Máster en Ingeniería Computacional y Matemática

Trabajo Final de Máster

Serial and parallel (CUDA) general purpose Monte Carlo code for atomistic simulations.

Nombre Estudiante: Antonio Díaz Pozuelo.

Nombre Director/a: Enrique Lomba García.

Fecha de Entrega: Julio de 2018.

Firma del director autorizando la entrega final del TFM:

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Serial and parallel (CUDA) general purpose Monte Carlo code for atomistic simulations.*
Nombre del autor:	*ANTONIO DÍAZ POZUELO*
Nombre del director/a:	*ENRIQUE LOMBA GARCÍA*
Nombre del PRA:	*JUAN ALBERTO RODRÍGUEZ VELÁZQUEZ*
Fecha de entrega:	07/2018
Titulación:	*MÁSTER EN INGENIERÍA COMPUTACIONAL Y MATEMÁTICA*
Área del Trabajo Final:	*HPC*
Idioma del trabajo:	*ESPAÑOL*
Palabras clave	*HPC MONTECARLO GPU*

Resumen del Trabajo (máximo 250 palabras)

El advenimiento de las *GPU*s (*Graphics Processing Units*) ha supuesto una revolución en la forma de abordar problemas físicos mediante métodos computacionales. En este Trabajo de Fin de Máster hemos pretendido explotar las enormes capacidades computacionales disponibles en las *GPUs* modernas con arquitectura *CUDA* (*Compute Unified Device Architecture*) en el campo de la Simulación Atomística, en concreto en lo que concierne a los métodos de Monte Carlo. Por lo tanto, se implementará un programa de *software* nuevo y se estudiará la aceleración, obtenida de paralelizar las partes del problema con mayor complejidad computacional, y cómo escala ésta respecto al tamaño del problema.

Los resultados obtenidos de las simulaciones *NVT* con cálculo de potencial químico y *NPT* demuestran, por un lado, que todas las versiones paralelas de los algoritmos son mucho más eficientes que las versiones serie. Por otro lado, según se aumenta el tamaño del problema, el tiempo de ejecución de las versiones paralelas de los algoritmos tiende a crecer cuadráticamente respecto al tamaño del problema al igual que en las versiones serie; con la excepción del movimiento traslacional en el colectivo *NVT*. Éste, para los tamaños de problema estudiados aquí, todavía presenta una dependencia casi lineal.

En conclusión, hemos demostrado que una simple paralelización del problema al nivel del cálculo de energías, sin utilizar la descomposición espacial por dominios, presenta un notable incremento de rendimiento utilizando *GPUs* en problemas de tamaño grande.

Abstract (in English, 250 words or less)

The advent of *GPUs* (*Graphics Processing Units*) has been a revolution in the way of addressing physical problems through computational methods. In this Master's Thesis, our aim is to exploit the enormous computational capabilities available in modern *GPUs* with *CUDA* architecture (*Compute Unified Device Architecture*) in the field of Atomistic Simulation, specifically with regard to Monte Carlo methods. Therefore, a new software program will be developed. We will study the speed-up obtained from parallelizing the parts of the problem with greater computational complexity and how that acceleration scales with respect to the size of the problem.

The results obtained from the *NVT* with chemical potential calculation and *NPT* simulations demonstrate, on the one hand, that all the parallel versions of the algorithms are much more efficient than the serial versions. On the other hand, as the size of the problem increases, the execution time of the parallel versions of the algorithms tends to grow quadratically with size as in the serial versions with the exception of *NVT* translational moves. These, for our sizes still present a quasi-linear dependence.

In conclusion, we have shown that a simple parallelization of the problem at the level of energy calculation without resorting to domain decomposition allows for a remarkable performance increase when using *GPUs* for large problem sizes.

Índice

1. Introducción 1
1.1 Contexto y justificación del Trabajo 1
1.3 Enfoque y método seguido 2
1.4 Planificación del Trabajo 2
1.5 Breve sumario de productos obtenidos 3
1.6 Breve descripción de los otros capítulos de la memoria 4
2. Fundamentos físicos 5
2.1 El algoritmo de Metrópolis en el colectivo canónico 7
2.2 ¿Dinámica Molecular o Monte Carlo? 9
2.3 Otras propiedades: el potencial químico 9
3. Arquitectura computacional 11
3.1 CPU vs GPU 11
3.2 CUDA 12
3.2.1 Kernels 13
3.2.2 Memoria 15
3.2.3 Programación heterogénea 15
3.2.4 Conclusión 16
4. Validación del modelo 18
5. Algoritmos 21
5.1 Rutinas de uso común 21
5.2 Traslación de partículas, algoritmo Metrópolis (*NVT* y *NPT*). 26
5.3 Cambio de volumen (*NPT*). 31
5.4 Potencial químico (*NVT*). 34
6. Implementación 38
7. Resultados 43
7.1 Configuración de la simulación 43
7.2 Tiempo total de CPU/GPU por paso de simulación 44
7.3 Speed-up 48
8. Conclusiones 50
8.1 Perspectivas de desarrollo futuro 51
9. Bibliografía 53
10. Anexos 54
A.1 Repositorio del proyecto 54

Lista de figuras

Figura 1: diagrama de Gantt.

Figura 2: ilustración de las condiciones de contorno periódicas.

Figura 3: cálculo de π mediante integración Monte Carlo.

Figura 4: potenciales de Morse y Lennard Jones con los parámetros utilizados en este estudio.

Figura 5: operaciones en coma flotante por segundo en CPU y GPU.

Figura 6: ancho de banda de memoria en CPU y GPU.

Figura 7: número de transistores para procesar datos en CPU y GPU.

Figura 8: modelo SIMD.

Figura 9: escalabilidad automática.

Figura 10: multiprocesador de streaming.

Figura 11: características de los tipos de memoria de la GPU.

Figura 12: grid, bloque e hilos en el modelo CUDA.

Figura 13: jerarquía de memorias de la GPU.

Figura 14: programación heterogénea.

Figura 15: configuración inicial DLP.

Figura 16: configuración inicial FCC.

Figura 17: convergencia energía NVT (CPU).

Figura 18: convergencia energía NVT (GPU).

Figura 19: histograma energía NVT (CPU).

Figura 20: histograma energía NVT (GPU).

Figura 21: convergencia energía NPT (CPU).

Figura 22: convergencia energía NPT (GPU).

Figura 23: histograma densidad NPT (CPU).

Figura 24: histograma densidad NPT (GPU).

Figura 25: reducción binaria (interleaved addressing).

Figura 26: tiempos (segundos) por paso y speed-up para diferentes número de hilos por bloque.

Figura 27: tiempos (segundos) por paso y speed-up para diferentes configuraciones de memoria.

Figura 28: algoritmo moveAtoms: tiempo por paso respecto al tamaño del problema (log-log) y regresión lineal (CPU y GPU).

Figura 29: algoritmo moveVolum: tiempo por paso respecto al tamaño del problema (log-log) y regresión lineal (CPU y GPU).

Figura 30: algoritmo chPotential: tiempo por paso respecto al tamaño del problema (log-log) y regresión lineal (CPU y GPU).

Figura 31: aplicación: tiempo por paso respecto al tamaño del problema (log-log) y regresión lineal (CPU y GPU).

Figura 32: moveAtoms: escalado del speed-up.

Figura 33: moveVolum: escalado del speed-up.

Figura 34: chPotential: escalado del speed-up.

Figura 35: aplicación: escalado del speed-up.

Figura 36: división de la caja de simulación en celdas de lado rc. Las partículas de la caja i, sólo interaccionan con las de las de las 9 cajas sombreadas.

Figura 37: división de la caja de simulación en tablero de damas. Las partículas diferentes cajas sombreadas no interactúan, por lo que intentos de traslación o inserción/borrado pueden ser paralelos.

1. Introducción

1.1 Contexto y justificación del Trabajo

El advenimiento de las *GPU*s (*Graphics Processing Units*) ha supuesto una revolución en la forma de abordar problemas físicos mediante métodos computacionales. En este Trabajo de Fin de Máster se pretende explotar las enormes capacidades computacionales disponibles en las *GPUs* modernas (miles de núcleos, cantidades muy considerables de memoria *GDDR* - *Graphics Double Data Rate* - de acceso mucho más rápido que la convencional, unidades de proceso dedicadas a cálculos numéricos específicos, etc.) en el campo de la Simulación Atomística, en concreto en lo que concierne a los métodos de Monte Carlo.

Desde que se definieron estándares eficientes para la paralelización de algoritmos, como el *MPI* (*Message Passing Interface*) a comienzos de los años noventa[1], existen numerosos paquetes de simulación mediante dinámica molecular[2] que explotan plenamente el paralelismo en base a la descomposición espacial por dominios. Este proceso se basa esencialmente en el carácter local de las fuerzas y las interacciones en la materia, esto es: en muchas situaciones de interés, partículas distantes no interaccionan, por lo que la celda unidad de simulación se puede descomponer en dominios en los que las fuerzas se pueden determinar independientemente. De esa forma, el número de operaciones necesario para simular un sistema de N partículas no escala con N^2 (número total de interacciones posibles si son de alcance infinito), sino que crece linealmente con N.

La situación es radicalmente diferente en lo que atañe a los algoritmos de Monte Carlo. En particular, el algoritmo de Metrópolis en el colectivo canónico es esencialmente serial, por lo que su paralelización dista de ser obvia. Es por ello que a fecha de hoy no se dispone de paquetes ampliamente difundidos con algoritmos de Monte Carlo para simulaciones atomísticas eficientemente paralelizados. Así pues el objetivo de este Trabajo de Fin de Master ha sido realizar una pequeña contribución en esa línea, aprovechando el potencial computacional de las *GPUs* y la ventaja que su arquitectura presenta frente a las *CPUs* (*Central Processing Unit*) en las que la paralelización descansa sobre el uso de capas de *MPI*, *OpenMP* (*Open Multi-Processing*) o *Posix Threads*. En este sentido, la estrategia de la paralelización se ha centrado en optimizar el cálculo de las energías, que en el caso *NVT* escala linealmente con *N* para cada intento de desplazamiento, y en el caso del *NPT* cuadráticamente, tal y como sucede con la determinación de las fuerzas en el método de Dinámica Molecular. Para ello se recurrirá a estrategias adaptadas a la arquitectura de la *GPU* y que pueden ser explotadas mediante la *API* (*Application Programming Interface*) *CUDA* (*Compute Unified Device Architecture*), en particular la parelización mediante el modelo de ejecución *SIMT* (*Single Instruction Multiple Thread*), recurriendo al uso de memoria compartida entre hilos (*threads*) y funciones matemáticas intrínsecas. En el trabajo se comparará el rendimiento de los algoritmos de *GPU* frente a los correspondientes de *CPU*, y finalmente se analizarán las perspectivas para implementar un algoritmo equivalente a la descomposición por dominios en el código de Monte Carlo canónico (*NVT*).

1.2 Objetivos del Trabajo

El objetivo principal del proyecto es realizar un estudio de la aceleración (*speed-up*) obtenida como resultado de paralelizar las partes de un código de Monte Carlo serial con mayor complejidad computacional. Además, se desea realizar un estudio de cómo escala dicha aceleración respecto al tamaño del problema.

1.3 Enfoque y método seguido

La construcción de la solución elegida se ha realizado implementando una aplicación nueva. Además, dicha implementación se ha realizado completamente de forma secuencial y, a su vez, de forma paralela en las partes necesarias para realizar el estudio de aceleración (*speed-up*).

Respecto a la parte paralela, se puede optar por paralelizar utilizando una *CPU* o una *GPGPU* (*General Purpose Graphical Processor Unit*). La paralelización mediante *CPU* se puede considerar una tecnología "clásica", mientras que la paralelización mediante *GPGPU* es una tecnología "actual" y en auge. Por lo tanto, se ha optado por utilizar una *GPGPU* dado que:

- Se desea seguir aprendiendo en el desarrollo de aplicaciones utilizando esta tecnología.
- La paralelización en *GPGPU* es una tecnología en continua expansión en el ámbito científico.
- La eficiencia de las *GPGPUs* es superior respecto de las *CPUs*.
- Las partes paralelizadas del problema se ajustan correctamente al modelo arquitectónico *SIMT* (*Single Instruction Multiple Thread*) que utilizan las *GPGPUs*, el cual es una evolución del modelo *SIMD* (*Single Instruction Multiple Data*).

Además, se decide utilizar *GPUs* de la marca *NVIDIA* con arquitectura *CUDA* dado que:

- Se desea seguir aprendiendo a implementar aplicaciones en *CUDA_C/C++*.
- Se trata de *GPUs* de ámbito general que se pueden encontrar en gran parte del ecosistema informático existente.
- Disponen de un *SDK* (*Software Development Kit*) maduro y basado en el lenguaje de programación *C*.

1.4 Planificación del Trabajo

Para realizar este proyecto se ha utilizado un enfoque metodológico en cascada, ya que cada etapa del proyecto nos dará como resultado un artefacto que se utilizará en la siguiente fase. Por lo tanto, los artefactos de cada etapa nos marcarán un desarrollo lineal e iterativo, con objeto de corregir y ajustar las fases según se desarrolle el proyecto. Podemos dividir el desarrollo del proyecto en varias etapas:

- Análisis y propuesta de soluciones (*PEC1*): se expone el problema a afrontar y la solución adoptada dentro de las posibles existentes. El

artefacto final será una guía que se basará en la selección de los artículos (bibliografía) mediante los cuales se realizará el diseño final del producto.

• Diseño (*PEC1*): tras definir como abordar el problema se diseñará un algoritmo *top-down* que resuelva el problema. Los artefactos intermedios serán los algoritmos de cada módulo dentro del programa y el artefacto final consistirá en la formulación completa del algoritmo.

• Implementación y pruebas (*PEC2*): por un lado, en esta fase se implementará el algoritmo definido en la fase anterior. Mediante los algoritmos intermedios se codificarán los módulos y se corregirá el diseño según se precise. Una vez todos los módulos funcionen correctamente se ensamblarán con objeto de dar forma al artefacto definitivo, la versión final del programa en código fuente. Por otro lado, es la etapa en la que se ejecutan los diversos módulos implementados para comprobar su correcto funcionamiento.

• Simulación (*PEC3*): finalmente, se ejecutará el artefacto definitivo de la fase de implementación en diferentes entornos (*CPU vs GPU*) y con diferentes parámetros (tamaños de los conjuntos de entrada) con objeto de estudiar su rendimiento. El artefacto final corresponderá a una serie de métricas que representarán las diversas simulaciones realizadas.

• Documentación (*PEC3*): en esta fase se utilizarán las métricas anteriores junto con toda la documentación recogida durante la realización del proyecto y se creará la presente memoria.

En la figura 1 se expone el diagrama de Gantt asociado al proyecto.

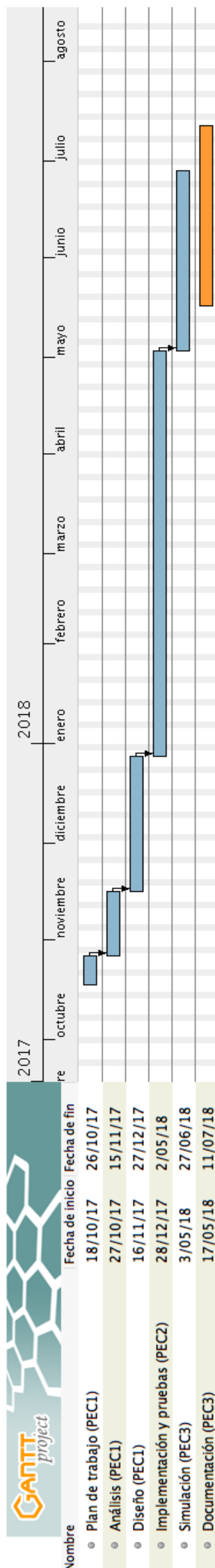


Figura 1: diagrama de Gantt.

1.5 Breve sumario de productos obtenidos

Por un lado, el producto obtenido en código fuente es la aplicación informática que realiza la simulación Monte Carlo y genera los resultados. Dicha aplicación se encuentra publicada (bajo licencia *GPLv3*) en el siguiente repositorio:

https://github.com/adpozuelo/MC

Por otro lado, el producto obtenido del estudio de los resultados de las simulaciones es esta memoria, en la que se describe el proyecto y las conclusiones alcanzadas.

Finalmente, en el anexo A.1 se detalla el contenido del repositorio del proyecto.

1.6 Breve descripción de los otros capítulos de la memoria

- En el capítulo 2 se exponen los fundamentos físicos del problema computacional a tratar.

- En el capítulo 3 se expone la arquitectura computacional utilizada y el modelo *CUDA*.

- En el capítulo 4 se expone el método utilizado para validar los resultados físicos generados por el modelo implementado.

- En el capítulo 5 se exponen los algoritmos diseñados e implementados en sus versiones serie (*CPU*) y paralela (*GPU*).

- En el capítulo 6 se exponen las decisiones tomadas respecto a la implementación.

- En el capítulo 7 se expone el resultado objetivo del proyecto: la aceleración (*speed-up*), y escalado de la misma respecto al tamaño del problema, de la versión paralela (*GPU*) respecto de la serie (*CPU*).

- En el capítulo 8 se exponen las conclusiones finales del Trabajo Final de Máster.

- La memoria se cierra con la bibliografía utilizada para la realización de este proyecto.

2. Fundamentos físicos

¿Qué es y qué pretende la Simulación Atomística o Molecular? El objetivo central de esta técnica computacional es la determinación de propiedades macroscópicas de la materia (como la presión, energía interna, etc.) a partir de nuestro conocimiento de las propiedades de sus constituyentes, átomos y/o moléculas, en particular a partir de las interacciones entre éstos. El conocimiento de las interacciones entre átomos se obtiene recurriendo al formalismo de la Mecánica Cuántica, a partir de la cual es posible deducir formas funcionales, en algunos casos relativamente simples, como las que emplearemos en este trabajo[3] (interacciones de *Lennard-Jones*, *Morse*, etc.).

Por una parte, el número de partículas presente en cantidades macroscópicas de materia es del orden número de Avogadro, N_A=6.022140857 × 10^{23}, que por su magnitud resulta totalmente inasequible para su tratamiento desde el punto de vista computacional. Será, por tanto, necesario reducir el número de partículas hasta conseguir que el tiempo necesario para calcular las propiedades que deseamos determinar, y la memoria requerida para almacenar las posiciones de las partículas (y potencialmente también sus velocidades y aceleraciones) sean razonables para los medios computacionales actuales. Esto hoy en día se traduce en números de partículas que pueden alcanzar hasta varios millones. No obstante, se ha demostrado, que la aplicación de las denominadas "condiciones periódicas de contorno" (*PBC* o *Periodic Boundary Conditions*) permite obtener resultados más que aceptables para propiedades (que no sean de superficie) con apenas miles o incluso cientos de partículas. Esta técnica básicamente se reduce a considerar que nuestro sistema está formado por un conjunto de partículas que se corresponde con una celda unidad de un sistema periódico, de tal forma que el movimiento de una partícula fuera de la celda es compensado por la entrada de otra en la posición correspondiente del lado opuesto de la celda, tal y como se esquematiza en la figura 2.

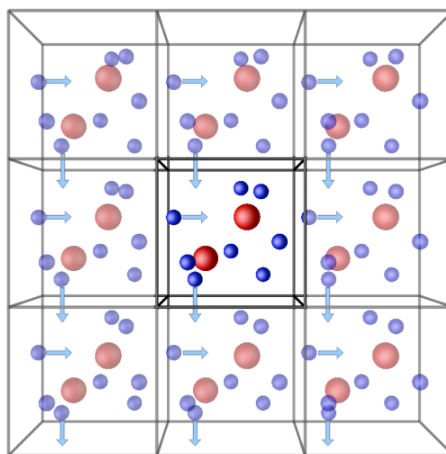


Figura 2: ilustración de las condiciones de contorno periódicas.

Por otra parte, la Mecánica Estadística nos dice que para un estado macroscópico dado – caracterizado por un volumen, temperatura, presión, etc. - existe una multitud de estados microscópicos del sistema compatibles, entendiéndose por tales, configuraciones de las partículas constituyentes, dadas por sus posiciones $\{R_1,\ldots,R_N\}$, y velocidades, $\{v_1,\ldots,v_N\}$. En un procedimiento de simulación atomística se determinarán valores medios de propiedades, como por ejemplo la energía interna del sistema, a partir del conjunto de configuraciones que éste puede adoptar, así se tendrá:

$$U_N = \frac{1}{N_c}\sum_{\{r_i\}} u(r_1, \ldots, r_N) \qquad (1)$$

donde el sumatorio sobre {R_i} indica que todas las configuraciones compatibles con el estado macroscópico, N_c, son incluidas en la suma. Ese conjunto de configuraciones posibles está constreñido por las condiciones macroscópicas, que en un experimento real (y por ende en la simulación) dependerá de cuales sean fijadas. *Gibbs* definió como colectivo el conjunto de microestados (en nuestro caso configuraciones) del sistema sometido a las restricciones macroscópicas que operan sobre el sistema[4]. Así, el colectivo canónico (*NVT*) corresponderá al conjunto de configuraciones compatibles con un número de partículas, volumen y temperatura dados, el isotérmico-isobárico (*NPT*) vendrá especificado por un número de partículas, presión y temperatura dados y el micro-canónico (*NVE*) por un número de partículas, un volumen y una energía total (cinética + potencial) dados. En el denominado límite termodinámico (p.e. número de partículas y volumen infinitos) los promedios calculados son independientes del colectivo elegido.

