las aceleraciones para el siguiente paso. Este método es algebráicamente equivalente al de Verlet. Hay algunas ventajas:

- Beneficios numéricos derivan del hecho de que en ninguna etapa se hacen diferencias entre dos cantidades grandes para obtener una pequeña; esto minimiza las pérdidas de precisión en las computadoras.
- Si es muy necesario conservar un espacio de almacenamiento reducido, las aceleraciones pueden ser acumuladas en las velocidades, así el espacio de almacenamiento en memoria requerido será $\theta(6N)$

Como vemos en la ecuación 3.2.23, Leap-Frog sigue sin manejar las velocidades de una manera completamente satisfactoria. Así que en este trabajo usaremos una versión más óptima de los dos métodos anteriormente vistos. A la optimización se le conoce como *Velocty Verlet* el cual usa los principios de las ecuaciones 3.2.19 pero tendrá algunas variaciones. Este algoritmo guarda las posiciones, velocidades y aceleraciones (fuerzas) de las partículas en un paso de tiempo dado y minimiza errores de redondeo [Swo82]. Las ecuaciones que rigen este método son:

$$\mathbf{r}(t+\Delta t) = \mathbf{r}(t) + \Delta t \mathbf{v}(t) + (1/2)\,\Delta t^2\,\mathbf{a}(t) \tag{3.2.24a}$$

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \frac{1}{2}\,\Delta t\,\left[\mathbf{a}(t) + \mathbf{a}(t+\Delta t)\right] \tag{3.2.24b}$$

De las ecuaciones anteriores se puede obtener el algoritmo de Velocity Verlet eliminando las velocidades. Este algoritmo sólo requiere guardar en memoria $\mathbf{r}$,$\mathbf{v}$ y $\mathbf{a}$, además de que están involucradas dos etapas, con una evaluación de las fuerzas entre ellas. Primeramente, las nuevas posiciones al tiempo $t+\Delta t$ son calculadas usando la ecuación 3.2.24a y las velocidades a medio paso de tiempo son calculadas usando:

$$\mathbf{v}\left(t+\frac{\Delta t}{2}\right)=\mathbf{v}(t)+\frac{1}{2}\,\Delta t\,\mathbf{a}(t) \tag{3.2.25}$$

Ahora pueden ser computadas las fuerzas y aceleraciones al tiempo $t+\Delta t$ y entonces el movimiento de las velocidades puede ser completado por:

$$\mathbf{v}(t+\Delta t)=\mathbf{v}\left(t+\frac{\Delta t}{2}\right)+\frac{1}{2}\,\Delta t\,\mathbf{a}(t+\Delta t) \tag{3.2.26}$$

En este punto, la energía cinética al tiempo $t+\Delta t$ puede ser calculada como:

$$E_K=\frac{1}{2}\sum_{i=1}^{N} m_i\|\mathbf{v}_i(t)\|^2 \tag{3.2.27}$$

Mientras que la energía potencial en este paso de tiempo será evaluada mediante el cálculo de las fuerzas con la ecuación 3.2.4.

El proceso total es mostrado en la figura 3.4 . Este método usa $9N$ palabras de espacio en memoria, es numéricamente estable, conveniente y sencillo de implementar haciendo este algoritmo uno de los más usados para simulación molecular (Véase [AT89]). Como veremos mas adelante los códigos casi serán transcripciones de las ecuaciones.

La figura 3.5 muestra las operaciones de cada uno de los tres métodos vistos en esta sección. Es necesario notar que los tres métodos son equivalentes pero las implementaciones dependen de las necesidades computacionales.

Algorithm 2 Proceso para obtener las posiciones siguientes con Verlet

1: **for** $t=1$ to T **do**
2: $\mathbf{r}(t+\Delta t)\longleftarrow 2\mathbf{r}(t)-\mathbf{r}(t-\Delta t)+\Delta t^2\mathbf{a}(t)$
3: Calculo de las nuevas aceleraciones
4: $\mathbf{v}(t)\longleftarrow \frac{\mathbf{r}(t+\Delta t)-\mathbf{r}(t-\Delta t)}{2\Delta t}$
5: Se realiza alguna operación con las velocidades (Calcular energía cinética, ajustar algún parámetro, etc.)
6: $\mathbf{r}(t-\Delta t)\longleftarrow \mathbf{r}(t)$
7: $\mathbf{r}(t)\longleftarrow \mathbf{r}(t+\Delta t)$
8: **end for**

3.2.3. Termostato usando un ensamble canónico

El *Teorema de Equipartición* [Gar08] para una distribución de Maxwell - Boltzmann relaciona la energía cinética vía el momento de las partículas, con la temperatura como cantidad macroscópica y está dado como:

$$\langle E_K\rangle=\left\langle\sum_{i=1}^{N}\frac{1}{2}m_i\|\mathbf{v}_i(t)\|^2\right\rangle=\left\langle\sum_{i=1}^{N}\frac{1}{2}m_i\left({v_x^i}^2+{v_y^i}^2+{v_z^i}^2\right)\right\rangle=\frac{3}{2}k_BT \tag{3.2.28}$$

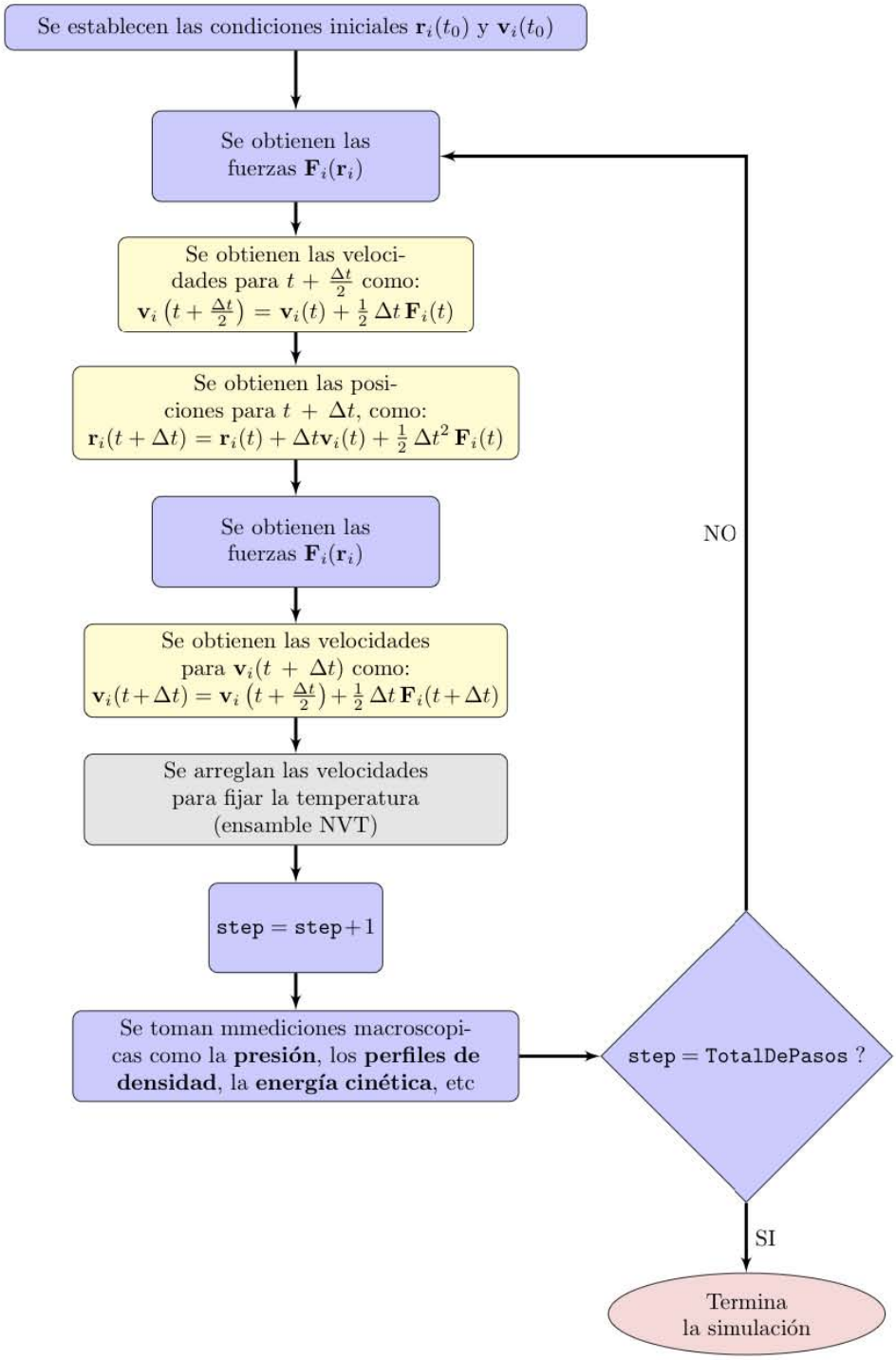


Figura 3.4: *Diagrama de flujo que muestra los distintos pasos de integración y cálculo de fuerzas dentro de una simulación. Los pasos de integración están implementados con el método de Velocity Verlet.*

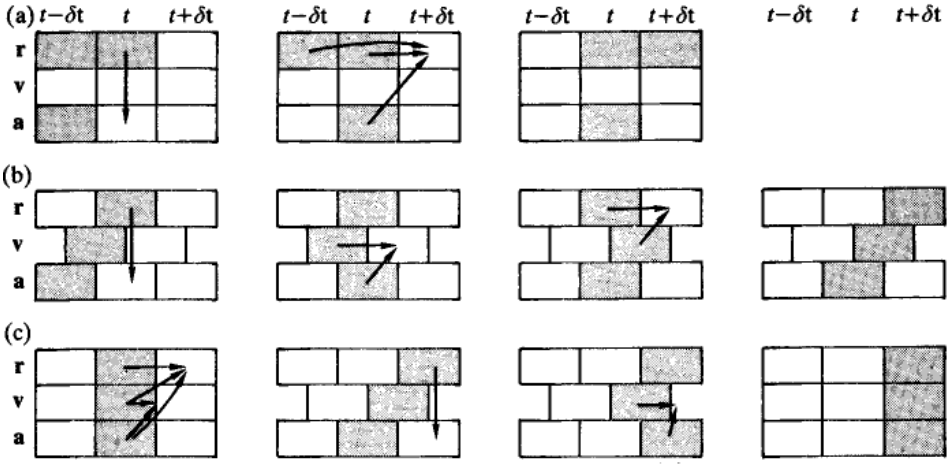


Figura 3.5: *Varias formas del algoritmo de Verlet: (a) Método original de Verlet, (b) Método Leap-Frog, (c) Método Velocity Verlet. Se muestran pasos sucesivos en la implementación de cada algoritmo. Las cajas sombreadas indican las variables que se almacenan en cada paso de tiempo.*

donde k_B es la constante de Boltzman y T es la temperatura del sistema en unidades reducidas (Véase Apéndice A).

La ecuación 3.2.28 puede ser reescrita sin los promedios, ya que este teorema es válido para cada uno de los pasos en una simulación numérica. Así pues, con esta consideración podemos relacionar la energía cinética en cada paso de tiempo con la temperatura total del sistema, fijada por el ensamble NVT ó ensamble canónico el cual considera el número de partículas, el vólumen y la temperatura del sistema como constantes en cada tiempo de la simulación [FS01].

Entonces, la ecuación que determinará la energía cinética con la temperatura instantánea como cantidad microscópica para una partícula en cada paso de tiempo queda como:

$$E_K = \frac{1}{2}m\left({v_x}^2 + {v_y}^2 + {v_z}^2\right) = \frac{3}{2}k_B T_{ins} \tag{3.2.29}$$

Por definición el Teorema de Equipartición dice que la contribución de cada grado de libertad es $\frac{1}{2}k_B T$. Definimos N_{df} como el número de grados de libertad del sistema, los cuales pueden ser de traslación, rotación o vibración. Además N_{df} también se puede definir como: $N_{df} = 3N - N_c$, siendo $3N$ los tres grados de libertad de traslación y N_c el número de limitaciones internas independientes (longitud de enlace y ángulo fijo) definidas en el modelo molecular. En el modelo usado en este trabajo $N_c = 0$ debido a que las partículas solo tendrán cambios en su traslación. Entonces para la energía total de un sistema de N partículas y tres grados de libertad es:

$$E_K = N\left(\frac{1}{2}k_B T_{ins} + \frac{1}{2}k_B T_{ins} + \frac{1}{2}k_B T_{ins}\right) = N\left(\frac{3}{2}k_B T_{ins}\right) \tag{3.2.30}$$

Por lo tanto la temperatura instantánea la podemos calcular como:

$$T_{ins} = \frac{2E_K}{3Nk_B} \tag{3.2.31}$$

Por otro lado, para mantener el ensamble es necesario usar termostato. Con este objetivo proponemos un factor de escalamiento en la velocidad, entre el paso actual y el paso siguiente, esto para cada partícula y cada componente, entonces tenemos:

$$\begin{aligned} \vartheta_x^2 &= (\lambda v_x)^2 \\ \vartheta_y^2 &= (\lambda v_y)^2 \\ \vartheta_z^2 &= (\lambda v_z)^2 \end{aligned} \tag{3.2.32}$$

donde ϑ es la velocidad en el paso siguiente, v es la velocidad en el paso actual y λ es el factor de escalamiento.

Utilizando la ecuación 3.2.31 y 3.2.29, podemos definir la temperatura en el siguiente paso que denotaremos como T' y estará dada como:

$$T' = \frac{2}{3Nk_B} \sum_{i=1}^{N} \frac{1}{2} m_i \left(\vartheta_x^{i\,2} + \vartheta_y^{i\,2} + \vartheta_z^{i\,2} \right) \tag{3.2.33}$$

Si sustituimos 3.2.32 en 3.2.33 tenemos:

$$\begin{aligned} T' &= \frac{2}{3Nk_B} \sum_{i=1}^{N} \frac{1}{2} m_i \left((\lambda v_x^i)^2 + (\lambda v_y^i)^2 + (\lambda v_z^i)^2 \right) \\ T' &= \frac{2\lambda^2}{3Nk_B} \sum_{i=1}^{N} \frac{1}{2} m_i \left(v_x^{i\,2} + v_y^{i\,2} + v_z^{i\,2} \right) T' = \lambda^2 T_{ins} \end{aligned} \tag{3.2.34}$$

Despejando el factor de escalamiento de la ecuación anterior tenemos:

$$\lambda = \sqrt{\frac{T'}{T_{ins}}} \tag{3.2.35}$$

Si fijamos T' como la temperatura del ensamble, entonces la temperatura del sistema nunca cambiará y se mantendrán las condiciones del ensamble. Este termostato es uno de los más simples, pero impreciso, sin embargo se han desarrollado métodos de mayor calidad y precisión para ajustar la temperatura, como son Nosé-Hoover o Berendsen [R07],[Hu04]. A pesar de la imprecisión de este termostato para nuestros fines da buenos resultados y es sencillo de implementar en cualquier lenguaje de programación. Además con ayuda de la física estadística podemos estar casi seguros de alcanzar con alta probabilidad los estados termodinámicos correctos del sistema, es decir, después de

una gran cantidad de pasos el estado termódinamico alcanzado por nuestro sistema es bastante cercano al alcanzado con termostatos más precisos.

En resumen, la temperatura instantánea la obtenemos en cada tiempo que modificamos las posiciones y las velocidades, y puede ser calculada como:

$$T_{ins} = \frac{2}{3N} \sum_{i=1}^{N} m_i \|\mathbf{v}_i(t)\|^2 = \frac{2E_k}{3N} \tag{3.2.36}$$

Después de integrar las ecuaciones de movimiento, numéricamente será conveniente *escalar* (ajustar) las velocidades del paso siguiente (debido a que se puede aprovechar el mismo ciclo de integración) con la siguiente asignación para cada partícula:

$$\mathbf{v}(t + \Delta t) \longleftarrow \lambda \mathbf{v}(t + \Delta t) \tag{3.2.37}$$

3.2.4. Condiciones de frontera periódicas

Como hemos visto anteriormente el espacio de solución de la Dinámica Molecular es una caja cúbica, la cual está delimitada en el intervalo $[0, L]$ en cada una de sus tres coordenadas Cartesianas. La caja cúbica es replicada a través del espacio como una lattice infinita. Cuando avanza la simulación, si una molécula se mueve en la caja original, su imagen periódica en cada una de las cajas vecinas se mueve en el mismo sentido. Así pues, si una partícula deja la caja central, una de las imágenes de la partícula en un caja vecina, entrará a la caja central por la cara opuesta. Esta caja simplemente forma un sistema de ejes convenientes para medir las coordenadas de N partículas. Un sistema periódico en dos dimensiones es mostrado en la figura 3.6. Las cajas duplicadas son etiquetadas como A,B,C, etc. Cuando la partícula 1 se mueve a través de la frontera su imagen 1_A,1_B, etc.(donde el subíndice indica a que caja pertenece) se mueve sobre sus fronteras correspondientes. Por lo tanto, la densidad de partículas en la caja central (y el sistema entero) se conserva.

3.2.5. Truncado del Potencial Lennard-Jones

En general, el centro de la Dinámica Molecular es el cálculo de la energía potencial de una configuración particular, y en este caso involucra calcular las fuerzas que actúan sobre todas las moléculas.

Considere ahora que queremos calcular la fuerza sobre la partícula 1 en la figura 3.7, o las contribuciones que involucran a la partícula 1 y cada una de las otras partículas en la caja de simulación. Claramente habrá $N - 1$ términos en la suma. Sin embargo, también tenemos que incluir todas las interacciones entre la partícula 1 y las imágenes en $i_A, i_B, etc.$ que están en las cajas vecinas. Este es un número infinito, debido a que tenemos infinitas cajas y en la práctica es imposible de calcular.

Para una función de energía potencial de corto-alcance, debemos restringir esta suma a una aproximación. Considere la partícula 1 en el centro de la región circular la cual

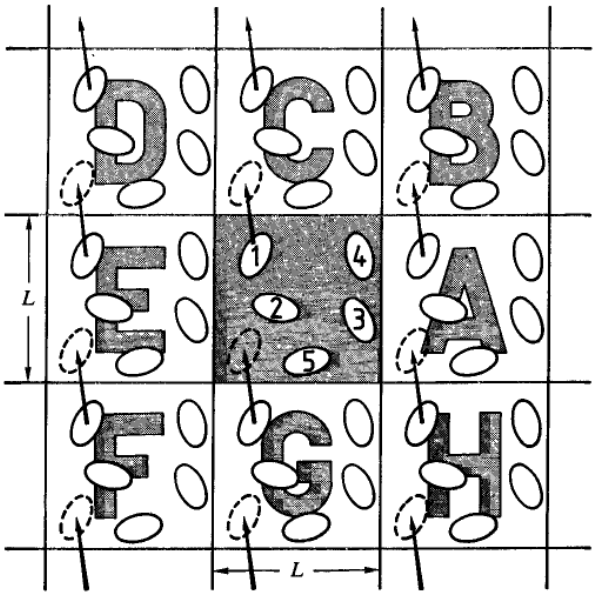


Figura 3.6: *Un sistema periodico en dos dimensiones.*

tiene el mismo tamaño y forma en todas las cajas de simulación. La partícula 1 interactúa con todas las partículas las cuales están dentro de esta región incluyendo las de las imágenes vecinas. Esto es llamado "*convención de imágen mínima*": por ejemplo, en la figura 3.7 la partícula 1 interactúa con las partículas 2,3_E,4_E y 5_C. Como vemos el cálculo de todas las interacciones toma $\frac{1}{2}N(N-1)$ operaciones. La contribución más grande al potencial y a las fuerzas viene de las partículas más cercanas a la partícula de interés, y para fuerzas de corto-alcance nosotros podemos aplicar un radio de corte esférico. Esto significa que el potencial $U_{ij}(\mathbf{r})$ es cero para $\mathbf{r} > \mathbf{r_c}$ donde $\mathbf{r_c}$ es el radio de corte. El círculo punteado en 3.7 representa el radio de corte y en este caso las moléculas 2 y 4_E contribuyen a la fuerza sobre 1, sin embargo 3_E y 5_C no contribuyen. En una caja de simulación cúbica de lado L, el número de vecinos explícitamente considerados es reducido por un factor de $4\pi\mathbf{r_c}^3/3L^3$ en 3 dimensiones, véase [AT89].

La introducción de un radio de corte esférico puede ser considerado como una pequeña perturbación, y la distancia de corte debe ser suficientemente larga para asegurar que no sea así. En la simulación de partículas Lennard-Jones, el valor del potencial en la frontera de una esfera de corte típica ($r_c = 2.5\sigma$) es sólo 1.6 % el del pozo [Véase 3.2]. Claramente, el costo de aplicar un corte esférico es que las propiedades termodinámicas (y otras, como la presión) no serán las mismas que si realizaramos una simulación sin truncar el potencial.

La distancia de corte no debe ser mayor que $\frac{1}{2}L$ para mantener la consistencia con la convención de mínima imagen.

3.2.6. Virial y Presión

Con el fin de determinar la presión de un sistema que se desarrolla en un espacio finito, introduciremos una función que es llamada *Función Virial de Clasius* [Cla70], la

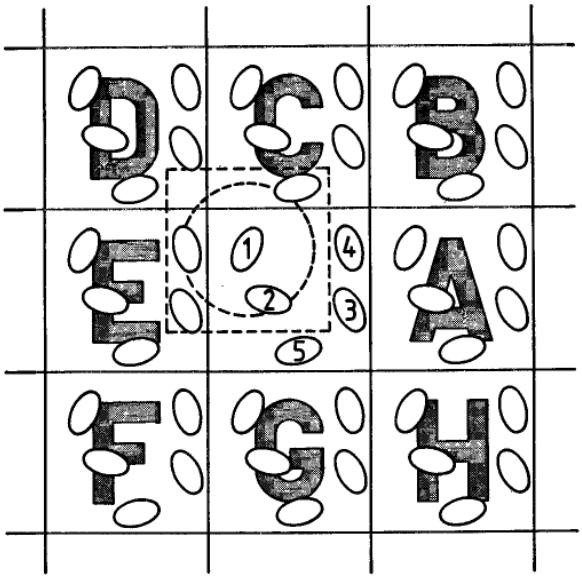


Figura 3.7: *Interacciones en un sistema periódico con la convención de mínima imagen*

cual se escribe como:

$$W^{Tot}(\mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_N) = \sum_{i=1}^{N} \mathbf{r}_i \cdot \mathbf{F}_i^{Tot} \quad , \tag{3.2.38}$$

donde $\vec{r_i}$ es la posición de la *i-ésima* partícula, $\vec{F_i}^{Tot}$ es la fuerza total actuando sobre la i-ésima partícula.

Promediando sobre la trayectoria de la dinámica y usando la ecuación 3.2.1, obtenemos

$$\left\langle W^{Tot} \right\rangle = \lim_{\tau\to\infty} \frac{1}{\tau} \int_0^\tau \sum_{i=1}^{N} \mathbf{r}_i(t) \cdot m_i \ddot{\mathbf{r}}_i(t) dt \quad , \tag{3.2.39}$$

Integrando por partes

$$\left\langle W^{Tot} \right\rangle = \lim_{\tau\to\infty} m_i \sum_{i=1}^{N} \frac{\dot{\mathbf{r}}_i(\tau) \cdot \mathbf{r}_i(\tau) - \dot{\mathbf{r}}_i(0) \cdot \mathbf{r}_i(0)}{\tau} - \lim_{\tau\to\infty} \frac{1}{\tau} \int_0^\tau \sum_{i=1}^{N} m_i \| \dot{\mathbf{r}}_i(t) \|^2 dt \quad , \tag{3.2.40}$$

Si el sistema está localizado en una región finita de espacio y las partículas no aceleran al infinito, entonces el primer término de la ecuación 3.2.40 es cero y por lo tanto tenemos:

$$\left\langle W^{Tot} \right\rangle = - \lim_{\tau\to\infty} \frac{1}{\tau} \int_0^\tau \sum_{i=1}^{N} m_i \| \dot{\mathbf{r}}_i(t) \|^2 dt = - \left\langle E_K \right\rangle = \frac{-3Nk_BT}{2} \quad , \tag{3.2.41}$$

Así, las dos ecuaciones resumidas que obtenemos son:

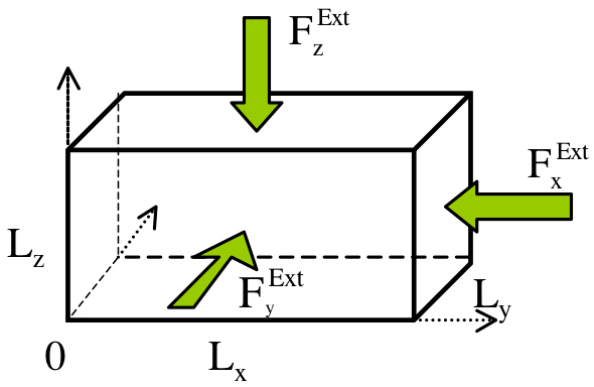


Figura 3.8: *Paralelepípedo que muetras las componentes de la presión*

$$\left\langle W^{Tot} \right\rangle = \lim_{\tau\to\infty} \frac{1}{\tau} \int_0^\tau \sum_{i=1}^{N} \mathbf{r}_i(t) \cdot m_i \ddot{\vec{\mathbf{r}}}_i(t) dt \quad , \tag{3.2.42}$$

$$\left\langle W^{Tot} \right\rangle = - \left\langle E_K \right\rangle = \frac{-3Nk_BT}{2} \quad , \tag{3.2.43}$$

La presión P puede ser definida considerando un contenedor con forma de paralelepípedo con lados L_x, L_y y L_z, como lo muestra la figura 3.8

La fuerza total actuando sobre la *i-ésima* partícula es compuesta de la fuerza interna $\mathbf{F}_i^{Int}$ y la fuerza externa $\mathbf{F}_i^{Ext}$ producida por las paredes del contenedor, entonces, $\mathbf{F}_i^{Tot} = \mathbf{F}_i^{Int} + \mathbf{F}_i^{Ext}$.

