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

Máster en Ingeniería Computacional y Matemática

Trabajo Final de Máster

Antonio Díaz Pozuelo

Julio de 2018

Contexto y justificación del Trabajo

- Explotar las **enormes capacidades computacionales** disponibles en las ***GPUs*** modernas en el campo de la Simulación Atomística, en concreto en lo que concierne a los métodos de Monte Carlo.
- El algoritmo de **Metrópolis** canónico es **esencialmente serial** (paralelización no es obvia).
- La estrategia de la paralelización se ha centrado en **optimizar el cálculo de las energías**.
- Objetivo principal:
  - Estudio de la aceleración (*speed-up*) *CPU vs GPU*.
  - Estudio de cómo escala la aceleración respecto al tamaño del problema.
- Repositorio: https://github.com/adpozuelo/MC (*GPLv3*)

Fundamentos físicos

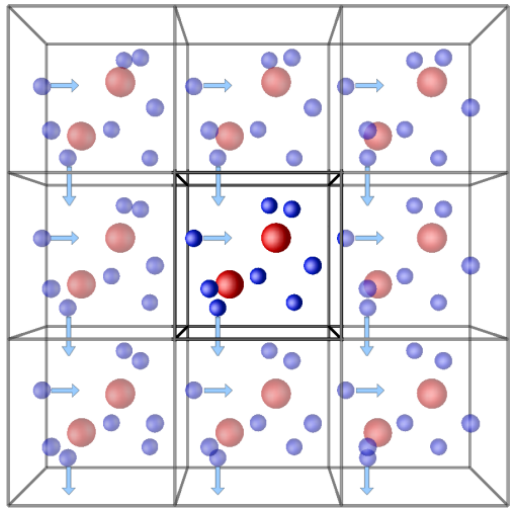

Figura 1: ilustración de las condiciones de contorno periódicas.

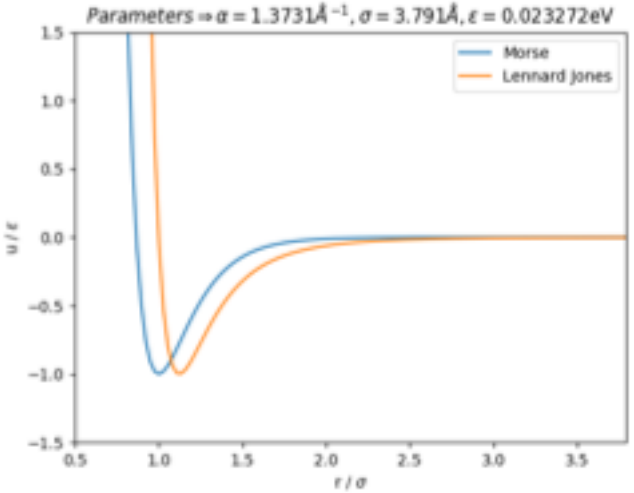

Figura 2: potenciales de Morse y Lennard Jones con los parámetros utilizados en este estudio.

- *Simulación Atomística:* determinación de **propiedades macroscópicas** de la materia **a partir** de nuestro conocimiento de las **propiedades de sus constituyentes** (configuración de partículas).
  - Interacciones[1] (*Lennard-Jones*, *Morse)*.
  - Aplicación de las denominadas "condiciones periódicas de contorno" (*PBC* o *Periodic Boundary Conditions*).
- Propiedades que vamos a estudiar:
  - **Energía interna** (configuracional): $U_N = \frac{1}{N_c} \sum_{\{r_i\}} u(r_1, \ldots, r_N)$
  - **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.
    - $\mu^{ex} = \mu - \mu^{id} = -k_B T \, log\langle e^{-U_{N+1}/k_B T} \rangle$
    - *Test-particle insertion* (*TPI*)[3]: $U_{N+1}(\boldsymbol{r}_{N+1})$ = energía de interacción de una partícula que se intenta insertar en la posición $\boldsymbol{r}_{N+1}$
- **Colectivo**: conjunto de microestados del sistema sometido a las restricciones macroscópicas que operan sobre el sistema[2]. **Consideraremos**:
  - Canónico (***NVT***): número de partículas, volumen y temperatura dados,
  - Isotérmico-isobárico (***NPT***): número de partículas , presión y temperatura dados.