Una vez determinado en qué colectivo se va a estudiar el sistema dado, los microestados (configuraciones {R_i}) tendrán un peso en (1) que dependerá del colectivo en cuestión, así en el colectivo canónico se tendrá para la energía interna:

$$U_N = \frac{1}{N_c} \sum_{\{r_i\}} \frac{exp\left(-\frac{u(r_1,\ldots r_N)}{k_B T}\right)}{Z(N,V,T)} u(r_1, \ldots r_N) , \qquad (2)$$

siendo k_B la constante de *Boltzmann*, y

$$Z(N,V,T) = \iiint e^{\left(-\frac{u(r_1,\ldots r_N)}{k_B T}\right)} dr_1 \ldots dr_N,$$

es decir, las configuraciones del sistema van a seguir una distribución de *Boltzmann* según su energía configuracional.

$$P(u;N,V,T) = \frac{exp\left(-\frac{u(r_1,\ldots r_N)}{k_B T}\right)}{Z(N,V,T)} , \qquad (3).$$

En este trabajo nos vamos a restringir a propiedades de equilibrio termodinámicas, no de transporte por lo que las velocidades no aparecen en los promedios. Ahora bien, la expresión contenida en la ecuación (2) no es otra cosa que una integral multidimensional sobre el espacio de fases (conjunto de configuraciones {R_i} y en su caso velocidades accesibles al sistema). Para resolver dicha integral multidimensional se pueden emplear diferentes procedimientos, que en definitiva van a constituir métodos para generar configuraciones del sistema compatibles con la distribución de *Boltzmann*. Se pueden clasificar los métodos de simulación atomística en dos grandes familias:

- *Métodos de dinámica molecular*: dadas las posiciones iniciales de un sistema de partículas {R_i^0} y sus velocidades {v_i^0}, así como el potencial de interacción entre ellas, es posible resolver las ecuaciones de movimiento de *Newton*. Dado que por tratarse de un sistema conservativo (la energía potencial se define en términos de una función potencial que depende únicamente de las posiciones de las partículas), conservará la energía total *E* a lo largo de la trayectoria; generando por tanto una serie

de configuraciones en el colectivo microcanónico (*NVE*), con una temperatura dada por $k_B T = \frac{1}{3N}\sum_{i=1}^{N} m_i v_i^2$ promediado sobre las configuraciones $\{R_i, v_i\}$. Las ecuaciones de movimiento se pueden acoplar a distintos tipos de termostatos[5] y de esa forma generar configuraciones a temperatura constante, esto es en el colectivo canónico *NVT* según la distribución dada por la ecuación (3).

- *Métodos de integración estocásticos, métodos de Monte Carlo*: integrales de volumen sobre geometrías complejas se realizan habitualmente recurriendo a métodos estocásticos. Uno de los ejemplos clásicos es la determinación del número π, calculado a partir de la relación entre el área de un cuadrado de lado unidad y el cuadrante de círculo inscrito, tal y como se ilustra en la figura 3. Generando pares de números aleatorios *(x,y)* en una distribución uniforme tales que $x,y \in [0,1]$, resulta evidente que la relación entre el número de puntos totales y aquellos que están dentro del círculo es $\pi/4$. Un método similar podría en principio aplicarse para determinar la integral (2), no obstante un muestreo uniforme de configuraciones $\{R_i\}$ sería extremadamente ineficiente, pues una gran parte de las configuraciones generadas corresponden a estados de energías muy elevadas, por lo que la distribución de *Boltzmann*, $P(u(\{R_i\});N,V,T) \to 0$. Es evidente que es necesario seguir una técnica que permita muestrear el espacio de configuraciones adaptada a la distribución por la que se van a regir. Una aproximación posible es el denominado muestreo de importancia (*importance sampling*), esencial para las técnicas de Monte Carlo en las simulaciones atomísticas. La idea central se basa en intentar muestrear más densamente aquellas regiones del espacio de configuraciones que más contribuyen a la integral (2) - o la correspondiente según en qué colectivo hagamos nuestra simulación -. A continuación se detalla uno de los posibles algoritmos que satisface estos requerimientos y que fue el primero en ser utilizado en el contexto del estudio de fluidos, el algoritmo de Metropolis[6].

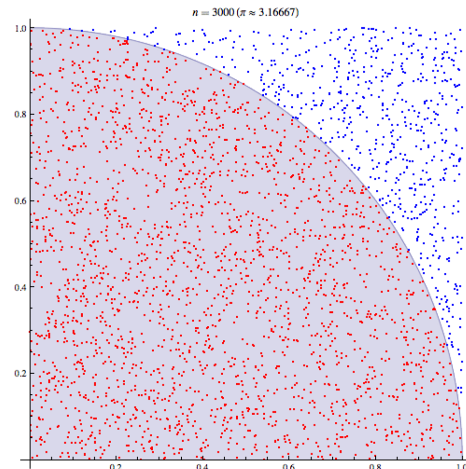


Figura 3: cálculo de π mediante integración de Monte Carlo.

2.1 El algoritmo de Metrópolis en el colectivo canónico

El algoritmo de Metrópolis y colaboradores se basa en la generación de una secuencia de configuraciones, en el caso que nos ocupa siguiendo una distribución de *Boltzmann*, mediante una cadena de *Markov*. En una cadena de *Markov*, la probabilidad de un evento (p.e. el paso de una configuración a otra) depende exclusivamente del estado inmediatamente anterior al evento, esto es, es independiente del proceso mediante el cual se llegó a ese estado anterior. En la práctica el algoritmo de Metrópolis se reduce a los siguientes pasos[4,7]:

1. Se elige una partícula aleatoriamente, *t*, y su posición se modifica de $\boldsymbol{R}_t$ a $\boldsymbol{R}'_t = \boldsymbol{R}_t + \Delta\boldsymbol{R}$, donde $\Delta\boldsymbol{R}$ representa un vector desplazamiento (cuyas componentes tienen una magnitud aleatoria $d_i \in [-\xi,\xi]$.
2. Se determina la energía potencial de la nueva configuración, $U_N(\boldsymbol{R}')$, y a partir de ella $\Delta U_N = U_N(\boldsymbol{R}') - U_N(\boldsymbol{R})$.
    a. Si $\Delta U_N \leq 0$ se acepta el movimiento
    b. Si $\Delta U_N > 0$, se genera un nuevo número aleatorio $\eta \in [0,1]$. Si $exp[-\Delta U_N / k_B T] > \eta$ se acepta el movimiento, en caso contrario se rechaza.
3. Se regresa al paso 1, y, al cabo de cierto número de pasos, ξ se reajusta para que la aceptación de movimientos se aproxime al 50 %.

Este algoritmo, satisface la propiedad de balance detallado, por la que la transición entre dos estados (configuraciones) del sistema consecutivos cumple $p_1 w_{1\to 2} = p_2 w_{2\to 1}$, donde p_i es la probabilidad del estado *i*, y $w_{i\to j}$ es la probabilidad de la transición $i \to j$. De esta forma se garantiza que la secuencia de configuraciones generada, además de ser una cadena *Markov*, converge al cabo de cierto número de configuraciones (tiempo de equilibrado) hacia la distribución de *Boltzmann*, lo que permitirá obtener un muestreo eficiente (de acuerdo con los criterios de *importance sampling*) en los promedios termodinámicos de propiedades de equilibrio, como la integral (2) correspondiente a la energía potencial o configuracional.

Adaptación al colectivo isotérmico-isobárico: en muchas ocasiones los experimentos de laboratorio se desarrollan a presión constante, permitiéndose variaciones de volumen. Las variables que permanecerán fijas a lo largo de la simulación atomística son *N*, *P*, y *T*. La cadena de *Markov* en este caso se genera mediante dos tipos de procesos estocásticos, por una parte movimientos de partículas, siguiendo el esquema indicado en el párrafo anterior, y por otra parte cambios de volumen del sistema, o lo que es equivalente reescalado de las coordenadas de las partículas constituyentes del mismo. La termodinámica dice que el potencial termodinámico relevante en este proceso será la entalpía, definida como $H_N = U_N + PV_N$ (el subíndice *N* resalta la dependencia con el número de partículas, *N*, de las propiedades extensivas). La nueva secuencia se generará entonces en los siguientes pasos[7]:

1. El tamaño del lado de la celda unidad se modifica aleatoriamente según $L' = L + \Delta L_{max}(2\xi - 1)$, donde ξ es un número aleatorio uniforme tal que $\xi \in [0,1]$.
2. Se determina $\Delta H = U(V') - U(V) + P(V - V') - Nk_B T \log(V' / V)$. Eligiendo un nuevo número aleatorio $\xi' \in [0,1]$, si $exp[-\Delta H / k_B T] > \xi'$ el cambio de volumen se acepta, en caso contrario se rechaza. ΔL_{max} se ajusta al cabo de cierto número de pasos para obtener una aceptación cercana al 50 %.

Por cada cambio de volumen es habitual realizar *N* intentos de desplazamiento de partículas.

2.2 ¿Dinámica Molecular o Monte Carlo?

Los procedimientos de simulación atomística reseñados en las líneas precedentes son adecuados para el estudio de propiedades de equilibrio de una gran variedad de sistemas tanto fluidos como sólidos. No obstante, la Dinámica Molecular permite la obtención de propiedades dinámicas, cosa que no es posible estrictamente mediante métodos de Monte Carlo, en los que el tiempo como tal no está definido. Por otra parte, los métodos de Monte Carlo son particularmente útiles para el estudio de transiciones de fase, en los que la evolución del sistema según las ecuaciones de *Newton* puede no muestrear adecuadamente el espacio de configuraciones del sistema. Esto es debido a la existencia de grandes barreras de energía libre que son prácticamente imposibles de atravesar en el orden temporal asequible a las simulaciones atomísticas (inferior o del orden del microsegundo). Sin embargo, es posible desarrollar algoritmos de Monte Carlo adaptados especialmente para este tipo de problemas. Se trata pues de técnicas complementarias.

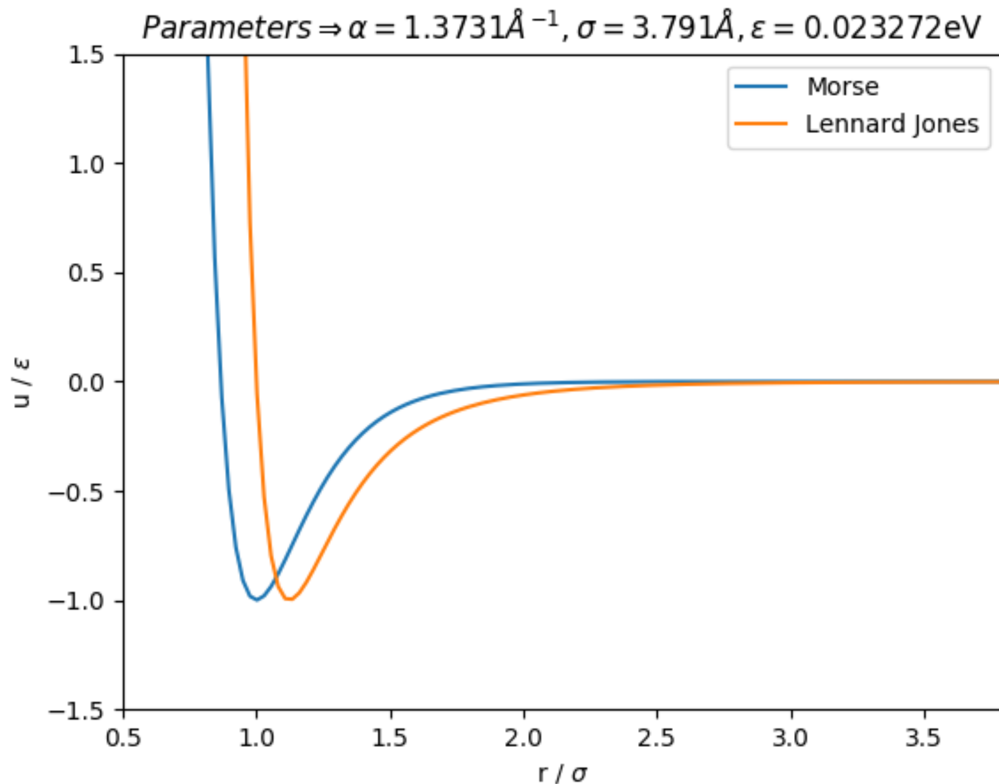


Figura 4: potenciales de Morse y Lennard Jones con los parámetros utilizados en este estudio.

2.3 Otras propiedades: el potencial químico

El potencial químico, μ, es una magnitud termodinámica que mide el cambio de energía libre como consecuencia de la variación del número de partículas en una muestra. Desde el punto de vista microscópico, el potencial químico se puede medir a partir del trabajo necesario para insertar una partícula en una muestra. En principio, el potencial químico de exceso, que mide la diferencia entre el potencial químico total, μ, y el de un gas ideal, μ^{id}, se puede calcular como:

$$\mu^{ex} = \mu - \mu^{id} = -k_B T \, log\langle e^{-U_{N+1}/k_B T}\rangle$$

Donde $U_{N+1}(\boldsymbol{r}_{N+1})$ es la energía de interacción de una partícula que se intenta insertar en la posición $\boldsymbol{r}_{N+1}$ con el resto de las partículas de muestra, y $\langle \ldots \rangle$ denota un promedio sobre configuraciones de un determinado colectivo, normalmente el colectivo canónico, *NVT*. Hay que señalar que las partículas de hecho no son insertadas en la muestra, sólo se calcula el cambio de energía que supondría su inserción. Este procedimiento, denominado *test-particle insertion* (*TPI*) es debido a Widom[14] y aunque no es el más eficiente estadísticamente, es simple y fácilmente paralelizable, por lo que será el que se considere en este trabajo. El procedimiento se puede mejorar implementando junto con inserciones de partículas borrado de partículas, tal y como se hace en el método de Bennet[15]. En ambos casos el procedimiento se realiza sobre configuraciones estadísticamente independientes del colectivo canónico, y cada intento de inserción es totalmente independiente de los demás, con lo que finalmente, si se realizan n_i intentos sobre N_c configuraciones del sistema estadísticamente independientes, el promedio quedaría como:

$$e^{-\mu^{ex}/k_B T} = \frac{1}{N_c n_i} \sum_j \sum_i exp\left(-(U_j^{N+1}(\vec{r}_1, .., \vec{r}_N, \vec{r}_i) - U_j^N(\vec{r}_1, .., \vec{r}_N)\right)$$

donde j denota la configuración e i la partícula que se intenta insertar n_i veces.

3. Arquitectura computacional

3.1 CPU vs GPU

Debido a la demanda de procesamiento en tiempo real o semi-real de grandes cantidades de datos (*Big Data*) y de la necesidad de gráficos *3D* en alta definición, las *GPUs* se han convertido en procesadores con altas capacidades de paralelización, *multi-cores* y multi-hilo. Dichas características forman unidades de procesamiento *GPU* con una potencia computacional enorme y un alto ancho de banda de memoria (figuras 5 y 6).

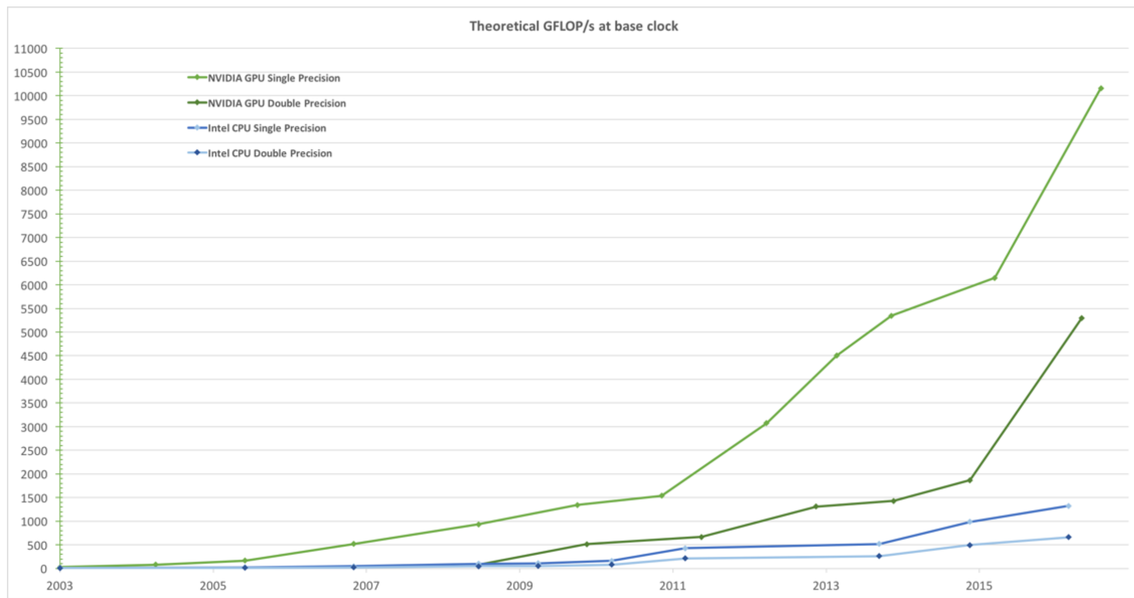


Figura 5: operaciones en coma flotante por segundo en CPU y GPU[9].

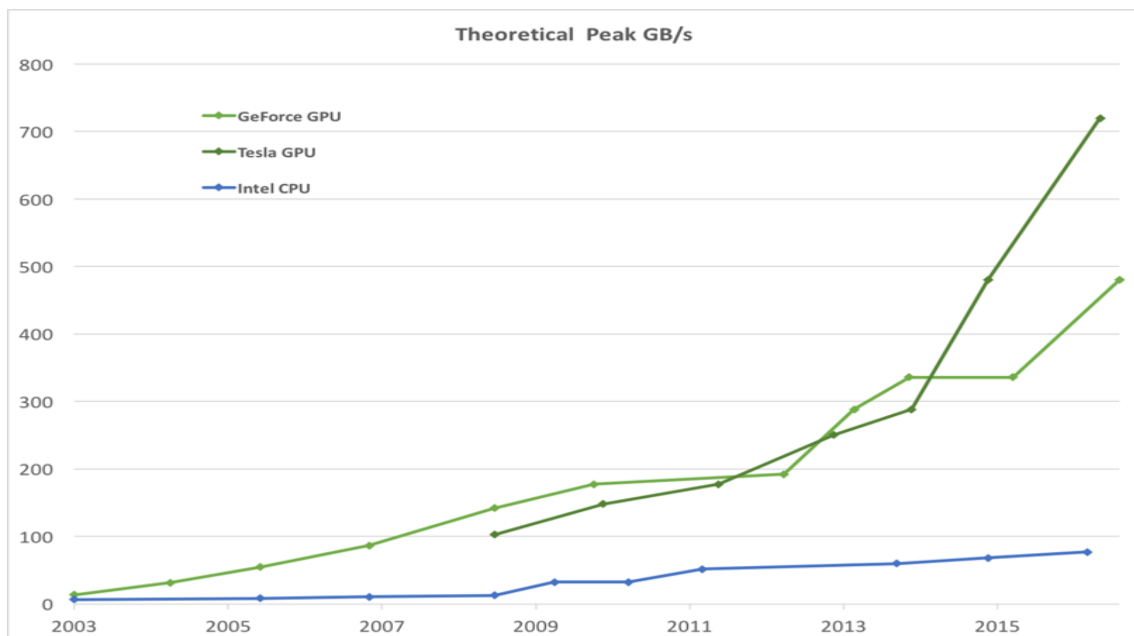


Figura 6: ancho de banda de memoria en CPU y GPU[9].

Figura 7: número de transistores para procesar datos en CPU y GPU[9].

El origen en la diferencia de capacidad computacional entre una *CPU* y una *GPU* radica en que ésta última esta diseñada para tareas de computo intensivo y altamente paralelizable (p.e. renderización gráfica). Por lo tanto, una GPU ha sido diseñada de forma que más transistores son dedicados al procesamiento de datos en lugar de al control de flujo o al caché de los datos (figura 7).