Así pues, la función virial total puede ser escrita como la suma del virial interno y el virial externo, $\left\langle W^{Tot} \right\rangle = \left\langle W^{int} \right\rangle + \left\langle W^{Ext} \right\rangle = -\frac{3Nk_BT}{2}$. La parte externa de la función virial para un contenedor con coordenadas de origen en la esquina $(0, 0, 0)$ es:

$$\left\langle W^{Ext} \right\rangle = L_x(-PL_yL_z) + L_y(-PL_xL_z) + L_z(-PL_xL_y) = -3PV \quad , \tag{3.2.44}$$

donde $V = L_xL_yL_z$.

Por lo tanto, para la función virial, tenemos:

$$\left\langle \sum_{i=1}^{N} \mathbf{r}_i \cdot \mathbf{F}_i^{Int} \right\rangle - 3PV = -3Nk_BT \quad , \tag{3.2.45}$$

Esta ecuación es conocida como *Ecuación Virial*. Todas las cantidades excepto la presión P son fácilmente accesibles en una simulación y lo que usaremos para calcular P es:

$$P = \frac{Nk_BT}{V} + \frac{1}{3V} \left\langle \sum_{i=1}^{N} \mathbf{r}_i \cdot \mathbf{F}_i^{Int} \right\rangle \quad , \tag{3.2.46}$$

Una interpretación de las ecuaciones para la presión son las interacciones de pares de las partículas, dada por D. Frenkel y B smith [FS01] de la siguiente manera:

$$\begin{aligned}
\sum_i \mathbf{r}_i \cdot \mathbf{F}_i^{Int} &= \sum_i \sum_{j \neq i} \mathbf{r}_i \cdot \mathbf{F}_{ij}^{Int} \\
&= \frac{1}{2} \sum_i \sum_{j \neq i} (\mathbf{r}_i \cdot \mathbf{F}_{ij}^{Int} + \mathbf{r}_j \cdot \mathbf{F}_{ij}^{Int}) \\
&= \frac{1}{2} \sum_i \sum_{j \neq i} \mathbf{r}_{ij} \cdot \mathbf{F}_{ij}^{Int} \\
&= \sum_i \sum_{j > i} \mathbf{r}_{ij} \cdot \mathbf{F}_{ij}^{Int} \\
&= -\sum_i \sum_{j > i} \mathbf{r}_{ij} \left. \frac{dU(r)}{dr} \right|_{r_{ij}}
\end{aligned} \tag{3.2.47}$$

Y por lo tanto la ecuación que determinará la presión dentro de una caja de simulación estará dada por:

$$P = \frac{N k_B T}{V} - \frac{1}{3V} \left\langle \sum_i \sum_{j > i} \mathbf{r}_{ij} \left. \frac{dU(r)}{dr} \right|_{r_{ij}} \right\rangle \quad , \tag{3.2.48}$$

donde N es el número de partículas, K_B es la constante de Boltzmann, V es el volumen de la caja, r_{ij} es la distancia entre partículas y $\left. \frac{dU(r)}{dr} \right|_{r_{ij}}$ es la derivada del potencial entre partículas.

La ecuacion 3.2.48 es la que se usará en la sección de algoritmos para encontrar la presión del sistema.

Capítulo 4

Algoritmos de Dinámica Molecular

El objetivo de este capítulo es describir los algoritmos que se implantaron durante el desarrollo de esta tesis. Estos algoritmos toman ideas de diferentes fuentes, en donde se explican los procesos que se requieren para realizar una simulación de dinámica molecular. Las fuentes principales consultadas son: [AT89], [FS01] y [Rap04].

4.1. Algoritmo General

A continuación se muestra un algoritmo básico para generar una dinámica de objetos que evolucionan en el tiempo, en donde se considera un método general de integración:

Algorithm 3 Algoritmo General de una Dinámica Molecular

Se inicializan los datos de la simulación
Cálculo de las fuerzas (este paso es opcional)
for $t = 1$ to T_{max} **do**
1. Integración de las ecuaciones de movimiento (este paso es opcional)
2. Cálculo de condiciones de frontera
3. Cálculo de interacciones
4. Medición de alguna propiedad
5. Integración de las ecuaciones de movimiento (este paso es opcional)
6. Reportar cantidades instantáneas (este paso es opcional)

end for

En el algoritmo de arriba algunos de los pasos son opcionales. En el caso de los pasos 1 y 6, debe ir al menos uno de los dos. El ciclo de la dinámica es considerado sobre la variable t y tiene que realizar T_{max} evoluciones en el tiempo. Este ciclo genera el cambio del sistema.

El algoritmo 3 **debe** ser modificado de acuerdo a las necesidades de la implementación computacional (lenguaje de programación, paralelización), incluso el orden de las secciones dentro del ciclo de la dinámica puede variar. En el caso de la integración de

las ecuaciones de movimiento (paso 1 y 6), son opcionales en el sentido de que pueden aparecer en el paso 1 o en el 6, pero incluso esta operación no necesariamente es una integración. Además, el método de integración puede depender del ensamble que se esté utilzando (NVT, NVE, NPT, etc). En la siguiente lista se describe cada una de las partes del algoritmo 3:

- **Se inicializan los datos de la simulación:** En esta sección se leen las posiciones, velocidades y demás propiedades de cada uno de las objetos. También se pueden leer los datos generales de la simulación como son: tamaño de la caja, número de objetos, número de pasos, número de pasos para hacer promedios, temperatura del sistema, presión del sistema, tamaño del paso de tiempo (Δt), etc.

- **Integración de las ecuaciones de movimiento:** En este paso se llevan a cabo operaciones sobre las posiciones y velocidades de cada uno de los objetos. Estas operaciones dependen del método que se este utilizando como puede ser LeapFrog, VelocityVerlet, integración de Newton, etc. Estas operaciones se hacen al principio o al final del ciclo de la dinámica.

- **Cálculo de condiciones de frontera:** En esta sección corregimos las posiciones para que se cumplan las condiciones de frontera y que no tengamos objetos fuera del espacio que comprende al sistema.

- **Cálculo de interacciones:** En este paso calcularemos las interacciones entre pares de objetos, obteniendo los nuevos valores de las fuerzas y alguna propiedad como la energía potencial o la presión del sistema. Como se explicó en el capítulo anterior, este es el cálculo más intensivo en tiempo de cómputo.

- **Medición de alguna propiedad:** En este paso medimos alguna propiedad usando las velocidades, las interacciones (fuerzas) y las posiciones actuales. Algunas de estas propiedades pueden ser la energía cinética, la presión, los perfiles de densidad, los perfiles de presión, aplicar un termostato, etc.

- **Reportar cantidades instantáneas:** En esta sección se pueden imprimir en pantalla o guardar en un archivo cantidades instantáneas, es decir cantidades del paso que esta corriendo, como son la energía cinética, la energía potencial, la temperatura, el número de paso, etc.

El algoritmo 4 es específico de una dinámica de partículas que utiliza el potencial de Lennard-Jones, las cuales son modeladas como objetos puntuales. Este algoritmo está basado en el algoritmo 3 y es el que se usará en el resto de este trabajo.

En el algoritmo 4, se implementa el método de integración explicado en la sección 3.2.2 y corresponde a *Velocity Verlet*. Como vemos en la línea 6 se obtienen las posiciones del paso siguiente y en la línea 5 se calculan las velocidades en $t+\frac{\Delta t}{2}$. Recordemos que las partículas que estamos modelando tienen su masa igual a 1, por lo tanto $\mathbf{a}_i(t) = \frac{\mathbf{F}_i(t)}{m_i} = \mathbf{F}_i(t)$. En la línea 10, para calcular las velocidades de un paso completo, es necesario

Algorithm 4 Algoritmo específico de una Dinámica de partículas Lennard-Jones

1: Se inicializan los datos de la caja
2: **Cálculo de fuerzas por partícula**
3: **for** $t = 1$ to T_{max} **do**
4: **for** $i = 1$ to N **do**
5: $\mathbf{v}_i\left(t + \frac{\Delta t}{2}\right) = \mathbf{v}_i(t) + \frac{1}{2}\,\Delta t\,\mathbf{a}_i(t)$
6: $\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \Delta t \mathbf{v}_i(t) + \frac{1}{2}\,\Delta t^2\,\mathbf{a}_i(t)$
7: **end for**
8: **Cálculo de fuerzas por partícula**
9: **for** $i = 1$ to N **do**
10: $\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i\left(t + \frac{\Delta t}{2}\right) + \frac{1}{2}\,\Delta t\,\mathbf{a}_i(t + \Delta t)$
11: **end for**
12: $kinetic = 0$
13: **for** $i = 1$ to N **do**
14: $kinetic = v_i^x v_i^x + v_i^y v_i^y + v_i^z v_i^z + kinetic$
15: **end for**
16: **Reporte de cantidades instantáneas**
17: **end for**

obtener las fuerzas, debido a que $\mathbf{a}_i(t + \Delta t) = \mathbf{F}_i(t + \Delta t)$. Es por eso que en la línea 8 son calculadas las fuerzas para cada partícula. En este último algoritmo, se observa que es posible aprovechar las iteraciones que se muestran en las líneas 9 y 13, de tal manera que se pueda obtener la energía cinética instantánea con las nuevas nuevas velocidades en un solo ciclo, esto lo hacemos usando la ecuación 3.2.27.

A continuación se muestra un breve mapeo entre el algoritmo 4 y el algoritmo 3 de una dinámica molecular:

- De la línea 4 a 7 se realiza la integración de medio paso de tiempo.
- En la línea 8 se realiza el cálculo de interacciones.
- De la línea 9 a la 11 se completa un paso completo de integración.
- De la línea 12 a la 15 se calcula la energía cinética.

Una modificación al algoritmo 4 es la reducción en el cálculo de la energía cinética aprovechando el ciclo de las velocidades y la incorporación de una sección para calcular promedios de cantidades macroscópicas o mediciones del sistema como son los perfiles de densidad, perfiles de presión, etc. El algoritmo 5 muestra estas modificaciones.

En el algoritmo 6 se calculan las interacciones entre los pares de cada una de las partículas como función de la posición de las mismas. Estas líneas están basadas en la ecuación 3.2.3.

La descripción del algoritmo es la siguiente:

Algorithm 5 Algoritmo reducido de una Dinámica de Partículas Lennard-Jones

1: Se inicializan los datos de la caja
2: **Cálculo de fuerzas por partícula**
3: **for** $t = 1$ to T **do**
4: **for** $i = 1$ to N **do**
5: $\mathbf{v}_i\left(t + \frac{\Delta t}{2}\right) = \mathbf{v}_i(t) + \frac{1}{2}\,\Delta t\,\mathbf{a}_i(t)$
6: $\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \Delta t \mathbf{v}_i(t) + \frac{1}{2}\,\Delta t^2\,\mathbf{a}_i(t)$
7: **end for**
8: **Cálculo de fuerzas por partícula**
9: $kinetic = 0$
10: **for** $i = 1$ to N **do**
11: $\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i\left(t + \frac{\Delta t}{2}\right) + \frac{1}{2}\,\Delta t\,\mathbf{a}_i(t + \Delta t)$
12: $kinetic = \|\mathbf{v}_i(t)\|^2 + kinetic$
13: **end for**
14: **Calculo de promedios**
15: **Reporte de cantidades instantaneas**
16: **end for**

1. Las líneas 3 y 4 son dos ciclos anidados que se pueden entender como: para la *i-ésima* partícula, iterar desde la siguiente hasta N. Esto realiza lo establecido en la ecuación 3.2.17.

2. En las líneas 5,6 y 7 se calculan las distancias entre las partículas para cada una de sus componentes.

3. En la línea 8 se encapsula el algoritmo de la Condición de Mínima Imagen, véase algoritmo 7.

4. En la línea 10 se realiza la distinción entre las partículas que están dentro del radio de corte de la *i-ésima* partícula. El valor de r_{cut} está elevado al cuadrado para poder realizar la comparación correcta de acuerdo a lo realizado en la línea 9.

5. En la línea 11 se suma la contribución de la energía potencial a cada uno de los pares. Nótese que falta multiplicar U por 2, debido a que solo se está obteniendo la mitad de las interacciones.

6. En la línea 12 se calcula un factor de interacción para obtener las fuerzas.

7. De la línea 13 a la 18, se calculan las fuerzas en cada una de las componentes tanto para la *i-ésima* partícula en donde se hace una suma, como para la *j-ésima*, en donde se hace una resta. Esto cumple con lo establecido en la ecuación 3.2.6

8. En la línea 19 se obtiene un factor para calcular el tensor de presión. Este sólo tiene motivos computacionales, pero es claramente lo descrito en la ecuación 3.2.48. En la línea 20 se calcula la traza del tensor de presiones.

Es de notarse que en este algoritmo se lleva acabo un ahorro de tiempo en cómputo en la línea 4 y en la línea 10. En la línea 4 se logra ahorrar la mitad de las interacciones, y en la línea 10 pueden no calcularse algunas interacciones debido a las partículas que no se encuentren dentro del radio de corte.

Algorithm 6 Algoritmo para el cálculo de fuerzas

1: $\mathrm{U} = 0$
2: $\mathrm{P} = 0$
3: **for** $i = 1$ to $N - 1$ **do**
4: **for** $j = i + 1$ to N **do**
5: $dx = x_i - x_j$
6: $dy = y_i - y_j$
7: $dz = z_i - z_j$
8: **Cálculo de la Condición Mínima Imagen** (véase algoritmo 7)
9: $r_{ij} = dx^2 + dy^2 + dz^2$
10: **if** $r_{ij} < r_{cut}^2$ **then**
11: $U = U + 4\frac{1}{r_{ij}^6}\left(\frac{1}{r_{ij}^6} - 1\right)$
12: $F_{ij} = 24\frac{1}{r_{ij}^6}\left(2\frac{1}{r_{ij}^6} - 1)\right)$
13: $F_i^x = F_i^x + F_{ij}\frac{dx}{r_{ij}}$
14: $F_i^y = F_i^y + F_{ij}\frac{dy}{r_{ij}}$
15: $F_i^z = F_i^z + F_{ij}\frac{dz}{r_{ij}}$
16: $F_j^x = F_j^x - F_{ij}\frac{dx}{r_{ij}}$
17: $F_j^y = F_j^y - F_{ij}\frac{dy}{r_{ij}}$
18: $F_j^z = F_j^z - F_{ij}\frac{dz}{r_{ij}}$
19: $pgbr = F_{ij}\frac{1}{r_{ij}}$
20: $P = P + pgbr \cdot dx^2 + pgbr \cdot dy^2 + pgbr \cdot dz^2$
21: **end if**
22: **end for**
23: **end for**

Las condiciones de mínima imagen, descritas en la sección 3.2.4, son mostradas en el algoritmo 7. Para sumar contribuciones de interacciones de partículas que están en las celdas contiguas hay que replicar las partículas. Así, al tener las distancias absolutas entre las partículas dadas por las variables dx, dy y dz y por definición la *i-ésima* partícula, está centrada en $\left(\frac{L}{2}, \frac{L}{2}\right)$, entonces si para la componente en x se cumple la condición de la línea 1, la partícula está fuera de la celda. Para meter la partícula solo tenemos que restar L. Esto es simular para valores negativos (línea 3). La condición de

mínima imagen se tiene que realizar para las tres componentes para que r_{ij} sea correctamente calculada (véase figura 3.7).Teniendo las distancias efectivas estamos listos para calcular r_{ij} en el algoritmo 6.

Algorithm 7 Algoritmo para calcular las condiciones de mínima imagen

1: **if** $dx > \frac{L_x}{2}$ **then**
2: $dx = dx - L_x$
3: **else if** $dx < -\frac{L_x}{2}$ **then**
4: $dx = dx + L_x$
5: **end if**
6: **if** $dy > \frac{L_y}{2}$ **then**
7: $dy = dy - L_y$
8: **else if** $dy < -\frac{L_y}{2}$ **then**
9: $dy = dy + L_y$
10: **end if**
11: **if** $dz > \frac{L_z}{2}$ **then**
12: $dz = dz - L_z$
13: **else if** $dz < -\frac{L_z}{2}$ **then**
14: $dz = dz + L_z$
15: **end if**

En el algoritmo 5 hay un paso que se llama *Cálculo de promedios* en nuestra implemetación solo realizamos el cálculo de la presión promedio y realizamos una medición de los perfiles de densidad. La presión promedio se muestra en el algoritmo 8.

Algorithm 8 Promedio de la presion

1: $P_{ideal} = \frac{N}{VT_{instantanea}}$
2: $P_{prome} = \frac{P}{3V} + P_{ideal}$

Respecto a los perfiles de densidad, la idea básica es partir la caja de simulación en rebanadas ó *bines* sobre cada uno de los ejes coordenados, y contar el número de partículas en cada una de las rebanadas. A lo largo de la simulación en algunos pasos se toman las medidas en las rebanadas y al final se toma el promedio. El grosor de las rebanadas dará la precisión con la que se quieran ver los perfiles de densidad. En el algoritmo 9 se muestra como hacer los perfiles, a continuación se describe este algoritmo:

- De la línea 1 a la 9 se inicializan los arreglos que son los contenedores de las rabanadas, hay tres arreglos porque son tres ejes coordenados. Cada posición del arreglo contiene el número de partículas por rebanada. La variables *MaxBinX*, *MaxBinY* y *MaxBinZ*. son el número de bines que tiene cada eje coordenado respectiamente.

- De la línea 10 a la 17, para cada partícula encontramos el bin al que pertenece en el eje coordenado con la operación round la cual devuelve el entero más cercano, y dividimos la componente de la posición de la partícula en el eje deseado entre el tamaño de las rebanadas sobre el eje coordenado, estos tamaños estan definidos pos las variables *DeltaBinX*, *DeltaBinY* y *DeltaBinZ*.

- Al final, debido a que haremos un promedio de las partículas en cada rebanada sobre toda la simulación es necesario guardar las cantidades de cada coordenada sobre cada una de las rebanadas y esto se hace en las líneas 18 a 26.

Algorithm 9 Promedio para los perfiles de densidad

1: **for** $i = 1$ to $MaxBinX$ **do**
2: ParticlePerBinX[i] = 0
3: **end for**
4: **for** $i = 1$ to $MaxBinY$ **do**
5: ParticlePerBinY[i] = 0
6: **end for**
7: **for** $i = 1$ to $MaxBinZ$ **do**
8: ParticlePerBinZ[i] = 0
9: **end for**
10: **for** $i = 1$ to N **do**
11: bin = round($\frac{r_i^x}{DeltaBinX}$)
12: ParticlePerBinX[bin]++
13: bin = round($\frac{r_i^y}{DeltaBinY}$)
14: ParticlePerBinY[bin]++
15: bin = round($\frac{r_i^z}{DeltaBinZ}$)
16: ParticlePerBinZ[bin]++
17: **end for**
18: **for** $i = 1$ to $MaxBinX$ **do**
19: DensityPerBinX[i] = ParticlePerBinX[i]/VolX + DensityPerBinX[bin];
20: **end for**
21: **for** $i = 1$ to $MaxBinY$ **do**
22: DensityPerBinY[i] = ParticlePerBinX[i]/VolX + DensityPerBinX[bin];
23: **end for**
24: **for** $i = 1$ to $MaxBinZ$ **do**
25: DensityPerBinZ[i] = ParticlePerBinX[i]/VolX + DensityPerBinX[bin];
26: **end for**

Por último, como vimos en el algoritmo 5 y en las líneas anteriores se realizaron algunos promedios, entonces es necesario finalizar éstos. En el algoritmo 10 se muestra como calcular la presión promedio final y la densidad por rebanada promedio final, que no es más que la multiplicación de la cantidad especificada por las veces que se suma

durante la simulación, ese valor es: $\frac{N_{promedios}}{N_{pasos}}$. Este únicamente se debe realizar al final de la simulación ya que se está considerando que ya se sumaron todos los valores promedio.

Algorithm 10 Promedio total de la presión y de los perfiles de densidad

1: $P_{prome} = P_{prome}\frac{N_{promedios}}{N_{pasos}}$
2: **for** $i = 1$ to $MaxBinX$ **do**
3: DensityPerBinX[i] = DensityPerBinX[i] * $\frac{N_{promedios}}{N_{pasos}}$;
4: **end for**
5: **for** $i = 1$ to $MaxBinY$ **do**
6: DensityPerBinY[i] = DensityPerBinY[i] * $\frac{N_{promedios}}{N_{pasos}}$;
7: **end for**
8: **for** $i = 1$ to $MaxBinZ$ **do**
9: DensityPerBinZ[i] = DensityPerBinZ[i] * $\frac{N_{promedios}}{N_{pasos}}$;
10: **end for**

Ahora estamos preparados para describir el software que se desarrolló con base en los algoritmos que se listaron en este capítulo.

Capítulo 5

Moldynamics: software para simulación de Dinámica Molecular

El cómputo científico es un área de estudio con más de 40 años de desarrollo, y actualmente está muy bien documentada. Sin embargo, no existe una manera "correcta" u "óptima" de desarrollar software para el estudio de problemas científicos, porque es una tarea extremadamente dependiente del tipo de problema que se está tratando. Por ejemplo, los estudios de experimentos en física de altas energías requieren miles de CPUs trabajando independientemente para el procesamiento de datos, mientras que las simulaciones de procesos hidrodinámicos en general requieren máquinas de cómputo paralelo masivo interconectadas. Por otro lado, varios de los problemas numéricos del trabajo diario en la física, pueden ser resueltos en una computadora portátil.

En este capítulo se explicará a grandes rasgos la implantación de los algoritmos que se describieron en el capítulo anterior, los cuales están inmersos en un software orientado a objetos. Este paradigma de programación, no es en general muy bienvenido en aplicaciones de cómputo científico, debido a que puede ocasionar bajo rendimiento. Sin embargo, la orientación a objetos ayuda a ordenar los códigos de tal manera que sea fácil su re-utilización en diversos problemas. Uno de los objetivos de esta tesis, es generar herramientas que permitan desarrollar simulaciones de dinámica molecular de manera simple y eficiente.

5.1. Estrategia General

En general, escribir un programa de computadora para estudiar un problema científico puede ser dividido en diferentes fases:

1. Análisis y Diseño
2. Implementación
3. Depuración y pruebas

4. Producción de datos

5. Análisis de Datos

6. Documentación

7. Publicación de resultados

En el proceso de desarrollo unificado [BRJ99], estas fases se realizan de manera iterativa, es decir, se plantea un problema simple y se ejecutan las tareas 1-6. Si todo transcurrió sin problemas entonces se pasa a la fase 7. Luego, se plantea un problema más complicado y se repiten las fases 1-6. En esta segunda iteración, es posible re-utilizar las herramientas desarrolladas durante la primera iteración. Este proceso se realiza hasta cumplir con todos los objetivos de nuestro problema original. A esto se le llama un proceso iterativo y aditivo.

5.2. Diseño de Moldynamics

Moldynamics es el nombre del software que se desarrolló en esta tesis, y es aquí donde se implementan cada uno de los algoritmos teóricos vistos en el capítulo 3.

Las características principales que se pretende contenga este software se listan a continuación:

Robusto : que el programa sea capaz de manejar los errores sin terminar de manera abrupta. Por ejemplo, para cada una de las variables que son controles importantes de la simulación se inicializan valores por omisión que permiten al usuario hacer una simulación aunque los parámetros sean incorrectos, y si por ejemplo se ingresan valores para el número de pasos de los promedios, mayor al número de pasos totales este se reinicializa, es decir, pone la variable de los pasos promedio en default ya que este debe ser estrictamente menor o igual al número de pasos totales.

Extensible: que sea fácil agregar nuevos componentes, sin tener que hacer grandes modificaciones.

Reusable: en este trabajo se harán simulaciones de partículas esféricas tipo Lennard-Jones. Sin embargo, se busca que partes del software se puedan utilizar en códigos con diferentes modelos moleculares como discos u ovoides formados por una interacción con un potencial de Gay-Berne (véase [Gay81]).

Sencillo: un objetivo principal en el desarrollo de moldynamics es que sea sencillo de usar para alguien que no tiene muchos conocimientos de programación y que pueda tener resultados rápidamente.

Óptimo: el costo computacional más caro de los procesos de una simulación de dinámica molecuar, está perfectamente detectado y es en el cálculo de las fuerzas. Es ahí donde es importante probar arquitecturas computacionales de alto desempeño (GPUs, procesadores cell, etc.) que sean más potentes y más precisas.