El algoritmo de Metrópolis[2,4]

- Generación de una secuencia de configuraciones, siguiendo distribución de *Boltzmann*, mediante una cadena de *Markov.*

NVT:

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 $\Delta U_N = U_N(\boldsymbol{R}') - U_N(\boldsymbol{R})$.
    1. Si $\Delta U_N \leq 0$ se acepta el movimiento
    2. Si $\Delta U_N > 0$, se genera un nuevo número aleatorio $\eta \in [0,1]$. Si $exp[-\Delta U_N / k_BT] > \eta$ 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 %.

NPT:

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 $\Delta H = U(V') - U(V) + P(V - V') - NkBT\ log(V' / V)$, siendo *H* la **entalpía**. Eligiendo un nuevo número aleatorio $\xi' \in [0,1]$, si $exp[-\Delta H / k_BT] > \xi'$ 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 %.

Arquitectura computacional

Figura 3: número de transistores para procesar datos en CPU y GPU[5].

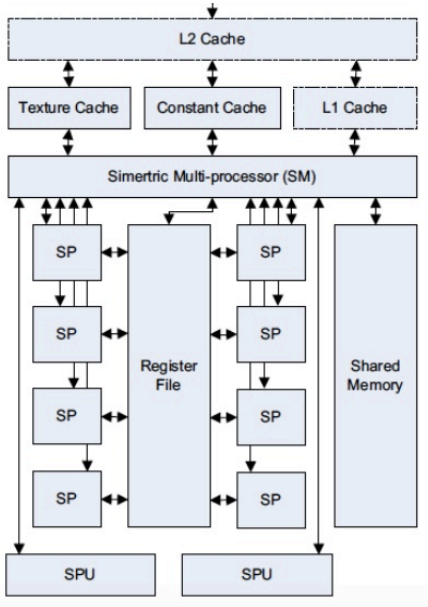

Figura 4: multiprocesador de streaming.

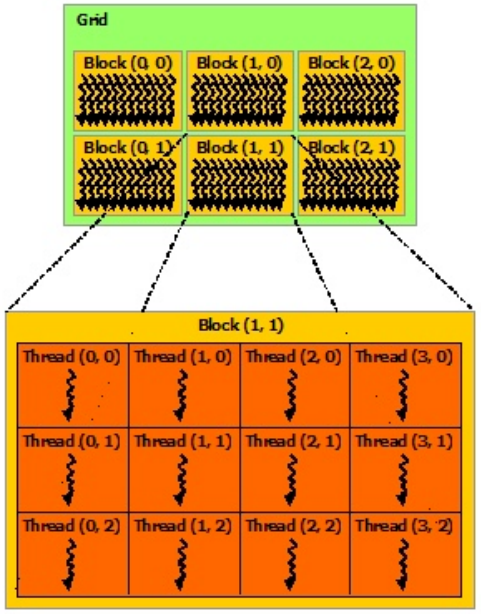

Figura 5: grid, bloques e hilos en el modelo CUDA[5].

- *GPU* diseñada para tareas de **computo intensivo** y **altamente paralelizable**.
- SIMT/SIMD: el mismo programa/algoritmo es ejecutado en paralelo sobre distintos conjuntos de datos.
- *CUDA* → **tres abstracciones base**: la jerarquía de memorias, de agrupación de hilos y las barreras de sincronización.
- Un **hilo** puede ser identificado utilizando una, dos o tres dimensiones → **bloques** de hilos de una, dos y tres dimensiones.
- Los bloques de hilos son organizados en **mallas** (*grids*) de una, dos o tres dimensiones.
- Cada bloque de hilos puede ser programado en cualquiera de los **multiprocesadores de *streaming*** disponibles en la *GPU* en cualquier orden y de forma concurrente o secuencialmente.