Concretamente, una GPU está especialmente diseñada para problemas de computación paralela de datos; esto es: el mismo programa/algoritmo es ejecutado en paralelo sobre distintos conjuntos de datos (figura 8) – *SIMD* (*Single Instruction Multiple Data*) -. Además, si la computación paralela de datos es altamente intensiva en cálculos aritméticos la latencia de acceso a memoria puede ser mitigada con dichos cálculos, en lugar de recurrir al uso masivo de memorias cachés de datos. El procesamiento paralelo de datos asigna elementos de tales datos a hilos de procesamiento paralelo (p.e. en un renderizado *3D* los conjuntos de píxeles y vértices son asignados a hilos paralelos).

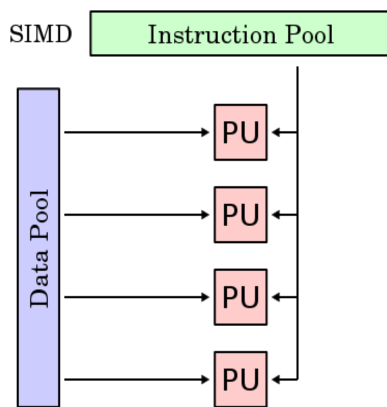


Figura 8: modelo SIMD[16].

3.2 CUDA

CUDA es una plataforma y un modelo de programación de propósito general para computación paralela que contiene un *SDK* que permite a los programadores utilizar el lenguaje C. Este modelo escalable de programación genera una abstracción que permite al programador utilizar los recursos arquitectónicos de la *GPU* (procesadores y memoria) mediante el uso de una *API* propia. Concretamente *CUDA* aporta tres abstracciones base: la jerarquía de memorias, de agrupación de hilos y las barreras de sincronización. Estas abstracciones proveen paralelismo de datos de grano fino (*fine-grained*) y de hilos, anidado junto al paralelismo de datos de

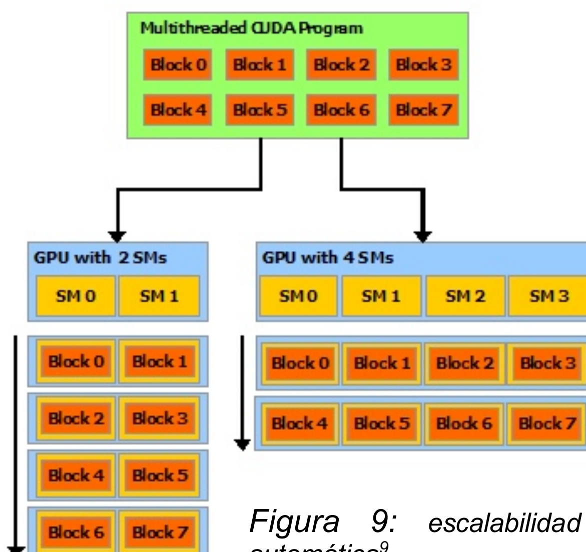


Figura 9: escalabilidad automática[9].

grano gordo (*coarse-grained*) y de tareas. Por lo tanto, el programador, utilizando estas abstracciones, puede particionar el problema en sub-problemas (grano-gordo) que pueden ser resueltos en paralelo de forma independiente en bloques de hilos; dichos sub-problemas deben ser particionados en elementos (grano-fino) que pueden ser resueltos cooperativamente y en paralelo por todos los hilos que forman un bloque. Esta descomposición permite a los hilos cooperar cuando resuelven cada sub-problema y, al mismo tiempo, genera una escalabilidad automática (figura 9).

De hecho, cada bloque de hilos puede ser programado en cualquiera de los multiprocesadores de *streaming* (figura 10) disponibles en la *GPU* en cualquier orden y de forma concurrente o secuencialmente (figura 9). Una *GPU* está construida mediante un conjunto (*array*) de multiprocesadores de *streaming* (*SMs*) . Por lo tanto, un programa *CUDA* se encuentra particionado en bloques de hilos que se ejecutan de forma independiente en cada uno de estos multiprocesadores de *streaming*. Esto quiere decir, obviamente, que una *GPU* con mayor número de multiprocesadores de *streaming* ejecutará un programa *CUDA* en menor tiempo que otra *GPU* con menor número de *SMs*.

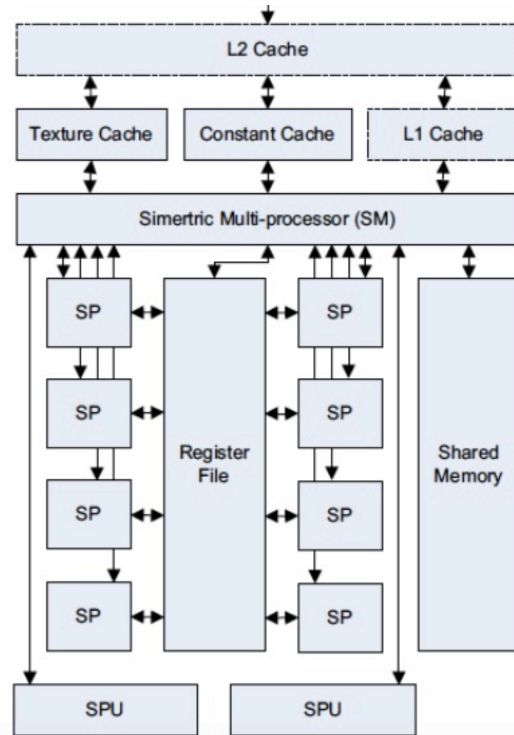


Figura 10: multiprocesador de streaming.

3.2.1 Kernels

CUDA extiende el lenguaje C permitiendo al programador definir funciones llamadas *kernels* que, cuando son llamadas, son ejecutadas *N* veces en paralelo por *N* hilos *CUDA* distintos. Cada hilo que ejecuta el *kernel* posee un identificador vectorial único con tres componentes denominado *threadIdx*. Por lo tanto, un hilo puede ser identificado utilizando una, dos o tres dimensiones; lo que genera, a su vez, bloques de hilos de una, dos y tres dimensiones. Esta jerarquía de hilos provee un método natural de acceso a elementos de estructuras de datos tales como vectores de un índice A(L) y matrices de dos índices A(L, M) o de tres índices A(L, M, N). Existe un límite en el número de hilos que puede contener un bloque (1024 hilos en las *GPU* actuales), dado que todos los hilos del bloque deben residir en el mismo multiprocesador de *streaming* y deben compartir los recursos limitados de memoria de dicho procesador.

Type	Band width	On chip	Cached	Scope
Register	8000 Tb/s	yes	no	One thread
Shared	1600 Tb/s	yes	N/A	Threads in a block
Global	177 Gb/s	no	yes	All threads and CPU
CPU Mapped	8 Gb/s	no	no	All threads and CPU

Figura 11: características de los tipos de memoria de la GPU.

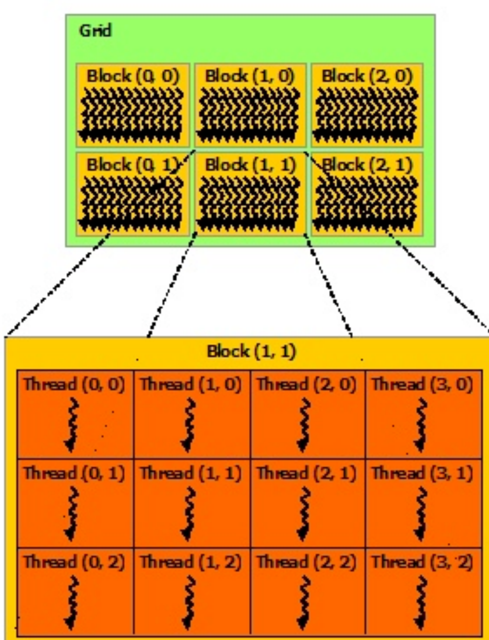


Figura 12: grid, bloques e hilos en el modelo CUDA[9].

Los bloques de hilos son organizados en mallas (*grids*) (figura 12) de una, dos o tres dimensiones. Normalmente, el número de bloques de hilos de un *grid* viene determinado por el tamaño de los datos que deben ser procesados o el número de procesadores del sistema. Cada bloque posee un identificador vectorial único con tres componentes denominado *blockIdx*. Por lo tanto, un bloque puede ser identificado utilizando una, dos o tres dimensiones. Al igual que en el caso de los hilos, esta jerarquía de bloques provee un método natural de acceso a elementos de estructuras de datos tales como vectores de un índice A(L) y matrices de dos índices A(L, M) o de tres índices A(L, M, N). Los bloques de hilos son ejecutados de forma independiente y de forma secuencial o paralela, lo que permite al programador escribir código que escala automáticamente con el número de procesadores.

Los hilos dentro de un bloque pueden cooperar compartiendo datos a través de la memoria compartida y sincronizando su ejecución para coordinar el acceso a dicha memoria. Por lo tanto, para evitar condiciones de carrera (*race conditions*) es necesario establecer puntos de sincronización en el *kernel*, de forma que éstos actúen a modo de barrera en la que los hilos del bloque deben esperar antes de que cualquiera de ellos pueda continuar.

3.2.2 Memoria

Los hilos *CUDA* pueden acceder a los datos desde distintos espacios de memoria durante su ejecución (figura 13). Cada hilo de cada bloque posee una memoria local privada; cada bloque de hilos posee, durante su ejecución, una memoria compartida a la que pueden acceder todos los hilos que conforman el bloque; y, finalmente, todos los hilos de todos los bloques tienen acceso a la memoria global de la *GPU*. Además, existen otros dos espacios de memoria (memoria constante, de sólo lectura, y texturizada, que permite reproducir la topología del algoritmo) que no se detallarán ya que no han sido necesarios para este proyecto. Por un lado, la memoria global, la constante y la texturizada son espacios de memoria persistentes entre lanzamientos de *kernels* de la misma aplicación. Por otro lado, la memoria local del hilo y la compartida del bloque son persistentes únicamente durante la ejecución del bloque de hilos en el que se crean. Con objeto de obtener una cooperación eficiente entre hilos del mismo bloque, la memoria compartida debe ser de muy baja latencia; por lo tanto, dicha memoria se encuentra muy cerca de cada *core* (figura 10) del procesador (incluso más cerca que la memoria caché *L1*) y el coste computacional de la rutina de sincronización de hilos es extremadamente bajo.

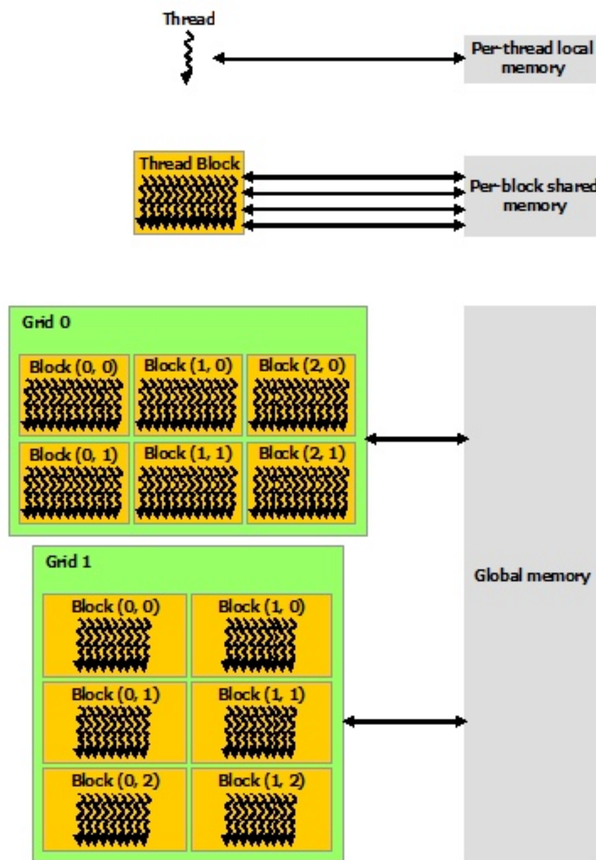


Figura 13: jerarquía de memoria de la GPU[9].

Finalmente, señalar que las operaciones de lectura/escritura en la memoria de la *GPU* se realizan en bloques de 128 bits; por lo que los datos en memoria deben estar alineados y ser coalescentes, con objeto de minimizar el número de operaciones de entrada salida y, por consiguiente, la latencia.

3.2.3 Programación heterogénea

El modelo de programación *CUDA* asume que los hilos *CUDA* se ejecutan en un dispositivo físico (*GPU*) que opera como coprocesador del procesador principal del sistema (*CPU*). Este modelo también asume que tanto la *CPU* como la *GPU* poseen espacios de memoria separados; físicamente la memoria de la *CPU* y la memoria de la *GPU* son espacios separados y distintos. Sin embargo, la *API* de *CUDA* crea una abstracción lógica que establece un puente entre ambas

memorias y presenta al programador una imagen simple y coherente de un espacio de direcciones común.

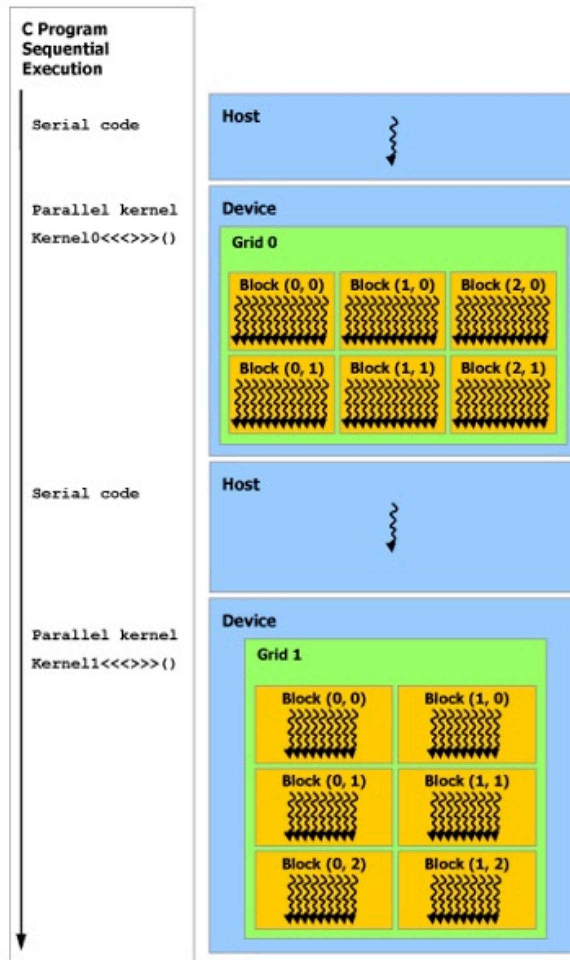


Figura 14: programación heterogénea[9].

Por lo tanto, un programa *CUDA* suele tener el siguiente flujo de ejecución (figura 14):

1. La parte serie del programa procesa la entrada y realiza la transferencia de datos desde el espacio de memoria de la *CPU* al espacio de memoria de la *GPU*.
2. A continuación, la parte serie llama al o los *kernels* que se ejecutarán en la *GPU*. Cabe señalar que los *kernels* se ejecutan de forma asíncrona, esto es: devuelven el control del programa a la *CPU* una vez lanzados.
3. Finalmente, la parte serie del programa realiza la transferencia de datos desde el espacio de memoria de la *GPU* al espacio de memoria de la *CPU*. Hay que destacar que si el *kernel* no ha terminado su ejecución en este momento, el control de programa queda suspendido hasta que esto ocurra. Con los datos procesados por la *GPU*, el programa serie genera la salida.

En conclusión, con objeto de minimizar la latencia y maximizar el ancho de banda de transferencia (figura 11), se deben minimizar las operaciones de transferencia de datos entre los distintos espacios de memoria (*CPU* y *GPU*).

3.2.4 Conclusión

Por lo tanto, utilizando las abstracciones expuestas anteriormente, minimizando la latencia y maximizando el ancho de banda de cada tipo de memoria (figura 11), una aplicación *CUDA* debe[11]:

1. Descomponer un problema en sub-problemas y alinear los datos en memoria de forma coalescente en el espacio de memoria de la *CPU* para, a continuación, transferir dichos datos al espacio de memoria de la *GPU* en el mismo estado. Cabe señalar que los datos deben ser transferidos el menor número de veces entre los espacios de memoria.
2. Dichos sub-problemas serán procesados en la *GPU* de forma paralela por cada bloque de hilos. Los datos a procesar deben ser leídos de la memoria global con el menor número de accesos y alojados en la

memoria compartida del bloque o en los registros de cada hilo. De esta forma se maximizará el ancho de banda de cada tipo de memoria y se minimizará la latencia.

3. Cada sub-problema debe ser dividido en tareas, las cuales serán procesadas de forma paralela por todos los hilos de cada bloque, los cuales cooperarán entre sí utilizando la memoria compartida. Dado que los datos en memoria se encuentran alineados, las operaciones de lectura/escritura se minimizarán debido a la localidad espacial de los datos. Además, y dado que los hilos que cooperan en un mismo bloque repiten las mismas operaciones se maximizará la localidad temporal de los datos mediante el uso de la memoria caché *L1*. Cabe señalar que la cooperación de hilos dentro de un bloque requiere del uso de barreras de sincronización, con objeto de evitar condiciones de carrera con los datos alojados en memoria compartida.
4. Puede ocurrir que se necesite cooperación entre los bloques de hilos que se ejecutan en un *kernel*. En este caso, se deberán utilizar operaciones atómicas (*atomics*) entre los bloques, con objeto de asegurar que las transacciones respetan la consistencia de los datos. Dado que estas operaciones atómicas son computacionalmente costosas, se debe minimizar su uso.

Finalmente, hay tres reglas que deben regir la programación de GPUs y que resumen los aspectos mencionados anteriormente:

1. El programador debe transferir los datos a la *GPU* y mantenerlos en ella.
2. El programador debe proporcionar suficiente trabajo a la *GPU*. Poco trabajo significa desperdicio de tiempo en ciclos ociosos (*idle cycles*). Por lo tanto, el tamaño del problema debe ser suficientemente grande para la *GPU*.
3. El programador debe centrarse en la reutilización de datos. Por lo tanto, hay que evitar las limitaciones del ancho de banda, tanto en transferencias *CPU⇔GPU* como en accesos a memoria global (usar preferentemente registros).

4. Validación del modelo

Aunque el estudio a realizar se centra en la aceleración (*speed-up*) del algoritmo y de cómo escala ésta respecto al tamaño al problema, es necesario comprobar que la implementación realizada produce un resultado correcto.

En una primera fase se ha comprobado que los resultados obtenidos por la implementación del método de Monte Carlo en serie (*CPU*) coincide con los resultados de referencia de un paquete de simulación de dominio público (*DLPOLY*[22]). Este paso se realiza para una muestra de dióxido de silicio (SiO_2) – 1536 Si y 3072 O -, partiendo el Monte Carlo de una configuración inicial generada por *DLPOLY*. En el cálculo no se consideran interacciones de Coulomb dada la dificultad de su paralelización. Una vez validado el código para *CPU* se comprueba la estabilidad del método analizando la convergencia al equilibrio desde configuraciones iniciales distintas.

Dadas dos configuraciones iniciales distintas de dióxido de silicio (SiO_2): un cristal cúbico centrado en las caras *FCC* (*Face Center Cubic*) – figura 16 - y una configuración desordenada generada por *DLPOLY* y suministrada mediante un fichero de entrada *CONFIG* (*DLP*) – figura 15 -.

- Ambas configuraciones poseen el mismo número de especies y el mismo número de partículas por especie. Concretamente, se tienen 3072 partículas de Oxígeno y 1536 partículas de Silicio.
- Las simulaciones a partir de ambas configuraciones se realizan a la misma temperatura (4000*K*), densidad (0.053679 *átomos* / $Å^3$) y presión (166.054 *bar*, sólo para *NPT*).