Para obtener los requerimientos antes planteados se sigue la estrategia general definida en la sección 5.1. Durante el desarrollo se utilizó el lenguaje de programación C++[], el cual permite un desarrollo usando la programación orientada a objetos. Además, algunas partes del software se pueden usar mediante el lenguaje Python [Pyt]. Finalmente, los cálculos que requieren mayor esfuerzo computacional se escribieron usando el API CUDA para aprovechar la arquitectura de las unidades gráficas de procesamiento (GPUs por sus siglas en inglés).

5.3. Clases y funciones de Moldynamics

Moldynamics está construido usando un paradigma de programación orientada a objetos. Las clases que contiene y algunas de sus relaciones se explican en esta sección.

5.3.1. Clase `Box`

Esta clase `Box` (el nombre hace referencia a la caja de simulación, véase la figura 5.1) encapsula las propiedades y métodos de la caja de simulación. Esta clase es muy importante y fue la primera que se implementó en *Moldynamics*. Las propiedades de la caja como su tamaño, el número de partículas, la posición, la velocidad y la fuerza de cada una de las partículas están implementadas con un arreglo dinámico dentro de esta clase. Esto hace que el recorrido de las partículas, el cual se realiza frecuentemente, sea lo más eficiente y rápido posible. Un ejemplo de estos recorridos es el integrador (véase el algoritmo 5). La conservación de la energía potencial y cinética dentro de la caja y la temperatura instantánea, son variables que en todo momento cambian dentro de la caja, por lo tanto están disponibles en esta clase.

También podemos apreciar los métodos que inicializan estas propiedades, los constructores que no tienen parámetros crean una caja con todas las propiedades por default, si tiene parámetros el constructor no puede establecer las propiedades de la caja. Los parámetros son variables públicas porque durante el proceso de la simulación se pueden modificar en cualquier momento. En cualquier momento se puede redefinir la caja en memoria con propiedades diferentes, esto es útil si queremos implementar un ensamble diferente como NPT (número de partículas, presión y temperatura fijos), lo que cambia durante la simulación es el volumen de la caja (Véase [HÖ5]). También tenemos un método (`PrintProperties`) para imprimir en pantalla las propiedades de la caja, eventualmente será necesario poner todas las fuerzas de las partículas a cero y esto lo hace el método `ForceZero`.

El hecho de tener en la clase `Box` encapsulados varios atributos importantes de la simulación, así como algunos métodos, permite versatilidad para reusarse en otros

problemas. Por ejemplo es posible realojar un objeto `Box` en memoria sin afectar otras clases del programa.

5.3.2. Clases `Pressure` y `SimParams`

La figura 5.1 muestra la relación entre la clase `Box` y las clases `Pressure` y `SimParams`. Por un lado `Pressure` es la clase que se encarga de mantener las estadísticas de la presión y esto es una propiedad intrínseca de la caja ya que depende de su volumen como se vió en 3.2.6. Hay métodos para poner el tensor y su promedio a cero. El método add agrega la contribución por interacción entre dos partículas, para esto se necesita un factor que es calculado como lo muestra el algoritmo 6, además son necesarias las distancias en cada una de las componentes cartesianas. También hay métodos para imprimir la presión instantánea y la promedio.

Por otro lado la clase `SimParams`, controla todos los parámetros de la simulación como son: el número de pasos, el número de pasos promedio, la frecuencia de salida las propiedades en pantalla, la cantidad de cambio en el tiempo Δt, la temperatura y el radio de corte. Un objeto de esta clase, actúa como un reloj", que actualiza todos los objetos de otras clases, que estén colaborando en la solución del problema. De esta manera, se pueden aislar los módulos del integrador, ya que la clase `Box` es una estructura global por diseño. Los métodos para controlar los pasos y las acciones entre los objetos son:

- `NextStep` incrementa en uno el contador de pasos.
- `EndStep` devuelve 1 si es el último paso o cero en caso contrario.
- `InitStep` devuelve 1 si es el primer paso o cero en caso contrario.
- `Average` devuelve 1 si es un paso para ejecutar un promedio o cero en caso contrario.
- `Output` devuelve un 1 si es un paso en que se deba ejecutar las salidas en pantalla o cero en caso contrario.

La clase contiene también otros métodos para imprimir en pantalla propiedades interiores, como el radio de corte, la temperatura instantánea, etc.

5.3.3. Clases `ReadBoxConfig` y `ReadSimConfig`

Los arreglos de las posiciones, las velocidades y las fuerzas son inicializados con valores aleatorios por default, sin embargo, es útil leer estos datos de algún archivo, ya que si alguna simulación requiere arrancar de algún estado guardado, como es frecuente por fallas eléctricas o simplemente se está experimentando con algún tipo de configuración especial de las partículas (en este trabajo, por ejemplo, todas las simulaciones se arrancan de un cristal, como se verá en los resultados). Estas tareas se pueden realizar con la clase `ReadBoxConfig`. Sólo basta pasar la dirección en memoria de un objeto `Box`

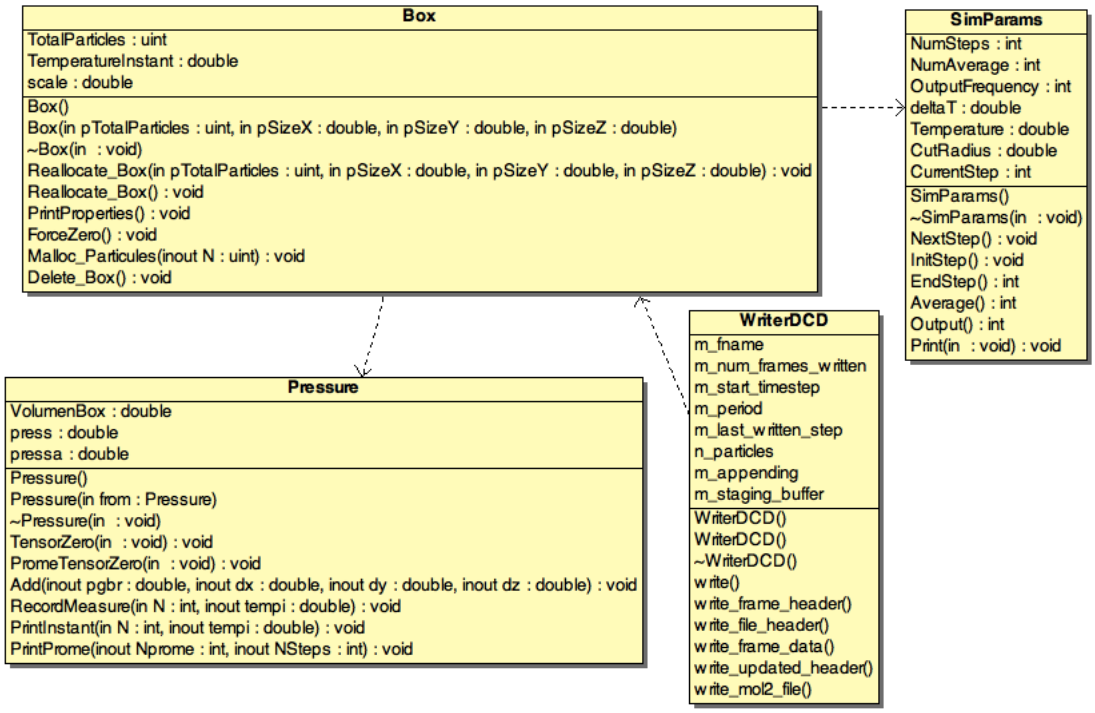


Figura 5.1: *Diagrama UML de la clase* `Box`*,* `Preassure` *y* `SimParams`*. Estas clases contienen los datos centrales de una simulación molecular.*

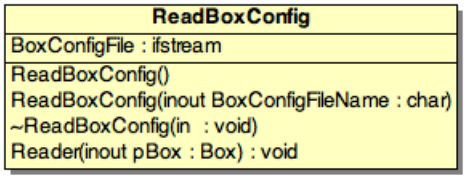


Figura 5.2: *Diagrama UML de la clase* `ReadBoxConfig`*. Esta clase lee una configuración de un sistema a partir de un archivo xml.*

al método `Reader` y éste la llenará con las características del archivo de entrada. El nombre del archivo se establece en el constructor. Es importante notar que el formato del archivo de entrada es en XML y se uso una biblioteca bastante eficiente llamada *Pugixml*. Anteriormente se describieron las ventajas de usar este formato.

En cuanto a la extensibilidad, usar formatos estándares como XML, permite el uso de lenguajes de script como Python, Perl o Bash, esto hace que tareas de análisis sobre los archivos sean más eficientes. Además, esto permite transformar archivos fácilmente para ser usados con diferentes programas de simulación como CHARMS o GROMACS [Lin].

La lectura de un archivo con los datos de la simulación se realiza con la clase `ReadSimConfig` que se muestra en la figura 5.3. Algunos parámetros de configuración de la simulación son:

- Número de pasos totales

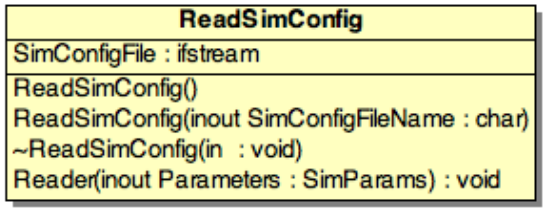


Figura 5.3: *Diagrama UML de la clase* `ReadSimConfig`*. La cual lee la configuración de una simulación a partir de un archivo xml.*

- Número de pasos en los cuales se realizaran promedios.
- Valor del Radio de corte.
- Valor de la Temperatura del ensamble.
- Valor booleano de si se desea hacer una película en formato DCD.
- Número de pasos para tomar fotos de la película.
- Número de pasos para imprimir alguna propiedad en pantalla.

El formato DCD, es un formato de un archivo en el cual se guardan las posiciones y otras propiedades de las partículas en formato binario. Contiene una cabecera que describe el orden del archivo, y se configura la misma para ser leido por programas de visualización como VMD [Sch].

Al igual que la clase que lee las propiedades de la caja, la clase que lee las propiedades de la simulación tiene un constructor donde como parámetro se le pasa el nombre del archivo que contienes las propiedades de la simulación. El método `Reader` realiza la lectura de las propiedades.

5.3.4. Clase `Integrator`

Como hemos visto el núcleo de la simulación es el integrador. La clase correspondiente se llama `Integrator` y se muestra en la figura 5.4. En esta clase el constructor asocia un objeto `Box` a la misma, esto sólo establece los parámetros iniciales de la simulación como son, el radio de corte, poner la traza del tensor de presiones a cero, inicializa las fuerzas a cero, incluso calcula las fuerzas en un primer paso. El método que realiza toda la simulación y asocia un ensamble de simulación a la caja se llama `VelocityVerletNVT`, en este punto la clase esta lista para agregar más métodos de integración con otros modelos como pueden ser NPT, NVE, etc, (Véase [HÖ5]).

5.3.5. Clase `Forces`

La clase `Integrator` depende de la clase `Forces`, en donde se calculan las interacciones con el potencial de Lennard-Jones para todas las partículas. El método `RunCPU`

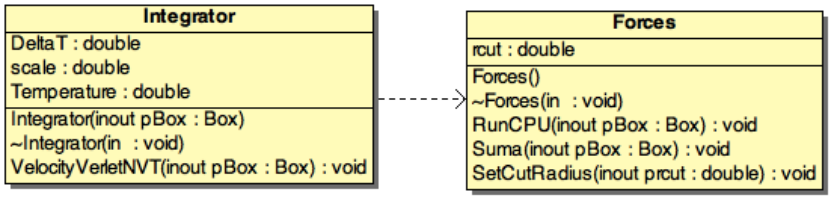


Figura 5.4: *Diagrama UML de las clases* `Integrator` *y* `Forces`*. Las cuales realizan los cálculos numéricos para una simulación molecular*

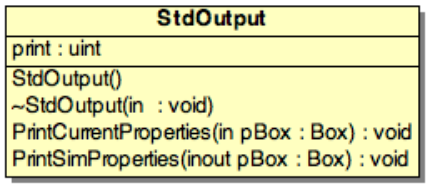


Figura 5.5: *Diagrama UML de la clase* `StdOutput`

realiza la tarea del algoritmo 6. Además este método considera las condiciones de mínima imagen que se describen en el algoritmo 7. Antes de ejecutar `RunCPU` es necesario establecer el radio de corte con el método `SetCutRadius`, en algunas ocasiones será necesario sumar las fuerzas para asegurarnos que son cero y probar que el programa esta funcionando correctamente. El método para realizar esta tarea se llama `Suma`. Esta clase está preparada para agregar nuevos métodos para calcular las fuerzas dependiendo de la arquitectura computacional requerida, alngunos son: `RunGPU` (que eventualmente realizaría el cálculo en un GPU) ó `RunMPI` (realizaría el cálculo en paralelo sobre un cluster). El cálculo intensivo es implementado en esta clase.

5.3.6. Clase `StdOutput`

Una parte esencial de un software de simulación científica es reportar en pantalla información importante. Para esto fue necesario crear una clase dedicada a esta tarea que sea muy flexible. En la figura 5.5, se muestra la clase que contiene dos métodos, uno que imprime las propiedades de la simulación en cada paso que se llama `PrintCurrentProperties`, y otra que imprime letreros al inicio y al final de la simulación. Ambos métodos reciben como parámetro un objeto Box, esto implica que en caso de cambiar por una caja de simulación de otro tipo, no será necesario modificar el funcionamiento esta clase.

5.3.7. Clase `DensityProfile`

La realización de mediciones se lleva a cabo en la clase `Forces` donde se calculan propiedades instantáneas, sin embargo, los promedios se deben obtener durante el ciclo de la simulación. Así pues, la clase `DensityProfile` es usada para calcular los perfiles de densidad, el constructor recibe como parámetro el nombre del archivo donde se

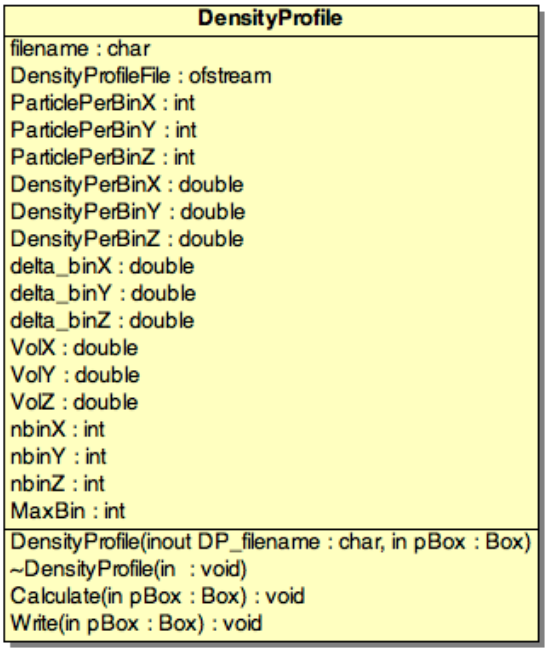


Figura 5.6: *Diagrama UML de la clase* `DensityProfile`*, la cual controla la construcción de los perfiles de densidad*

escribirán los perfiles, y un objeto Box, de donde se obtiene el tamaño de la caja para posteriormente calcular el tamaño de las rebanadas. El método `Calculate` se ejecuta cada que se va a realizar un promedio y está basado en el algoritmo 9. El método `Write`, se ejecuta fuera de la simulación y escribirá los perfiles en un archivo en formato de columnas separadas por espacio.

5.3.8. Clases de Utilerías

Alguna de las utilerías importantes en una simulación puede ser tomar el tiempo de un proceso ejecutado, lo cual se realiza con la clase `Timer` que contiene los siguientes métodos:

- `StartTimer1`: Arranca el timer 1
- `EndTimer1`: finaliza el timer 1, y para de contar el tiempo
- `TimeDiff`: Realiza las operaciones de diferencia entre un timer en segundos
- `PrintT1`: Imprime en pantalla las propiedades del timer 1 en segundos, minutos y horas.
- `PrintT1`: Imprime la fecha del día corriente.

La otra utilería importante realiza la toma de fotos que son guardadas en formato binario (DCD). Como foto se interpreta a un archivo binario (lo cual implica alta

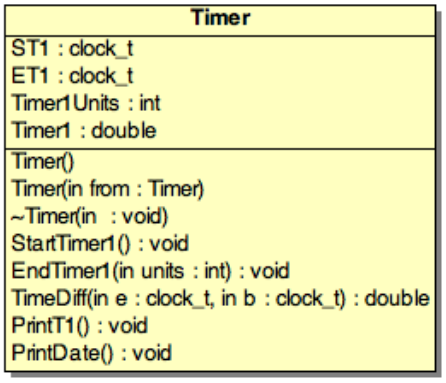


Figura 5.7: *Diagrama UML de la clase `Timer`. Esta clase toma tiempos de cómputo de distintas operaciones.*

compresión) con las posiciones de las partículas en un tiempo dado. La creación de animaciones es deseable cuando se están analizando estructuras moleculares como se hace en este trabajo, ya que se puede inspeccionar la evolución de las partículas dentro de la caja desde cualquier vista. La clase que realiza esta labor se llama `WriterDCD`.

La clase `WriterXYZ`, escribe la configuración de las partículas en formato XYZ. Cada renglón describe la posición de una partícula, siendo la columna 1 el tipo de parícula (puede ser cualquier elemento químico), en la columna 2 la posición en el eje X , en la columna 3 la posición en el eje Y, y por último en la columna 4 la posición en Z.

Por otro lado, la clase `WriterBox` escribe la configuración de la caja en un formato que se creó especialmente para este trabajo. La finalidad es que se puedan leer archivos en este formato usando Python para realizar tareas de post-procesamiento. Por ejemplo, estos archivos se pueden usar con técnicas de *clustering*[], análisis de cantidades, entre otras.

Es importante aclarar que la clase `WriterDCD` tiene un dependencia de la clase `Box` la cual se muestra en la figura 5.8.

Las clases para el control de las configuraciones de la caja se muestran en la figura 5.1, esto tiene como finalidad que se pueda construir un archivo DCD a partir de cualquier tipo de caja en trabajos futuros.

5.3.9. Diagrama de clases y sus relaciones

En la figura 5.9 se muestra un *diagrama de clases*, el cual contiene todas las *dependecias* y los métodos de cada una de las clases que componen el software desarrollado durante este trabajo.

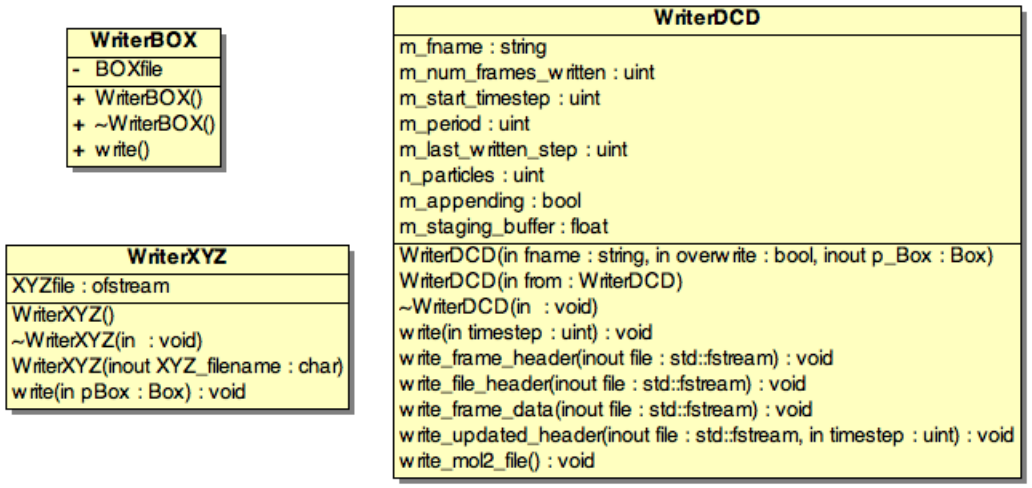


Figura 5.8: *Diagramas UML de la clases que escriben los archivos de salida*

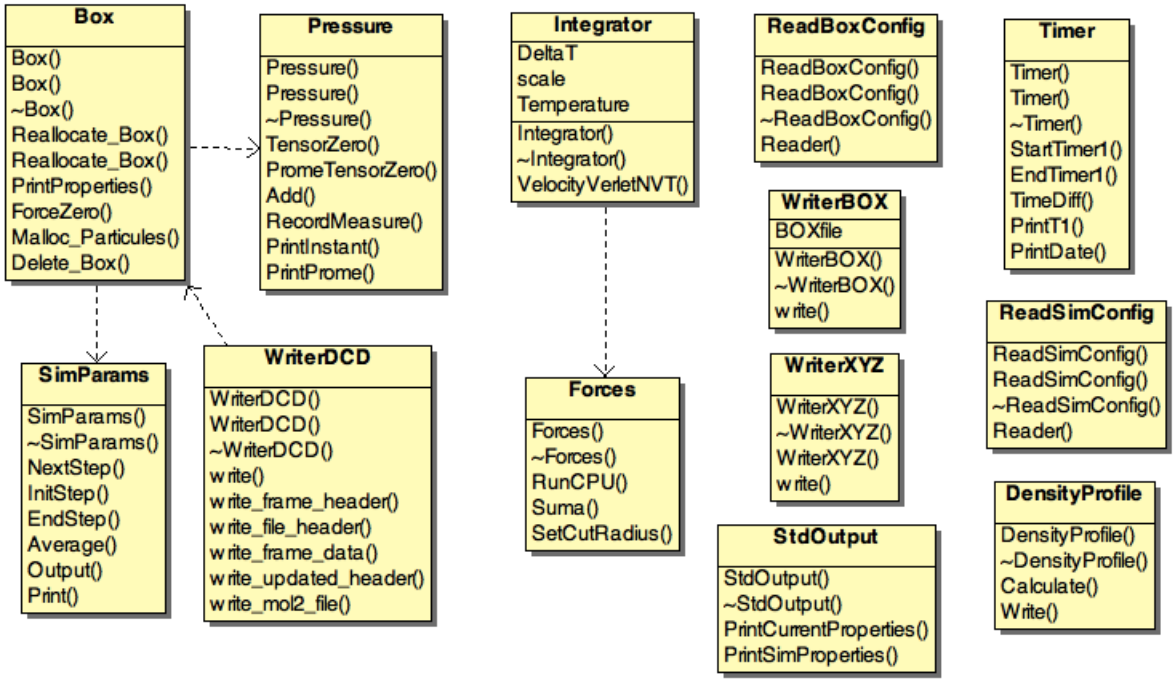


Figura 5.9: *Diagrama UML de la clases, el cual describe todas las dependencias dentro del software*

Capítulo 6

Paralelización en la GPU

Desde hace alguno años se ha visto un incremento en las aplicaciones científicas que hacen uso de unidades de procesamiento gráfico (GPUs) para reducir el tiempo de cómputo de sus procesos. Muchos autores reportan aceleraciones en sus códigos desde 5 hasta 100 veces más rápidos que usando procesadores normales (CPUs) [NVIc]. La razón de estas aceleraciones estriba en la diferencia fundamental que existe en el diseño de GPUs y CPUs. En un CPU se busca optimizar las operaciones secuenciales haciendo uso de un control lógico muy sofisticado que permite que las instrucciones que se ejecutan sobre un hilo (del inglés thread), se ejecute en paralelo manteniendo la apariencia de una ejecución serial. Un factor importante es que los CPUs están provistos de memorias cache relativamente grandes que reducen el tiempo de acceso a datos e instrucciones. Sin embargo, ni el control lógico ni la memoria cache tienen un impacto importante en el rendimiento de la mayoría de las aplicaciones. En la actualidad los CPUs con múltiples núcleos (hasta 16), están diseñados para mejorar el rendimiento de un código secuencial, aunque también es posible diseñar algoritmos paralelos para estas arquitecturas. Por otro lado, la filosofía de diseño y construcción de un GPU se enfoca en maximizar el área del chip dedicada a las operaciones de punto flotante. La idea es optimizar la ejecución en paralelo de un número grande de hilos. Actualmente, los GPUs están diseñados como motores de cálculo numérico, de tal manera que este tipo de aquitectura no será óptima en otras tareas que no tengan que ver con operaciones de punto flotante. Por lo tanto, casi todas las aplicaciones usan una combinación CPU-GPU, ejecutando las partes del código secuencial en el CPU y las partes numéricas intensivas en el GPU. Es por esta razón que CUDA (Compute Unified Device Architecture), el modelo de programación diseñado por la compañía NVIDA, está diseñado para soportar ejecuciones de una aplicación en una combinación CPU-GPU.

6.1. CUDA

CUDA es un modelo de programación diseñado por el fabricante NVIDIA para realizar operaciones aritméticas sobre una GPU. Desde el punto de vista del hardware, CUDA ve la GPU como un conjunto de procesadores SIMD (Single Instruction Multiple

Data). Hoy en día el número de procesadores en las GPUs oscila entre los 100 y 512 procesadores (CUDA cores) por tarjeta gráfica.