CUDA

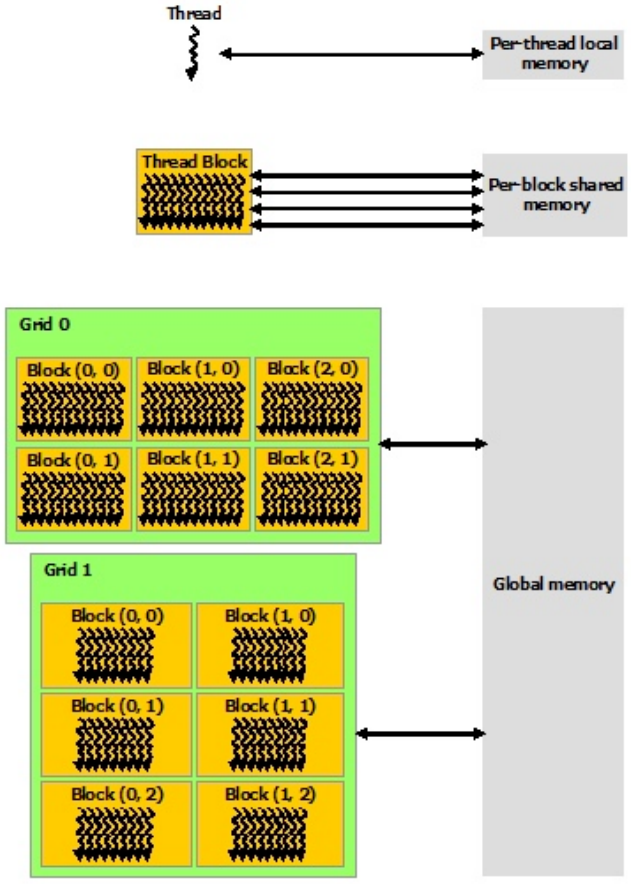

Figura 6: jerarquía de memoria de la GPU[5].

- Funciones llamadas ***kernels*** que, cuando son llamadas, son ejecutadas *N* veces en paralelo por *N* hilos *CUDA* distintos.
- Los **hilos dentro de un bloque pueden cooperar** (memoria compartida).
- Para **evitar condiciones de carrera** (*race conditions*) es necesario establecer **puntos de sincronización** en el *kernel* → hilos del bloque deben esperar antes de que cualquiera de ellos pueda continuar.
- Cada **hilo** de cada bloque posee una **memoria local privada**.
- Cada **bloque** de hilos posee, durante su ejecución, una **memoria compartida** que comparten dichos hilos.
- Todos los hilos de todos los bloques (**malla/grid**) tienen acceso a la **memoria global** de la *GPU.*
- Operaciones *I/O* en la memoria de la *GPU* se realizan en bloques de 128 bits –> **datos en memoria deben estar alineados y ser coalescentes**.
- Físicamente la **memoria de la *CPU* y** la memoria de la ***GPU* son espacios separados y distintos**.
- *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**.

Programación con CUDA

- Utilizando las **abstracciones, minimizando la latencia y maximizando el ancho de banda** de cada tipo de memoria:
  1. Descomponer un problema en sub-problemas
  2. Alinear los datos en memoria de forma coalescente
  3. Transferir datos al espacio de memoria de la *GPU*.
  4. Sub-problema debe ser dividido en tareas (bloque de hilos *GPU*).
  5. Procesar tareas maximizando la cooperación entre hilos (memoria compartida y barreras de sincronización).
  6. Cooperación entre bloques de hilos → operaciones atómica (*atomics*), minimizar su uso.
- Tres reglas básicas:
  1. El programador debe transferir los **datos** a la ***GPU*** y **mantenerlos en ella**.
  2. El programador debe **proporcionar suficiente trabajo a la *GPU***.
  3. El programador debe centrarse en la **reutilización de datos**. Transferencias *CPU*⇔*GPU* y accesos a memoria global (**usar preferentemente registros**).