Se realizan dos simulaciones (*LAT* y *DLP*) por cada tipo de problema (*NVT* y *NPT*) y de arquitectura (*CPU* y *GPU*) con las siguientes características:

- 50000 pasos de simulación, de los cuales 10000 pasos son de equilibrado.
- El desplazamiento máximo será de 0.3*Å*, tanto para las coordenadas de la partículas (*NVT* y *NPT*) como para el volumen (*NPT*).
- El potencial seleccionado será *Morse* [$V(r) = \varepsilon(1 - e^{\alpha(r-\sigma)})^2$] con radio de corte de interacción entre partículas de 9*Å* y con los siguientes parámetros (figura 4):
    - O ⇔ O: $\varepsilon = 0.023272 eV$, $\alpha = 1.3731 Å^{-1}$, $\sigma = 3.791 Å$
    - O ⇔ Si: $\varepsilon = 1.99597 eV$, $\alpha = 2.6518 Å^{-1}$, $\sigma = 1.628 Å$
    - Si ⇔ Si: $\varepsilon = 0.007695 eV$, $\alpha = 2.0446 Å^{-1}$, $\sigma = 3.7598 Å$

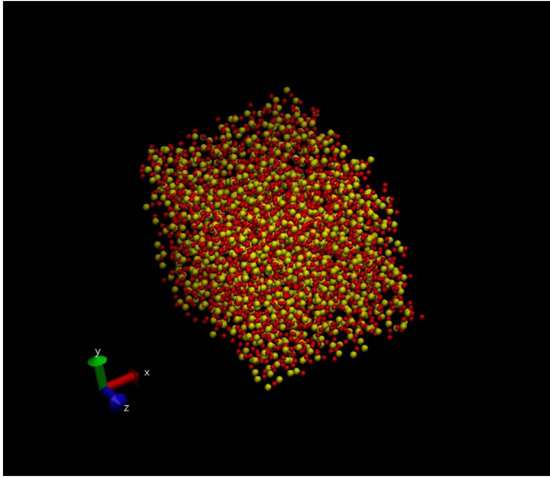


Figura 15: *configuración inicial DLP.*

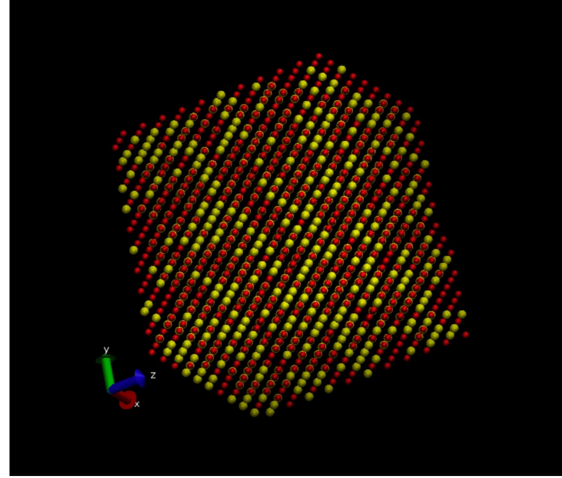


Figura 16: *configuración inicial FCC.*

Por lo tanto, dadas las mismas condiciones termodinámicas para dos configuraciones iniciales distintas y tras la simulación, si el procedimiento está bien implementado ambas configuraciones deben converger, finalmente, a un mismo estado de equilibrio de energía (*NVT* y *NPT*) y de densidad (*NPT*).

Los resultados obtenidos son los siguientes:

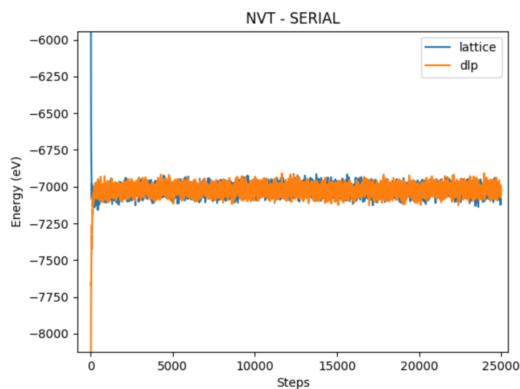


Figura 17: *convergencia energía NVT (CPU)*

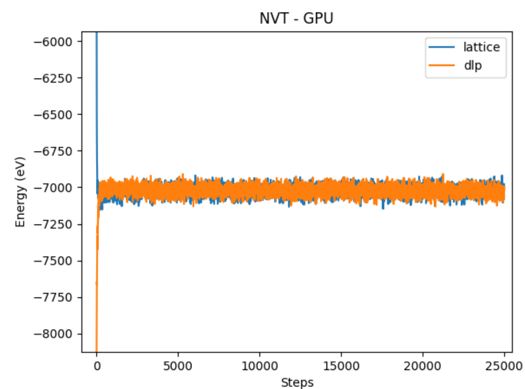


Figura 18: *convergencia energía NVT (GPU)*

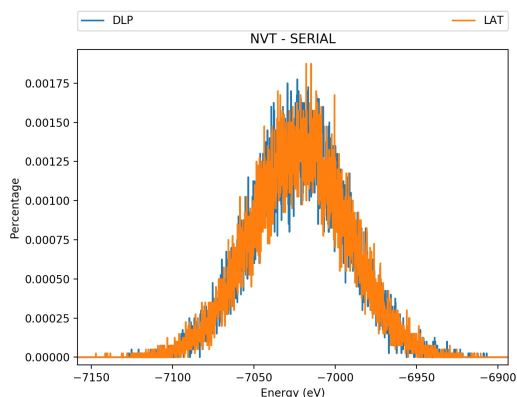


Figura 19: *histograma energía NVT (CPU)*

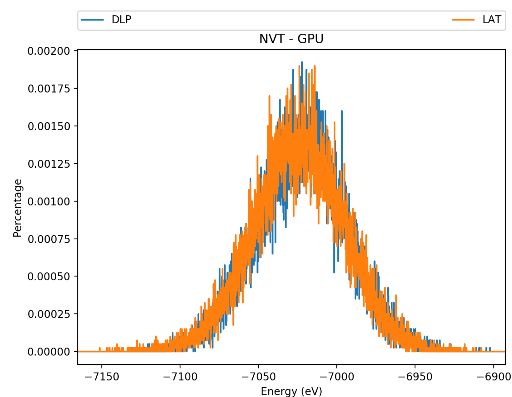
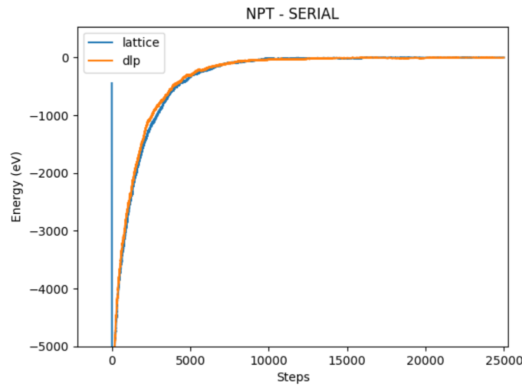


Figura 20: *histograma energía NVT (GPU)*



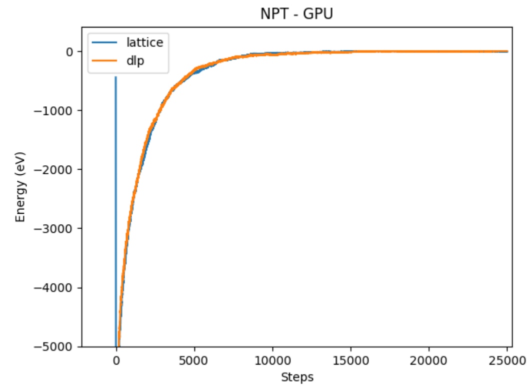
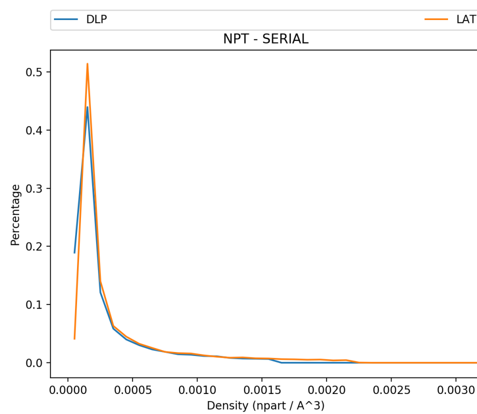


Figura 21: convergencia energía NPT (CPU)

Figura 22: convergencia energía NPT (GPU)



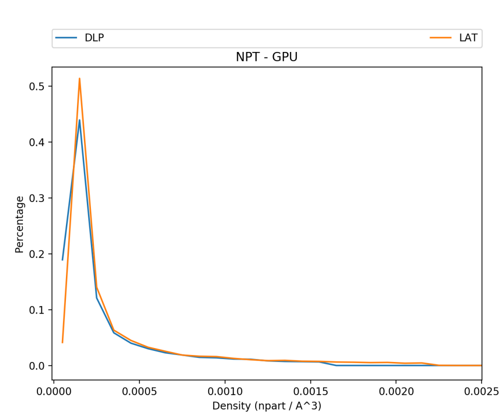


Figura 23: histograma densidad NPT (CPU)

Figura 24: histograma densidad NPT (GPU)

Por lo tanto, tras analizar las gráficas anteriores se puede concluir que la implementación (tanto en *CPU* como en *GPU*) de la solución produce resultados coincidentes para ambos problemas (*NVT* y *NPT*); dado que ambas configuraciones iniciales (*FCC* y *DLP*) convergen hacia las mismas distribuciones de energía (figuras 17, 18, 19, 20, 21 y 22) y de densidad (figuras 23 y 24). En ambos casos el muestreo estocástico conduce a situaciones de equilibrio similares, y que previamente han sido validadas mediante resultados independientes de Dinámica Molecular.

Finalmente, señalar que en el repositorio de código de la implementación se encuentra el directorio *sim/convergence*, el cual contiene todas las simulaciones realizadas y los scripts *Python* para producir las gráficas presentadas en este estudio de validez del modelo.

5. Algoritmos

A continuación se exponen los algoritmos diseñados e implementados en sus versiones serie (*CPU*) y paralela (*GPU*). Cabe señalar que sólo se presentan los algoritmos relevantes al estudio de aceleración o *speed-up*, dado que los demás algoritmos (entrada/salida, estadística, inicialización, lógica de la aplicación, etc.) no son relevantes a efectos de paralelización.

5.1 Rutinas de uso común

En primer lugar se expondrán las subrutinas de uso común siguientes: **el cálculo de la distancia al cuadrado entre dos partículas con *PBC*** (*Periodic Boundary Conditions*) (algoritmo 1), **el cálculo del potencial de interacción entre dos partículas** (algoritmo 2) y **la energía potencial de una configuración dada** (algoritmo 3). Hay que señalar aquí que todas las distancias a efectos de cálculo están escaladas con la longitud del lado de la caja de simulación, esto es: x’ = x / Lx, y’ = y / Ly, z’ = z / Lz, y sólo hemos considerado cajas ortorrómbicas.

Algoritmo 1

***dist2[cpu][gpu]**: distancia al cuadrado entre dos partículas.*
Requiere:
    ndim: número de dimensiones.
    $\vec{r}_i$: partícula i.
    $\vec{r}_j$: partícula j.
    $\overrightarrow{runit}$: valores de normalización de los lados de la caja de simulación.
Devuelve:
    La distancia al cuadrado entre dos partículas.

1: $\overrightarrow{rdd} = \vec{r}_i - \vec{r}_j$ 2:   for k = 0 to ndim – 1: 3:   if $\overrightarrow{rdd}_k > 0{,}5$: 4: $\overrightarrow{rdd}_k -= 1$ 5:   if $\overrightarrow{rdd}_k < -0{,}5$: 6: $\overrightarrow{rdd}_k += 1$ 7: $rd^2 = \sum_{k=0}^{ndim-1} \left(\overrightarrow{rdd}_k \cdot \overrightarrow{runit}_k\right)^2$ 8: return rd^2

Algoritmo 2

***fpot[cpu][gpu]**: potencial de interacción entre dos partículas.*
Requiere:
    rd^2: distancia al cuadrado entre las dos partículas.
    keyp: potencial seleccionado.
    σ: mínimo del potencial Morse y el cero en el potencial Lennard-Jones.
    Tamaño aproximado de las partículas.

σ^2: mínimo (al cuadrado) del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño (al cuadrado) aproximado de las partículas.
ε: profundidad del potencial.
α: decaimiento del potencial (Morse).

Devuelve:
El potencial de interacción entre dos partículas.

1: if keyp == 'morse':
2: rd = $\sqrt{rd^2}$
3: eng = $\varepsilon\left(1 - e^{-\alpha(rd-\sigma)}\right)^2$
4: if keyp == 'lj':
5: eng = $4\varepsilon\left[\left(\frac{\sigma^2}{rd^2}\right)^6 - \left(\frac{\sigma^2}{rd^2}\right)^3\right]$
6: return eng

El **algoritmo serie para el cálculo de la energía de una configuración dada** (algoritmo 3) calcula el potencial de interacción de cada par de partículas (evitando la repetición de pares *ij* = *ji*). Dicho potencial entre pares es acumulado para obtener la energía total del sistema dado.

Este algoritmo posee un complejidad computacional de:

$$O = natoms(natoms - 1) = natoms^2 - natoms$$

Lo que implica que el algoritmo serie para el cálculo de la energía de una configuración dada escala cuadráticamente con el tamaño de la muestra.

Algoritmo 3

***energy[cpu]**: cálculo de la energía de una configuración de partículas dada.*
Requiere:
ndim: número de dimensiones.
natoms: número de átomos/partículas de la configuración dada.
$\vec{r}$: posiciones de las partículas de la configuración dada.
$\overrightarrow{runit}$: valores de normalización de los lados de la caja de simulación.
keyp: potencial seleccionado.
rc^2: radio (al cuadrado) de corte en la interacción entre partículas.
σ: mínimo del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño aproximado de las partículas.
σ^2: mínimo (al cuadrado) del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño (al cuadrado) aproximado de las partículas.
ε: profundidad del potencial.
α: anchura del potencial (Morse).

Devuelve:
La energía de una configuración dada.

```
1:  eng = 0
2:  for i = 0 to natoms - 1:
3:        for j = i + 1 to natoms - 1:
4:                    rd^2 = dist2(ndim, r_i, r_j, runit)
5:                    if rd^2 < rc^2:
6:                              eng += fpot(rd^2, keyp, σ, σ^2, ε, α)
7:  return eng
```

El **algoritmo paralelizado en *GPU* para el cálculo de la energía de una configuración dada** (algoritmo 4) se basa en aplicar las premisas vistas en el apartado 3.2.4 al problema dado. Concretamente, se divide el vector (alineado y transferido una única vez a la memoria de la *GPU*) de posiciones de partículas (*natoms* partículas) en un número de bloques igual a *nblocks* = (*natoms* + (*NTHREAD* - 1)) / *NTHREAD*. Por lo tanto, se divide el problema en bloques de *NTHREADS* hilos cada bloque. Cada hilo dentro del bloque tendrá dos identificadores: *i* le identifica a nivel global dentro del *grid* (*partícula A*) y *tx* le identifica a nivel local dentro del bloque (*partícula B*) - líneas 1 y 2 - . Cada bloque en ejecución de la *GPU* instancia, en memoria compartida, un vector de posiciones de partículas $\overrightarrow{rsh}$ con *NTHREADS* posiciones; una para cada *partícula B*. A continuación, cada *partícula B* recorre todos los bloques (incluido al que pertenece) para obtener las coordenadas $\overrightarrow{rsh}_{tx} = \vec{r}_{(tx+m\cdot blockDim)}$ de todas las *partículas B* separadas entre si por *NTHREADS* posiciones del vector de la configuración $\vec{r}$ (línea 5). Dentro de cada iteración (línea 4) y una vez leída desde la memoria global la posición de la *partícula B* correspondiente (línea 5), todos los hilos del bloque deben esperar (línea 6) una barrera (*syncthreads*). Esto es debido a que las siguientes instrucciones recorrerán el vector $\overrightarrow{rsh}$ y sus datos deben ser consistentes: si algún hilo empieza a leer de este vector y otro hilo no ha transferido la información desde la memoria global a la compartida (línea 5), se producirá una condición de carrera. A continuación, si la *partícula A* es válida (línea 7) y es distinta de la *partícula B* (línea 9), la primera recorrerá todos los elementos del vector $\overrightarrow{rsh}$ (línea 8) para calcular la energía de interacción entre si misma y todas las *partículas B* (líneas 10 y 12), siempre y cuando ambas partículas estén dentro del radio de corte (línea 11). A continuación, todos los hilos dentro del bloque deben esperar una barrera (línea 13) dado que si alguno entra en la siguiente iteración que recorre los bloques (línea 4) y transfiere la información desde la memoria global a la compartida (línea 5), los datos del vector $\overrightarrow{rsh}$ no serán consistentes y se producirá una condición de carrera. Una vez que cada partícula A ha recorrido todos los bloques y calculado su energía de interacción respecto a todas las partículas B, almacena dicha energía en memoria global (una única vez). Por lo tanto, y hasta aquí, se ha dividido el problema en sub-problemas (bloques), cada sub-problema en tareas (hilos dentro del bloque), los hilos dentro del bloque han cooperado (almacenando posiciones de *partículas B* para el uso de las *partículas A*), se ha minimizado la lectura/escritura en memoria global y se ha maximizado el uso de memoria compartida y registros.

A continuación, se debe reducir el vector que contiene la energía de interacción de cada partícula. Para realizar esta tarea se recurre a un **algoritmo *interleaved***

***addressing* de reducción binaria** (algoritmo 5, figura 25) que se ejecuta mediante otro *kernel* (línea 16) sin necesidad de transferir el vector de energías $\overrightarrow{eng}$ entre espacios de memoria (*CPU*⇔*GPU*). Este algoritmo divide el problema (vector energías de *natoms* elementos) en *nblocks* = (*natoms* + (*NTHREAD* - 1)) / *NTHREAD* bloques de *NTHREADS* hilos cada uno. Al igual que en el caso anterior, cada hilo dentro del bloque tendrá dos identificadores: *i* le identifica a nivel global dentro del *grid* (*elemento A*) y *tx* le identifica a nivel local dentro del bloque (*elemento B*) - líneas 1 y 2 –. Cada bloque en ejecución de la *GPU* instancia, en memoria compartida, un vector $\overrightarrow{sh}$ con *NTHREADS* posiciones; una para cada *elemento B* del bloque. A continuación, si el *elemento A* es válido, almacena su valor (energía de interacción) en la posición *tx* que le corresponde en el vector en memoria compartida (línea 5) desde la memoria global. En la línea 8 todos los *elementos B* tienen que respetar una barrera de sincronización, dado que en los siguientes pasos se utilizará el vector $\overrightarrow{sh}$ de la memoria compartida y, éste, debe ser consistente. A continuación se iterará en el tamaño del vector $\overrightarrow{sh}$ dividiendo el mismo entre dos en cada paso (líneas 9, 10 y 13) y almacenando dicho valor en *s*. En cada iteración, los *elementos B* (hilos del bloque) cuyo índice sea inferior a *s* (línea 11), sumarán su valor del vector $\overrightarrow{sh}_{tx}$ y el separado por *s* posiciones (línea 12); tal y como se ilustra en la figura 25. Tras cada suma, todos los hilos dentro del bloque deben respetar una barrera de sincronización (línea 14). Finalmente, y dado que cada bloque produce una reducción parcial del vector completo, todos los bloques deben reducir sus resultados parciales mediante operaciones atómicas. Para realizar esta labor, el hilo cero de cada bloque realiza dicha operación atómica acumulando su resultado en un acumulador en memoria global (línea 16). Por lo tanto, en la reducción binaria, se ha dividido el problema en sub-problemas (bloques), cada sub-problema en tareas (hilos dentro del bloque), los hilos dentro del bloque han cooperado (almacenando valores de *elementos A* para el uso de los *elementos B*), se ha minimizado la lectura/escritura en memoria global y se ha maximizado el uso de memoria compartida y registros.

Tal y como se ha observado, en el primer *kernel* (cálculo de energía de interacción) las *partículas/elementos B* son los encargados de cooperar para almacenar las coordenadas de las partículas en memoria compartida de modo que las *partículas/elementos A* procesen dichos datos. En el segundo *kernel* (reducción binaria) ocurre al revés: los *elementos A* son los encargados de almacenar los valores desde la memoria global a la memoria compartida. Entonces, *los elementos B* son los encargados de cooperar para reducir dicho vector de la memoria compartida. Finalmente, el algoritmo paralelo de reducción binaria requiere de una cooperación entre bloques que debe realizarse mediante operaciones atómicas para asegurar la consistencia de los datos y evitar accesos múltiples simultáneos a la misma posición de memoria.

Este algoritmo posee una complejidad computacional de (*energy[gpu]* + *binarureduction[gpu]*):

$$O = \frac{natoms \cdot nblocks \cdot blockDim}{blockDim \cdot nSM} + \frac{natoms \cdot log_2 blockDim}{blockDim \cdot nSM} =$$

$$= \frac{natoms \cdot nblocks}{nSM} + \frac{natoms \cdot log_2 blockDim}{blockDim \cdot nSM} = \frac{natoms}{nSM}\left(nblocks + \frac{log_2 blockDim}{blockDim}\right)$$

Siendo *nSM* el número de procesadores de *streaming* de la *GPU* y *blockDim* = *NTHREADS* el número de hilos por bloque. Dado que *nblocks* = (*natoms* + (*NTHREAD* - 1)) / *NTHREAD* el algoritmo paralelizado en *GPU* para el cálculo de la energía de una configuración dada (incluida la reducción binaria) escala cuadráticamente respecto al número de átomos.