En el modelo de programación de CUDA, se tienen multiprocesadores o SMs (del inglés streaming multiprocessors), los cuales contienen una cantidad definida de procesadores o SPs (del inglés streaming processors). Cada SM es visto como un dispositivo multi-core que es capaz de ejecutar un gran número de hilos en paralelo. Estos hilos son organizados en bloques de hilos. Los hilos en el mismo bloque pueden cooperar juntos porque comparten eficientemente datos y sincronizan su ejecución en acceso a memoria con otros hilos. Cada hilo tiene un identificador único, similar a una coordenada en 2D y/o 3D, lo cual hace que puedan leer y modificar datos de la memoria al mismo tiempo sin conflictos con otros hilos. Sin embargo, hilos en diferentes bloques no se pueden comunicar ni sincronizar. Teóricamente, tener a los hilos mayormente ocupados puede ayudar a reducir las deficiencias en la velocidad de transferencia de datos de y hacia la memoria global. De acuerdo a [NVIb], el número máximo de hilos que pueden correr concurrentemente sobre un multiprocesador es 768. En la práctica, el número de hilos está limitado por la memoria compartida incrustada en el chip, y por lo tanto, el número máximo de hilos es dependiente de la aplicación particular que se esté ejecutando.

CUDA contiene un controlador de hardware (*driver*), una interfaz de programación para desarrollo de aplicaciones (API, por sus siglas en inglés), una interfaz en tiempo de ejecución, y varias bibliotecas matemáticas de alto nivel para usos comunes [NVIb]. Los datos y la porción de cómputo numérico que se realizará en paralelo, se aíslan en subprogramas conocidos como *kernel*s en CUDA. Durante la ejecución de un programa en CUDA, el kernel se carga dentro del hardware del GPU para posteriormente ser utilizado durante los cálculos numéricos intensivos. En el apéndice B se describe un ejemplo simple de una suma de vectores con CUDA.

Un multiprocesador tiene una memoria empotrada en el chip que se divide en cuatro tipos (véase la figura 6.1):

1. Un conjunto de registros por procesador;

2. Una memoria compartida;

3. Una cache constante de sólo lectura; y

4. Una cache de textura de sólo lectura.

Cada una de las memorias anteriores son de acceso rápido y son usadas en operaciones de entrada y salida de gran demanda. El intercambio de datos bien administrado entre estas memorias deriva en un buen rendimiento. Por otro lado, aunque la memoria global es la que tiene la velocidad de acceso más baja en la transferencia de datos, ésta puede ser aprovechada porque todos los hilos tienen acceso directo a ella.

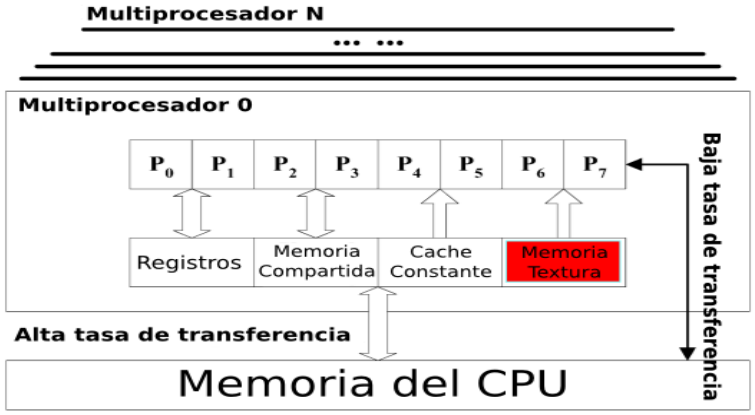


Figura 6.1: *Organización de la memoria de una GPU (tomada del manual de NVIDIA).*

6.2. Pruebas de rendimiento del GPU

En esta sección se reportan los resultados de varios ejemplos de programas en CUDA que realizan operaciones simples de álgebra lineal. El objetivo de estas pruebas es medir el rendimiento del hardware en donde se correrán todos los códigos de la dinámica molecular. Dicho hardware es un servidor Dell T7500 con las siguientes características:

- CPU
  - 8 procesadores Intel(R) Xeon(R) CPU X5677 @ 3.47GHz
  - Tamaño del cache: 12288 KB
  - Tamaño de la memoria RAM: 4 GB
  - Sistema Operativo: Linux Ubuntu 10.10 (maverick)
- GPU
  - Tarjeta de video: Tesla C1060 con las siguientes características (salida del programa `deviceQuery` de los ejemplos de CUDA):

```
CUDA Driver Version:       4.0
CUDA Runtime Version:    3.20
CUDA Capability Major/Minor version number:    1.3
Total amount of global memory:                 4294770688 bytes
Multiprocessors x Cores/MP = Cores:
          30 (MP) x 8 (Cores/MP) = 240  (Cores)
Total amount of constant memory:               65536 bytes
Total amount of shared memory per block:       16384 bytes
Total number of registers available per block: 16384
Warp size:                                     32
Maximum number of threads per block:           512
```

```
Maximum sizes of each dimension of a block:    512 x 512 x 64
Maximum sizes of each dimension of a grid:
           65535 x 65535 x 1
Maximum memory pitch:                          2147483647 bytes
Texture alignment:                             256 bytes
Clock rate:                                    1.30 GHz
Concurrent copy and execution:                 Yes
Run time limit on kernels:                     No
Integrated:                                    No
Support host page-locked memory mapping:       Yes
Compute mode:                                  Default
            (multiple host threads can use this device simultaneously)
Concurrent kernel execution:                   No
Device has ECC support enabled:                No
Device is using TCC driver mode:               No
```

Los códigos que se muestran en las secciones siguientes tienen las siguientes características:

- **C nativo**: Los códigos de las pruebas usando este lenguajes se optimizaron lo mejor posible manteniendo un código simple. Se usó el compilador de INTEL (`icc`) con la bandera de máxima optimización `-O3` (veáse `man icc`).
- `MKL`: Los códigos que usan esta biblioteca fueron optimizados y compilados con el compilador `icc` y la bandera de optimización `-O3`. Específicamente las bibliotecas que se usaron fueron: `mkl_intel_lp64`, `mkl_sequential` y `mkl_core`.
- `CUBLAS`: Los códigos que usan esta bibioteca se compilaron con el compilador `nvcc`.

Todos lo códigos fueron ejecutado en el hardware antes descrito.

6.2.1. Prueba 1: Operación SAXPY usando MKL y CUDA

La operación SAXPY (**S**ingle-precision real **A**lpha **X** **P**lus **Y**), está definida dentro del conjunto de subprogramas contenidos en la biblioteca conocida como Basic Linear Algebra Subprograms (BLAS). El uso de esta operación es bastante común en aplicaciones científicas. SAXPY es una combinación de la multiplicación de escalar por un vector y una suma vectorial, y está definida como:

$$\mathbf{y} = \alpha\mathbf{x} + \mathbf{y} \quad , \tag{6.2.1}$$

donde α es un escalar, mientras que $\{\mathbf{x}, \mathbf{y}\}$ son vectores. En el listado 6.1 vemos el código fuente de una rutina en lenguaje C, la cual implementa la operación SAXPY descrita en la ecuación 6.2.1.

Listing 6.1: *Operación SAXPY en C nativo*

```
1 void saxpy(float* x, float* y, int n, float a) {
2   int i;
3   for (i = 0; i < n; i++) {
4     Y[i] = a * X[i] + Y[i];
5   }
6 }
```

La biblioteca de MKL (Math Kernel Library) [INT],desarrollada por la compañía INTEL, es una biblioteca de funciones matemáticas altamente optimizada y diseñada para correr con varios hilos en el CPU, todas las rutinas que requiren máximo rendimiento. Entre otras, la operación SAXPY está implementada en MKL y es una función que se declara cómo sigue:

```
cblas_saxpy(int n, float alpha, float X, int incX, float Y, int incY);
```

donde `n` es el tamaño de los vectores, `alpha` es un escalar y (`*X`, `*Y`) son apuntadores a las localidades de memoria donde se almacenan los vectores. Los parámetros `incX` e `incY` son el incremento de los vectores. Usualmente, `incX=1` y entonces el vector `X` corresponde directamente a un arreglo unidimensional. Si `incX>1`, esta variable especificará cuantos elementos en los arreglos se deberían "saltar" entre cada elemento del vector `X`, a continuación algunos ejemplos:

```
cblas_saxpy(100, a, x, 1, y, 1);
cblas_saxpy(50, a, &x[50], 1, &y[50], 1);
cblas_saxpy(100, a, x, 2, y, 2);
```

En la primera línea se sumarán los 100 elementos de `x` y `y`. En la siguiente línea sólo se sumarán los últimos 50 elementos de `x` y `y`. En la última línea sólo se sumarán los elementos con índice par de los arreglos. En las pruebas se usó el ejemplo de la primera línea.

CUBLAS [NVIa] es una implementación de BLAS sobre la plataforma de CUDA. Esta implementación esta contenida propiamente en la API de CUDA, permitiendo un acceso transparente a los recursos del GPU, sin necesidad de usar operaciones complicadas. El modelo básico con el cual las aplicaciones acceden a operaciones realizadas con CUBLAS es el siguiente:

1. Reservar la memoria para crear vectores y matrices dentro del GPU.

2. Llenar el espacio reservado con los datos

3. Llamar una secuencia de funciones de CUBLAS.

4. Subir los resultados del espacio de memoria del GPU a la memoria del *host*.

Además de lo antes mencionado, CUBLAS viene acompañado de una serie de funciones de ayuda para crear y destruir objetos dentro de la memoria del GPU, escribir y leer datos de la misma. En el listado 6.2 se muestra el código de la operación SAXPY usando CUBLAS. A continuación se describe como funciona dicho código:

- En las primeras 10 líneas se muestra un extracto de código que incluye los encabezados de definiciones (iostream, cuda, cublas, etc.), declaración e inicialización de variables. La variable `f` es de tipo `common` la cual es una estructura definida para almacenar datos y realizar algunas operaciones de la prueba como es la medida en tiempo de ejecución.
- En la línea 12 se inicializa la biblioteca de CUBLAS
- En las líneas 13 y 14 se reserva el espacio de memoria para los vectores `x` y `y`. Note que las variables `xptr` y `yptr`, son las variables que apuntan a los vectores dentro del espacio de memoria del GPU, mientras que las variables `x` y `y` apuntan a los arreglos alojados dentro del CPU (también llamado el *host*).
- En las líneas 15 y 16 se copian los datos alojados en el host a la memoria global de GPU.
- De la línea 20 a la 25 se realiza la prueba. La operación se realiza en el GPU con la función `cublasSaxpy`. Con el ciclo ubicado en la línea 21, se ejecuta la función las vece que la variable `maxIter` lo indique. `cublasSaxpy` tiene la notación estándar de BLAS, así que no cambia en nada con la función de MKL. Es importante ejecutar la función `cudaThreadSynchronize`, para asegurarnos que todos los hilos dentro del GPU han terminado de operar.
- En la línea 28 se copian los datos del vector `y` al host, para realizar algunas pruebas de precisión.
- En las líneas 29 y 30 se libera la memoria en el GPU.
- La línea 31 es necesaria para terminar correctamente el uso de la biblioteca CUBLAS.

Este código además contiene algunas funciones de las clases de utilerías que se desarrollaron en este trabajo para medir tiempos (`BenchAllocateStart`, `BenchAllocateStop`, `BenchRunStart`, `BenchRunStop`, `BenchRunStop`,`BenchDeallocateStop`), algunas funciones para leer y escribir datos (`ReadSize`, `ReadData`, `SaveResult`, `WriteData`).

Listing 6.2: *Operación SAXPY en el GPU*

```
1 #include <iostream> // y demas encabezados ...
2
3 int main (int argc, char *argv[]) {
4 long int maxIter = 1000;
5 int n = 1e6;
```

```
6  float alpha = 1.0f, *x = new float[n], *y = new float[n], *xptr, *yptr;
7  common f;
8  //
9  // inicializar los datos ....
10 //
11 f.BenchAllocateStart();
12 cublasInit();
13 cublasAlloc(n, sizeof(*y), (void**)&yptr);
14 cublasAlloc(n, sizeof(*x), (void**)&xptr);
15 cublasSetVector(n, sizeof(*x), x, 1, xptr, 1);
16 cublasSetVector(n, sizeof(*y), y, 1, yptr, 1);
17 f.BenchAllocateStop();
18
19 // Perform benchmark
20 f.BenchRunStart();
21 for(long int i = 0; i < maxIter; i++) {
22  cublasSaxpy(n, alpha, xptr, 1, yptr, 1);
23 }
24 cudaThreadSynchronize();
25 f.BenchRunStop();
26
27 f.BenchDeallocateStart();
28 cublasGetVector(n, sizeof(*yptr), yptr, 1, y, 1);
29 cublasFree(xptr);
30 cublasFree(yptr);
31 cublasShutdown();
32 f.BenchDeallocateStop();
33
34 f.SaveResult(y);
35 f.WriteData("output.CUBLAS_saxpy");
36
37    return 0;
38 }
```

Los ejemplos codificados para medir el rendimiento del código en C nativo y usando la biblioteca MKL, usan exactamente las mismas utilerías para escritura y lectura de datos, medición de tiempos, etc. Estos códigos se muestran en el Apéndice C.

En la figura 6.2(a) se muestra una gráfica con los tiempos utilizados por la operación SAXPY para los ejemplos con CUBLAS, MKL y C nativo. En esta prueba el tamaño de los vectores (N) es de 640,000 elementos y la operación se realizó 100,000 veces. Como se aprecia, el GPU es aproximadamente 5 veces más rápido que el código en C nativo y 2 veces más rápido que el de MKL. En una prueba con $N = 320,000$ y un número de iteraciones de $10,000,00$ los tiempos de cálculo que se obtuvieron se muestran en la figura 6.2(b). Por último, se realizó una prueba para un valor de N que no fuera múltiplo del número de multiprocesadores disponibles. Es importante tomar en cuenta lo anterior, pues este desbalance puede disminuir el rendimiento considerablemente. Para $N = 711,000$ se obtuvieron los tiempos mostrados en la figura 6.2(c). En todos los casos se observa la misma aceleración, es decir el código de CUBLAS es aproximadamente 5 y 2 veces más rápido que los correspondientes con C nativo y MKL. Finalmente, los

tiempos para subir y bajar los datos del CPU a la memoria del GPU se muestran en la figura 6.2(d). Se observa que ninguno de estos tiempos de transferencia afecta en las mediciones anteriores, pues son bajos comparados con el tiempo de cálculo.

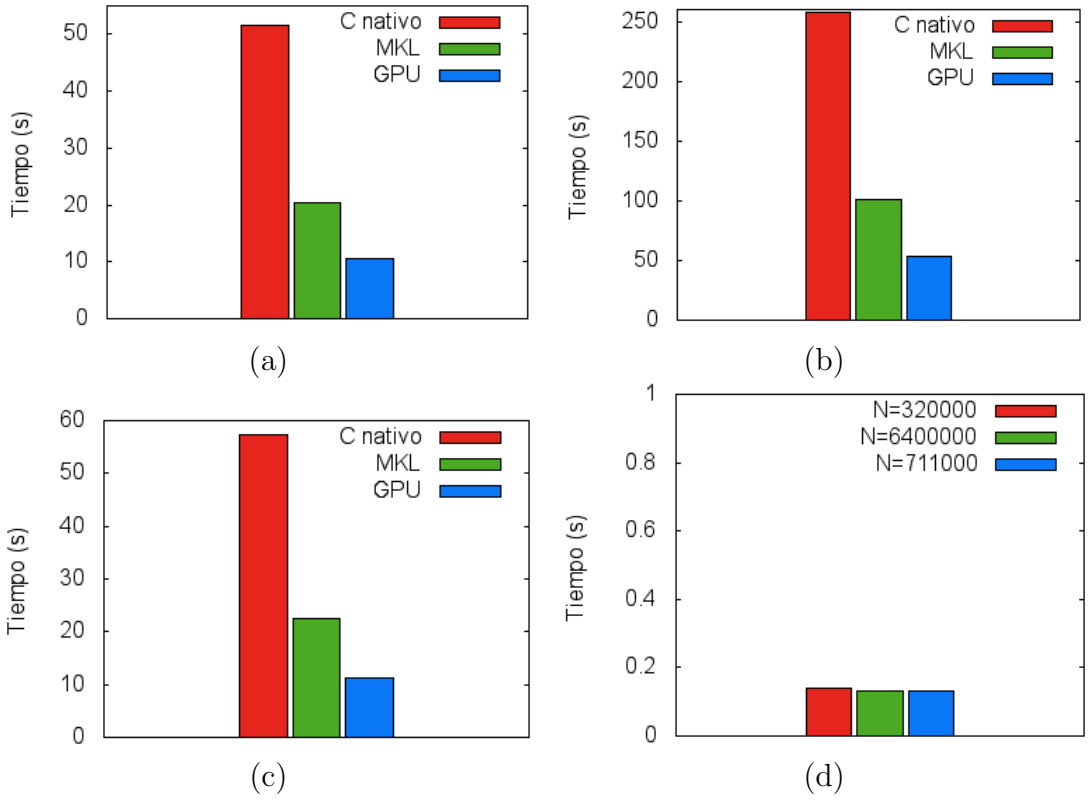


Figura 6.2: *(a) N = 640,000 y 10^5 iteraciones; (b) N = 320,000 y 10^6 iteraciones; (c) N = 711,000 y 10^6 iteraciones; (d) Tiempos de transferencia de datos del CPU al GPU.*

6.2.2. Prueba 2: Multiplicación Matriz por Matriz

La operación sgemm (Single-precision General Matrix Multiply) es una subrutina de BLAS que realiza la multiplicación de matrices y está definida como:

$$\mathbf{C} = \alpha\mathbf{AB} + \beta\mathbf{C} \quad , \tag{6.2.2}$$

donde α y β son escalares, mientras que $\mathbf{A}$, $\mathbf{B}$ y $\mathbf{C}$ son matrices de tamaño $N \times N$.

Es importante destacar que cuando se usan matrices en el cómputo de alto rendimiento es necesario tener una convención de como están representadas en memoria. Hay dos maneras de recorrer las matrices por renglones o por columnas. La figura 6.3(a), muestra los tiempos de recorrido por renglones y por columnas para 1000 iteraciones y diferentes valores de N.

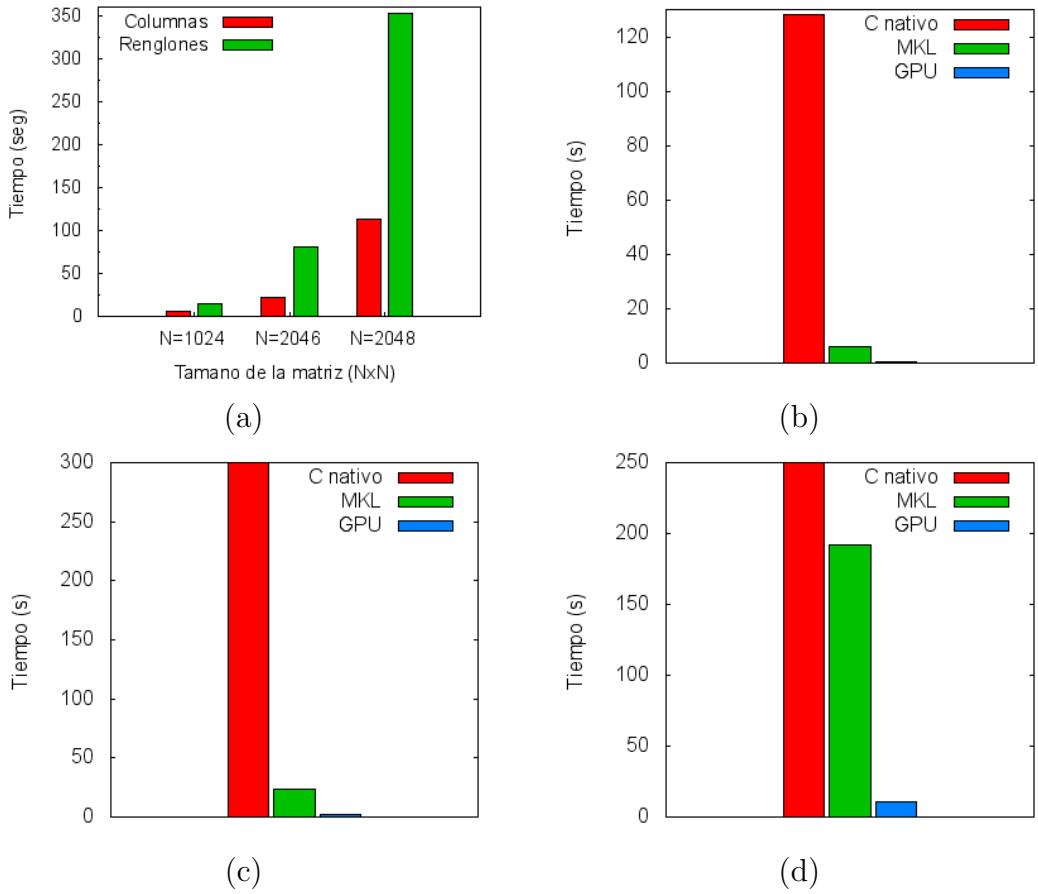


Figura 6.3: *(a) Tiempo de recorridos de una matriz por columnas y por renglones; (b) $N = 256$ y 2500 iteraciones; (c) $N = 512$ y 2500 iteraciones; (d) $N = 1024$ y 2500 iteraciones. Nota: las columnas para el caso de lenguaje C no se muestran completas en las gráficas (c) y (d).*

En la figura 6.3(b) se muestran los tiempos para el caso $N = 256$. Como se puede notar, el GPU es mucho más rápido que el código en C y es aproximadamente 10 veces más rápido que el código en MKL. Es de notarse que aquí se realizaron menos iteraciones que en SAXPY debido al tamaño y la complejidad del problema. La tabla 6.1 muestra los tiempos para $N = 256$, $N = 512$ y $N = 1024$, mientras que las figuras 6.3 (b)–(d) muestran estos datos gráficamente. Todas las pruebas fueron realizadas con 2500 iteraciones. Para $N = 1024$ el código en C ya tarda un tiempo bastante largo (horas), por lo cual ya no está dentro de la escala de consideración (segundos).

Sólo basta medir el tiempo de alojamiento de memoria y su destrucción para las matrices en el GPU. El tiempo de alojamiento se muestra en la figura 6.4(a), en donde se muestra que dicho tiempo se mantiene constante y es muy pequeño en comparación con el tiempo del cálculo. Respecto al tiempo para la destrucción, se muestra en la figura 6.4(b) que éste ya no es constante y crece con N. La influencia de este tiempo se nota

	256	512	1024
C nativo	128.14	3619.04	–
MKL	6.08	38.32	191.71
GPU	0.63	1.68	10.62
Alojamiento	0.14	0.15	0.15
Destrucción	0.15	1.14	3.59
T_C / T_{GPU}	**139.28**	**1837.08**	–
T_{MKL} / T_{GPU}	**6.61**	**19.45**	**13.35**

Cuadro 6.1: *Tiempos de cálculo y transferencia de memoria para la operación SGEMM con C, MKL y el GPU. En las últimas dos columnas se muestra la razón de tiempos de los códigos en C y MKL entre el tiempo en el GPU.*

especialmente en el ejemplo $N = 1024$, pues en este caso la razón de tiempos de MKL entre el de GPU baja con respecto del ejemplo $N = 512$, véase tabla 6.1.

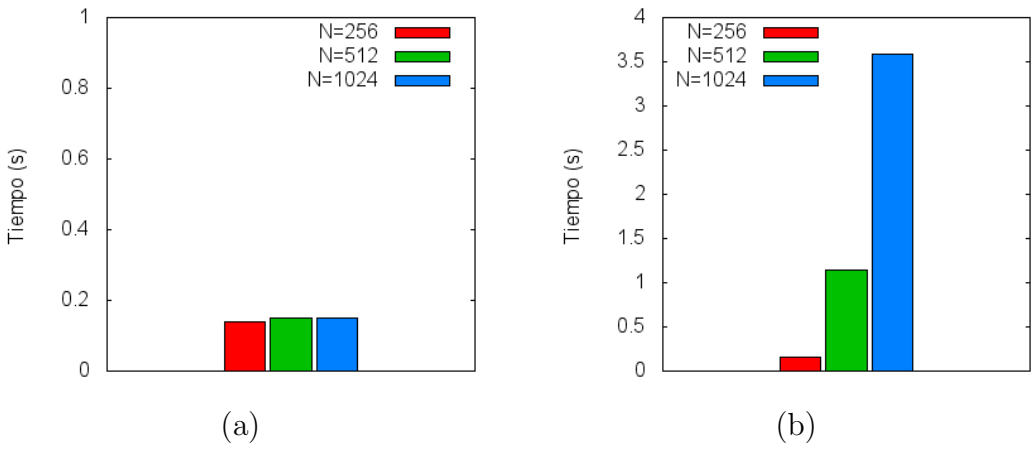


Figura 6.4: *(a) Tiempo de la transferencia y alojamiento en memoria del GPU de las matrices; (b) tiempo de destrucción de las matrices.*

6.3. Cálculo de fuerzas de dinámica molecular basado en CUDA

Muchos algoritmos paralelos para simulación de Dinámica Molecular han sido propuestos e implementados por diferentes investigadores ([MO99],[Pli95]). Los detalles de estos algoritmos varían demasiado, ya que son dependientes de la aplicación y de la arquitectura computacional que se usará. Generalmente desde el punto de vista de la descomposición de datos, estos pueden ser catalogados como:

1. **Descomposición de datos por partícula (DP)**. Cada procesador es asignado a un subconjunto de N/P partículas al inicio de la simulación, donde N es el número de partículas y P es el número de procesadores. Como cada procesador debe mantener copias idénticas de la información de las partículas, este método es también llamado de réplica de datos [Pli95]. Este método ha sido ampliamente usado en arquitecturas de memoria compartida.