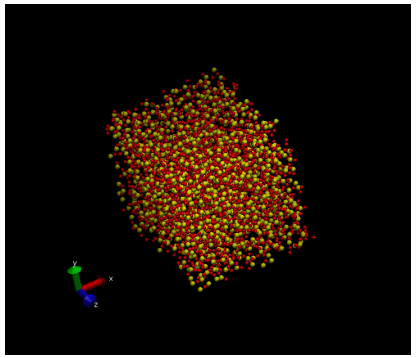

Figura 7: configuración inicial DLP.

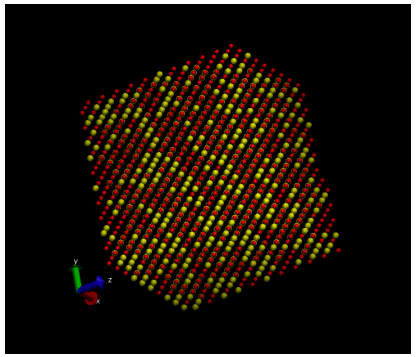

Figura 8: configuración inicial FCC.

Validación del modelo

- **Fase 1**: resultados Monte Carlo en serie (*CPU*) = resultados de referencia *DLPOLY*[6].
- **Fase 2**: dos configuraciones iniciales distintas de dióxido de silicio (SiO_2): un cristal cúbico centrado en las caras *FCC* (*Face Center Cubic*) y una configuración desordenada generada por *DLPOLY.* Ambas sometidas a las mismas condiciones termodinámicas y a la misma simulación Monte Carlo.
  - Procedimiento implementado correctamente → ambas configuraciones deben converger a un mismo estado de equilibrio de energía (*NVT* y *NPT*) y de densidad (*NPT*).

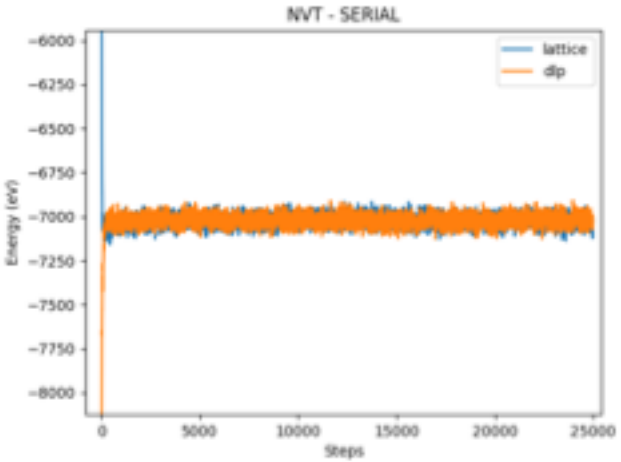

Figura 9: convergencia energía NVT (CPU)

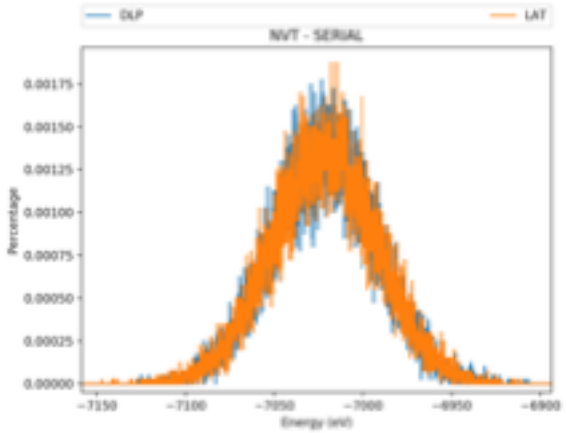

Figura 10: histograma energía NVT (CPU)