Algoritmo 4

energy[gpu]*: cálculo de la energía de una configuración de partículas dada.*
Requiere:
- ndim: número de dimensiones.
- natoms: número de átomos/partículas de la configuración dada.
- nblocks: número de bloques a ejecutar por la GPU.
- $\vec{r}$: posiciones de las partículas de la configuración dada.
- $\overrightarrow{runit}$: valores de normalización de los lados de la caja de simulación.
- keyp: potencial seleccionado.
- rc^2: radio (al cuadrado) de corte en la interacción entre partículas.
- σ: mínimo del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño aproximado de las partículas.
- σ^2: mínimo (al cuadrado) del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño (al cuadrado) aproximado de las partículas.
- ε: profundidad del potencial.
- α: decaimiento del potencial (Morse).

Devuelve:
- La energía de una configuración dada.

```
1:  i = threadIdx + blockDim • blockIdx
2:  tx = treadIdx
3:  eng = 0, eng⃗ = {0}
4:  for m = 0 to nblocks:
5:        rsh⃗_tx = r⃗_(tx+m·blockDim)
6:        syncthreads
7:        if i < natoms:
8:              for j = 0 to blockDim - 1:    ¡ Comprobar número de partículas dentro del bloque !
9:                    if i != j:
10:                         rd² = dist2(ndim, r⃗_i, rsh⃗_j, runit⃗)
11:                         if rd² < rc²:
12:                               eng += fpot(rd², keyp, σ, σ², ε, α)
13:       syncthreads
14: if i < atoms:
15:       eng⃗_i = eng
16: eng = binaryreduction[gpu](natoms, eng⃗)
17: return eng / 2
```

Algoritmo 5

binaryreduction[gpu]*: reducción binaria de un vector dado.*

Requiere:

nitems: número de elementos del vector.

$\overrightarrow{reduce}$: vector a reducir.

Devuelve:

La suma de todos los elementos del vector $\overrightarrow{reduce}$.

1: i = threadIdx + blockDim • blockIdx
2: tx = threadIdx
3: sum = 0
4: if i < nitems:
5: $\overrightarrow{sh}_{tx} = \overrightarrow{reduce}_i$
6: else:
7: $\overrightarrow{sh}_{tx} = 0$
8: *syncthreads*
9: s = blockDim / 2
10: while s > 0:
11: if tx < s:
12: $\overrightarrow{sh}_{tx} += \overrightarrow{sh}_{tx+s}$
13: s /= 2
14: *syncthreads*
15: if tx == 0:
16: *atomicAdd*(sum, $\overrightarrow{sh}_{tx}$)
17: return sum

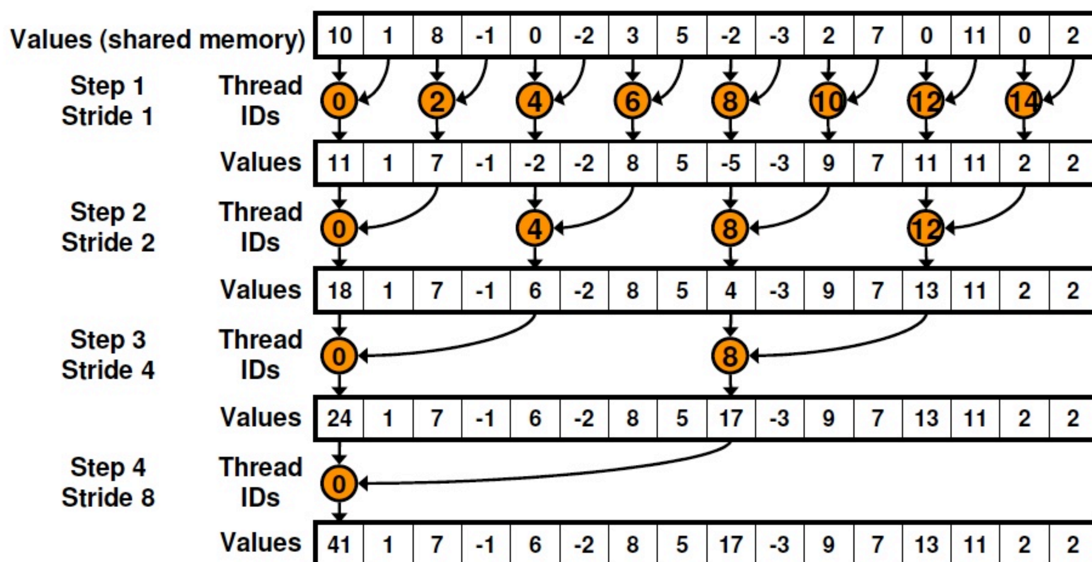


Figura 25: reducción binaria (interleaved addressing)

5.2 Traslación de partículas, algoritmo Metrópolis (*NVT* y *NPT*).

Respecto al **algoritmo Metrópolis para la traslación de partículas (*MC-NVT*), la versión serie** (algoritmo 6) selecciona de forma aleatoria una partícula a

mover (*ntest*) en cada una de las *natoms* iteraciones a ejecutar (líneas 1 y 2). Después, para la partícula $\vec{r}_{ntest}$, en cada iteración, se modifican sus coordenadas $\vec{p}_{test}$ guardando las coordenadas originales (línea 3). A continuación, dentro de la iteración, se calcula la energía potencial de la antigua (línea 11) y de la nueva configuración (*e0* y *e1*, respectivamente) (línea 13): utilizando las coordenadas originales (línea 8) y las nuevas de la partícula (línea 9). Con las dos energías potenciales, se calcula la diferencia de energías (*deltae*) (línea 14), y a partir de ahí:

- Si la diferencia es negativa (*deltae* < *0*) se acepta el movimiento ($\vec{r}_{ntest} = \vec{p}_{test}$) (líneas 16 a 17).
- Si la diferencia no es negativa se genera un número aleatorio que es comparado con el criterio de aceptación ($e^{(-deltae/kt)}$ > *uniformRNG*(float, 0, 1)). Si el número aleatorio es menor que dicho criterio, el movimiento es aceptado ($\vec{r}_{ntest} = \vec{p}_{test}$) (líneas 18 a 21).

Este algoritmo posee un complejidad computacional de:

$$O = natoms^2$$

Esto es, el algoritmo serie *MC-NVT* escala cuadráticamente respecto al número de átomos. A continuación, se presenta en pseudocódigo el algoritmo serie de generación de traslaciones correspondiente al Monte Carlo *NVT*.

Algoritmo 6
***moveatoms[cpu]**: algoritmo Metrópolis para realizar intentos de traslación de átomos de la configuración dada.* Requiere: ndim: número de dimensiones. natoms: número de átomos/partículas de la configuración dada. ntrial: número de intentos realizados. naccept: número de movimiento aceptados. $\vec{r}$: posiciones de las partículas de la configuración dada. $\overrightarrow{runit}$: valores de normalización de los lados de la caja de simulación. $\overrightarrow{rdmax}$: desplazamiento máximo de la partícula. keyp: potencial seleccionado. rc^2: radio (al cuadrado) de corte en la interacción entre partículas. σ: mínimo del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño aproximado de las partículas. σ^2: mínimo (al cuadrado) del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño (al cuadrado) aproximado de las partículas. ε: profundidad del potencial. α: decaimiento del potencial (Morse). kt: constante de Boltzmann por la temperatura.

```
1:  for i = 0 to natoms – 1:
2:        ntest = uniformRNG(int, 0, natoms - 1)
3:        p⃗_test = r⃗_ntest + rdmax⃗ • (2 • uniformRNG(float, 0, 1) – 1)
4:        e0 = 0, e1 = 0, ntrial++
5:        for j = 0 to natoms - 1:
6:                  if j == ntest:
7:                        continue
8:                  rd² = dist2(ndim, r⃗_j, r⃗_ntest, runit⃗)
9:                  rdn² = dist2(ndim, r⃗_j, p⃗_test, runit⃗)
10:                 if rd² < rc²:
11                        e0 += fpot(rd², keyp, σ, σ², ε, α)
12:                 if rdn² < rc²:
13:                       e1 += fpot(rdn², keyp, σ, σ², ε, α)
14:       deltae = e1 – e0
15:       if deltae < 0:
16:                 r⃗_ntest = p⃗_test
17:                 naccept++
18:       else:
19:                 if e^(−deltae/kt) > uniformRNG(float, 0, 1)
20:                       r⃗_ntest = p⃗_test
21:                       naccept++
```

El **algoritmo paralelizado en *GPU MC-NVT*** (algoritmo 7) selecciona de forma aleatoria una partícula a mover (*ntest*) en cada una de las *natoms* iteraciones a ejecutar (líneas 1 y 2). Después, para cada partícula en cada iteración, se modifican sus coordenadas guardando las coordenadas originales $\vec{r}_{ntest}$ y las nuevas $\vec{p}_{test}$ (línea 3). A continuación, dentro de la iteración, se llama al *kernel* de la *GPU* (línea 5) que devuelve las dos energías potenciales de las configuraciones, la original y la desplazada (*e0* y *e1*, respectivamente): utilizando las coordenadas originales y las de prueba de la partícula *ntest*. Con las dos energías potenciales, se calcula la diferencia de energías (*deltae*) (línea 6) de forma que:

- Si la diferencia es negativa (*deltae* < *0*) se acepta el movimiento ($\vec{r}_{ntest} = \vec{p}_{test}$) (líneas 7 a 9).
- Si la diferencia no es negativa se genera un número aleatorio que es comparado con el criterio de aceptación ($e^{(-deltae/kt)}$ > *uniformRNG*(float, 0, 1)). Si el número aleatorio es menor que dicho criterio, el movimiento es aceptado ($\vec{r}_{ntest} = \vec{p}_{test}$) (líneas 10 a 13).

Respecto al *kernel* de la *GPU* el algoritmo se basa en aplicar las premisas vistas en el apartado 3.2.4 al problema dado. Concretamente, se divide el vector (alineado y transferido una única vez a la memoria de la *GPU*) de posiciones de partículas (*natoms* partículas) en un número de bloques igual a *nblocks* = (*natoms* + (*NTHREAD* - 1)) / *NTHREAD*. Por lo tanto, se divide el problema en bloques de *NTHREADS* hilos cada bloque. Cada hilo dentro del bloque tendrá dos identificadores: *j* le identifica a nivel global dentro del *grid* (*partícula A*) y *tx* le identifica a nivel local dentro del bloque - líneas 1 y 2 - . Cada bloque en ejecución de la *GPU* instancia, en memoria compartida, dos vectores de energías de interacción $\overrightarrow{eng0}$ y $\overrightarrow{eng1}$ con *NTHREADS* posiciones cada uno. En dichos vectores, si cada *partícula A* es válida (línea 3) y no es igual a la partícula a

trasladar (línea 4) almacenará su energía de interacción respecto a la partícula *ntest* seleccionada para trasladar (líneas 10 y 14): para las coordenadas originales (línea 7) y usando las coordenadas de prueba (línea 8). Una vez que todas las *partículas A* han calculado y almacenado su energía de interacción en memoria compartida, todos los hilos del bloque en ejecución deben respetar una barrera de sincronización (línea 19). Esto es debido a que en las siguientes líneas se realizará una reducción binaría (figura 25, apartado 5.1) de los vectores instanciados en memoria compartida ($\overrightarrow{eng0}$ y $\overrightarrow{eng1}$) dentro del bloque. Finalmente, y dado que cada bloque produce una reducción parcial del vector completo, todos los bloques deben reducir sus resultados parciales mediante operaciones atómicas. Para realizar esta labor, el hilo cero de cada bloque realiza dicha operación atómica acumulando su resultado en una acumulador en memoria global (líneas 28 y 29). Por lo tanto, en el algoritmo, se ha dividido el problema en sub-problemas (bloques), cada sub-problema en tareas (hilos dentro del bloque), los hilos dentro del bloque han cooperado (reducción binaria), se ha minimizado la lectura/escritura en memoria global y se ha maximizado el uso de memoria compartida y registros.

Este algoritmo posee un complejidad computacional de:

$$O = natoms \frac{natoms}{blockDim \cdot nSM} log_2 blockDim$$

Siendo *nSM* el número de procesadores de *streaming* de la *GPU* y *blockDim* = *NTHREADS* el número de hilos por bloque. En conclusión, el algoritmo paralelo *MC-NVT* para *GPU* escala cuadráticamente respecto al número de átomos.

Cabe señalar que *blockDim • nSM* es el número total de *cores CUDA* (*ncCUDA*) que posee la *GPU*, por lo tanto:

$$\lim_{ncCUDA \to natoms} O = natoms \frac{natoms}{ncCUDA} log_2 blockDim = natoms \cdot log_2 blockDim$$

De modo que, cuantos más *cores CUDA* posea la *GPU* menor será la complejidad computacional del algoritmo. A continuación, se presenta el pseudocódigo correspondiente a la rutina que realiza los intentos de traslación en la *GPU*.

Algoritmo 7
moveatoms[gpu]*: algoritmo Metrópolis para realizar intentos de traslación de átomos de la configuración dada.* Requiere: ndim: número de dimensiones. natoms: número de átomos/partículas de la configuración dada. naccept: número de movimiento aceptados. ntrial: número de intentos realizados. $\vec{r}$: posiciones de las partículas de la configuración dada. $\overrightarrow{runit}$: valores de normalización de los lados de la caja de simulación. $\overrightarrow{rdmax}$: desplazamiento máximo de la partícula.

keyp: potencial seleccionado.
rc^2: radio (al cuadrado) de corte en la interacción entre partículas.
σ: mínimo del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño aproximado de las partículas.
σ^2: mínimo (al cuadrado) del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño (al cuadrado) aproximado de las partículas.
ε: profundidad del potencial.
α: decaimiento del potencial (Morse).
kt: constante de Boltzmann por la temperatura.

1: for i = 0 to natoms – 1:
2: ntest = *uniformRNG*(int, 0, natoms - 1)
3: $\vec{p}_{test} = \vec{r}_{ntest} + \overrightarrow{rdmax}$ • (2 • *uniformRNG*(float, 0, 1) – 1)
4: e0 = 0, e1 = 0, ntrial++
5: (e0 , e1) = *kernel[gpu]* {
 1: j = threadIdx + blockDim • blockIdx
 2: tx = threadIdx
 3: if j < natoms:
 4: if j == ntest:
 5: $\overrightarrow{eng0}_{tx}$ = 0, $\overrightarrow{eng1}_{tx}$ = 0
 6: if j != ntest:
 7: rd^2 = *dist2*(ndim, $\vec{r}_j$, $\vec{r}_{ntest}$, $\overrightarrow{runit}$)
 8: rdn^2 = *dist2*(ndim, $\vec{r}_j$, $\vec{p}_{test}$, $\overrightarrow{runit}$)
 9: if $rd^2 < rc^2$:
 10: $\overrightarrow{eng0}_{tx}$ = *fpot*(rd^2, keyp, σ, σ^2, ε, α)
 11: else:
 12: $\overrightarrow{eng0}_{tx}$ = 0
 13: if $rdn^2 < rc^2$:
 14: $\overrightarrow{eng1}_{tx}$ = *fpot*(rdn^2, keyp, σ, σ^2, ε, α)
 15: else:
 16: $\overrightarrow{eng1}_{tx}$ = 0
 17: else:
 18: $\overrightarrow{eng0}_{tx} = 0$, $\overrightarrow{eng1}_{tx} = 0$
 19: *syncthreads*
 20: s = blockDim / 2
 21: while s > 0:
 22: if tx < s:
 23: $\overrightarrow{eng0}_{tx} \mathrel{+}= \overrightarrow{eng0}_{tx+s}$
 24: $\overrightarrow{eng1}_{tx} \mathrel{+}= \overrightarrow{eng1}_{tx+s}$
 25: s /= 2
 26: *syncthreads*
 27: if tx == 0:
 28: *atomicAdd*(e0, $\overrightarrow{eng0}_{tx}$)
 29: *atomicAdd*(e1, $\overrightarrow{eng1}_{tx}$)
 30: return e0, e1
 }
6: deltae = e1 – e0
7: if deltae < 0:
8: $\vec{r}_{ntest} = \vec{p}_{test}$
9: naccept++
10: else:
11: if $e^{(-deltae/kt)}$ > *uniformRNG*(float, 0, 1)
12: $\vec{r}_{ntest} = \vec{p}_{test}$
13: naccept++

5.3 Cambio de volumen (*NPT*).

Respecto al **cambio de volumen (*MC-NPT*)**, previamente en **el algoritmo serie** (algoritmo 8) se almacenan la energía, el volumen, las dimensiones de la caja de simulación y los valores de normalización de los lados de la caja originales (líneas 1 a 4). A continuación, se modifica la longitud de cada lado de la caja, aplicando un desplazamiento aleatorio, (líneas 5 al 7) para calcular el nuevo volumen (línea 8). Además, se actualizan las coordenadas con nuevos valores de normalización de los lados de la caja (línea 9). Después, se llama a la función de cálculo de la energía de la configuración (línea 10, capítulo 5.1) para obtener la energía de dicha configuración tras el cambio de volumen. Con las dos energías potenciales (antes y después del cambio de volumen), se calcula la diferencia de energías, ΔU (línea 11). Además, se genera un número aleatorio *x* (línea 13) y se calcula el criterio de aceptación siguiente (línea 12):

$$c = e^{\frac{-\Delta U + pres \cdot (vNew - vOld)}{kt + natoms \cdot \log\left({vNew}/{vOld} \right)}}$$

Entonces, si *c* > *x* se acepta el movimiento de volumen (líneas 14 y 15). En caso contrario, no se acepta el movimiento de volumen y se restauran los valores originales guardados antes del cambio (líneas 16 a 20).

Este algoritmo posee la complejidad computacional de *energy[cpu]*, por lo tanto:

$$O = natoms(natoms - 1) = natoms^2 - natoms$$

Esto es, el algoritmo serie *MC-NPT* escala cuadráticamente respecto al número de átomos. A continuación, se presenta en pseudocódigo el algoritmo serie de intento de cambio de volumen de la caja de simulación correspondiente al Monte Carlo *NPT*.

Algoritmo 8
movevolume[cpu]*: algoritmo para realizar intentos de cambios de volumen en la configuración dada.* Requiere: energy: energía de la configuración. v: volumen de la caja de simulación ndim: número de dimensiones. natoms: número de átomos/partículas de la configuración dada. nvaccept: número de cambios de volumen aceptados. $\vec{r}$: posiciones de las partículas de la configuración dada. $\overrightarrow{runit}$: valores de normalización de los lados de la caja de simulación. $\overrightarrow{side}$: longitud de los lados de la caja de simulación. pres: presión de la caja de simulación. vdmax: desplazamiento máximo del lado de la caja de simulación. keyp: potencial seleccionado. rc^2: radio (al cuadrado) de corte en la interacción entre partículas.

σ: mínimo del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño aproximado de las partículas.
σ^2: mínimo (al cuadrado) del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño (al cuadrado) aproximado de las partículas.
ε: profundidad del potencial.
α: decaimiento del potencial (Morse).
kt: constante de Boltzmann por la temperatura.

1: energyOld = energy
2: vOld = v
3: $\overrightarrow{runitOld} = \overrightarrow{runit}$
4: $\overrightarrow{sideOld} = \overrightarrow{side}$
5: for i = 0 to ndim – 1:
6: x = *uniformRNG*(float, 0, 1)
7: side[i] += (2 · x – 1) · vdmax
8: vNew= $\sum_{k=0}^{ndim-1} side_k$
9: $\overrightarrow{runit} = \overrightarrow{side}$
10: energy = ***energy[cpu]***(ndim, natoms, $\vec{r}$, $\overrightarrow{runit}$, keyp, rc^2, σ, σ^2, ε, α)
11: deltae = energy – energyOld
12: $c = e^{\frac{-deltae+pres\cdot(vNew-vOld)}{kt+natoms\cdot\log\left({}^{vNew}/_{vOld}\right)}}$
13: x = *uniformRNG*(float, 0, 1)
14: if c > x:
15: nvaccept++
16: else:
17: energy = energyOld
18: v = vOld
19: $\overrightarrow{runit} = \overrightarrow{runitOld}$
20: $\overrightarrow{side} = \overrightarrow{sideOld}$

Para el **algoritmo MC-NPT paralelo en GPU** (algoritmo 9) realiza los mismos pasos que el serie excepto la línea 10, en donde se llama a la versión paralela de la función del cálculo de la energía de la configuración (capítulo 5.1).

Este algoritmo posee la complejidad computacional de *energy[gpu]*, por lo tanto:

$$O = \frac{natoms \cdot nblocks \cdot blockDim}{blockDim \cdot nSM} + \frac{natoms \cdot log_2 blockDim}{blockDim \cdot nSM} =$$

$$= \frac{natoms \cdot nblocks}{nSM} + \frac{natoms \cdot log_2 blockDim}{blockDim \cdot nSM} = \frac{natoms}{nSM}\left(nblocks + \frac{log_2 blockDim}{blockDim}\right)$$

Siendo *nSM* el número de procesadores de *streaming* de la *GPU* y *blockDim* = *NTHREADS* el número de hilos por bloque. Dado que *nblocks* = (*natoms* + (*NTHREAD* - 1)) / *NTHREAD* el algoritmo *MC-NPT* paralelo en *GPU* escala cuadráticamente respecto al número de átomos.