2. **Descomposición con base en el cálculo de las fuerzas (DF)**. En este método un subconjunto del ciclo del cálculo de fuerzas inherente en el algoritmo 6 es asignado a cada procesador. Esto reduce costos en las comunicaciones y costos de memoria en un factor de $\sqrt{P}$ comparado con el método de descomposición por partículas. Sin embargo, DF puede no mantener el balance de carga tan fácilmente como DP [Pli95].

3. **Descomposición del dominio espacial (DS)**. Este método corresponde a la descomposición geométrica del dominio de simulación física. Cada procesador computa sólo las fuerzas sobre átomos en su subdominio correspondiente. Conforme la simulación avanza, los procesadores intercambiaran partículas cuando éstas se muevan de un subdominio a otro. DS es más deseable que DP y DF para simulaciones de dinámica molecular de gran escala, o con un Δt muy corto, y es muy usado sobre arquitecturas de grano grueso, como pueden ser los clusters [Pli95].

En este trabajo se toma ventaja del paralelismo inherente a las simulaciones y en particular del método de DP. Las principales razones para escoger este método son:

- Se puede alcanzar una buena escalabilidad y un buen balanceo de carga.
- De acuerdo al modelo de CUDA que se describió en la sección 6.1, el hardware del GPU puede ser visto como un sistema multiprocesador con memoria compartida, por lo tanto, el método de DP puede dar un buen rendimiento en tales sistemas, véase [Pli95].

El algoritmo en CUDA aquí implementado no cambia mucho con respecto al código en C de Moldynamics, ya que cada hilo se encargará de calcular las fuerzas de cada partícula. Como se muestra en la figura 6.5, la idea es que un hilo se encargue del número de partícula encerrado en rojo, y que calcule todas las interacciones de su columna tanto las interacciones negras como las grises (a diferencia del algoritmo en el CPU), al final solo dividiremos a la mitad todas las contribuciones a las fuerzas.

6.3.1. Fuerzas

Como se ha visto en secciones anteriores la parte más intensiva en cómputo es el cálculo de las fuerzas. En el apéndice D se da el listado completo del código fuente para el GPU. La explicación de algunas partes fundamentales de este código se da a continuación:

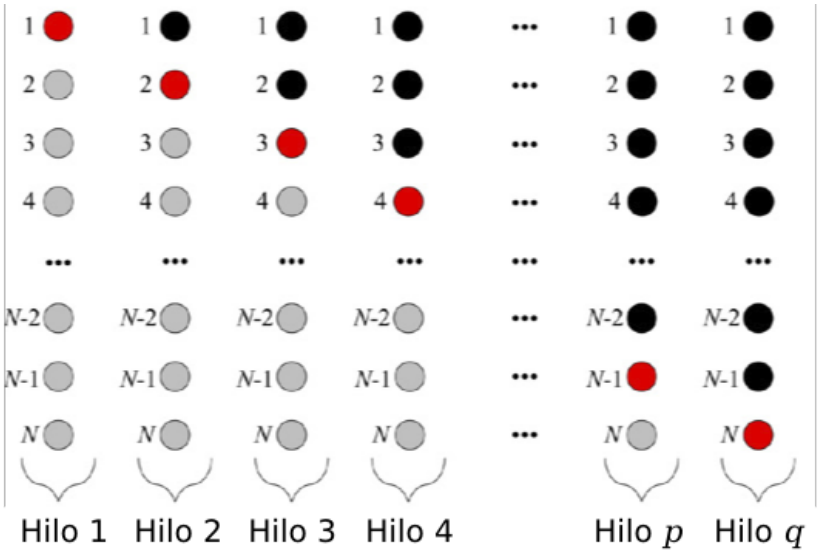


Figura 6.5: *Distribución de las partículas en los hilos*

- El kernel para este cálculo recibe como parámetro dos arreglos de tipo flotante de 4 posiciones, es decir cada elemento del arreglo contienen 4 espacios de memoria de tamaño flotante las cuales representan las coordenadas cartesianas. Un arreglo guarda las posiciones y otro las fuerzas de todas las partículas:

  ```
  __global__ static void MDGPU_ljForcesN2(float4* pPos, float4* pFor)
  ```

- Se obtiene el identificador de cada hilo con la siguiente macro:

  ```
  tid = threadIdx.x+blockDim.x * ( blockIdx.x+gridDim.x * blockIdx.y )
  ```

  Se utiliza `blockDim.x` y `blockDim.y`, porque las posiciones de las partículas están guardadas en una textura de dos dimensiones, entonces para obtener el índice del hilo el cual es unidimensional se tienen que obtener las coordenadas del bloque.

  Como se mostró en la sección 6.1 dentro de un GPU existen diferentes tipos de memoria. Establecido por [SK10], se dice que la memoria más sofisticada dentro de un GPU es la memoria de textura. Afortunadamente, esta memoria se puede usar tanto para procesamiento gráfico como para computo de propósito general.

  Como la memoria constante, la memoria de textura esta constituida como una memoria `cache` dentro del chip de procesamiento del GPU, esto hace que el acceso a la misma tenga una tasa de transferencia muy superior a otros dispositivos. Además accesos constantes a este tipo de memoria mejoraran el rendimiento de los programas. Específicamente los caches de textura están diseñados para el procesamiento de aplicaciones gráficas (optimizado con OpenGL [Ope] y DirectX [Dir]), y es por eso que exhiben un gran rendimiento para aplicaciones donde los hilos ejecutan tareas que estan *espacialmente* divididas, es decir, que un hilo requiere datos de hilos contiguos.

  En la aplicación que nosotros desarrollamos esta distibución *espacial* esta regida por la partícula y su posición.

- Se obtiene el valor de la posición de la partícula apuntada por el hilo `tid`, es importante recordar que este *kernel* lo ejecutarán todos los hilos, así que cada hilo obtendrá los valores de su posición (véase la figura 6.5) de la textura para llevarlos a la memoria cache. El número de partículas máximo que podemos simular con este código es 65,535, el cual es el valor máximo de un entero.

  ```
  f4Pos = tex1Dfetch( g_tPosTex, (int) tid );
  ```

- Las variables para almacenar las posiciones de las partículas están definidas de manera global como variables alojadas en memoria compartida del GPU:

  ```
  // Copia a la memoria compartida
  s_fvPosX[threadIdx.x] = f4Pos.x;
  s_fvPosY[threadIdx.x] = f4Pos.y;
  s_fvPosZ[threadIdx.x] = f4Pos.z;

  // sincroniza todos los hilos
  __syncthreads(
  ```

  La función `__syncthreads` es una barrera a donde tienen que llegar todos los hilos para seguir ejecutando el *kernel*. Si no se sincronizaran los hilos en este punto, entonces se podrían leer datos incorrectos y los cálculos posteriores serían también incorrectos.

  El GPU Tesla C1060 tiene 16384 bytes de memoria compartida por bloque, si un flotante mide 4 bytes, entonces podemos almacenar a lo más 1024 partículas en cada bloque.

- Cada hilo tiene una variable donde se van acumulando las contribuciones de las fuerzas calculadas, el nombre de esta variable es `f4For`, la cual es una estructura de datos de 4 flotantes, y cada elemento representa una coordenada cartesiana.

- El cálculo de las fuerzas de cada partícula se dividirá en 2 ciclos:

  1. Cálculo de las interacciones con las partículas (hilos) del mismo bloque:

     Listing 6.3: *Ciclo sobre las párticulas dentro del bloque inicial*

```
1    for (i=0; i<nBlockSize; ++i)
2    {
3      // Seobtiene la posicion de la memoria compartida
4      f4Pos2.x = s_fvPosX[i];
5      f4Pos2.y = s_fvPosY[i];
6      f4Pos2.z = s_fvPosZ[i];
7
8      // Se calcula la distancia
9      f4R.x = f4Pos.x - f4Pos2.x;
```

```
10      f4R.y = f4Pos.y - f4Pos2.y;
11      f4R.z = f4Pos.z - f4Pos2.z;
12
13      // Se aplica condiciones a la frontera periodicas
14      MDGPU_pbcDistance( f4R );
15
16      // Se calcula la fuerza de LJ que es una constante
17      fC = MDGPU_ljForce( f4R );
18
19      // Se suma la contribucion de la fuerza a la particula tid
20      f4For.x += f4R.x * fC;
21      f4For.y += f4R.y * fC;
22      f4For.z += f4R.z * fC;
23    }
```

Este primer ciclo es bastante sencillo: primero se obtienen las posiciones de las partículas de los otros bloques (líneas 4 a la 6), esas posiciones están almacenadas en memoria compartida. En la línea 17 se calculan las condiciones de mínima imagen. Esta función se detalla en el listado D.2 del apéndice XXXX, el detalle de las operaciones está contenido en el algoritmo 7.

2. Cálculo de las interacciones con las partículas de los demás bloques:

Listing 6.4: *Ciclo sobre los bloques que faltan*

```
1     // ciclo sobre todos los bloques que faltan
2     const int nBlocks = gridDim.x * gridDim.y;
3     for (k=1; k<nBlocks; ++k)
4     {
5       // se obtienen los indice del k-esimo bloque
6       i = tid + k * nBlockSize;
7
8       // i = i mod N
9       if (i >= c_nParticles) i -= c_nParticles;
10
11      // Asegurarse que todos han cargado correctamente sus
12      // datos en la memoria compartida
13      __syncthreads();
14
15      // Se obtienen las posiciones de las particulas de otro
            bloque
16      f4Pos2 = tex1Dfetch( g_tPosTex, (int) i );
17
18      // Se copian a memoria compartida
19      s_fvPosX[threadIdx.x] = f4Pos2.x;
20      s_fvPosY[threadIdx.x] = f4Pos2.y;
21      s_fvPosZ[threadIdx.x] = f4Pos2.z;
22
23      // Asegurar que todos lo hilos han cargado en memoria
            compartida
24      __syncthreads();
```

```
25
26      // ciclo sobre todas las particulas del bloque
27      for (i=0; i<nBlockSize; ++i)
28      {
29        // obtine la posicion del bloque en memoria compartida
30        f4Pos2.x = s_fvPosX[i];
31        f4Pos2.y = s_fvPosY[i];
32        f4Pos2.z = s_fvPosZ[i];
33
34        // calculo de la distancia
35        f4R.x = f4Pos.x - f4Pos2.x;
36        f4R.y = f4Pos.y - f4Pos2.y;
37        f4R.z = f4Pos.z - f4Pos2.z;
38
39        // se aplican condiciones a la frontera
40        MDGPU_pbcDistance( f4R );
41
42        // Calculo de la fuerza constante (fij)
43        fC = MDGPU_ljForce( f4R );
44
45        // Se suma la contribucion de la fuerza de esta particula
46        f4For.x += f4R.x * fC;
47        f4For.y += f4R.y * fC;
48        f4For.z += f4R.z * fC;
49      }
50    }
51    // Se escribe el total de la fuerza calculada
52    pFor[tid] = f4For;
53  }
```

En la línea 2 de este último código, se calcula el número total de bloques de la simulación, este cálculo es bastante sencillo. El número de bloques es bidimensional.

En este segundo ciclo se involucran las interacciones con las partículas fuera del bloque. Para hacer esto se requiere de dos ciclos anidados: el primero para apuntar a un bloque que falta usando la variable `k` y el ciclo interno para obtener los hilos del bloque actual usando la variable `i`. La idea de este segundo ciclo es pasar las posiciones de las partículas correspondientes a un bloque a la memoria compartida de otro bloque.

En la línea 6 se usa provisionalmente la variable `i` para almacenar el índice *unidimensional* del k-ésimo bloque. Este índice depende el hilo que esté ejecutando el código porque su dirección de bloque es distinto, así que esta línea se puede leer como el índice del k-ésimo bloque relativo al hilo ejecutado. Los últimos hilos tendrán una dirección mayor al número de partículas lo cual no puede ser posible, una manera de tratar esto es usar la operación módulo con el número de partículas (línea 9), con esto aseguramos direccionar las partículas correspondientes en una cola.

Una vez calculado el índice de la partícula en otro bloque se pueden leer las

posicones de esta partícula de la textura y copiarla a memoria compartida (líneas 16 a 21). Posteriormente se sincronizan los hilos para no leer datos incorrectos en el ciclo interior.

Ya que se intercambiaron los espacios de memoria entre los bloques, cada hilo dentro del bloque calcula sus interacciones con las particulas del mismo de acuerdo a lo explicado en pasos anteriores. Esto se realiza en el ciclo interior que va de las líneas 27 a la 50. También es necesario calcular las condiciones de mínima imagen y la contribución de la fuerza.

Al final de estos ciclos se tendrán todas las contribuciones de todas las partículas, así que cada hilo escribe su fuerza total en el arreglo que será utilizado para realizar la integración.

En la línea 43 se calcula la constante de fuerza de tipo Lennard Jones. Esta rutina sólo requiere la distancia de las partículas almacenada en `f4R`. La función `MDGPU_ljForce`, calcula una constante que se requiere para escalar las posiciones y obtener la contribución de la interacción en las 3 coordenadas cartesianas de las partículas. Esto se realiza en las líneas 46-48. Los detalles de la rutina `MDGPU_ljForce` son mostrados en el listado D.3, el detalle de estas líneas es explicado en el algoritmo 6. En este punto sólo queda destacar que dentro del código de la rutina `MDGPU_ljForce`, se discriminan las partículas que están fuera del radio de corte, y la interacción de la partícula con ella misma.

Es importante remarcar algunas cosas:

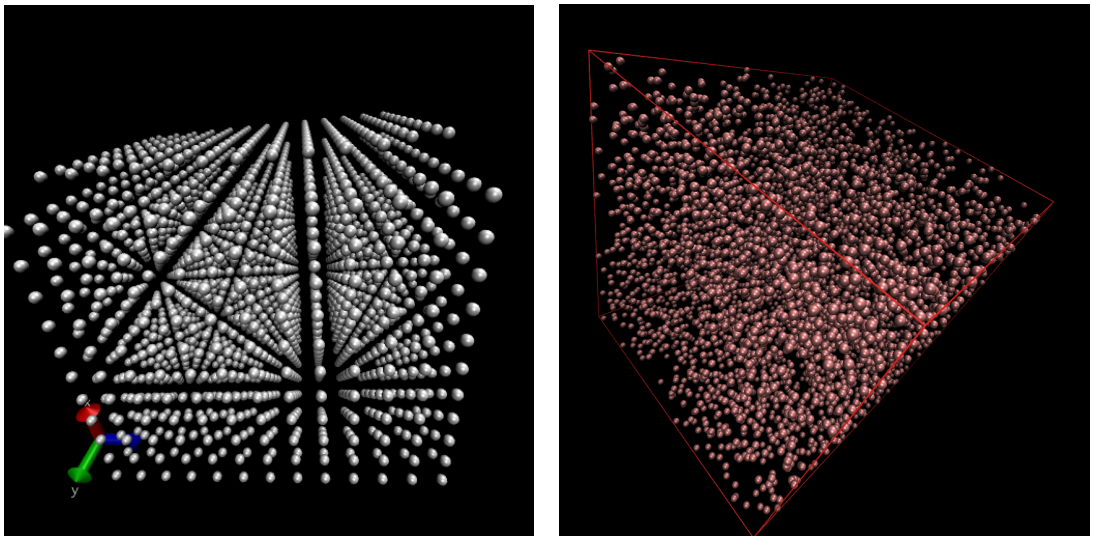
- El cálculo de las fuerzas se realiza en T número de pasos, es decir el *kernel* listado en 6.4 se ejecutará T veces.
- Para conseguir un buen rendimiento en el GPU se realiza un intercambio de bloques entre los hilos, y entonces poder usar la memoria compartida de los mismos. Esto es fundamental, de otra manera no se obtiene reducción del tiempo de cálculo en el GPU, e incluso el código puedes ser más lento que en un CPU.
- El uso de rutinas de dispositivo como son `MDGPU_pbcDistance` y `MDGPU_ljForce` hacen los códigos mas legibles en CUDA y disminuyen los errores de codificación.
- El direccionamiento de los hilos es fundamental, y en estos códigos se realizan macros para no tener que estar repitiendo código, esto también evita errores.
- El uso de texturas también acelera el cálculo ya que el GPU esta optimizado para acceder a la memoria de esta manera.
- La integración se realiza como lo muestra el algoritmo 2. Lo cual es bastante sencillo en el GPU ya que es una simple sumas de vectores.

- Toda la simulación es calculada dentro del GPU sólo hay una transferencia del CPU al GPU al inicio de la simulación donde se copian algunas constantes, las posiciones y las velocidades iniciales. Posteriormente se realizan todos lo pasos de integración dentro del GPU. Y finalmente se obtienen algunas medidas escalares como son: la energía cinética, la energía potencial, la presión y la temperatura

Capítulo 7

Resultados

Una simulación en dinámica molecular comienza con un arreglo de partículas uniformemente distribuidas a manera de cristal (véase la figura 7.1(a)). El cálculo de la interacción de partículas se realiza como se explicó en capítulos anteriores, de tal manera que, luego de unos cuantos pasos de integración, las partículas se han movido a posiciones distintas, (véase la figura 7.1(b)). En lo que sigue se mostrarán los resultados obtenidos para diferentes casos de estudio.



(a) (b)

Figura 7.1: *(a) Un arreglo cristalino de partículas al inicio de una simulación($N = 2000$); (b) Caja de simulación después de unos cuantos pasos obtenida con Moldynamics ($N = 2000$).*

7.1. Formación de estructuras moleculares

En la figura 7.2 se presentan los perfiles de densidad para una caja de simulación con las siguientes propiedades: $T = 0.85$, $\rho = 0.1$ y $N = 2000$, donde T es la temperatura del sistema, ρ la densidad y N el número de partículas. Los resultados mostrados en la figura 7.2 fueron obtenidos después de 10^6 pasos de simulación. En estas gráficas se muestra la variación de la densidad de partículas en rebanadas que son paralelas a los planos coordenados. Por ejemplo, se toman rebanadas paralelas al plano xy de un ancho δz, y en esa rebanada se calcula la densidad de partículas. Entonces se grafica esta densidad contra la posición en la dirección z. Esto se hace también para las otras direcciones (x y y). En la figura 7.2 se muestran tres gráficas correspondientes a estas direcciones. En estas gráficas se observa que la densidad de las partículas es mayor hacia el centro de la caja, de tal manera que esto corresponde a la formación de una esfera. Es importante recordar que estos perfiles son un promedio a lo largo de toda la simulación.

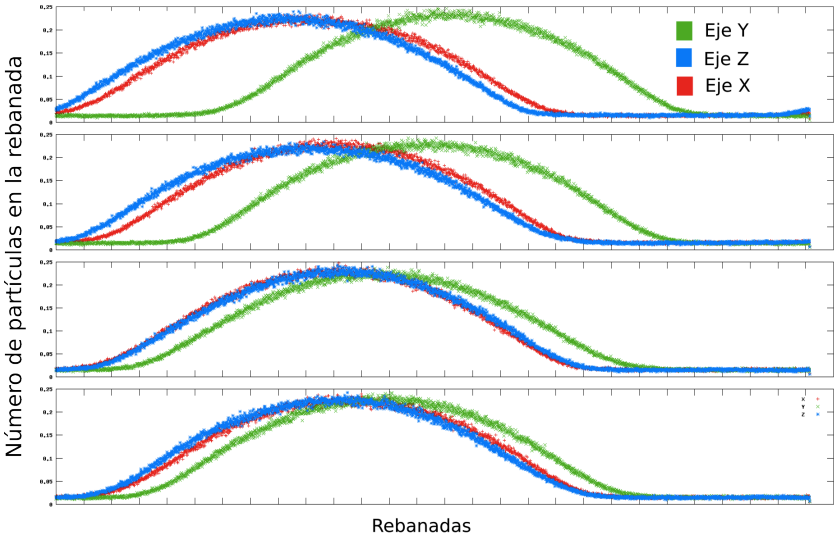


Figura 7.2: *Perfiles de densidad que muestran la formación de una esfera con* 2000 *partículas. Cada perfil fue tomado a los* 100000, 200000, 300000, 400000, 500000 *pasos (en orden descendente).*

La figura 7.3 muestra la distribución de las partículas dentro de la caja. Se observa la acumulación de partículas en en centro de la caja formando una esfera.

En la figura 7.4 se presentan estructuras obtenidas con el software Moldynamics. En las figuras 7.4(a) y (b) se muestra una estructura cilíndrica obtenida a los 60000 en una simulación con densidad de $\rho = 0.22$. Los perfiles mostrados en 7.5 pertenecen a esta estructura. En la figura 7.4(c) y (d) se muestra una estructura en forma de losa obtenida a los 50000 en una simulación con densidad de $\rho = 0.48$. Los perfiles mostrados en 7.6 pertenecen a esta estructura. Estos resultados concuerdan con los reportados por *Errington et al.* en [MSE06].

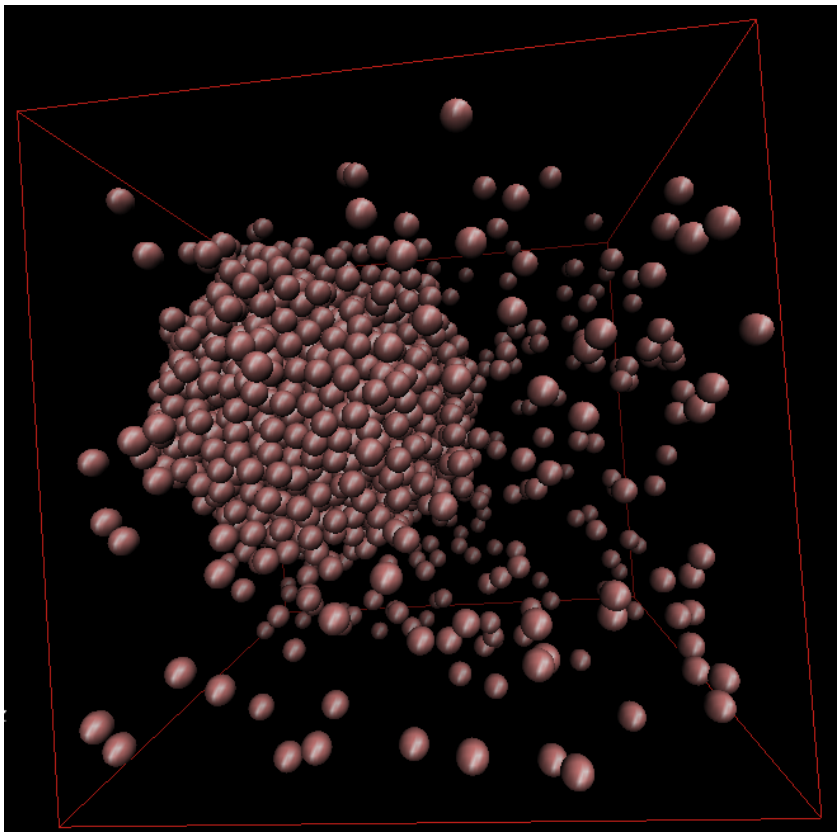


Figura 7.3: *Formación de una esfera en la caja de simulación con* 2000 *partículas (tiempo de formación 72400 pasos). Los perfiles mostrados en la figura 7.2 corresponden a esta estructura.*

Después de un cierto número de pasos, los perfiles de densidad pueden confirmar la formación de algunas estructuras, sin embargo, no es posible determinar las formas características de las estructuras (tamaño de la esfera, el radio del cilindro, etc). Esto debido a que los perfiles de densidad son un promedio de la distribución de las partículas en cada rebanada de la caja. Si deseamos observar estructuras características es necesario la inspección visual de las mismas ó usar una técnica de clustering [YM06, BCX$^+$06, Nak97].

Es importante destacar que durante la simulación, aparecen estructuras interesantes antes de la formación total de las estructuras finales. Por ejemplo, antes de crearse la esfera se observa la formación de un cierto número de islas, las cuales se van agregando a una esfera cada vez mas grande (véase figuras 7.7(a)–(b)). De igual manera, antes de la formación del cilíndro, aparecen durante la simulación una serie de tubos conectados (véase figura 7.7(c)–(d)).