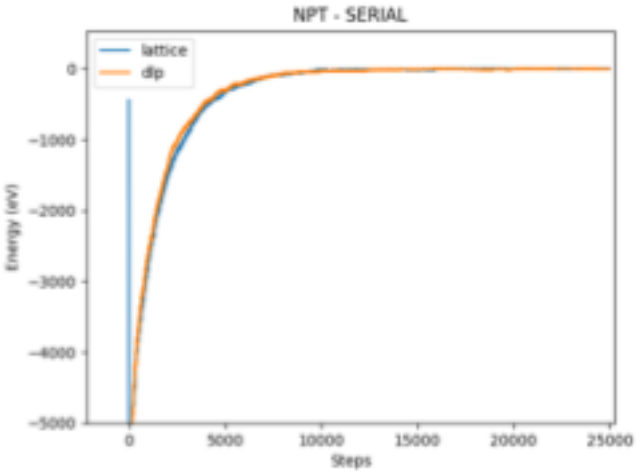

Figura 11: convergencia energía NPT (CPU)

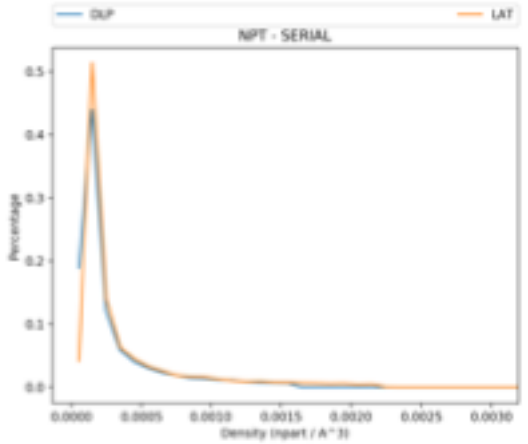

Figura 12: histograma densidad NPT (CPU)

Algoritmos

MC-NVT

RNG-MKL

- $CPU \rightarrow O = natoms^2$
- $GPU_{intrinsics} \rightarrow O = natoms \frac{natoms}{blockDim \cdot nSM} log_2 blockDim$

MC-NPT

RNG-MKL

- $CPU \rightarrow O = natoms^2 - natoms$
- $GPU_{intrinsics} \rightarrow O = \frac{natoms}{nSM} \left(\frac{natoms + (NTHREAD - 1))}{NTHREAD} + \frac{log_2 blockDim}{blockDim} \right)$

¡Todos los algoritmos escalan cuadráticamente!

MC-NVT-ChPotential

RNG-MKL

- $CPU \rightarrow O = O = nsp \cdot natoms^2$

RNG-cuRAND

- $GPU_{intrinsics} \rightarrow O = nsp \frac{natoms^2}{blockDim \cdot nSM} log_2 blockDim$

Resultados

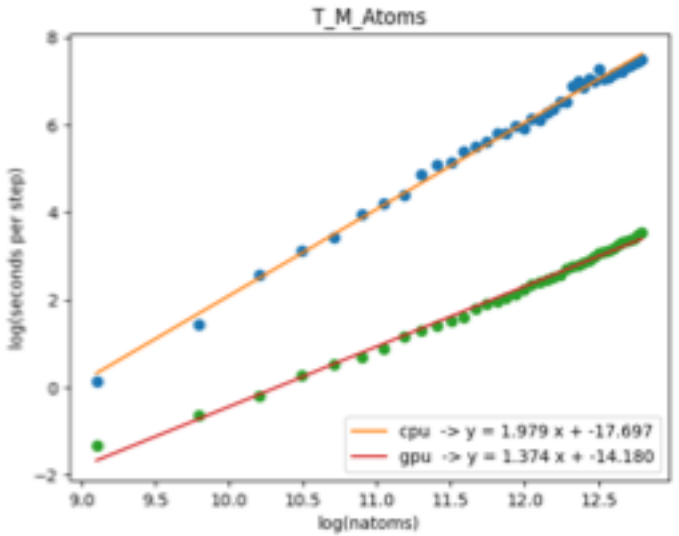

Figura 13: algoritmo moveAtoms : tiempo por paso respecto al tamaño del problema (log-log) y regresión lineal (CPU y GPU).

- MC-NVT (*moveAtoms*)
  - GPU mucho más rápido que CPU.
  - CPU: $O(N^2)$.
  - GPU: menor exponente, aún no escala cuadráticamente (casi lineal). $\approx O(N)$
  - Curva cóncava creciente.
  - Para tamaños mayores se espera mejor rendimiento.