A continuación, se presenta el pseudocódigo correspondiente a la rutina que realiza los intentos de cambio de volumen en la *GPU*.

Algoritmo 9

***movevolume[gpu]**: algoritmo para realizar intentos de cambios de volumen en la configuración dada.*

Requiere:

energy: energía de la configuración.
v: volumen de la caja de simulación
ndim: número de dimensiones.
natoms: número de átomos/partículas de la configuración dada.
nvaccept: número de cambios de volumen aceptados.
$\vec{r}$: posiciones de las partículas de la configuración dada.
$\overrightarrow{runit}$: valores de normalización de los lados de la caja de simulación.
$\overrightarrow{side}$: longitud de los lados de la caja de simulación.
pres: presión de la caja de simulación.
vdmax: desplazamiento máximo del lado de la caja de simulación.
keyp: potencial seleccionado.
rc^2: radio (al cuadrado) de corte en la interacción entre partículas.
σ: mínimo del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño aproximado de las partículas.
σ^2: mínimo (al cuadrado) del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño (al cuadrado) aproximado de las partículas.
ε: profundidad del potencial.
α: decaimiento del potencial (Morse).
kt: constante de Boltzmann por la temperatura.

1: energyOld = energy
2: vOld = v
3: $\overrightarrow{runitOld} = \overrightarrow{runit}$
4: $\overrightarrow{sideOld} = \overrightarrow{side}$
5: for i = 0 to ndim – 1:
6: x = *uniformRNG*(float, 0, 1)
7: side[i] += (2 · x – 1) · vdmax
8: vNew= $\sum_{k=0}^{ndim-1} side_k$
9: $\overrightarrow{runit} = \overrightarrow{side}$
10: energy = ***energy[gpu]***(ndim, natoms, $\vec{r}$, $\overrightarrow{runit}$, keyp, rc^2, σ, σ^2, ε, α)
11: deltae = energy – energyOld
12: $c = e^{\frac{-deltae + pres\cdot(vNew - vOld)}{kt + natoms\cdot\log\left(vNew/vOld\right)}}$
13: x = *uniformRNG*(float, 0, 1)
14: if c > x:
15: nvaccept++
16: else:
17: energy = energyOld
18: v = vOld
19: $\overrightarrow{runit} = \overrightarrow{runitOld}$
20: $\overrightarrow{side} = \overrightarrow{sideOld}$

5.4 Potencial químico (*NVT*).

Por un lado, el **algoritmo serie para el cálculo del potencial químico** (algoritmo 10) realiza para cada tipo de especie (*nsp*) (línea 1) y para un número de intentos de inserción (*chpotit*) (línea 2) las siguientes operaciones:

- Genera de forma aleatoria tres coordenadas que corresponden a la nueva partícula a insertar $\overrightarrow{rnew}_k$ (líneas 3 y 4).
- Calcula la energía potencial de interacción (*eng*) entre esta nueva partícula y todas las existentes en la configuración dada (líneas 6 a 9).
- Cada iteración (intento de inserción), almacena la contribución al potencial químico de exceso en el vector $\overrightarrow{eng}_j$ = $e^{(-eng/kt)}$ (línea 10).

Finalmente, para cada tipo de partícula *nsp* y después de todos los intentos de inserción, se calcula el valor medio del potencial en exceso *chpot* = $\frac{\sum_{k=0}^{chpotit-1}\overrightarrow{eng}_k}{chpotit}$ para cada tipo de especie (línea 11).

Este algoritmo posee un complejidad computacional de:

$$O = nsp \cdot chpotit \cdot natoms$$

Dado que para este trabajo $chpotit = natoms$, finalmente se tiene que:

$$O = nsp \cdot natoms^2$$

Esto es, el algoritmo serie del cálculo del potencial químico también escala cuadráticamente con el tamaño de la muestra.

Algoritmo 10
chpotential[cpu]*: cálculo del potencial químico[14] (algoritmo Widom) mediante la inserción (independiente) de un número dado de partículas en la configuración dada.* Requiere: ndim: número de dimensiones. natoms: número de átomos/partículas de la configuración dada. nsp: número de especies. chpotit: número de inserciones a realizar. $\vec{r}$: posiciones de las partículas de la configuración dada. $\overrightarrow{runit}$: valores de normalización de los lados de la caja de simulación. keyp: potencial seleccionado. rc^2: radio (al cuadrado) de corte en la interacción entre partículas. σ: mínimo del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño aproximado de las partículas. σ^2: mínimo (al cuadrado) del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño (al cuadrado) aproximado de las partículas. ε: profundidad del potencial. α: decaimiento del potencial (Morse).

kt: constante de Boltzmann por la temperatura.

```
1:  for i = 0 to nsp – 1:
2:       for j = 0 to chpotit – 1:
3:             for k = 0 to ndim - 1:
4:                  rnew_k = uniformRNG(float, 0, 1)
5:             eng = 0
6:             for n = 0 to natoms – 1:
7:                  rd^2 = dist2(ndim, rnew, r_n, runit)
8:                  if rd^2 < rc^2:
9:                       eng += fpot(rd^2, keyp, σ, σ^2, ε, α)
10:            eng_j = e^(-eng/kt)
11:      chpot = (Σ_{k=0}^{chpotit-1} eng_k) / chpotit
```

El **algoritmo paralelo en *GPU* para el cálculo del potencial químico** (algoritmo 11) realiza para cada tipo de especie (*nsp*) (línea 1) una llamada al *kernel* de la *GPU* (línea 3) que devuelve la suma de los potenciales en exceso de todas las inserciones a realizar (*chpotit*).

Respecto al *kernel* de la *GPU* el algoritmo de Widom en paralelo se basa en aplicar las premisas vistas en el apartado 3.2.4 al problema dado. Concretamente, se divide el vector (alineado y transferido una única vez a la memoria de la *GPU*) de posiciones de partículas (*natoms* partículas) en un número de bloques igual a *nblocks* = (*chpotit* + (*NTHREAD* - 1)) / *NTHREAD*. Por lo tanto, se divide el problema en bloques de *NTHREADS* hilos cada bloque. Cada hilo dentro del bloque tendrá dos identificadores: *idx* le identifica a nivel global dentro del *grid* (*intento de inserción*) y *tx* le identifica a nivel local dentro del bloque - líneas 1 y 2 - . Cada bloque en ejecución de la *GPU* instancia, en memoria compartida, un vector de potenciales en exceso $\overrightarrow{eng}$ con *NTHREADS* posiciones. En dicho vector, si cada *intento de inserción* es válido (línea 4) cada partícula a insertar, representada por tres coordenadas aleatorias (líneas 5 y 6), almacenará su potencial en exceso (línea 11). Cabe señalar que en la versión serie de este algoritmo los número aleatorios para generar las coordenadas de la partícula a insertar se obtenían de la librería Intel MKL[8] (*Math Kernel Library*). Sin embargo, en la versión paralela para *GPU* se utiliza la librería *cuRAND*[23], de forma que cada hilo de cada bloque posee una semilla propia que asegura la independencia estadística de los números aleatorios obtenidos. Una vez que se han realizado todos los *intentos de inserción* y se han almacenado los potenciales en exceso en memoria compartida, todos los hilos del bloque en ejecución deben respetar una barrera de sincronización (línea 14). Esto es debido a que en las siguientes líneas se realizará una reducción binaria (figura 25, apartado 5.1) del vector instanciado en memoria compartida $\overrightarrow{eng}$ dentro del bloque. Finalmente, y dado que cada bloque produce una reducción parcial del vector completo, todos los bloques deben reducir sus resultados parciales mediante operaciones atómicas. Para realizar esta labor, el hilo cero de cada bloque realiza dicha operación atómica acumulando su resultado en una acumulador en memoria global (línea 22). Por lo tanto, en el algoritmo, se ha dividido el problema en sub-problemas (bloques), cada sub-problema en tareas (hilos dentro del bloque), los hilos dentro del bloque han cooperado (reducción

binaria), se ha minimizado la lectura/escritura en memoria global y se ha maximizado el uso de memoria compartida y registros.

Este algoritmo posee un complejidad computacional de:

$$O = nsp\frac{chpotit \cdot natoms}{blockDim \cdot nSM}log_2 blockDim$$

Dado que para este trabajo $chpotit = natoms$, finalmente se tiene que:

$$O = nsp\frac{natoms^2}{blockDim \cdot nSM}log_2 blockDim$$

Siendo *nSM* el número de procesadores de *streaming* de la *GPU* y *blockDim* = *NTHREADS* el número de hilos por bloque. En conclusión, el algoritmo paralelo en *GPU* para el cálculo del potencial químico escala cuadráticamente respecto al número de átomos.

Cabe señalar que *blockDim • nSM* es el número total de *cores CUDA* (*ncCUDA*) que posee la *GPU*, por lo tanto:

$$\lim_{ncCUDA \to natoms} O = nsp\frac{natoms^2}{ncCUDA}log_2 blockDim = nsp \cdot natoms \cdot log_2 blockDim$$

De modo que, cuantos más *cores CUDA* posea la *GPU* menor será la complejidad computacional del algoritmo.

Algoritmo 11
chpotential[gpu]: *cálculo del potencial químico[12] mediante la inserción (independiente) de un número dado de partículas en la configuración dada.* Requiere: ndim: número de dimensiones. natoms: número de átomos/partículas de la configuración dada. nsp: número de especies. chpotit: número de inserciones a realizar. $\vec{r}$: posiciones de las partículas de la configuración dada. $\overrightarrow{runit}$: valores de normalización de los lados de la caja de simulación. keyp: potencial seleccionado. rc^2: radio (al cuadrado) de corte en la interacción entre partículas. σ: mínimo del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño aproximado de las partículas. σ^2: mínimo (al cuadrado) del potencial Morse y el cero en el potencial Lennard-Jones. Tamaño (al cuadrado) aproximado de las partículas. ε: profundidad del potencial. α: decaimiento del potencial (Morse). kt: constante de Boltzmann por la temperatura.
1: for i = 0 to nsp – 1:

```
2:      chpot = 0
3:      chpot = kernel[gpu] {
                1:  idx = threadIdx + blockDim • blockIdx
                2:  tx = threadIdx
                3:  eng = 0
                4:  if idx < chpotit:
                5:      for k = 0 to ndim - 1:
                6:          rnew_k = uniformRNG(float, 0, 1)
                7:      for n = 0 to natoms – 1:
                8:          rd² = dist2(ndim, rnew, r_n, runit)
                9:          if rd² < rc²:
                10:             eng += fpot(rd², keyp, σ, σ², ε, α)
                11:     eng_tx = e^(−eng/kt)
                12: else:
                13:     eng_tx = 0
                14: syncthreads
                15: s = blockDim / 2
                16: while s > 0:
                17:     if tx < s:
                18:         eng_tx += eng_tx+s
                19:     s /= 2
                20:     syncthreads
                21: if tx == 0:
                22:     atomicAdd(chpot, eng_tx)
                23: return chpot
                }
4:      chpot /= chpotit
```

6. Implementación

En primer lugar, señalar que a continuación se exponen las decisiones de diseño, implementación y optimización de las partes del código de las que se desea estudiar la aceleración resultante de la paralelización. Como ya hemos mencionado, no se expondrán detalles relativos a la entrada/salida, estadística, inicialización, lógica de la aplicación, etc.

Las decisiones de diseño, implementación y optimización del código son las siguientes:

- El diseño del algoritmo está particularmente orientado a sistemas con potenciales de muy largo alcance, esto es con radios de corte altos. Por lo tanto, no se han implementado listas de vecinos para las partículas de la configuración dada. La paralelización del cálculo de energías será el elemento central de la implementación. A efectos de reducir el tiempo de cálculo de la versión serie, se ha impuesto un radio de corte de la interacción de 9*Å* en todos los casos estudiados. No obstante el escalado cuadrático del cálculo de distancias con *N* se mantiene.
- Como compilador base de *C/C++* se ha utilizado el compilador *icpc* (*Intel C/C++ Compiler*) tanto para el código *C/C++* (*.cpp*) como para el código *CUDA_C/C++* (*.cu*). Esta decisión se ha tomado dado que las simulaciones se han realizado en procesadores *Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz*. Asimismo, señalar que el compilador de Intel sobre procesadores de la misma marca es, aproximadamente, un 40% más rápido que el compilador *GNU gcc*. Finalmente, indicar que los ficheros de código son compilados con la opción de optimización *-fast*, la cual incluye las siguientes opciones: *ipo*, *-O3*, *-no-prec-div*, *-static*, *-fp-model fast=2* y *-xHost*.
- La dimensionalidad del problema, *NDIM*, se define mediante una *macro* dado que existe un alto número de bucles cuya condición de fin es dicho valor (se ejecutan *NDIM* veces). Además, la ejecución de dichos bucles crece cuadráticamente respecto al número de partículas - O($natoms^2$) - por lo que afectan gravemente al rendimiento de la aplicación. Por lo tanto, definiendo *NDIM* en tiempo de preprocesado (y no de ejecución) se consigue que el compilador "des-enrolle" (*unroll*) dichos bucles dado que, además, el valor de *NDIM* es bajo. Finalmente, señalar que la aceleración (*speed-up*) conseguida con esta decisión es aproximadamente de un factor cuatro.
- El producto escalar necesario para el cálculo de la distancia entre dos partículas se ha implementado sin utilizar librerías externas. En un principio se utilizó la función *cblas_ddot* de la librería *MKL*[8] de Intel. Sin embargo, durante el proceso de optimización se comprobó que el factor de aceleración obtenido al no utilizar la librería externa es de un factor dos, por lo que se desestimó su uso. Finalmente, señalar que la librería está optimizada para realizar productos escalares de vectores muy grandes, por lo que al ejecutar dichas operaciones con vectores del tamaño de *NDIM* (valores pequeños) el rendimiento se ve mermado.

- Respecto a la obtención de números aleatorios se han utilizado dos librerías:
    - En todos los algoritmos excepto en el cálculo en paralelo para *GPU* del potencial químico (algoritmo 11) se ha utilizado la librería *MKL*[8] de Intel para la generación de número aleatorios. Concretamente se ha utilizado la función *vsRngUniform* dado que genera números aleatorios $X \in [0, 1]$ en una distribución continua y uniforme.
    - En el cálculo en paralelo para *GPU* del potencial químico (algoritmo 11) se ha utilizado la librería *cuRAND*[23] de *CUDA* dado que para cada hilo (intento de inserción) se necesitan tres números aleatorios que representan las coordenadas de la partícula a insertar. Cabe señalar que, dentro de la *GPU*, cada hilo de cada bloque necesita generar su propia semilla para obtener números aleatorios estadísticamente independientes a los producidos por otros hilos. Finalmente, se ha utilizado la función *curand_uniform* dado que genera números aleatorios $X \in [0, 1]$ en una distribución continua y uniforme.
- Las posiciones de las partículas de la configuración $\vec{r}$ se han normalizado respecto a las longitudes (x, y, z) de la caja de simulación. Por un lado, se facilita la implementación de las condiciones periódicas de contorno. Por otro lado, las operaciones entre valores normalizados (en la misma magnitud) generan menores errores de representación.
- El radio de corte del potencial de interacción entre dos partículas siempre se utiliza al cuadrado y se compara respecto a la distancia euclidiana al cuadrado. De esta forma sólo es necesario ejecutar la raíz cuadrada (coste computacional alto) del cálculo de la distancia si las partículas se encuentran dentro del radio de corte y sólo para el potencial *Morse*. Además, utilizar el radio de corte, la distancia y el parámetro σ^2 todos al cuadrado permite calcular el potencial de *Lennard Jones* con menor coste computacional.
- Todas las variables que representan energías son normalizadas respecto del primer parámetro ε del potencial seleccionado; con objeto de generar menores errores de representación.
- En la *GPU* todos los valores en doble precisión se han convertido a precisión simple. Dado que los registros de la *GPU* son de 32 *bits*[9], utilizar precisión simple aporta una aceleración de entre el 30 y el 50% respecto a usar doble precisión.
- La *GPU* en la que se han realizado las simulaciones es una *NVIDIA GeForce GTX 1080 Ti* con *28 Streaming Multiprocessors, 128 CUDA Cores/SM: total 3584 CUDA Cores*, por lo que se ha establecido el valor de la macro *NTHREADS* a 128. Por lo tanto, con esta configuración, la *GPU* puede procesar simultáneamente 28 bloques de 128 hilos cada uno, asegurando una utilización óptima y eficiente de la *GPU*. Además, tal y como se demuestra en la siguiente tabla (figura 26), el número de hilos por bloque de la *GPU* no es un factor determinante para la optimización de problema dado.

9000 particles	32 threads (sec)	64 threads (sec)	128 threads (sec)	CPU (sec)
mAtoms	0,254321	0,254583	0,259153	1,743833
Speed-Up	6,86	6,85	6,73	1,00
mVolume	0,010834	0,010978	0,010598	0,627554
Speed-Up	57,93	57,17	59,21	1,00
18000 particles	32 threads (sec)	64 threads (sec)	128 threads (sec)	CPU (sec)
mAtoms	0,504437	0,508895	0,505808	6,957894
Speed-Up	13,79	13,67	13,76	1,00
mVolume	0,028253	0,028020	0,027836	2,499374
Speed-Up	88,46	89,20	89,79	1,00
36000 particles	32 threads (sec)	64 threads (sec)	128 threads (sec)	CPU (sec)
mAtoms	1,303637	1,304556	1,309842	22,006388
Speed-Up	16,88	16,87	16,80	1,00
mVolume	0,115345	0,104334	0,104179	8,199619
Speed-Up	71,09	78,59	78,71	1,00
72000 particles	32 threads (sec)	64 threads (sec)	128 threads (sec)	CPU (sec)
mAtoms	2,820667	3,195734	3,286373	77,374562
Speed-Up	27,43	24,21	23,54	1,00
mVolume	0,444516	0,421195	0,449225	29,397113
Speed-Up	66,13	69,79	65,44	1,00

Figura 26: tiempos (segundos) por paso y speed-up para diferentes número de hilos por bloque.

- Todas las funciones matemáticas utilizadas dentro de la *GPU* son del tipo *intrinsic*[10]. Dichas funciones se encuentran implementadas y son ejecutadas dentro de las *SFUs*[9] (*Special Purpose Units*) (figura 10). Por lo tanto, mediante el uso de las funciones *intrinsics* se obtiene una aceleración de entre el 30 y el 50% respecto de las funciones de la *STL* (*Standard Template Library*).
- La gestión de la memoria, y sus tipos, en la *GPU* es un factor determinante en el rendimiento de la aplicación por lo que se han tenido las siguientes consideraciones:
    - Respecto a la gestión de las transferencias de datos *CPU⇔GPU*, se han reducido al máximo posible y se han evitado transferencias de datos redundantes[11]. De este modo, el objetivo es evitar el cuello de botella en el ancho de banda que genera el *bus PCI express* en la jerarquía de memorias. Por un lado, sólo se instancia memoria en la *GPU* y se copia el contenido de todos los datos de la *CPU* a la *GPU* una única vez. Por otro lado, existen numerosos ejemplos como cuando se acepta el movimiento de una partícula en el algoritmo Metrópolis y se debe actualizar la posición de la partícula *ntest*, sólo se transfiere la posición de esta partícula a la *GPU*. Por lo tanto, se mantiene el vector de la configuración $\vec{r}$ dentro de la *GPU* entre iteraciones y sólo se transfieren tres valores (x, y, z) en precisión simple entre *CPU⇔GPU*. Otro ejemplo se produce en el cálculo de la energía de una configuración dada, cuando se desea reducir el vector de energías de interacción entre

partículas. Dicho vector se mantiene en la memoria de *GPU* entre las llamadas de los *kernels*, de modo que se evita transferir, desde la *CPU*, de nuevo información existente dentro de la memoria de la *GPU*.