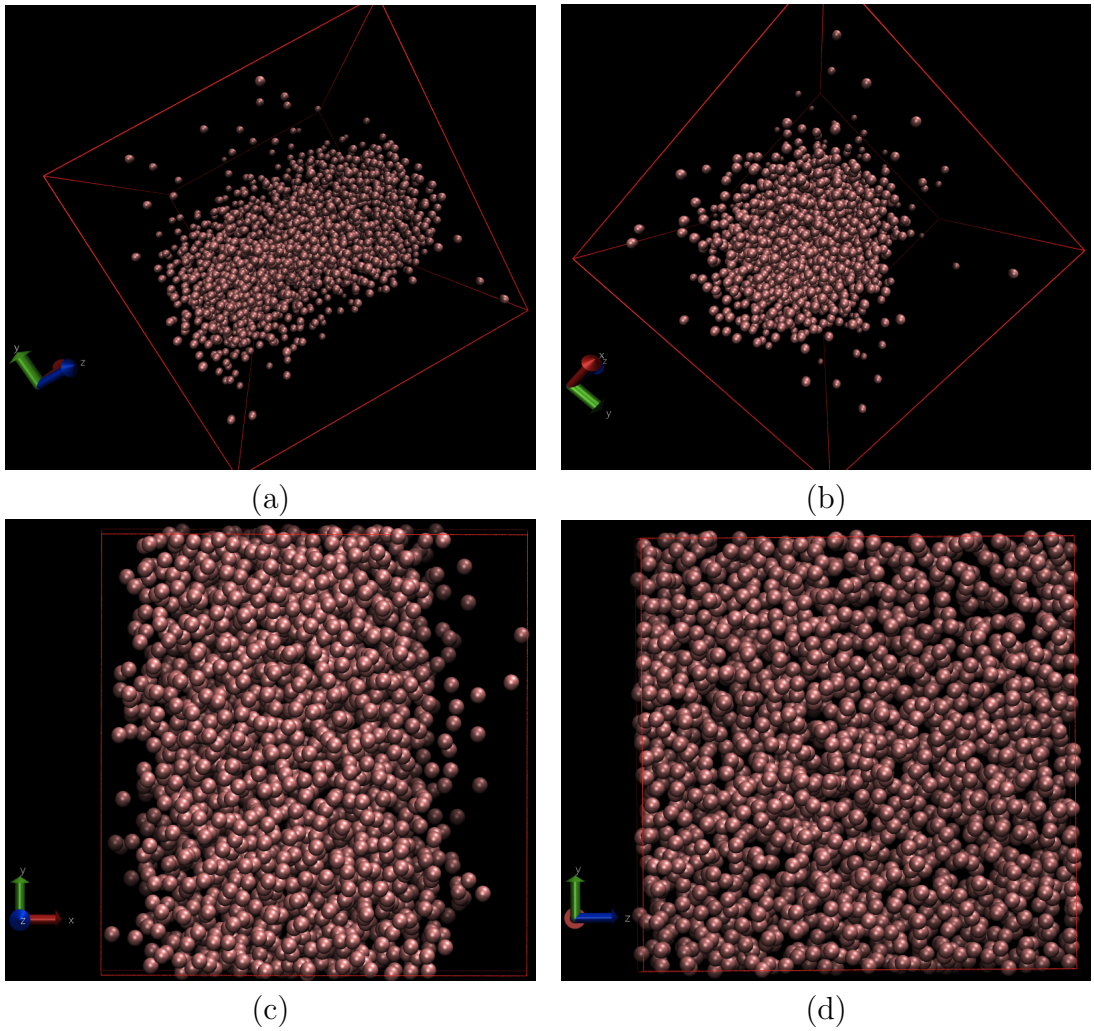


Figura 7.4: *(a) Cilíndro después de 60000 pasos de simulación; (b) Cara frontal del cilindro; (c) Losa de partículas después de 50000 pasos de simulación; (d) Toma de frente de la losa.*

7.2. Caso de estudio

La curva que describe la presión como función de la densidad o el volumen debido a la relación $\rho = N/V$, considerando la temperatura constante, se conoce en este ámbito como isoterma. El cálculo de la presión no es un problema trivial como se vió en 3.2.6.

Las isotermas nos ayudarán a describir el comportamiento a través del Loop de Van Der Waals (véase [MSE06]) para un líquido tipo Lennard-Jones (véase figura 1.1). Es importante recordar que en las isotermas de este trabajo, cada punto es el promedio de la traza del tensor de presión calculado en cada paso en coordenadas cartesianas de acuerdo al algoritmo 8. Las simulaciones se realizaron para al menos 10^6 pasos de tiempo.

Por otro lado, es importante conocer el diagrama de fase de un sistema. Este dia-

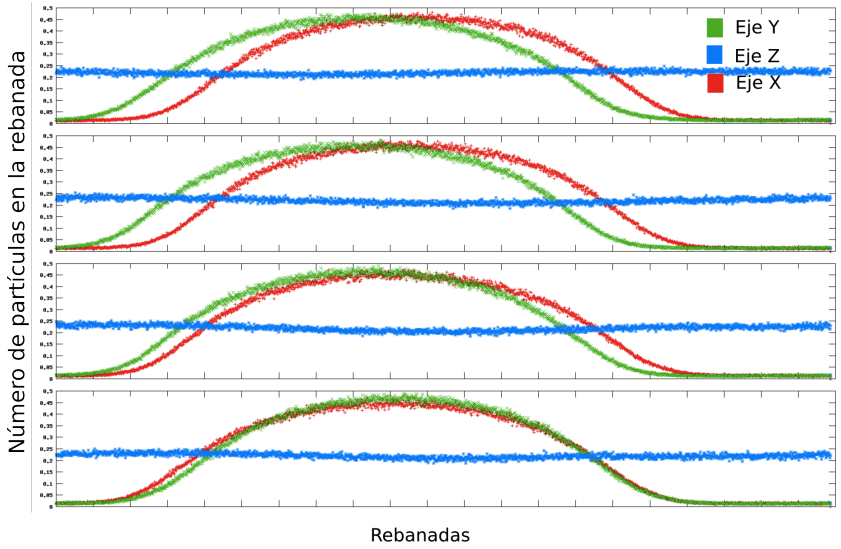


Figura 7.5: *Perfiles de densidad que muestran la formación de un* ***cilíndro*** *con* 2000 *partículas. Cada perfil fue tomado a los* 100000, 200000, 300000, 400000, 500000 *pasos (en orden descendente).*

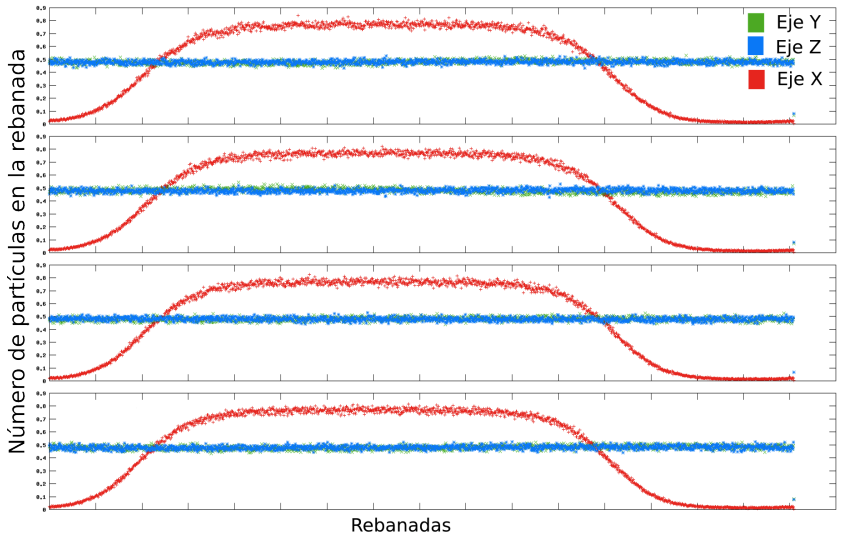


Figura 7.6: *Perfiles de densidad que muestran la formación de una* ***losa*** *con* 2000 *partículas. Cada perfil fue tomado a los* 100000, 200000, 300000, 400000, 500000 *pasos (en orden descendente).*

grama es una gráfica de temperatura como función de la densidad. Los puntos de este diagrama determinan la curva de coexistencia o curva binodal, véase figura 7.8. Esta curva binodal describe como se divide un sistema durante el barrido de las densidades. Por ejemplo, en el caso de la esfera, la densidad dentro de ella tendrá un valor que estará a la derecha de la curva binodal, mientras que la densidad fuera de la esfera estará a

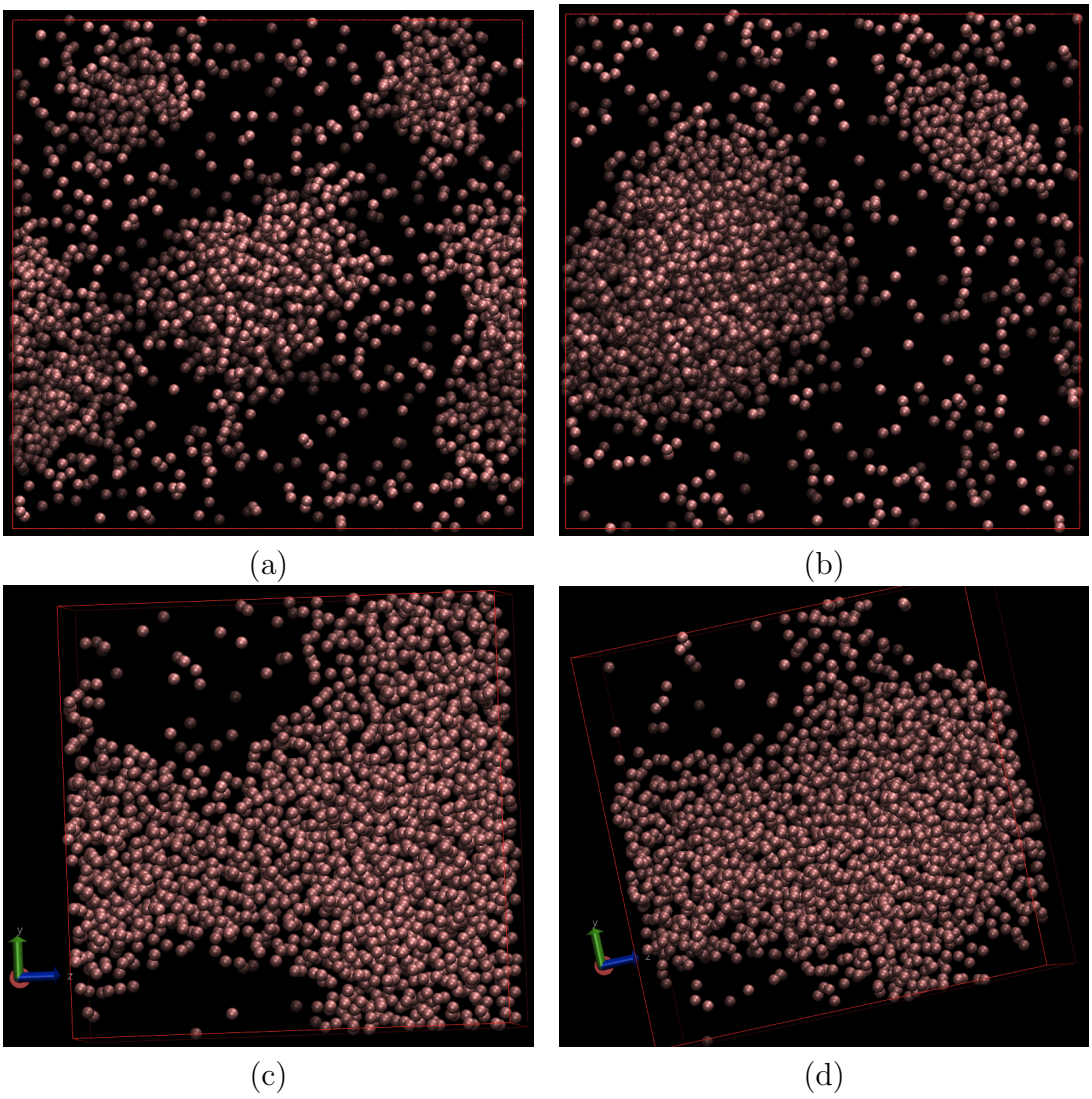


Figura 7.7: *Islas antes de la formación de la esfera. (a) 5 islas en 320 pasos; (b) 2 islas en 475 pasos. Tubos conectados antes de la formación del cilíndro. (c) Tubos formados en 285 pasos; (d) Tubo en camino de formación (2010 pasos).*

la izquierda de dicha curva. Esta gráfica también contiene la curva espinodal la cual nos permite diferenciar los estados estables, metaestables e inestables, véase figura 7.8. En esta figura la curva azul es la curva binodal, mientras que la curva roja es la curva espinodal del sistema. Este diagrama debe tener clara concidencia con la gráfica de presión contra densidad, ya que el máximo del loop de Van Der Waals, debe coincidir con el punto izquierdo de la curva espinodal en el diagrama de fase, mientras que el mínimo del loop de Van Der Waals debe coincidir con el punto a la misma temperatura del lado derecho de la curva espinodal. Es importante notar que el diagrama de fase se puede obtener experimentalmente o teóricamente, y sirve de guía para saber que densidades

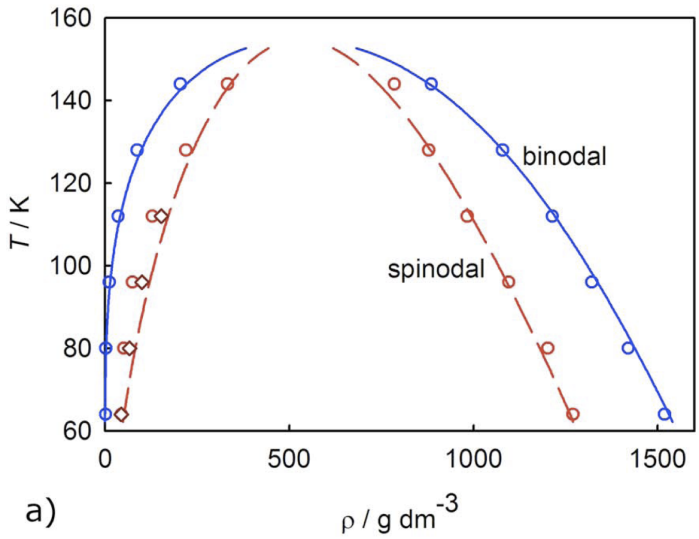


Figura 7.8: *Diagrama de fase que muestra la curva de coexistencia*

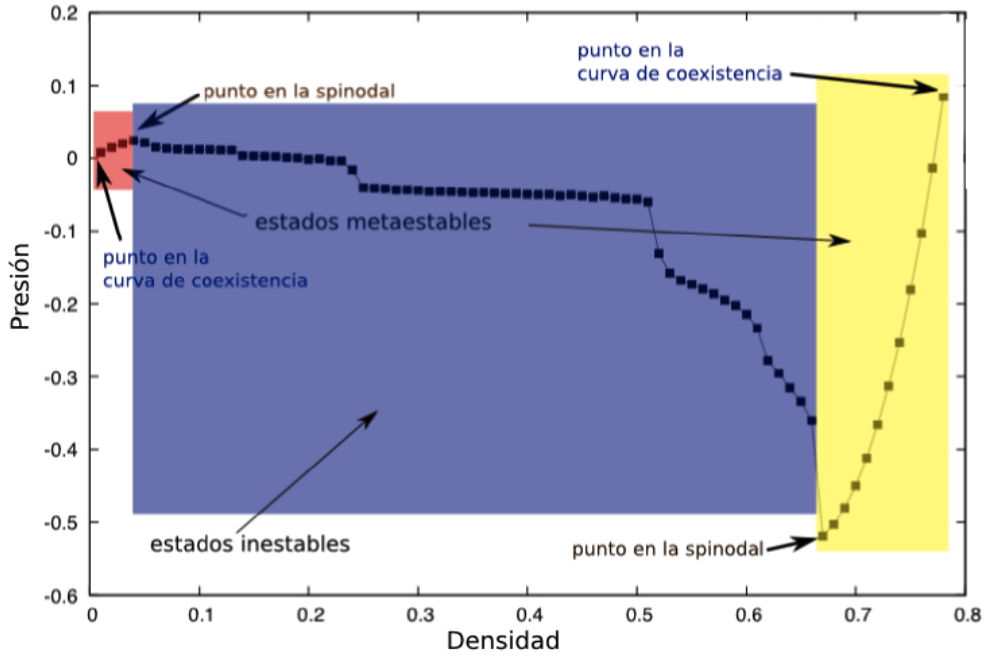
se obtienen del sistema cuando se hace estable.

Es importante conocer el diagrama de fase debido a que existe plena coincidencia entre éste y el loop de Van der Walls. De hecho las isotermas las podemos considerar como cortes transversales del diagrama de fase, esto ayudará a saber que propiedades y estados debemos obtener en nuestro sistema.

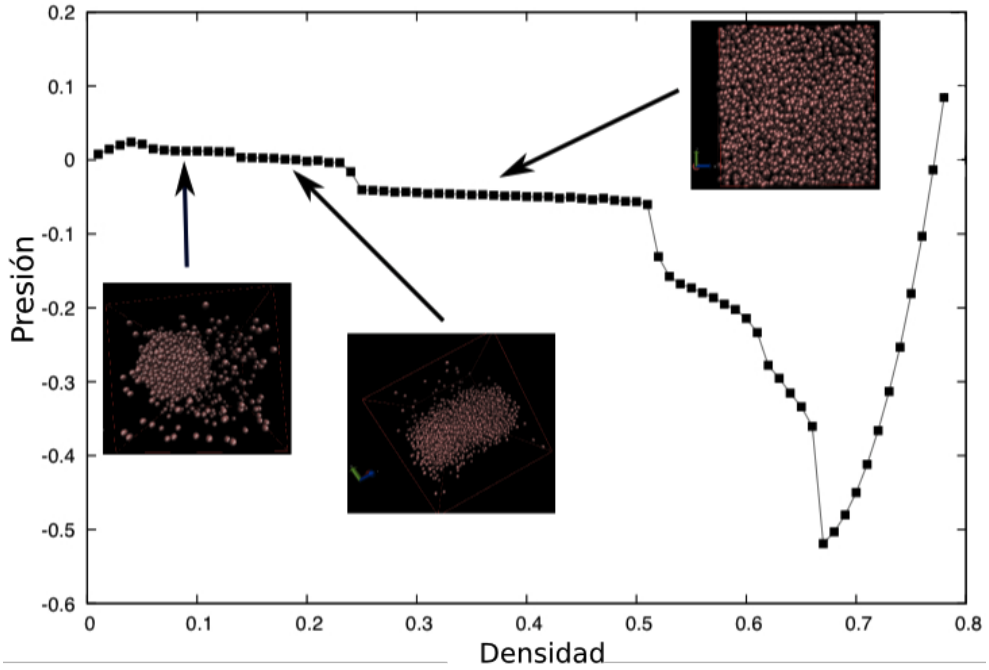
En la figura 7.9(a) se muestran los estados y los puntos de coincidencia con los diagramas de fase. Así con varias isotermas solo se puede conseguir la curva espinodal del diagrama de fase pero no la curva de coexistencia, esta última se puede obtener simulando con un ensamble diferente como NPT (Véase [HÖ5]). Entonces con la figura 7.9(a), podemos describir que sistemas se obtendrán:

- En densidades aproximadamente menores a 0.04 se obtendrán sistemas homogéneos, es decir, si inspeccionamos las cajas visualmente veremos todas las partículas distribuidas y desordenadas en la caja.
- En densidades entre 0.04 y 0.67 vemos los estados inestables donde no se puede predecir un tipo de formación. En teoría dentro de la caja el sistema se debería de partir en dos densidades, las densidad del cúmulo cílindrico, esférico etc. debe ser similar a 0.67, pero debido a que los volúmenes de los cúmulos son extraños es muy difícil medir la densidad.
- Para densidades aproximadamente mayores a 0.67 se deben obtener sistemas homogéneos.

En este trabajo lo que se desea saber es que pasa con el sistema dentro de la curva de coexistencia o dentro del loop de Van Der Valls. Como se explicó en la sección anterior, las estructuras que se forman están distribuidas sobre la isoterma como lo muestra la figura 7.9(b). Cada una de las estructuras, las esferas, los cilindros, los bloques y



(a)



(b)

Figura 7.9: *Curva obtenida para una simulación de 2000 partículas a una temperatura de* 0.85. *(a) Ubicación de los puntos de la espinodal y la curva de coexistencia. (b) Estructuras moleculares en forma de esfera, cilíndro y bloque ubicadas en los cambios sobre la curva de la isoterma.*

(las cavidades esféricas y cilíndricas), muestran cambios notables y bien definidos en la isoterma. Estos cambios son discretos, es decir se podría inducir que el loop de Van Der Waals en su interior no es continuo, y además los cambios están bien definidos como ya lo había previsto *Errington et al.* en [MSE06].

7.3. Aceleración y precisión del GPU

El costo computacional del cálculo de una isoterma típica se puede obtener de la siguiente manera: Para la simulación de la figura 7.9(b) el tiempo que tardan en ejecutarse 10^5 pasos es de 1 hora aproximadamente en una arquitectura Intel(R) Xeon(R) CPU E5440 @ 2.83GHz, se sabe que en la mayoría de los casos el sistema es estable después de 10^6 pasos. Esto da un costo en tiempo de cómputo de 10 horas. El número de puntos calculados para esta isoterma es 80, es decir, se realizaron 80 simulaciones lo cual nos acercó bastante a la solución del problema pero lo deseable sería realizar al menos el doble de las simulaciones. El costo en tiempo de cómputo de 80 simulaciones es 800 horas, es decir aproximadamente 33 días de cómputo en un sólo procesador. Este dato es lo más importante de este trabajo ya que es el motivo del mismo. Sería muy tardado calcular la curva espinodal con este método y obtener las estructuras para cada isoterma, entonces se vuelve fundamental encontrar arquitecturas de cómputo que reduzcan el tiempo de cálculo de estas simulaciones.

7.3.1. Simulaciones convencionales

La figura 7.10 muestra los tiempos de cálculo durante una simulación de dinámica molecular usando un CPU y un GPU para distinto número de partículas. Las simulaciones se realizaron para 100000 pasos de integración, a una temperatura de 0.85 con una densidad de 0.1 que es donde se obtiene la esfera en las estructuras moleculares.

Los tiempos de cálculo en el GPU y en el CPU se muestran en la tabla 7.1. Como se observa, conforme se aumenta el número de partículas la razón de tiempo de cálculo en el CPU (T_{CPU}) entre el tiempo de cálculo en el GPU (T_{GPU}) aumenta. Esto coincide con los resultados obtenidos por NVIDIA [NVIc]. Además, esto comprueba que cuando el GPU realiza cálculos para un número de partículas relativamente chico, esta razón es cercana a uno. Sin embargo, cuando se tiene una cantidad muy grande de partículas, se puede aprovechar la potencia del GPU para realizar operaciones aritméticas más rápido.

No. de partículas	256	512	1024	2048	4096
T_{CPU} / T_{GPU}	1.5	2.8	5.5	10.5	20

Cuadro 7.1: *Razón T_{CPU} / T_{GPU} para distinto número de partículas*

Nótese que el crecimiento de T_{CPU} / T_{GPU} es exponencial, lo cual es de espererse debido a que el tamaño del problema también crece de manera exponencial.

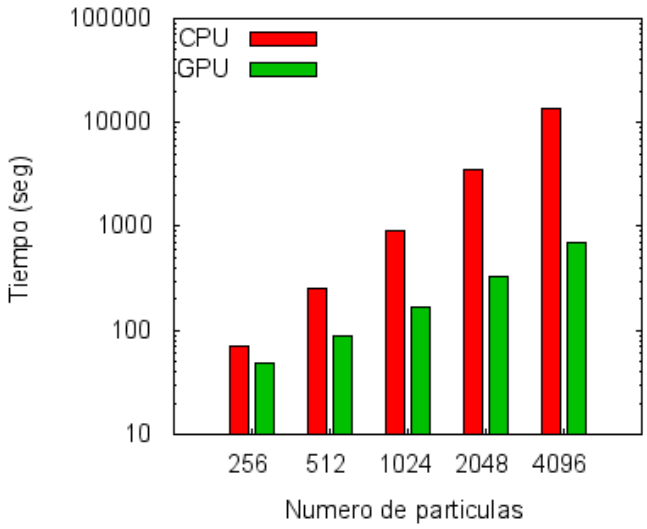


Figura 7.10: *Tiempos de cálculo del GPU y el CPU para diferente número de partículas*

7.3.2. Simulación de un sistema de partículas muy grande

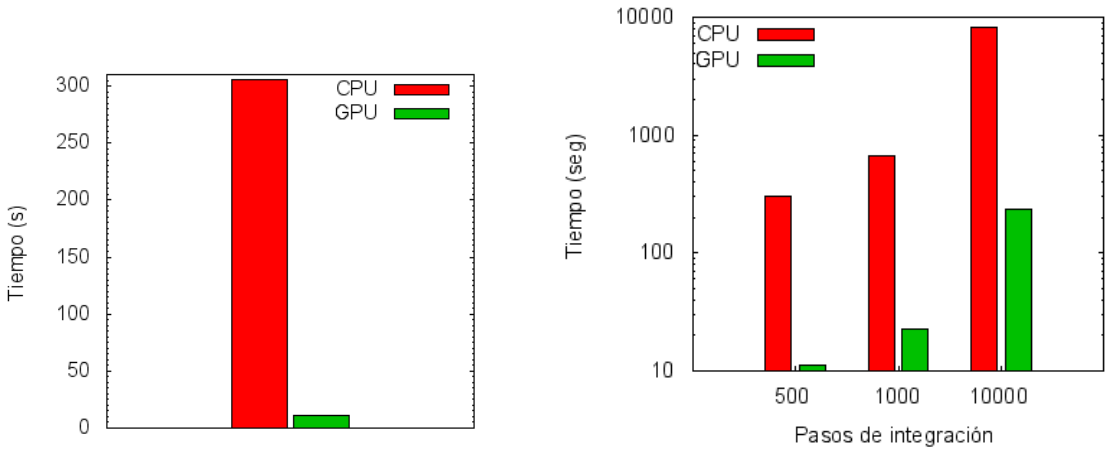
En la figura 7.11(a) se muestran los resultados para un sistema considerablemente grande el cual es de 10240 partículas. Este sistema es mucho más grande que aquel donde se obtuvieron las estructuras moleculares el cual fue de 2048 partículas. Para esta simulación se realizaron 500 pasos de integración, a una temperatura de 0.85 y una densidad de 0.1.