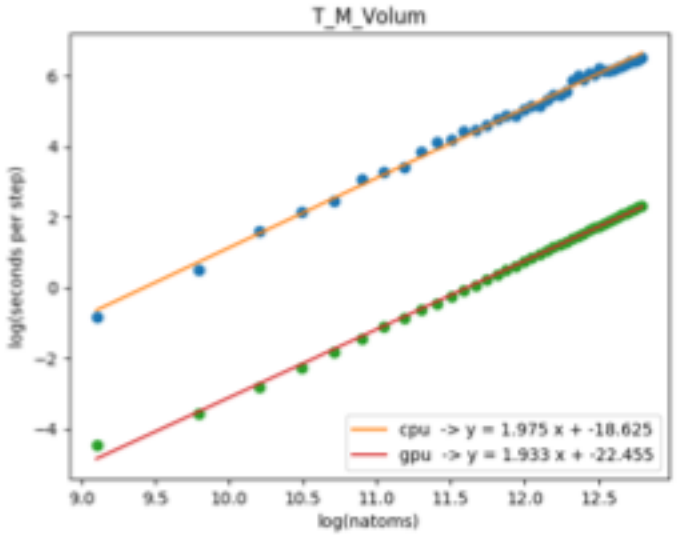

Figura 14: algoritmo moveVolum : tiempo por paso respecto al tamaño del problema (log-log) y regresión lineal (CPU y GPU).

- MC-NPT (*moveVolume*)
  - GPU mucho más rápido que CPU.
  - CPU y GPU: mismo exponente, ambos escalan cuadráticamente $O(N^2)$.
  - Rectas casi paralelas.

- *CPU Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz* con *GPU NVIDIA GeForce GTX 1080 Ti*.
- **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.
- Todos los algoritmos siguen la ley de potencias.

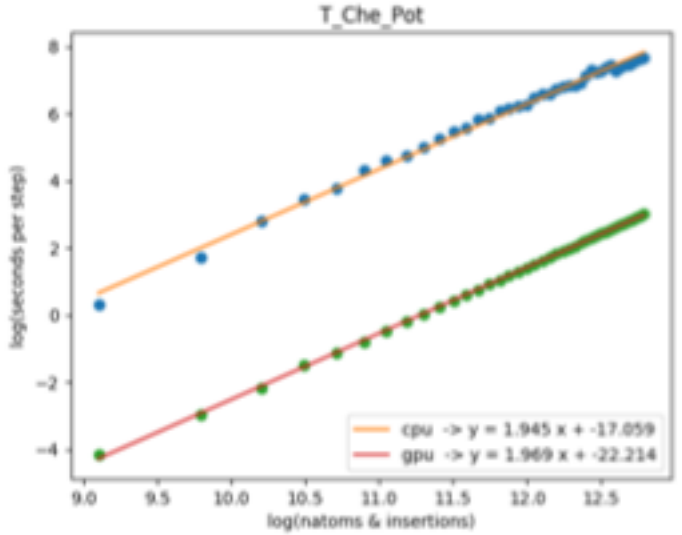

Figura 15: algoritmo chPotential : tiempo por paso respecto al tamaño del problema (log-log) y regresión lineal (CPU y GPU).

- MC-NVT (*chPotential*)
  - GPU mucho más rápido que CPU.
  - CPU y GPU: mismo exponente, ambos escalan cuadráticamente $O(N^2)$.
  - Rectas casi paralelas

Resultados

- *CPU Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz* con *GPU NVIDIA GeForce GTX 1080 Ti*.
- ***Speed-up***: $\lambda = \frac{t_{serie}}{t_{gpu}}$
- t_{serie} *y* t_{gpu} *corresponden al tiempo por paso en CPU y en GPU, respectivamente*.