- Además, se ha evitado al máximo el acceso a la memoria global desde los *kernels*[11] de la *GPU*, tanto para utilizar variables repetitivas como para almacenar resultados temporales o acumulados. Por ejemplo, se han almacenado las posiciones de las partículas en cada hilo dentro de registros (8000 *Tb/s*) y sólo se accede a memoria global (177 *Gb/s*) para guardar el resultado final (figura 11).
- Dentro de la *GPU* el acceso a memoria debe ser uniforme por lo que la memoria debe ser coalescente[11]. Además, las operaciones de lectura se producen en bloques de 128 *bits* por lo que una correcta alineación de los datos evita degradaciones de rendimiento en la aplicación. Por lo tanto, con objeto de evitar dichos problemas de rendimiento, todos los datos de la aplicación se han estructurado bajo el esquema *SoA* (*Structures of Arrays*) y se han alineado en vectores de una dimensión. Por ejemplo, el vector de la configuración $\vec{r}$ se podría haber representado como un vector de dos dimensiones (matriz). Sin embargo, se ha preferido mantener una estructura lineal coalescente para optimizar el rendimiento dentro de la *GPU*.
- Los tres *kernels GPU* implementados basan su estrategia en la cooperación de los hilos dentro del bloque en ejecución. Esto es debido a que todos los hilos de un bloque comparten una memoria de 48*k* denominada memoria compartida[9]. Dicha memoria tiene baja latencia y un ancho de banda de 1600 *Tb/s*, el cual es muy superior a los 177 *Gb/s* de la memoria global (figura 11). Por lo tanto, esta estrategia se basa en leer, una única vez o pocas veces, los datos a procesar de la memoria global a registros y/o a memoria compartida, realizar las operaciones necesarias acumulando o almacenando resultados en registros y/o en memoria compartida y, por último, almacenar el resultado final en memoria global. Finalmente, señalar que las operaciones a realizar por los hilos pueden ser independientes (se obtiene el beneficio del ancho de banda y la baja latencia) o colaborativas (a los beneficios anteriores se suma la reducción de operaciones de lectura).
- La configuración[9] de la memoria compartida y la memoria caché *L1* se puede ajustar[11] mediante las funciones *cudaDeviceSetCacheConfig()* y *cudaFuncSetCacheConfig()* para conseguir una optimización máxima. Esto es, de los *64KB* disponibles para ambas memorias se han probado las configuraciones siguientes: *48KB* para memoria compartida / *16KB* para memoria caché *L1* y *16KB* para memoria compartida / *48KB* para memoria caché *L1*. Los resultados (figura 27) demuestran que, para el problema dado, ambas configuraciones producen un valor similar (tiempo por paso en segundos). Por lo tanto, la configuración utilizada en las simulaciones será: *48KB* para memoria compartida / *16KB* para memoria caché *L1*.

9000 particles	48KB shared / 16KB L1 (s)	16KB shared / 48KB L1 (s)	CPU (s)
chPotential (10000)	0,015979	0,016005	2,113662
Speed-Up	132,27	132,07	1,00
18000 particles	48KB shared / 16KB L1 (s)	16KB shared / 48KB L1 (s)	CPU (s)
chPotential (10000)	0,030376	0,030385	3,040563
Speed-Up	100,10	100,07	1,00
9000 particles	48KB shared / 16KB L1 (s)	16KB shared / 48KB L1 (s)	CPU (s)
chPotential (100000)	0,147874	0,148491	22,804175
Speed-Up	154,21	153,57	1,00
18000 particles	48KB shared / 16KB L1 (s)	16KB shared / 48KB L1 (s)	CPU (s)
chPotential (100000)	0,288418	0,288096	33,386319
Speed-Up	115,76	115,89	1,00

Figura 27: tiempos (segundos) por paso y speed-up para diferentes configuraciones de memoria.

7. Resultados

7.1 Configuración de la simulación

Por un lado se expondrá la configuración de la simulación realizada para llevar a cabo este estudio:

- *NPT* y *NVT*, tanto en *CPU* como en *GPU*:
  - *N* → Número de partículas SiO_2: desde 9000 hasta 360000, en intervalos de 9000 partículas cada uno. Por lo tanto se tienen dos especies: *nsp* = 2.
  - La configuración inicial de partículas corresponde a un cristal cúbico centrado en las caras (*FCC*).
  - Temperatura: 4000*K*.
  - Densidad: 0.053679 *átomos* / $Å^3$.
  - Presión (sólo *NPT*): 166.054 *bar*.
  - El desplazamiento máximo será de 0.3*Å*, tanto para las coordenadas de la partículas (*NVT* y *NPT*) como para el cambio de volumen (*NPT*).
  - El potencial seleccionado será *Morse* con radio de corte 9*Å* y con los siguientes parámetros (figura 4):
    - O ⇔ O: ε = 0.023272*eV*, α = 1.3731$Å^{-1}$, σ = 3.791*Å*
    - O ⇔ Si: ε = 1.99597*eV*, α = 2.6518$Å^{-1}$, σ = 1.628*Å*
    - Si ⇔ Si: ε = 0.007695*eV*, α = 2.0446$Å^{-1}$, σ = 3.7598*Å*
- *NVT* con cálculo de potencial químico:
  - Número de inserciones por cada especie: desde 9000 hasta 360000, en intervalos de 9000 inserciones cada uno. Por lo tanto, se tendrá el mismo número de inserciones por especie que de partículas totales.
  - Todas las demás características igual que en el caso anterior (*NVT*).
- Los procesadores *CPU* utilizados son *Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz* con *GPU NVIDIA GeForce GTX 1080 Ti*.

Se define como paso de simulación *N* intentos de traslación *MC-NVT*, un intento de cambio de volumen *MC-NPT* y *nsp·N* intentos de inserción en la determinación del potencial químico.

7.2 Tiempo total de CPU/GPU por paso de simulación

Los resultados obtenidos estudiando el tiempo por paso de simulación han sido los siguientes:

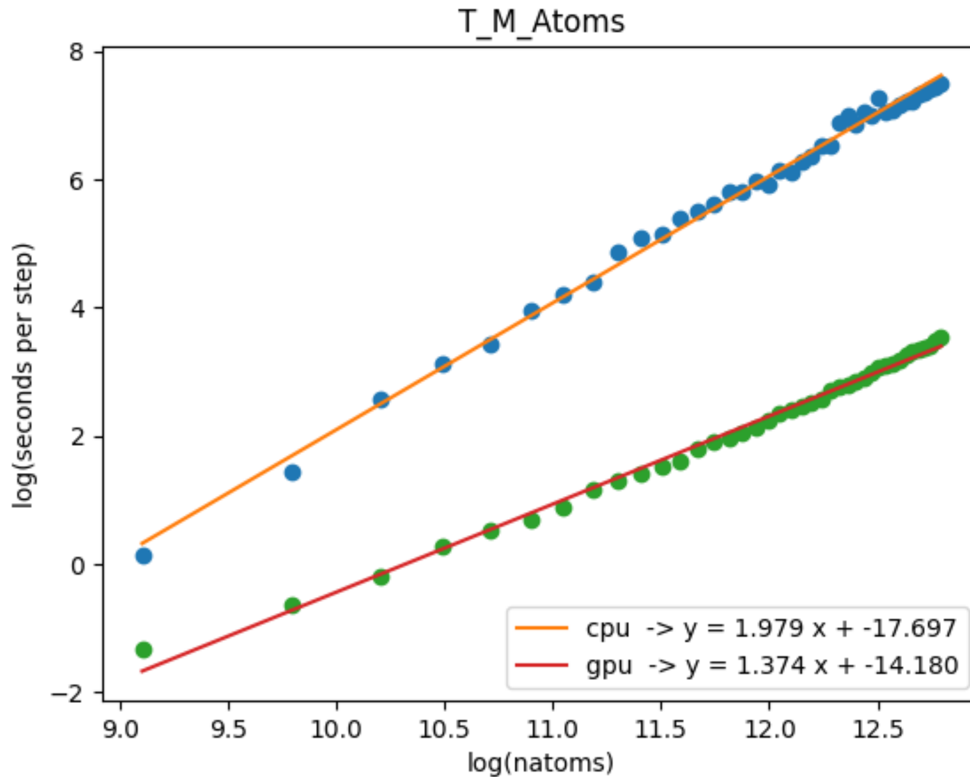


Figura 28: algoritmo moveAtoms : tiempo por paso respecto al tamaño del problema (log-log) y regresión lineal (CPU y GPU).

Respecto al algoritmo *moveAtoms*, se puede observar cómo la tasa de crecimiento del tiempo por paso en la versión *CPU* es mayor que en la versión *GPU*. Además, ambas versiones del algoritmo siguen una ley de potencias por lo que se ha procedido a realizar una regresión lineal con objeto de obtener el exponente de dicha distribución (figura 28). Tal y como se puede observar, la versión *CPU* del algoritmo posee un exponente de ~1,98 mientras que la versión *GPU* posee un exponente de ~1,37. La versión *GPU* aún no escala cuadráticamente con el tamaño del problema. Se aprecia, no obstante, que la curva *log-log* para *moveAtoms* en *GPU* es cóncava y la pendiente va a aumentando; por lo que cabe esperar que el rendimiento (speed-up) de la *GPU* mejore aún para tamaño mayores. Mientras que el tiempo crece cuadráticamente con *N* para la *CPU*, como era de esperar, en la *GPU* estamos aún cerca del régimen lineal. El algoritmo *NVT* para *GPU* tiene una componente serial (*N* intentos de traslación consecutivos) y una componente paralelizada que óptimamente alcanzan el límite $(N-1) / N_{cores}$; de ahí que no se aprecie aún el régimen cuadrático.

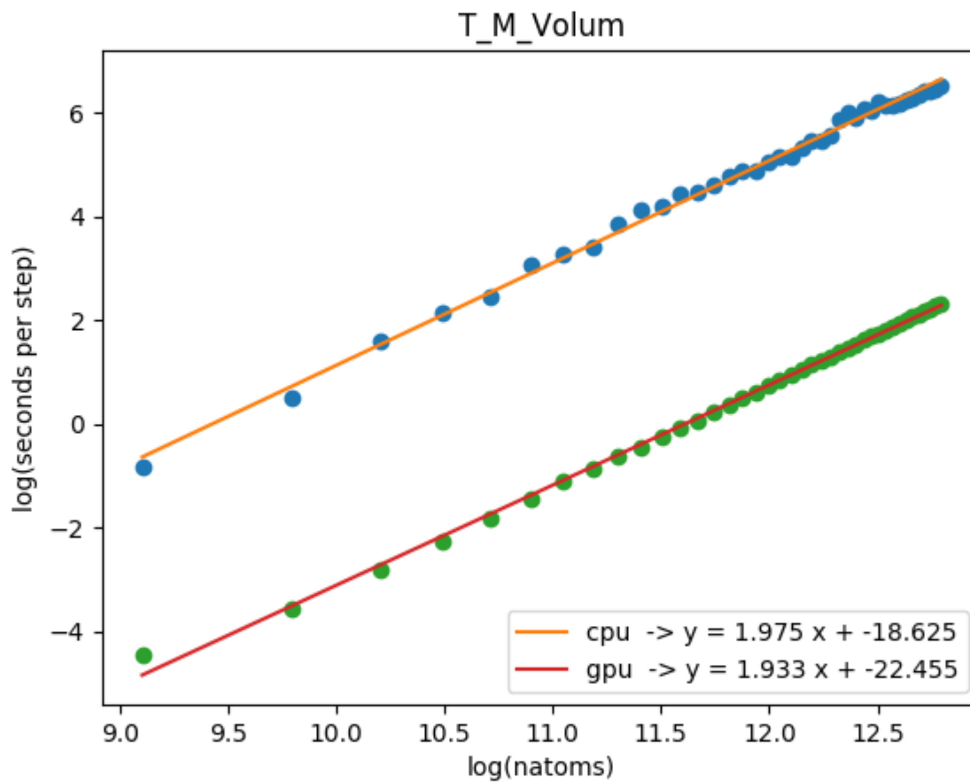


Figura 29: algoritmo moveVolum : tiempo por paso respecto al tamaño del problema (log-log) y regresión lineal (CPU y GPU).

Respecto al algoritmo *moveVolum*, al igual que en el caso anterior, se puede observar cómo la tasa de crecimiento del tiempo por paso en la versión *CPU* es mayor que en la versión *GPU*. Además, ambas versiones del algoritmo siguen una ley de potencias por lo que se ha procedido a realizar una regresión lineal con objeto de obtener el exponente de dicha distribución (figura 29). Tal y como se puede observar, la versión *CPU* del algoritmo posee un exponente de ~1,98 mientras que la versión *GPU* posee un exponente de ~1,93. Por lo tanto, ambas versiones crecen a un ritmo muy similar (rectas casi paralelas), siendo la versión *GPU* mucho más rápida que la versión *CPU*. Se puede concluir que ambas versiones muestran la dependencia cuadrática con el tamaño de la muestra, característica de un algoritmo que implica la determinación de $O(N^2)$ interacciones.

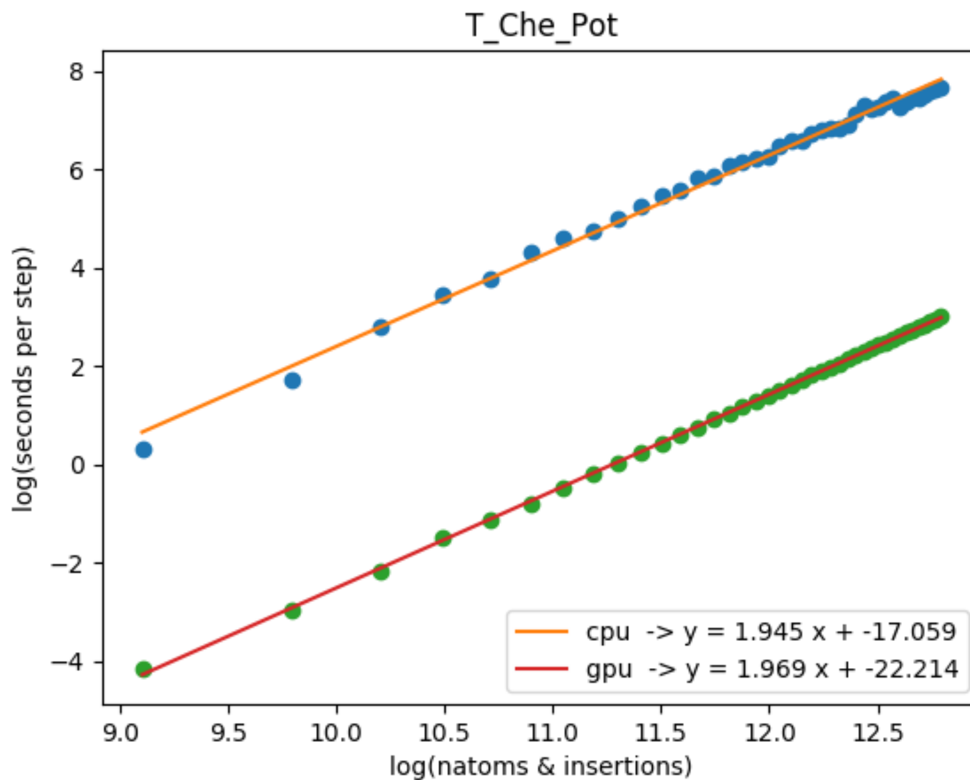


Figura 30: algoritmo chPotential : tiempo por paso respecto al tamaño del problema (log-log) y regresión lineal (CPU y GPU).

Respecto al algoritmo *chPotential*, se puede observar cómo la tasa de crecimiento del tiempo por paso en la versión *CPU* es mayor que en la versión *GPU*. Además, ambas versiones del algoritmo siguen una ley de potencias por lo que se ha procedido a realizar una regresión lineal con objeto de obtener el exponente de dicha distribución (figura 30). Tal y como se puede observar, la versión *CPU* del algoritmo posee un exponente de ~1,94 mientras que la versión *GPU* posee un exponente de ~1,97. Por lo tanto, ambas versiones crecen a un ritmo muy similar (rectas casi paralelas), siendo la versión *GPU* mucho más rápida que la versión *CPU*. Finalmente, esto significa que *chPotential*, con N intentos de inserción por paso corresponde a un algoritmo $O(N^2)$.

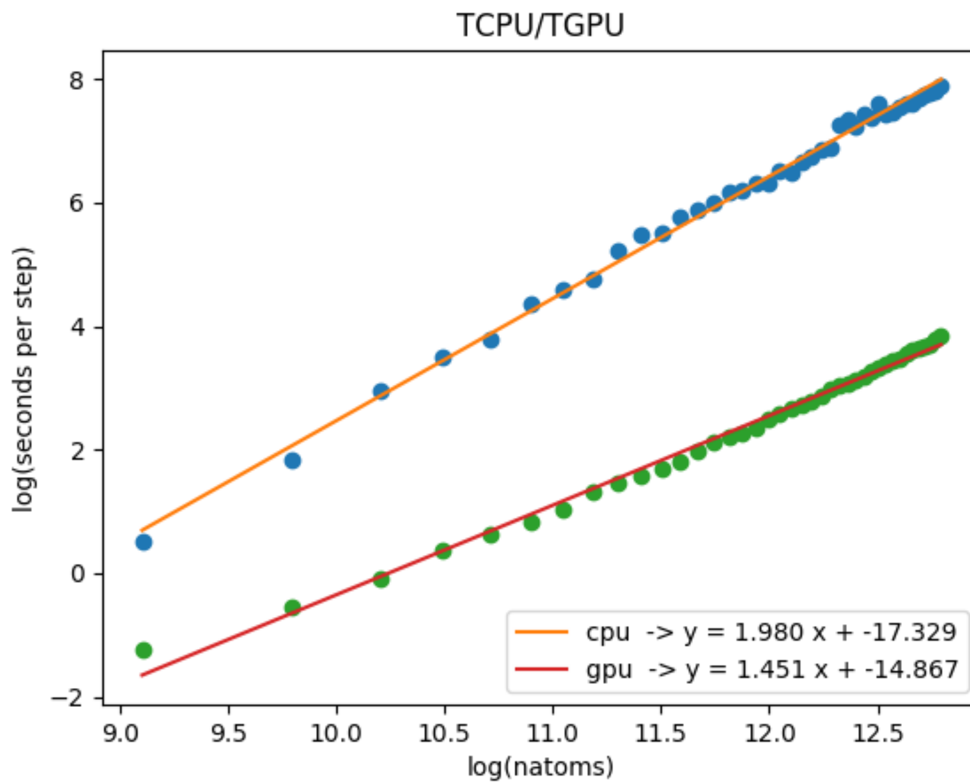


Figura 31: aplicación : tiempo por paso respecto al tamaño del problema (log-log) y regresión lineal (CPU y GPU).

Respecto al tiempo por paso de la aplicación como un todo, se puede observar cómo la tasa de crecimiento del tiempo por paso en la versión *CPU* es mayor que en la versión *GPU*. Además, ambas versiones del algoritmo siguen una ley de potencias por lo que se ha procedido a realizar una regresión lineal con objeto de obtener el exponente de dicha distribución (figura 31). Tal y como se puede observar, la versión *CPU* del algoritmo posee un exponente de ~1,98 mientras que la versión *GPU* posee un exponente de ~1,45. Por lo tanto, la versión *GPU* de la aplicación como un todo, además de utilizar menos tiempo por paso, escala con *N* a menor velocidad que la versión *CPU* del mismo. Esto significa que la aplicación no ha alcanzado el tamaño del problema máximo que consiga el rendimiento óptimo.

7.3 Speed-up

Con objeto de completar el estudio de optimización se debe conocer el factor de aceleración (*speed-up*) entre ambas versiones de algoritmos (*CPU vs GPU*), según se aumenta el tamaño del problema. Por lo tanto, se definirá el *speed-up* como:

$$\lambda = \frac{t_{serie}}{t_{gpu}}$$

Donde t_{serie} y t_{gpu} corresponden al tiempo por paso (*N* intentos de traslación, un cambio de volumen o *N* intentos de inserción) en *CPU* y en *GPU*, respectivamente.

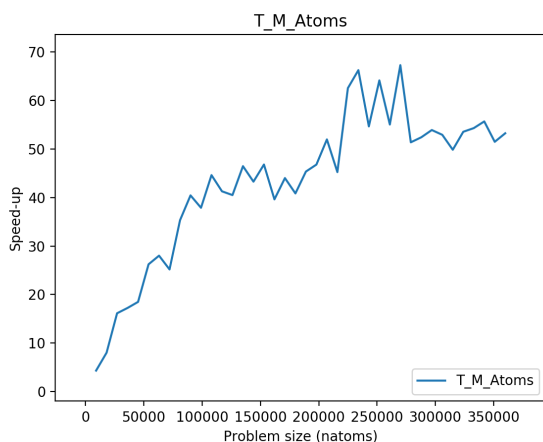


Figura 32: moveAtoms: escalado del speed-up

Estudiando el comportamiento de λ para el algoritmo *moveAtoms* se puede comprobar que a mayor tamaño del problema (*natoms*) mayor aceleración se obtiene (figura 32). Sin embargo, también se puede apreciar un comportamiento asintótico general que indica una próxima saturación en la aceleración. Finalmente, cabe señalar que el máximo de aceleración obtenida ha sido de un factor ~67.