Arquitectura	500	1000	10000
GPU	11.2	22.71	235.8
CPU	305.7	664.14	8157.15

Cuadro 7.2: *Tiempos de cómputo para diferentes cantidades de pasos*

El GPU es aproximadamente 27.3 veces más rápido que un CPU. Si decidiéramos obtener una isoterma en el CPU con 10^6 de pasos de integración, obtendríamos que la simulación de un sistema de una densidad dada tardaría aproximadamente 7.07 días. Si la isoterma tuviera 80 densidades diferentes esto requeriría 566 días, es decir 1.5 años. Sin embargo en el GPU una simulación de 10^6 solo tomaría 6.2 horas. El cálculo de la isoterma con 80 densidades diferentes, como el generado para encontrar las estructuras moleculares de este trabajo, solo requiere 20 días. Este simple cálculo es lo que impresiona a expertos en el campo de la dinámica molecular. Es importante remarcar que la medida de estos tiempos es únicamente de los pasos de integración, es decir, lo realizado de las líneas 2 a la 17 en el algoritmo 4.

La figura 7.11(b) muestra una comparación de los tiempos del GPU y CPU, para diferente número de pasos de integración. Obsérvese que en el eje de tiempos se usó una escala logarítmica, pues los tiempos que toma el CPU son muy grandes comparados



(a) (b)

Figura 7.11: *(a) Tiempos obtenidos en el GPU y en el CPU para N=10240. (b) Tiempos de cálculo para diferentes pasos de integración entre GPU y el CPU.*

Arquitectura	1 densidad (días)	Isoterma (días)	Isoterma (años)
GPU	7.07	566	1.5
CPU	0.258	20	-

Cuadro 7.3: *Aproximación sobre el tiempo de cálculo de una Isoterma con* 10^6 *pasos*

con los que toma el GPU. Esta gráfica muestra como el GPU también puede reducir el tiempo de cálcuo respecto al numéro de pasos de integración, aunque la ganancia es mínima. La tabla 7.4 muestra los valores obtenidos de la razón T_{CPU} / T_{GPU}. Estos resultados muestran que es importante realizar cálculos para un número elevado de partículas y durante un periodo de tiempo grande.

	500	1000	10000
T_{CPU} / T_{GPU}	27.3	29.2	34.6

Cuadro 7.4: *Razón* T_{CPU} / T_{GPU} *para distinto número de pasos de integración con* $N = 10240$

7.3.3. Transferencia de datos entre CPU y GPU

En la figura 7.12 se muestra una gráfica del tiempo que se tarda la transferencia de datos de varios sistemas de partículas, de la memoria del CPU a la memoria global del GPU. La transferencia involucra varias número de punto flotante, como son la temperatura, las dimensiones de la caja, etc. También se deben transferir dos arreglos que se usan para almacenar las posiciones y las velocidades. La figura 7.12 muestra que el tiem-

po de transferencia crece con el número de partículas, de la misma manera como crece el número de datos, y en este caso es exponencial. Sin embargo, algo que es muy importante es que los tiempos de transferencia son relativamente pequeños con respecto a los tiempos del cálculo numérico. Estos tiempos de transferencia serían muy importantes si se deseara hacer intercambios durante los pasos de integración. Esto sucede cuando se deben almacenar los resultados para realizar una inspección de los mismos una vez finalizada la simulación. Dado que el análisis de los datos es siempre importante, este hecho impone limitaciones en la simulación de una dinámica molecular en el GPU. Por ejemplo, la generación de animaciones, las cuales son muy importantes en la inspección visual, los perfiles de densidad que son de mucha utilidad para encontrar estructuras promedio a lo largo de la simulación, ó los perfiles de presión que son indispensables en el estudio de la formación de las esferas, cilíndros etc.

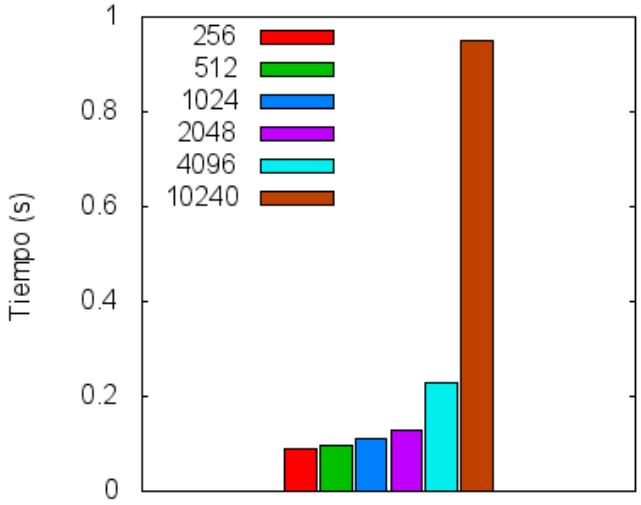


Figura 7.12: *Tiempo de transferencia de memoria entre el CPU y el GPU*

Respecto a la velocidad de transferencia del GPU al CPU, incluyendo la liberación del espacio de memoria en el GPU, en este caso el tiempo es prácticamente cero, ya que al final de la simulación sólo se requieren algunos escalares.

7.3.4. Precisión del GPU

La precisión de un cálculo se mide con el valor de la traza del tensor de presión, véase sección 3.2.6. Esta cantidad es acumulativa sobre los pasos de integración, lo que puede dar una muy buena medida de precisión: se mide el resultado en el CPU y se compara con el obtenido en el GPU. Dado que los códigos en el CPU han sido validados por varios autores, si el resultado del GPU se aproxima a su contraparte en el CPU entonces se tendrá buena precisión. En la figura 7.13 se muestran los resultados para algunos los sistemas usando 100000 pasos de integración, una temperatura a 0.85 y una densidad de 0.1.

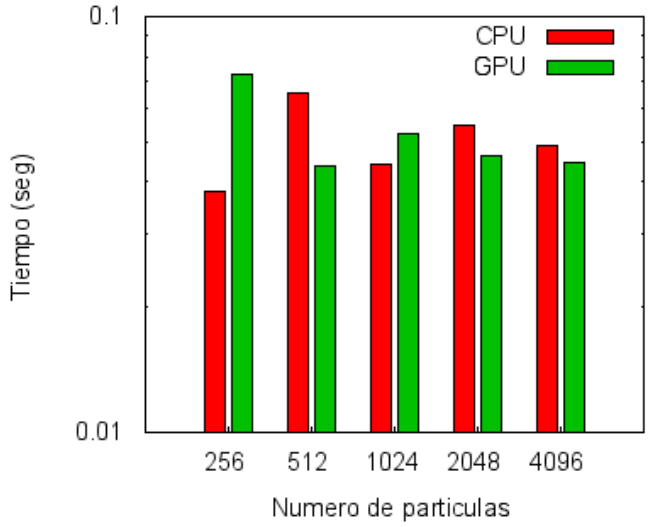


Figura 7.13: *Precisión entre el CPU y el GPU*

La gráfica de la figura (7.13) muestra el valor obtenido de la presión en el CPU y en el GPU para diferentes números de partículas, todo en simple precisión. Como se observa el cambio en el valor de la presión es muy pequeño. En dinámica molecular es aceptable un error de 0.1 en la presión. En nuestro caso el error obtenido es menor. Si se hiciera el cálculo en doble precisión, se obtendría un mejor resultado, pero los tiempos de cálculo cambiarían.

Capítulo 8

Conclusiones

En este trabajo se revisó un método de simulación molecular, basado en el potencial de Lenard Jones, desde un punto de vista algorítmico y computacional. Este es uno de los pocos trabajos que incluye la descripción de los desarrollos computacionales en detalle y que además, propone una estrategia de implementación en GPUs que acelera de manera considerable el cáculo de las distribuciones moleculares.

El uso de GPUs en aplicaciones de cómputo científico no siempre es conveniente. Sin embargo, en este trabajo se ha demostrado que para los algoritmos de dinámica molecular aquí descritos, el tiempo de cómputo se reduce sustancialmente mediante el cálculo de las fuerzas en el GPU.

El hecho de tener la capacidad de reducir el tiempo de cómputo en este tipo de aplicaciones es muy importante para realizar simulaciones moleculares con un número de partículas mayor al que se usa comúnmente. Como resultado de esto último, es posible tener un mejor entendimiento de los procesos que ocurren en estas áreas de estudio. En este trabajo, a pesar de haber usado simple precisión, los resultados se acercan lo suficiente a lo que se reporta en la literatura.

Un resumen de los resultados obtenidos en este trabajo es el siguiente:

1. Se obtiene una curva isoterma bien definida entre las densidades de 0.01 y 0.8 para un número de partículas de $N = 2048$.

2. Dentro de la isoterma se pudieron observar las estructuras moleculares cilíndricas, esféricas y con forma de losa, esto reproduce los resultados reportados por otros autores.

3. Usando el GPU se pudieron reducir los tiempos de cálculo hasta 20 veces con respecto al CPU.

4. Cuanto mayor es el número de pasos de integración es posible alcanzar una mayor aceleración. Y de los resultados, podemos aproximar que el código sobre el GPU es 34 veces más rápido que el código sobre el CPU.

5. La velocidad de transferencia de datos entre el CPU y el GPU no influye mucho en los tiempos totales.

6. Usando simple precisión, los resultados obtenidos entre el CPU y el GPU son muy similares con una diferencia global menor al 1 %.

Con respecto a las preguntas realizadas en la sección 1.2 se tienen las siguientes respuestas:

1. ¿Se forman cúmulos de partículas con algún tipo de estructura geométrica?

   Respuesta : Si se forman cúmulos con estructuras esféricas, cilíndricas y con forma de losa, para ciertos parámetros de la simulación. Esto reproduce varios resultados reportados en la literatura.

2. ¿Se puede reducir el tiempo de cómputo usando GPUs?

   Respuesta : Efectivamente es posible reducir sustancialmente el tiempo de cálculo de una simulación molecular. Particularmente, esto se logró cuando se realiza el cálculo de las fuerzas en el GPU.

3. ¿Existe alguna relación entre la presión, la densidad y la formación de estructuras en los cúmulos de esferas.?

   Respuesta : En la figura 7.9 del capítulo anterior, se muestra una curva obtenida para una simulación de 2000 partículas a una temperatura de 0.85. En esta curva se muestra la ubicación de los puntos de la espinodal y la curva de coexistencia, en donde se pueden identificar las densidades a las cuales se encuentran las estructuras moleculares en forma de esfera, cilíndro y de losa.

Como trabajo futuro quedan pendientes los siguientes temas más específicos:

1. El software Moldynamics debe ser modificado para usar doble precisión en sus cálculos. Además, debe incorporar nuevos algoritmos de integración como son el uso del termostato de *Nose Hover* (véase [R07]), ó la integración con métodos diferentes a *Velocity Verlet*. También es importante la incrustación de diferentes potenciales.

2. Respecto a la paralelización será necesario incorporar la descomposición de fuerzas con MPI en sistemas de memoria distribuida, con lo cual podemos medir con más detalle el rendimiento con respecto al CPU.

3. Incorporar algoritmos que solo tomen en cuenta la vecindad de de las partículas usando radios de corte. Esto reducirá los cálculos necesarios para obtener las fuerzas y permitirá realizar comparaciones con códigos convencionales de dinámica molecular.

4. En este trabajo se obtuvieron las estructuras moleculares con 2000 partículas, sin embargo, sería interesante obtener las estructuras que se forman para tamaños mucho más grandes (10240) simulados en el GPU.

5. Obtener las densidades de los volúmenes dentro de la caja, lo cual es un problema muy difícil, ya que estos volúmenes son bastante irregulares y se mueven dentro de la caja.

6. Realizar la búsqueda de las estructuras para densidades después de la losa, las cuales según [MSE06] son cavidades esféricas y cilíndricas.

Apéndices

Apéndice A

Unidades reducidas

Para sistemas consistentes de un tipo de partícula o moléculas, como es el caso de este trabajo, es buena idea usar la masa de la partícula como unidad fundamental, entonces se supone $m_i = 1$. Como consecuencia el momento de la partícula $\mathbf{p}_i$ y la velocidad $\mathbf{v}_i$, se convierten en cantidades numéricamente idénticas, así como también se cumple la relación $\mathbf{f}_i = \mathbf{a}_i$. Este enfoque puede ser extendido más lejos. Si las partículas interactúan por un potencial a pares de una forma simple, es decir, como el potencial Lennnard-Jones (eq. 3.2.3) el cual es caracterizado por pocos parámetros (σ y ϵ), entonces a partir de esto se pueden definir unidades fundamentales de energía, longitud, etc. De estas definiciones, unidades de otras cantidades (presión, tiempo, momento, etc.) se obtienen directamente. A continuación se listan las formas de las unidades reducidas:

$$\text{densidad:} \quad \rho^* = \rho\sigma^3 \tag{A.0.1}$$

$$\text{temperatura:} \quad T^* = k_B\frac{T}{\epsilon} \tag{A.0.2}$$

$$\text{energía:} \quad E^* = \frac{E}{\epsilon} \tag{A.0.3}$$

$$\text{presión:} \quad P^* = P\frac{\sigma^3}{\epsilon} \tag{A.0.4}$$

$$\text{tiempo:} \quad t^* = \sqrt{\frac{\epsilon}{m\sigma^2}}t \tag{A.0.5}$$

$$\text{fuerza:} \quad \mathbf{f}^* = \mathbf{f}\frac{\sigma}{\epsilon} \tag{A.0.6}$$

$$\text{torque:} \quad \tau^* = \frac{\tau}{\epsilon} \tag{A.0.7}$$

$$\text{tensión superficial:} \quad \gamma^* = \gamma\frac{\sigma^2}{\epsilon} \tag{A.0.8}$$

Apéndice B

Suma de vectores en CUDA

La suma de dos vectores se define como la suma de cada uno de sus elementos y éste es guardado en un tercer vector. La figura B.1 muestra este proceso.

El programa en C tradicional que realiza la suma de dos vectores se muestra en el listado B.1. Aunque este código casi no requiere explicación es importante destacar algunos datos:

- La línea 2 es la función que suma el vector `a[]` al vector `b[]` y el resultado lo almacena en el vector `c[]`
- En el ciclo de la línea 4 a la 7, se observa que la variable `tid` va apuntando a cada uno de los elmentos empezando por el 0 y acabando en $N-1$.
- En la línea 6 se realiza el incremento de la variable, lo cual es obligatorio en una arquitectura de un sólo CPU.
- En el programa principal primero se llenan los arreglos (línea 13 a 16), el arreglo `a[]` con la secuencia $\{0, -1, -2, -3, ..., -(N-1)\}$, y el arreglo `b[]` con la secuencia $\{0, 1, 4, 9, 25, ..., (N-1)^2\}$.
- En la línea 17 se realiza la suma de los vectores devolviendo el resultado en el vector `c[]`

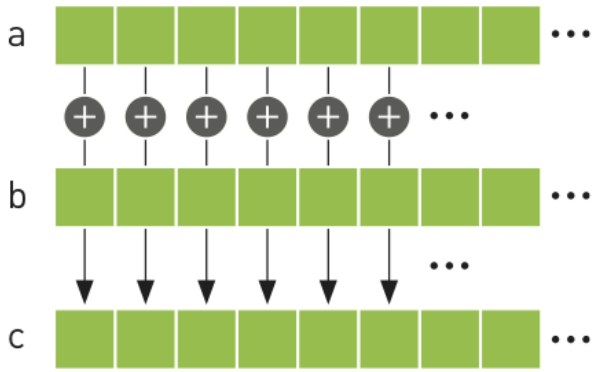


Figura B.1: *Sumando dos vectores*

- En el ciclo de la línea 19 a la 21 se imprimen cada uno de los elementos en pantalla para revisar los resultados de la operación.

Listing B.1: *Código para sumar dos vectores en C*

```
1  #define N 10
2  void add( int *a, int *b, int *c ) {
3    int tid = 0; // En el CPU los arreglos comienzan en cero
4    while (tid < N) {
5      c[tid] = a[tid] + b[tid];
6      // Como se tiene un CPU, se tiene que
7      // realizar el incremento del apuntador
8      id += 1;
9    }
10 }
11
12 int main( void ) {
13   int a[N], b[N], c[N];
14   // Se rellenan los arreglos 'a' y 'b' sobre el CPU
15   for (int i=0; i<N; i++) {
16     a[i] = -i;
17     b[i] = i * i;
18   }
19   add( a, b, c );
20   // imprime los resultados en pantalla
21   for (int i=0; i<N; i++) {
22     printf( "%d + %d = %d\n", a[i], b[i], c[i] );
23   }
24   return 0;
25 }
```

Así mismo este problema se puede resolver en un GPU escribiendo la función `add()` como una función de dispositivo ó una función dentro del GPU. Esto debería ser muy parecido al listado B.1. Pero antes de ver la función `add()` es importante ver el `main()` que se muestra en el listado B.2.

Listing B.2: *Código para sumar dos vectores usando CUDA*

```
1  #define N 10
2  int main( void ) {
3
4    int a[N], b[N], c[N];
5    int *dev_a, *dev_b, *dev_c;
6
7    // Alojamiento de los arreglos en el CPU
8      a = (int*)malloc( N * sizeof(int) );
9      b = (int*)malloc( N * sizeof(int) );
10     c = (int*)malloc( N * sizeof(int) );
11
12   // Alojamineto de los vectore en el GPU
13   HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
14   HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
```

```
15    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );
16
17    // Se llenan los arreglos 'a' y 'b' en el CPU
18    for (int i=0; i<N; i++) {
19      a[i] = -i;
20      b[i] = i * i;
21    }
22
23    // Se copian los areglos 'a' y 'b' al GPU
24    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
          cudaMemcpyHostToDevice ) );
25    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
          cudaMemcpyHostToDevice ) );
26
27    add<<<N,1>>>( dev_a, dev_b, dev_c );
28
29    // Se copia el arreglo 'c' a la memoria del CPU
30    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
          cudaMemcpyDeviceToHost ) );
31
32    // se muestran los resultados en pantalla
33    for (int i=0; i<N; i++) {
34      printf( "%d + %d = %d\n", a[i], b[i], c[i] );
35    }
36
37    // Se libera la memoria alojada en el GPU
38    cudaFree( dev_a ); cudaFree( dev_b ); cudaFree( dev_c );
39    return 0;
40  }
```

Los puntos mas importantes de este listado son:

- Los apuntadores que se crean en la línea 4 serán referencias a los arreglos dentro del GPU.

- En la líneas 8,9 y 10, se realiza el alojamiento de cada uno de los vectores en la memoria del CPU. Esto es indispensable para posteriormente ligar esta memoria con la del GPU, es importante aclarar que la memoria del GPU y la del CPU son diferentes pero están "ligadas" con apuntadores.

- En las líneas 13, 14 y 15, se realiza el alojamiento de cada uno de los vectores en la memoria del GPU, esta tarea se hace con la función `cudaMalloc`. En este caso se reserva un espacio de memoria de N elementos de tamaño `int`. Esta operación se puede ver como casar los apuntadores con una dirección de memoria en el GPU. En resumen, dos arreglos son reservados para las entradas `dev_a` y `dev_b`, además de un arreglo `dev_c` para guardar el resultado. La directiva `HANDLE_ERROR` sirve para reconocer si existe un error en el llamado a las funciones de CUDA, si existe un error se imprime en pantalla de que tipo fue y se termina el programa.

- De las líneas 18 a 21, se llenan los arreglos de la misma forma que en el código en C tradicional. Este paso no es necesario incluso se pueden llenar los vectores en el GPU.

- En las líneas 24 y 25 se copian los segmentos de memoria apuntados por `dev_a` y `dev_b` al GPU, este paso se realiza después de llenar los datos de los arreglos en la memoria del CPU, de no ser así las modificaciones en los arreglos no tendrían efecto en la memoria del GPU. Este paso es donde se ligan los apuntadores `a` y `b` con `dev_a` y `dev_b`. El parámetro `cudaMemcpyHostToDevice` le dice a la función `cudaMemcpy` que la copia de los segmentos es del CPU (host) al GPU (device), sin embargo con esta misma función se pueden realizar copias del GPU al CPU, e incluso del GPU al GPU.

- En la línea 27 se ejecuta el código de dispositivo desde el CPU, esto lo indica la notación de"triple angle bracket" <<<>>>.

- El resultado de la línea 27 queda almacenado en la memoria del GPU, así que será necesario transferir este segmento al CPU, esta tarea se realiza en la línea 30. Debe remarcarse que estas transferencias son muy lentas.

- En la línea 36 se libera toda la memoria del GPU, es necesario ejecutar la instrucción `cudaFree`, para cada uno de los apuntadores.

La manera de ejecutar un código en el GPU desde el CPU, es usando la siguiente notación:

```
kernel <<< num_blocks , num_threads >>> ( param1, param2, ... )
```

El primer parámetro dentro de los "angle brackets" representa el número de bloques paralelos en los cuales deseamos que se ejecute el kernel dentro del GPU.

Por ejemplo si se arranca un kernel como `kernel <<< 2,1 >>> ()`, se estarían creando 2 copias de el kernel y estas estarían corriendo en paralelo, cada hilo se llamará una invocación de un bloque. Con un kernel llamado como `kernel <<< 256 , 16 >>> ()` estarían corriendo 256 bloques cada uno con 16 hilos, en total tendríamos 4096 hilos ejecutándose en "paralelo".

Ahora podemos dar un vistazo a la rutina `add` que se muestra en el listado

Listing B.3: *Código de dispositivo para sumar dos vectores*

```
1 __global__ void add( int *a, int *b, int *c ) {
2   int tid = blockIdx.x; //Maneja los datos de este indice
3   if (tid < N)
4     c[tid] = a[tid] + b[tid];
5 }
```

Cada función que se ejecuta en el GPU debe ser antecedida por el calificador `__global__` al nombre de la función, este mecanismo alerta al compilador que una función debe ser compilada para correr en el GPU. El resto de la función se escribe en C tradicional.

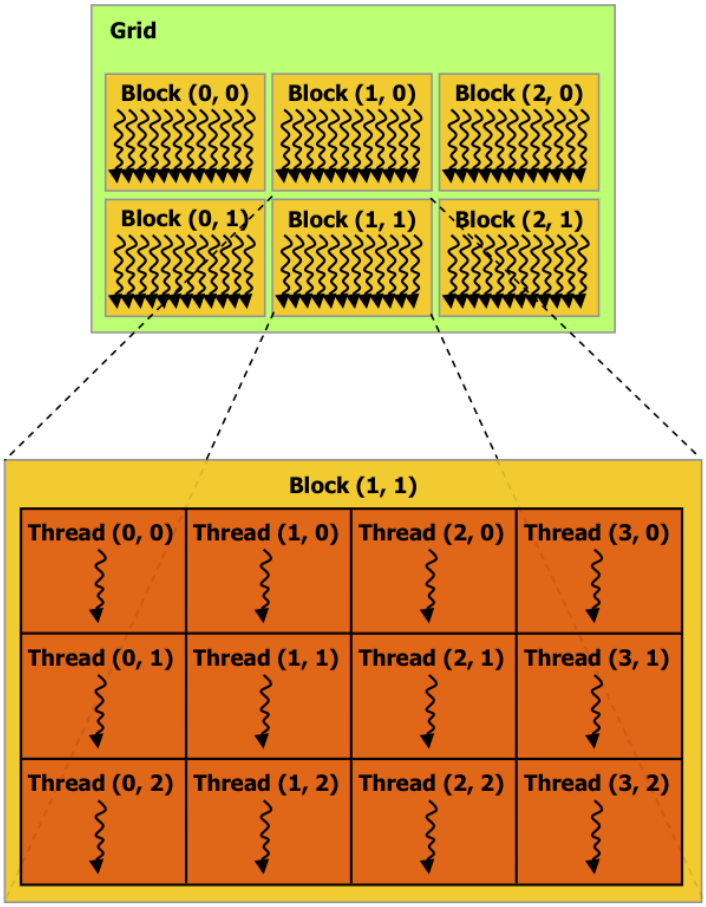


Figura B.2: *Malla de bloques de hilos. Fuente [KH10]*

Debido a que este kernel se ejecuta en paralelo es necesario identificar cada hilo, esto se hace con variables que estan incluidas en la API de CUDA, un ejemplo es la variable `blockIdx.x`.