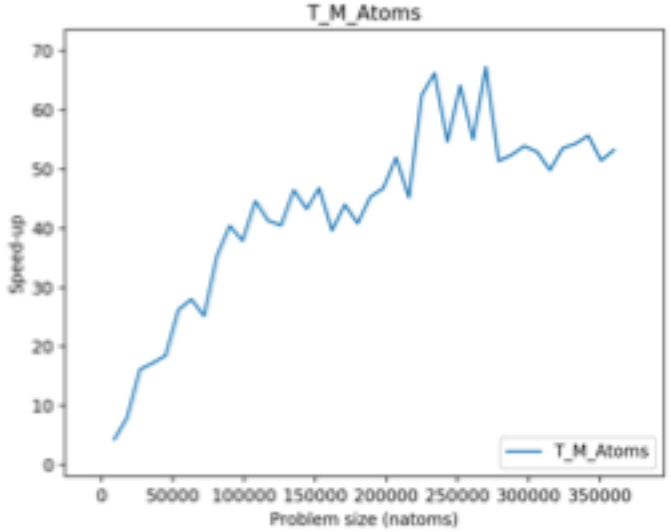

Figura 16: moveAtoms: escalado del speed-up

- MC-NVT (*moveAtoms*)
  - Mayor N → mayor aceleración.
  - Curva convexa creciente.
  - Comportamiento asintótico general que indica una próxima saturación.
  - Max(λ) = ~67.

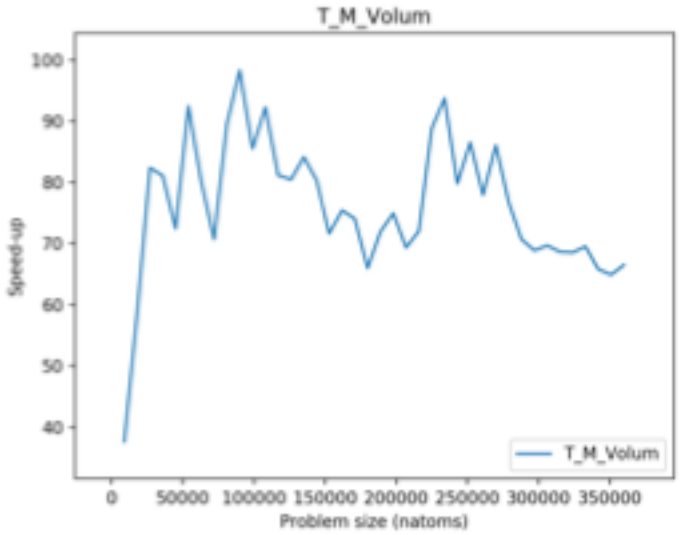

Figura 17: moveVolum: escalado del speed-up

- MC-NPT (*moveVolume*)
  - Aceleración muy alta ya con tamaños pequeños del problema.
  - Saturación en la optimización.
  - Max(λ) = ~98.
  - *Embarrassing parallel.*

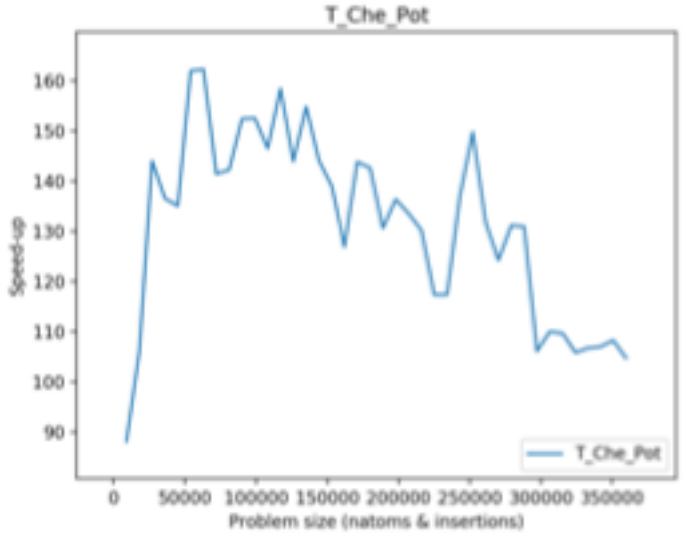

Figura 18: chPotential: escalado del speed-up

- MC-NVT (*chPotential*)
  - Aceleración muy alta ya con tamaños pequeños del problema.
  - Saturación en la optimización.
  - Max(λ) = ~162.
  - *Embarrassing parallel.*
  - **Necesario estudio de oscilaciones.**