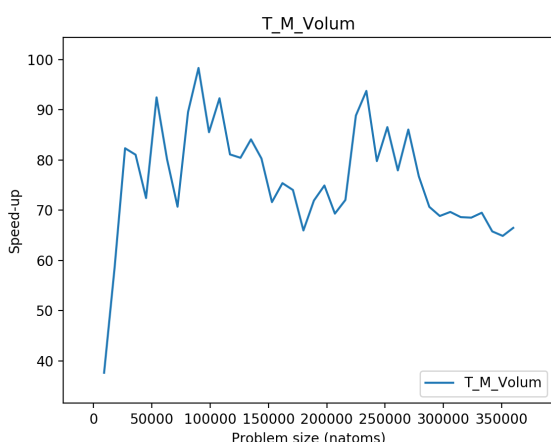


Figura 33: moveVolum: escalado del speed-up

Estudiando el comportamiento de λ para el algoritmo *moveVolum* se puede comprobar que se obtiene una aceleración muy alta ya con tamaños pequeños del problema (figura 33). Sin embargo, y tras una saturación en la optimización, se puede comprobar cómo la aceleración decrece lentamente según se aumenta el número de partículas de la configuración. Finalmente, cabe señalar que el máximo de aceleración obtenida ha sido de un factor ~98. Esta eficiencia tan alta para tamaños pequeños es debida a la dependencia cuadrática – $N (N – 1) / 2$ - del cálculo de energías, para una configuración que se paraleliza de forma más inmediata que los *N* intentos (serializados) de traslaciones y en la que la parte paralelizada es únicamente el cálculo de la interacción de una partícula con las *N-1* restantes.

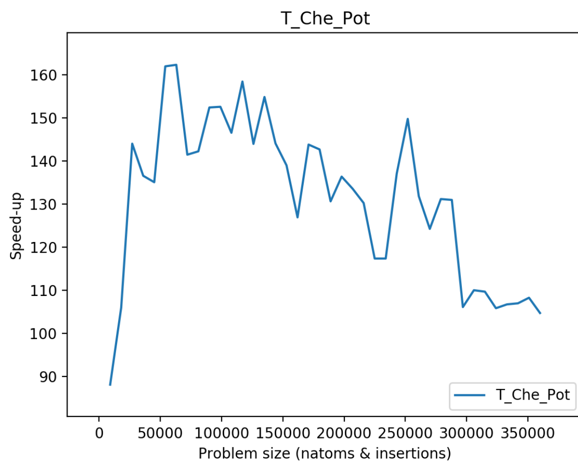


Figura 34: chPotential: escalado del speed-up

Estudiando el comportamiento de λ para el algoritmo *chPotential* se puede comprobar que, como en el caso anterior, se obtiene una aceleración muy alta con tamaños pequeños del problema (figura 34). Sin embargo, y tras una saturación en la optimización, se puede comprobar cómo la aceleración decrece incluso más rápido que en el caso anterior (*moveVolum*). Finalmente, cabe señalar que el máximo de aceleración obtenida ha sido de un factor ~162. Este algoritmo y el anterior (*moveVolum*) son ejemplos de problemas *embarrassing parallel* con una naturaleza completamente paralela. Es necesario realizar un estudio más en profundidad de los accesos a memoria, usos de registros y accesos al caché para poder dar una explicación de las fuertes oscilaciones y caídas de rendimiento que aparecen para tamaños >10^5 partículas.

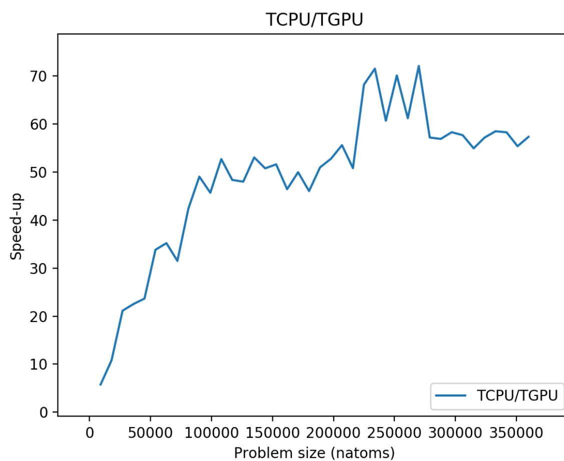


Figura 35: aplicación: escalado del speed-up

Estudiando el comportamiento de λ para el tiempo total de la aplicación se puede comprobar que el factor de aceleración limitante (figura 35) es, obviamente, el menor de los factores de cada uno de los algoritmos que forman dicha aplicación, esto es *moveAtoms* (figura 32). A mayor tamaño del problema (*natoms*) mayor aceleración se obtiene, sin embargo también se puede apreciar un comportamiento asintótico general que indica una próxima saturación en la aceleración. Finalmente, cabe señalar que se obtiene un *speed-up* significativo (~20x) a partir de ~30000 partículas y que el máximo de aceleración obtenida ha sido de un factor ~72.

En el repositorio de código de la implementación se encuentra el directorio *sim/benchmarks* que contiene todas las simulaciones y los scripts *Python* para producir las gráficas y resultados presentados en este estudio.

8. Conclusiones

Las conclusiones obtenidas tras la realización del presente trabajo son las siguientes:

- Todos los algoritmos presentan una tasa de crecimiento del tiempo de ejecución por paso mayor en su versión serie que en su versión paralela. Por lo tanto, se demuestra que las versiones paralelas de dichos algoritmos son más eficientes que las versiones serie.
- Para la muestra utilizada, se ha encontrado el máximo de rendimiento paralelo para los algoritmos *moveVolume* y *chPotential*; pudiéndose concluir que ambas versiones han alcanzado la dependencia cuadrática con el tamaño de la muestra. Sin embargo, para el algoritmo *moveAtoms* no se ha alcanzado el máximo de rendimiento. Por lo tanto, la tasa de crecimiento del tiempo por paso de ambas versiones, serie y paralela, de *moveVolume* y *chPotential* crecen al mismo ritmo; mientras que para *moveAtoms* la versión paralela crece a menor ritmo que la versión serie. Con seguridad, este último algoritmo, alcanzará el régimen cuadrático aumentando el tamaño del problema.
- El comportamiento de la aceleración o *speed-up* (*CPU vs GPU*) de los algoritmos *moveVolume* y *chPotential* responde a los denominados problemas *embarrassing parallel*. Por lo tanto, se obtienen aceleraciones muy altas desde el principio de la muestra, lo que conlleva una pronta saturación de la aceleración. Sin embargo, dicho comportamiento para el algoritmo *moveAtoms* difiere ya que presenta una aceleración baja que aumenta progresivamente con el tamaño del problema (para la muestra utilizada). Finalmente, para *moveVolume* y *chPotential* se espera un comportamiento asintótico hacia el valor de saturación de la aceleración, formando una curva decreciente cóncava; mientras que para *moveAtoms* se espera el mismo comportamiento asintótico hacia el valor de saturación de la aceleración, pero formando una curva creciente convexa.
- Finalmente, el comportamiento de la aplicación, serie y paralela, respecto a la tasa de crecimiento del tiempo por paso y a su aceleración corresponde a su factor limitante, esto es al algoritmo que obtenga los valores más bajos. Por lo tanto, para la muestra utilizada, la aplicación se comporta como el algoritmo *moveAtoms* dado que, para este estudio, es el que ha obtenido valores más bajos. Sin embargo, se espera que la aceleración de dicho algoritmo (curva decreciente convexa) aumente y se cruce con la de los demás algoritmos (curva creciente cóncava); llegando a un punto de intersección que será el factor de aceleración máxima de la aplicación.

En conclusión, hemos demostrado que explotando las peculiaridades de diseño de las *GPUs* es posible paralelizar con bastante eficiencia incluso algoritmos básicamente seriales como *MC-NVT*, con buenos rendimientos para muestras muy grandes. Otros algoritmos de tipo $O(N^2)$ como el *MC-NPT* consiguen eficiencias comparables a las obtenidas en Dinámica Molecular (*LAMMPS*[24]) para tamaños relativamente pequeños.

8.1 Perspectivas de desarrollo futuro

Es evidente que los algoritmos presentados en este trabajo distan de ser óptimos. Hemos mencionado que tal y como se ha formulado la implementación del cálculo de energías, la solución presentada es adecuada únicamente para interacciones de muy largo alcance (y que no incluyan interacciones entre partículas cargadas, que precisan de un tratamiento especial, p.e. mediante sumas de Ewald, que aquí no hemos considerado). En los problemas habituales las interacciones son de medio y corto alcance. Estas interacciones por una parte se pueden tratar mediante las implementaciones habituales de paralelismo, mediante descomposición por dominios, pero especialmente, incluso en los algoritmos en serie se puede aprovechar la localidad de las interacciones para reducir el escalado de $O(N^2)$ a $O(N)$, mediante las técnicas de listas de vecinos y/o listas de celdas (*cell list*)[17]. La idea detrás de esta última técnica se esboza en la figura 36, la caja básica de simulación se subdivide en celdas de lado igual al radio de corte de la interacción, r_c, en una situación bidimensional, las interacciones de la celda central sólo necesitan tener presente las 9 celdas sombreadas (27 celdas en *3D*). De esta forma el algoritmo para determinar la energía total (o las fuerzas) en una dinámica molecular escala linealmente con el número de partículas totales, sólo las inmediatamente circundantes a una partícula dada deben ser tenidas en cuenta. Este algoritmo ha sido generalizado de forma eficiente para *GPUs* por Howard y colaboradores[18], así como en el ejemplo *Particles,* incluido en la implementación de *CUDA*[19]. En un futuro es nuestra intención implementar en los algoritmos anteriormente presentados la estrategia implementada en por Howard y colaboradores.

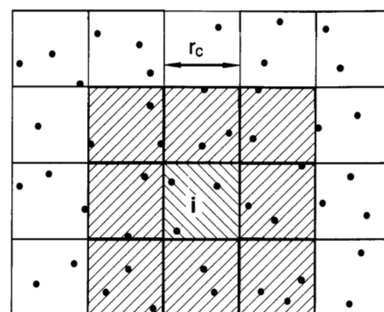


Figura 36: división de la caja de simulación en celdas de lado r_c. Las partículas de la caja i, sólo interaccionan con las de las de las 9 cajas sombreadas.

Finalmente, el otro desafío que se plantea es la posibilidad de implementar una descomposición por dominios, a la manera habitual en Dinámica Molecular. Este tipo de descomposición no resulta muy adecuada para métodos de Monte Carlo dado el carácter fundamentalmente serial de la cadena de Markov. No obstante, Anderson y colaboradores[13] han diseñado una estrategia de división en *tablero de damas* (*checkerboard*) como la que se ilustra en la Figura 37. Se observa que las partículas pertenecientes a cajas sombreadas (el lado de la caja es de nuevo r_c) no interaccionan, por lo que cualquier intento de traslación o inserción/borrado de partículas dentro de esas cajas puede realizarse en paralelo. Este algoritmo no satisface el balance detallado, pero si el balance global[20], por lo que la convergencia hacia la distribución de energías del colectivo canónico está garantizada.

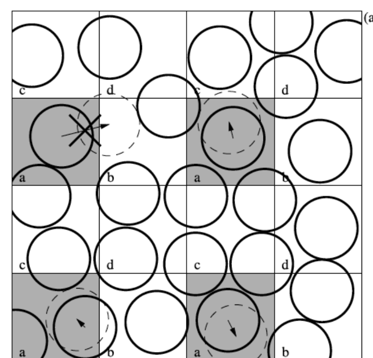


Figura 37: división de la caja de simulación en tablero de damas. Las partículas diferentes cajas sombreadas no interactúan, por lo que intentos de traslación o inserción/borrado pueden ser paralelos.

Es claro, que un algoritmo como el de Anderson puede mejorar significativamente el escalado del algoritmo *NVT* presentado en este trabajo, de manera especialmente visible para números de partículas < 100000. Es muy posible que mejore los resultados hasta en un orden de magnitud. El efecto sobre el *NPT* y el cálculo del potencial químico, dada la estructura mucho más paralela del algoritmo implementado aquí, será previsiblemente mejor. Es también muy probable que su implementación de buenos resultados para simulaciones en el colectivo Gran Canónico (a potencial químico, μ, fijo) donde se realizan inserciones/destrucciones de partículas de forma secuencial.

Por lo tanto, estas son las líneas de evolución previsible del trabajo presentado a corto plazo.

9. Bibliografía

[1] W. Gropp, E. Lusk y A. Skjellum, *Using MPI: Portable, Parallel Programming with the Message Passing Interface*, MIT Press, 1994.

[2] S. Plimpton, *Fast Parallel Algorithms for Short-Range Molecular Dynamics*, J Comp Phys, 117, 1-19 (1995); M.J. Abraham, D. van der Spoel, E. Lindahl, B. Hess, and the GROMACS development team, GROMACS User Manual version 5.1.2, www.gromacs.org (2016).

[3] A.J. Stone, *The theory of intermolecular forces*, Clarendon Press, 1997.

[4] K.P.N. Murhty, *An introduction to Monte Carlo simulations in Statistical Physics*, arXiv:cond-mat/0104167 [cond-mat.stat-mech] (2003).

[5] D.C. Rapaport, *The Art of Molecular Dynamics Simulation*, Cambridge University Press, 2011.

[6] N. Metropolis and S. Ulam, *The Monte Carlo method*, J. Amer. Statistical Assoc. 44, 335 (1949); N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller, *Equation of state calculation by fast computing machine*, J. Chem. Phys. 21, 1087 (1953).

[7] M.P. Allen y D.J. Tildesley, *Computer Simulation of Liquids*, Oxford University Press, 1987.

[8] https://software.intel.com/en-us/mkl-developer-reference-c (2018).

[9] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (2018).

[10] https://docs.nvidia.com/cuda/cuda-math-api/index.html (2018).

[11] https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html (2018).

[12] Kevin B. Daly, Jay B. Benziger, Pablo G. Debenedetti y Athanassios Z. Panagiotopoulos, *Massively parallel chemical potential calculation on graphics processing units*, Computer Physics Communications, 2012.

[13] J.A. Anderson, E. Jankowski, T.L Grubb, M. Engel, y S.C. Glotzer, *Massively parallel Monte Carlo for many-particle simulations on GPUs* , *Journal of Computational Physics,* 2013, 254, 27 - 38 .

[14] B. Widom, J. Chem. Phys., 39, 2808 (1963).

[15] C.H. Bennet, J. Comp. Phys., 22, 245 (1976).

[16] De I, Cburnett, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2233539.

[17] D. Frenkel y B. Smit, *Understanding Molecular Simulation*, Academic Press, Londres, 2002.

[18] M.P. Howard, J.A. Anderson, A. Nikoubashman, S.C. Glotzer, S. y A.Z. Panagiotopoulos, *Efficient neighbor list calculation for molecular simulation of colloidal systems using graphics processing units Comp. Phys. Comm.,* 2016, 203, 45 – 52.

[19] S. Green, *Particle Simulation using CUDA*, NVIDIA, 2013.

[20] R, Ren, R. y G. Orkoulas, *Parallel Markov chain Monte Carlo simulations*, *J. Chem. Phys.,* 2007,126, 211102.

[21] http://www.ks.uiuc.edu/Research/vmd/ (2018).

[22] W. Smith y J.T. Todorov, *A short description of DL_POLY*, Molecular Simulation, 32, 935 (2006).

[23] https://docs.nvidia.com/cuda/curand/index.html (2018)

[24] S. Plimpton, *Fast Parallel Algorithms for Short-Range Molecular Dynamics*, J Comp Phys, 117, 1-19 (1995); http://lammps.sandia.gov/

10. Anexos

A.1 Repositorio del proyecto

A continuación se detalla el contenido del repositorio del proyecto https://github.com/adpozuelo/MC:

Fichero/Directorio	Descripción/Contenido
bin/	Directorio que contendrá el fichero binario resultado de la compilación y enlazado (*link*).
Makefile	Fichero utilizado por *make* para compilar y enlazar el programa en el fichero binario *bin/mc.exe*
data/	Directorio que contiene los fichero de entrada del programa y *DPOLY* para realizar simulaciones (tanto *NPT* como *NVT*). Se entregan ejemplos.
doc/	Directorio que contiene la documentación del proyecto (memoria y presentación). La documentación del código se encuentra en los ficheros de cabecera y de código fuente del programa.
include/	Directorio con las cabeceras (*headers*) del programa. Cada cabecera contiene la documentación de las funciones que define.
include/energy.h	Fichero cabecera de las funciones relacionadas con la energía potencial (inicialización, cálculo de la energía potencial y calculo del potencial químico).
include/io.h	Fichero cabecera de las funciones relacionadas con la entrada/salida del programa (lectura de ficheros de entrada y salida, inicialización de los mismos y salida a pantalla de la información).
include/moves.h	Fichero cabecera de las funciones relacionadas con la generación de traslaciones: el algoritmo Metrópolis (*moveAtoms*) y el cambio en el volumen *NPT* (*moveVolume*).
include/thermo.h	Fichero cabecera de las funciones relacionadas con la estadística (histogramas, promedios, etc.).
include/util.h	Fichero cabecera de las funciones de utilidad general tales como el cálculo de la distancia al cuadrado entre dos partículas, la generación de un cristal centrado en las caras (*FCC*) o el identificador del tipo de interacción de cada partícula.
include/gpu.h	Fichero cabecera de la función *GPU* implementada. Dicha función tiene varios modos de ejecución tal y como se documenta en este fichero.
obj/	Directorio que contendrá los ficheros objetos producidos en la compilación.
results/	Directorio en donde se escribe la salida del programa mediante los siguientes ficheros: • *thermoins.dat*: fichero con la estadística y datos de salida instantánea de cada paso: número de

	movimientos ejecutados, porcentaje de acierto tanto de translaciones de partículas (*NVT*) como de volumen (*NPT*), energía total y acumulada y, sólo para *NPT*, dimensiones y volumen de la caja de simulación en cada paso. • *thermoaver.dat*: fichero que contiene los mismos datos que el fichero *thermoins.dat* pero no incluye la fase de equilibrado del sistema. • *conf.xyz*: fichero que contiene las posiciones de las partículas que componen la configuración. El fichero se puede leer con el programa *VMD*[21]. • *chpotential.dat*: fichero con los potenciales químicos calculados. • *ehisto.dat*: fichero que contiene el histograma de energías generado en la simulación *NVT*. • *rho_histo.dat*: fichero que contiene el histograma de densidad generado en la simulación *NPT*.
sim/	Directorio que contiene todas las simulaciones realizadas para llevar a cabo este estudio: • *benchmarks*: directorio con los resultados del estudio de aceleración y escalado de la misma respecto al tamaño del problema. Contiene el *script study.py* que genera las gráficas y los datos expuestos en esta memoria. • *convergence*: directorio con los resultados del estudio de validez del modelo realizado. Contiene los scripts *convergence.py* y *histograms.py* que generan las gráficas expuestas en esta memoria. • *test_cache_l1*: directorio con los resultados del estudio del uso de distintas distribuciones en el tamaño de las memorias caché *L1* y compartida dentro de la *GPU*. • *test_nthreads*: directorio con los resultados del estudio del uso de bloques con distinto número de hilos dentro de la *GPU*.
src/	Directorio con los ficheros de código fuente del programa. Cada fichero de código se encuentra documentado explicando qué hace cada línea considerada relevante.
src/energy.cpp	Fichero de código con las funciones relacionadas con la energía potencial (inicialización, cálculo de la energía potencial y calculo del potencial químico).
src/input.cpp	Fichero de código con las funciones relacionadas con la entrada del programa (lectura de ficheros de entrada.
src/output.cpp	Fichero de código con las funciones relacionadas con la salida del programa (escritura de los ficheros de

	salida, inicialización de los mismos y salida a pantalla de la información).
src/moves.cpp	Fichero de código con las funciones relacionadas con la generación de traslaciones: el algoritmo Metrópolis (*moveAtoms*) y el cambio en el volumen *NPT* (*moveVolume*).
src/thermo.cpp	Fichero de código con las funciones relacionadas con la estadística (histogramas, promedios, etc.).
src/util.cpp	Fichero de código con las funciones de utilidad general tales como el cálculo de la distancia al cuadrado entre dos partículas, la generación de un cristal centrado en las caras (*FCC*) o el identificador del tipo de interacción de cada partícula.
src/gpu.cu	Fichero de código con la función *GPU* implementada. Dicha función tiene varios modos de ejecución tal y como se documenta en este fichero. Este fichero contiene los *kernels* y las funciones __*device*__ que se ejecutan dentro de la *GPU*.
src/main.cpp	Fichero código principal de la aplicación que gestiona la entrada de argumentos y procesa la lógica y ejecución del programa.
LICENSE	Licencia *GPLv3* que ampara a la aplicación.
README.md	Fichero con los requisitos y las instrucciones de descarga e instalación de la aplicación.

Agradecimientos:

- A mi familia por su paciencia y apoyo.
- Al Profesor de Investigación D. Enrique Lomba García por su apoyo, ayuda y dirección del presente trabajo.
- A la Dra. Eva González Noya por su apoyo, ayuda y orientación.
- Al departamento de Mecánica Estadística y Material Condensada del Instituto de Química Física Rocasolano (CSIC) por permitirme utilizar el sistema *HPC* (*High Performace Computing*) Ladon-Hidra para realizar las pruebas y simulaciones.