Por conveniencia, `threadIdx` es un vector de 3 componentes, así que los hilos pueden ser identificados usando un *índice de hilo* de una dimension, dos dimensiones y tres dimensiones, adicionalmente se puede tener un *índice de bloque* de una dimensión, dos dimensiones y tres dimensiones. Esto da un sentido natural para invocar cómputo a través de elementos en un dominio tales como un vector, una matriz, ó un volúmen,

El índice de un hilo y su *thread ID* se puende relacionar de una manera muy sencilla: para una bloque unidimensional el índice de un hilo es igual al *thread ID*, para un bloque bidimiensional de tamaño (D_x, D_y)el *thread ID* de un hilo de indices (x, y) es $(x+yD_x)$; para un bloque tridimensional de tamaño (D_x, D_y, D_z), el *thread ID* de un hilo de índices (x, y, z) es $(x + yD_x + zD_xD_y$.

Apéndice C

Código fuente para el Benchmark de la operación SAXPY

Listing C.1: *Operación SAXPY usando C nativo.*

```
1  #include <stdio.h>
2  #include <iostream>
3  #include "common.h"
4
5  using namespace std;
6
7  void Csaxpy(int N, float a, float *X, float *Y){
8    for(int i=0;i<N;i++) Y[i] = a * X[i] + Y[i];
9  }
10
11
12 int main (int argc, char *argv[]) {
13
14   long int maxIter = 1000;
15   int n = 100;
16   float alpha = 1.0f;
17   common f;
18
19   f.ReadSize(maxIter,n);
20   cout << "Tamano N: " << n << endl;
21   cout << "Numero de Iteraciones: " << maxIter << endl;
22
23   f.BenchAllocateStart();
24   float *x = new float[n];
25   float *y = new float[n];
26   f.BenchAllocateStop();
27
28   f.ReadData(x,y);
29
30   Csaxpy(n, alpha, x, y);
31
32   // Perform benchmark
```

```
33    cout << "Make Benchmark" << endl;
34    f.BenchRunStart();
35    for(long int i = 0; i < maxIter; i++) {
36      Csaxpy(n, alpha, x, y);
37    }
38    f.BenchRunStop();
39
40    f.SaveResult(y);
41
42    f.BenchDeallocateStart();
43    if(x != NULL) delete[] x;
44    if(y != NULL) delete[] y;
45    f.BenchDeallocateStop();
46
47    f.WriteData("output.C_saxpy");
48
49    return 0;
50  }
```

Listing C.2: *Operación SAXPY usando la biblioteca MKL*

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <iostream>
4  #include <ctime>
5  #include "common.h"
6  #include <mkl_cblas.h>
7  #include <mkl.h>
8
9  using namespace std;
10
11 int main (int argc, char *argv[]) {
12
13
14   long int maxIter = 1000;
15   double speed, gflops;
16   clock_t begin;
17   clock_t end;
18   int n = 1e6;
19   float alpha = 1.0f;
20   common f;
21
22   f.ReadSize(maxIter,n);
23   cout << "Tamano N: " << n << endl;
24   cout << "Numero de Iteraciones: " << maxIter << endl;
25
26   f.BenchAllocateStart();
27   float *x = new float[n];
28   float *y = new float[n];
29   f.BenchAllocateStop();
30
31   f.ReadData(x,y);
```

```
32
33    cblas_saxpy(n, alpha, x, 1, y, 1);
34
35    // Perform benchmark
36    /*mkl_set_num_threads(8);
37    mkl_domain_set_num_threads(8,MKL_BLAS);
38    mkl_set_dynamic(0);*/
39    cout << "maxthreads= " << mkl_get_max_threads() << endl;
40    cout << "Make Benchmark" << endl;
41    f.BenchRunStart();
42    for(long int i = 0; i < maxIter; i++) {
43      cblas_saxpy(n, alpha, x, 1, y, 1);
44    }
45    f.BenchRunStop();
46
47    f.SaveResult(y);
48
49    f.BenchDeallocateStart();
50    if(x != NULL) delete[] x;
51    if(y != NULL) delete[] y;
52    f.BenchDeallocateStop();
53
54    f.WriteData("output.MKL_saxpy");
55
56    return 0;
57  }
```

Listing C.3: *Utilerias*

```
1  #include "common.h"
2
3  common::common() {
4  }
5
6  common::~common() {
7
8  }
9
10 void common::ReadSize(long int & max_iter,int & N){
11   ifstream datain("./data.input");
12   if (!datain.is_open()) {
13     cout << "Error input data file" << endl;
14     exit(10);
15   }
16   datain >> max_iter;
17   datain >> N;
18   this->N = N;
19   datain.close();
20 }
21
22
23 void common::ReadData(float *X, float *Y){
```

```
24    ifstream datain("./data.input");
25    if (!datain.is_open()) {
26      cout << "Error input data file" << endl;
27      exit(10);
28    }
29    datain >> prueba;
30    datain >> prueba;
31    cout << "Reading " << N << " elements...." << endl;
32    for(int i=0;i<N;i++){
33      datain >> X[i];
34      datain >> Y[i];
35    }
36    datain.close();
37  }
38
39  void common::SaveResult(float *Y){
40    Result = new float[N];
41    for(int i=0;i<N;i++)Result[i]=Y[i];
42  }
43
44  void common::WriteData(char *filename){
45      MakeBench();
46      ofstream dataout(filename);
47      if (!dataout.is_open()) {
48        cout << "Error output data file" << endl;
49        exit(11);
50      }
51      dataout << speed_allocate  << "\t\t#SPEED ALLOCATE SECONDS" << endl;
52      dataout << speed_run  << "\t\t#SPEED KERNEL SECONDS" << endl;
53      dataout << speed_deallocate << "\t\t#SPEED DEALLOCATE SECONDS" << endl
            ;
54      dataout.setf(ios::scientific);
55      dataout.precision(13);
56      for(int i=0;i<N;i++){
57        dataout << Result[i] << endl;
58      }
59      dataout.close();
60  }
61
62  void common::BenchAllocateStart(void){
63    begin_allocate=clock();
64  }
65
66  void common::BenchAllocateStop(void){
67    end_allocate=clock();
68  }
69
70  void common::BenchRunStart(void){
71    begin_run=clock();
72  }
73
74  void common::BenchRunStop(void){
```

```
75    end_run=clock();
76  }
77
78  void common::BenchDeallocateStart(void){
79    begin_deallocate =clock();
80  }
81
82  void common::BenchDeallocateStop(void){
83    end_deallocate =clock();
84  }
85
86  void common::MakeBench(void){
87    speed_allocate = timediff(end_allocate,begin_allocate);
88    speed_run =  timediff(end_run,begin_run);
89    speed_deallocate =  timediff(end_deallocate,begin_deallocate);
90    PrintBench();
91  }
92
93  void common::PrintBench(void){
94    printf("Allocate     %.6f Time (s) \n",speed_allocate);
95    printf("Kernel       %.6f Time (s) \n",speed_run);
96    printf("Deallocate   %.6f Time (s) \n",speed_deallocate);
97  }
98
99  double common::timediff(clock_t e, clock_t b){
100   double ticks = e - b;
101   double diff = ticks / CLOCKS_PER_SEC;
102   return diff;
103 }
```

Apéndice D

Código fuente para el cálculo de Fuerzas dentro del GPU

Listing D.1: *Descomposición de fuerzas en el GPU*

```
1  __global__
2  static void MDGPU_ljForcesN2( float4* pPos, float4* pFor )
3  {
4    // Se obtine el ID del hilo y este es ubicado en un arreglo lineal
5    const unsigned int tid = LINEAR_ADDRESS_T1D;
6
7    // Se obtiene el tamano de los bloques
8    const int nBlockSize = blockDim.x * blockDim.y * blockDim.z;
9
10   // ciclo para todas las particulas
11   int i;                // apunta a las otras particula
12   int k;              // indice del bloque para recorrer la malla
13   float  fC;            // constante de la fuerza (fij)
14   float4 f4R;         // distancia entre las particulas
15   float4 f4For;         // Fuerza total de las particulas
16   float4 f4Pos;         // Posicion de la particula tid
17   float4 f4Pos2;        // Posicion de las otras particulas
18
19   // Carga de la posicion de la memoria cache
20   f4Pos = tex1Dfetch( g_tPosTex, (int) tid );
21
22   // Copia a la memoria compartida
23   s_fvPosX[threadIdx.x] = f4Pos.x;
24   s_fvPosY[threadIdx.x] = f4Pos.y;
25   s_fvPosZ[threadIdx.x] = f4Pos.z;
26
27   // sincroniza todos los hilos para asegurar que todos han cargado su
         posicion
28   __syncthreads();
29
30   // Unicializa el contador de la fuerza a cero
31   f4For = make_float4( 0.0, 0.0, 0.0, 0.0 );
```

```
32
33    // Ciclo obre todas las particulas dentro del bloque inicial
34    for (i=0; i<nBlockSize; ++i)
35    {
36      // Seobtiene la posicion de la memoria compartida
37      f4Pos2.x = s_fvPosX[i];
38      f4Pos2.y = s_fvPosY[i];
39      f4Pos2.z = s_fvPosZ[i];
40
41      // Se calcula la distancia
42      f4R.x = f4Pos.x - f4Pos2.x;
43      f4R.y = f4Pos.y - f4Pos2.y;
44      f4R.z = f4Pos.z - f4Pos2.z;
45
46      // Se aplica condiciones a la frontera periodicas
47      MDGPU_pbcDistance( f4R );
48
49      // Se calcula la fuerza de LJ que es una constante
50      fC = MDGPU_ljForce( f4R );
51
52      // Se suma la contribucion de la fuerza a la particula tid
53      f4For.x += f4R.x * fC;
54      f4For.y += f4R.y * fC;
55      f4For.z += f4R.z * fC;
56    }
57
58    // ciclo sobre todos los bloques que faltan
59    const int nBlocks = gridDim.x * gridDim.y;
60    for (k=1; k<nBlocks; ++k)
61    {
62      // se obtienen los indice del k-esimo bloque
63      i = tid + k * nBlockSize;
64
65      // i = i mod N
66      if (i >= c_nParticles) i -= c_nParticles;
67
68      // Asegurarse que todos han cargado correctamente sus
69      // datos en la memoria compartida
70      __syncthreads();
71
72      // Se obtienen las posiciones de las particulas de otro bloque
73      f4Pos2 = tex1Dfetch( g_tPosTex, (int) i );
74
75      // Se copian a memoria compartida
76      s_fvPosX[threadIdx.x] = f4Pos2.x;
77      s_fvPosY[threadIdx.x] = f4Pos2.y;
78      s_fvPosZ[threadIdx.x] = f4Pos2.z;
79
80      // Asegurar que todos lo hilos han cargado en memoria compartida
81      __syncthreads();
82
83      // ciclo sobre todas las particulas del bloque
```

```
84      for (i=0; i<nBlockSize; ++i)
85      {
86        // obtine la posicion del bloque en memoria compartida
87        f4Pos2.x = s_fvPosX[i];
88        f4Pos2.y = s_fvPosY[i];
89        f4Pos2.z = s_fvPosZ[i];
90
91        // calculo de la distancia
92        f4R.x = f4Pos.x - f4Pos2.x;
93        f4R.y = f4Pos.y - f4Pos2.y;
94        f4R.z = f4Pos.z - f4Pos2.z;
95
96        // se aplican condiciones a la frontera
97        MDGPU_pbcDistance( f4R );
98
99        // Calculo de la fuerza constante (fij)
100       fC = MDGPU_ljForce( f4R );
101
102       // Se suma la contribucion de la fuerza de esta particula
103       f4For.x += f4R.x * fC;
104       f4For.y += f4R.y * fC;
105       f4For.z += f4R.z * fC;
106     }
107   }
108   // Se escribe el total de la fuerza calculada
109   pFor[tid] = f4For;
110 }
```

Listing D.2: *Condiciones de frontera periodicas en el GPU*

```
1  __device__
2  void MDGPU_pbcDistance( float4& f4Rij )
3  {
4    // check distance for periodic boundary conditions
5    f4Rij.x = (f4Rij.x >  c_f3HalfBoxSize.x) ? (f4Rij.x - c_f3BoxSize.x) :
         f4Rij.x;
6    f4Rij.x = (f4Rij.x < -c_f3HalfBoxSize.x) ? (f4Rij.x + c_f3BoxSize.x) :
         f4Rij.x;
7
8    f4Rij.y = (f4Rij.y >  c_f3HalfBoxSize.y) ? (f4Rij.y - c_f3BoxSize.y) :
         f4Rij.y;
9    f4Rij.y = (f4Rij.y < -c_f3HalfBoxSize.y) ? (f4Rij.y + c_f3BoxSize.y) :
         f4Rij.y;
10
11   f4Rij.z = (f4Rij.z >  c_f3HalfBoxSize.z) ? (f4Rij.z - c_f3BoxSize.z) :
         f4Rij.z;
12   f4Rij.z = (f4Rij.z < -c_f3HalfBoxSize.z) ? (f4Rij.z + c_f3BoxSize.z) :
         f4Rij.z;
13 }
```

Listing D.3: *Calculo de la constante de fuerza en el GPU*

```
1  __device__
```

```
2  float MDGPU_ljForce( float4 f4Rij )
3  {
4    float fC;
5    float fInvR6;
6    float fInvR8;
7
8    // Calculo de la distancia al cuadrado
9    const float fR2 = f4Rij.x * f4Rij.x + f4Rij.y * f4Rij.y + f4Rij.z *
         f4Rij.z;
10
11
12   // La fuerza es cero si la particula esta fuera del radio de corte
13   // La fuerza es cero si es la auto-fuerza
14   if ((fR2 > c_fLJCutoffSqr) || (fR2 < 0.001))
15   {
16     fC = 0.0;
17   }
18   else //la interaccion es valida
19   {
20     // Se calculan los inversos de la potencia R:
21     // R^-2, R^-4, R^-6 and R^-8
22     const float fInvR2 = 1.0 / fR2;
23     const float fInvR4 = fInvR2 * fInvR2;
24     fInvR6 = fInvR2 * fInvR4;
25     fInvR8 = fInvR4 * fInvR4;
26
27     // Se calcula la constante de fuerza de LG
28     fC = (48.0 * fInvR6 - 24.0 ) * fInvR8 - c_fLJForceShift;
29   }
30   return fC;
31 }
```

Glosario

ángulo fijo En química, es el ángulo que forman dos enlaces de alguna molécula o arreglo de moléculas..

lattice Es un embaldosado en una celda unitaria, es decir, se ordenan todos los elmentos igualitariamente dentro de una celda unitaria. En física es lo opuesto a lo contínuo en el espacio. Los modelos de lattice nacieron en el ámbito de la física de materia condensada, donde los atomos se forman en un cristal automáticamente..

Virial Una cierta función que relaciona un sistema de fuerzas y sus puntos de aplicación, el término fue primeramente usado por Clasius (véase [Cla70]) en la investigación sobre problemas de física molecular.

longitud de enlace En química, es la longitud de un enlace entre dos átomos de una molécula o arreglo de moléculas..

nucleación Es la formación loaclizada de una fase termodinámica distinta. Algunos ejemplos de fases que se pueden forma vía la nucleación en líquidos son: burbujas gaseosas, cristales, regiones vidriosas, etc. La creación de gotas líquidas en gases saturados es también característica de la nucleación. La mayoría de los procesos de nucleación son obtenidos por cambios físicos más que químicos, pero en pocas excepciones como la nucleación electroquímica,, las reacciones químicas son importantes..

Potencial químico En termodinámica, dentro de la física y en termoquímica dentro de la química , potencial químico, cuyo símbolo es μ, es un término introducido en 1876 por el físico estadounidense Willard Gibbs, que él definió como sigue:
"Podemos decir que el potencial químico es la fuerza impulsora que induce el cambio en el sistema. Si suponemos que se añade una cantidad infinitesimal de cualquier sustancia a una masa homogénea cualquiera en un estado de tensión hidrostática, que la masa permanece homogénea y su entropía y volumen permanecen constantes, el incremento de la energía interna de la masa dividida por la cantidad de la sustancia añadida es el potencial para esa sustancia en la masa considerada.".

termostato Es una construcción matemática que permite fijar la temperatura de un sistema usando algún parámetro.

pozo Denotamos como pozo a una parte donde una gráfica tiene un mínimo global. Puede haber pozos de distintas formas los más frecuentes son cuadrados y curvos..

Bibliografía

[AAS+07] Sadaf R. Alam, Pratul K. Agarwal, Melissa C. Smith, Jeffrey S. Vetter, and David Caliga. Using FPGA Devices to Accelerate Biomolecular Simulations. *Computer*, 40(3):66–73, March 2007.

[AT89] M P Allen and D J Tildesley. *Computer simulation of liquids*. Clarendon Press, New York, NY, USA, 1989.

[AW57] B J Alder and T E Wainwright. Phase Transition for a Hard Sphere System. *The Journal of Chemical Physics*, 27(5):1208–1209, 1957.

[AW62] B J Alder and T E Wainwright. Phase Transition in Elastic Disks. *Phys. Rev.*, 127(2):359–361, July 1962.

[BCX+06] Kevin Bowers, Edmond Chow, Huafeng Xu, Ron Dror, Michael Eastwood, Brent Gregersen, John Klepeis, Istvan Kolossvary, Mark Moraes, Federico Sacerdoti, John Salmon, Yibing Shan, and David Shaw. Scalable Algorithms for Molecular Dynamics Simulations on Commodity Clusters. *ACM/IEEE SC 2006 Conference (SC'06)*, (November):43–43, November 2006.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[Cla70] R Clausius. Ueber einen auf die Wärme anwendbaren mechanischen Satz. *Annalen der Physik*, 217(9):124–130, 1870.

[DA74] Germund Dahlquist and Björk Ake. *Numerical Methods*. Prentice–Hall, New Jersey,, 1974.

[Dir] DirectX. http://www.microsoft.com/download/en/details.aspx?id=35.

[EP05] Erich Elsen and Vijay Pande. N-Body Simulations on GPUs. *interactions*, 2005.

[FS01] Daan Frenkel and B Smit. *Understanding Molecular Simulation, Second Edition: From Algorithms to Applications (Computational Science)*. Academic Press, 2 edition, November 2001.

[Gar08] Leopoldo García-Colíin Scherer. *Introducción a la física estadística*. 2008.

[Gay81] J. G. Gay. Modification of the overlap potential to mimic a linear site–site potential. *The Journal of Chemical Physics*, 74(6):3316, 1981.

[GZP+06] Mark Giampapa, Yuri Zhestkov, Michael C Pitman, Frank Suits, Alan Grossfield, Jed Pitera, William Swope, Ruhong Zhou, and Scott Feller. Blue Matter : Strong Scaling of Molecular Dynamics on Blue Gene / L. *Proc. International Conf. on Computational Science (ICCS 2006)*, 3992:846–854, 2006.

[HÖ5] Philippe H Hünenberger. Thermostat Algorithms for Molecular Dynamics Simulations. *Springer*, pages 105–149, 2005.

[Hoc70] R W Hockney. Potential calculation and some applications. *Methods Comput. Phys.*, 1970.

[Hu04] Y Hu. Constant temperature molecular dynamics simulations of energetic particle?solid collisions: comparison of temperature control methods. *Journal of Computational Physics*, 200(1):251–266, October 2004.

[INT] INTEL. http://software.intel.com/en-us/articles/intel-mkl/.

[Jon24] J. E. Jones. On the Determination of Molecular Fields. II. From the Equation of State of a Gas. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 106(738):463–477, October 1924.

[KH10] David B Kirk and Wen-mei W Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[Lin] Erik Lindahl. http://www.gromacs.org/.

[MO99] R Murty and D Okunbor. Eficient parallel algorithms for molecular dynamics simulations. *Parallel Computing*, 25(3):217–230, March 1999.

[Mor00] R G Mortimer. *Physical chemistry*. Harcourt/Academic Press, 2000.

[MSE06] Luis G MacDowell, Vincent K Shen, and Jeffrey R Errington. Nucleation and cavitation of spherical, cylindrical, and slablike droplets and bubbles in small systems. *The Journal of chemical physics*, 125(3):34705, July 2006.

[MW65] Joseph E Mayer and Wm. W Wood. Interfacial Tension Effects in Finite, Periodic, Two Dimensional Systems. *The Journal of chemical physics*, 42(12):4268–4274, 1965.

[Nak97] Aiichiro Nakano. Fuzzy clustering approach to hierarchical molecular dynamics simulation of multiscale materials phenomena. *Computer Physics Communications*, 105:11, 1997.

[NH03] Thomas Neuhaus and Johannes S Hager. Exponential Slowing Down in Simulations of First-Order Phase Transitions. *October*, 113(October):47–83, 2003.

[NVIa] NVIDIA. http://developer.download.nvidia.com/compute/cuda/1_0/CUBLAS_Library_1.0

[NVIb] NVIDIA. http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA

[NVIc] NVIDIA. http://www.nvidia.com/object/molecular_dynamics.html.

[Ope] OpenGL. http://www.opengl.org/.

[PBC+03] Vijay S Pande, Ian Baker, Jarrod Chapman, Sidney P Elmer, Siraj Khaliq, Stefan M Larson, Young Min Rhee, Michael R Shirts, Christopher D Snow, Eric J Sorin, and Bojan Zagrovic. Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing. *Biopolymers*, 68(1):91–109, January 2003.

[PH01] Michel Pleimling and Alfred Hüller. Crossing the Coexistence Line at Constant Magnetization. 104(September):971–989, 2001.

[Pli95] S Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, March 1995.

[Pot73] D. Potter. *Computational Physics*. Wiley, New York, NY, USA, 1973.

[Pyt] Python. http://python.org/.

[R07] Victor R. Berendsen and Nose-Hoover thermostats Temperature in MD MD at constant Temperature - NVT ensemble. pages 1–4, 2007.

[Rap04] D. C. Rapaport. *The Art of Molecular Dynamics Simulation*. Cambridge University Press, 2nd editio edition, 2004.

[RBK78] M Rao, B J Berne, and M H Kalos. Introducción a la física estadística. 68(4):1325–1336, 1978.

[SCC+09] Gerardo A Laguna Sánchez, Mauricio Olguín Carbajal, Nareli Cruz Cortés, Ricardo Barrón, and Jesús A Álvarez Cedillo. Comparative Study of Parallel Variants for a Particle Swarm Optimization Algorithm Implemented on a Multithreading GPU. 7(3):292–309, 2009.

[SCE+07] David E. Shaw, Jack C. Chao, Michael P. Eastwood, Joseph Gagliardo, J. P. Grossman, C. Richard Ho, Douglas J. Ierardi, István Kolossváry, John L. Klepeis, Timothy Layman, Christine McLeavey, Martin M. Deneroff, Mark a. Moraes, Rolf Mueller, Edward C. Priest, Yibing Shan, Jochen Spengler, Michael Theobald, Brian Towles, Stanley C. Wang, Ron O. Dror, Jeffrey S. Kuskin, Richard H. Larson, John K. Salmon, Cliff Young, Brannon Batson, and Kevin J. Bowers. Anton, a special-purpose machine for molecular dynamics simulation. *ACM SIGARCH Computer Architecture News*, 35(2):1, June 2007.

[Sch] Klaus Schulten. http://www.ks.uiuc.edu/Research/vmd/.

[Sim] Carles Simó. New Families of Solutions in N -Body Problems. *New Solutions*, (2).

[SJC+00] Home Search, Collections Journals, About Contact, My Iopscience, and I P Address. Droplets in the coexistence region of the two-dimensional Ising model Droplets in the coexistence region of the two-dimensional Ising. 199, 2000.

[SK10] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1 edition, July 2010.

[Swo82] William C. Swope. A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *The Journal of Chemical Physics*, 76(1):637, 1982.

[TAKB06] Michela Taufer, Chahm An, Andreas Kerstens, and Charles L Brooks III. Predictor@Home: A "Protein Structure Prediction Supercomputer' Based on Global Computing. *IEEE Trans. Parallel Distrib. Syst.*, 17(8):786–796, August 2006.

[Ver67] Loup Verlet. Computer .Experiments.on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Phys. Rev.*, 159(1):98, July 1967.

[vMAF+08] J. a. van Meel, a. Arnold, D. Frenkel, S.F. Portegies Zwart, and R. G. Belleman. Harvesting graphics power for MD simulations. *Molecular Simulation*, 34(3):259–266, March 2008.

[WJ57] W.˜W. Wood and J.˜D. Jacobson. Preliminary Results from a Recalculation of the Monte Carlo Equation of State of Hard Spheres. *The Journal of chemical physics*, 27:1207–1208, November 1957.

[YM06] Kenji Yasuoka and Mitsuhiro Matsumoto. Molecular dynamics of homogeneous nucleation in the vapor phase. I. Lennard-Jones fluid. *journal of chemical physics*, 109(19):8451–8462, 2006.