Conclusiones

- Se ha demostrado que las **versiones paralelas** de los algoritmos **son mucho más eficientes** que las versiones serie.
- Para la muestra utilizada:
  - **Máximo de rendimiento** paralelo para los algoritmos ***moveVolume* y *chPotential* → dependencia cuadrática**.
  - Algoritmo ***moveAtoms* no** se ha alcanzado el **máximo de rendimiento → dependencia casi lineal**.
  - ***Speed-up*** de los algoritmos ***moveVolume* y *chPotential* → *embarrassing parallel*: aceleraciones muy altas** desde el **principio** de la muestra, lo que conlleva una **pronta saturación** de la aceleración.
  - ***Speed-up*** del algoritmo ***moveAtoms* → aceleración baja** que **aumenta progresivamente** con el tamaño del problema. Se esperan mejores rendimiento al aumentar *N*.
- 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 (***moveAtoms***).
- 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*[7])** para tamaños relativamente pequeños.

Futuro

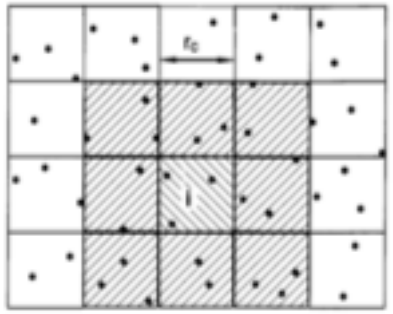

Figura 19: división de la caja de simulación en celdas de lado r_c. Las partículas de la caja i, sólo interaccionan con las de las de las 9 cajas sombreadas.

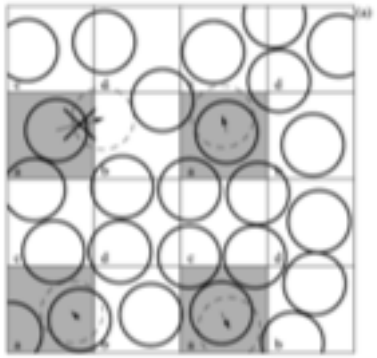

Figura 20: división de la caja de simulación en tablero de damas. Las partículas diferentes cajas sombreadas no interactúan, por lo que intentos de traslación o inserción/borrado pueden ser paralelos.

- La solución presentada es **adecuada** únicamente para **interacciones de muy largo alcance y partículas sin carga**.
- Para **interacciones de medio y corto alcance**.
  - Listas de vecinos y/o listas de celdas (*cell list*)[8].
  - Celdas igual al radio de corte.
- **Otro desafío**: equivalente a descomposición por dominios, a la manera habitual en Dinámica Molecular.
  - Estrategia de división en ***tablero de damas***[9] (*checkerboard*).
  - **Partículas de cajas sombreadas no interaccionan**.
  - Intento de traslación (*NVT*) o inserción/borrado de partículas (potencial químico) dentro de esas cajas **puede realizarse en paralelo**.

Bibliografía

1. A.J. Stone, *The theory of intermolecular forces*, Clarendon Press, 1997.
2. K.P.N. Murhty, *An introduction to Monte Carlo simulations in Statistical Physics*, arXiv:cond-mat/0104167 [cond-mat.stat-mech] (2003).
3. B. Widom, J. Chem. Phys., 39, 2808 (1963).
4. M.P. Allen y D.J. Tildesley, *Computer Simulation of Liquids*, Oxford University Press, 1987
5. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (2018).
6. W. Smith y J.T. Todorov, *A short description of DL_POLY*, Molecular Simulation, 32, 935 (2006).
7. S. Plimpton, *Fast Parallel Algorithms for Short-Range Molecular Dynamics*, J Comp Phys, 117, 1-19 (1995); http://lammps.sandia.gov/
8. D. Frenkel y B. Smit, *Understanding Molecular Simulation*, Academic Press, Londres, 2002.
9. 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 .