Laboratorio I de tiny MD: optimización secuencial

Fernando Blanco[1] and Ignacio J. Chevallier-Boutell[2,3]

[1]Universidad Nacional de Rosario, Facultad de Ciencias Exactas, Ingenieria y Agrimensura, Santa Fe, Argentina
[2]Universidad Nacional de Córdoba, Facultad de Matemática, Astronomía, Física y Computación, Córdoba, Argentina
[3]CONICET, Instituto de Física Enrique Gaviola (IFEG), Córdoba, Argentina

1. Introducción a la dinámica molecular

La dinámica molecular (Molecular Dynamic, MD) es una técnica de simulación computacional con la cual generamos una secuencia de puntos en el espacio de fase asociado a nuestro sistema en estudio, los cuales están conectados por el tiempo. De allí que podamos determinar una trayectoria.

La MD se basa en la dinámica Newtoniana y, por lo tanto, se enmarca dentro de la mecánica clásica: el movimiento se produce debido a la aplicación de una fuerza.

Tenemos cuatro etapas fundamentales:

1. **Determinar las condiciones iniciales:**

   Debemos indicarle al programa tanto las posiciones iniciales como las velocidades iniciales de cada una de las partículas que conforman nuestro sistema.

   En lo que respecta a las posiciones, dependerá del sistema y modelo a utilizar. Podemos tener sistemas no periódicos o bien sistemas que sean periódicos en 1, 2 o 3 direcciones.

   Si los enlaces de nuestro sistema son de corto alcance, no hay tanto problema en qué tamaño de caja de simulación utilizar; en cambio, si son de largo alcance, hay que tener más cuidado para no sobreestimar contribuciones: agregaríamos contribuciones ficticias a las contribuciones reales.

   Una vez definido el tamaño de la caja, debemos definir el radio de corte para el potencial: debe ser menor a la mitad de la caja. Este radio determina hasta dónde se deben considerar las contribuciones energéticas entre las partículas de la caja real junto a las partículas de la caja virtual.

   Si definimos r_c como el cutoff, el potencial deberá satisfacer

   $$V_c(r) = \begin{cases} V(r) & r \leq r_c \\ 0 & r_c < r \end{cases}$$

   A partir de esto, se recurre a la convención de imagen mínima para hacer el cálculo de las fuerzas.

   Respecto a las velocidades, debemos indicar una distribución inicial. Generalmente se recurre a una distribución Boltzmann.

2. **Calcular las fuerzas:**

   Las fuerzas son calculadas según $\vec{F} = -\nabla V$, donde V es el potencial que nosotros le asignamos a nuestro sistema. Es una parte clave del cálculo pues indica todas las interacciones que se pueden desarrollar. Vemos entonces que determina la calidad y precisión del cálculo.

   Los potenciales de interacción se asignan según el objetivo del estudio y la disponibilidad tecnológica. A grandes rasgos se clasifican en empíricos, semi-empíricos y ab-initio.

   Un tipo usual de potenciales de interacción son los potenciales de a pares. En éstos es donde interviene el radio de corte.

3. **Resolver las ecuaciones de movimiento:**

   Con el fin de minimizar los errores, se utiliza el formalismo Hamiltoniano donde debemos resolver $6N$ ODE de primer orden. Para resolverlas, debemos integrar numéricamente por lo que recurrimos las técnicas de diferencias

finitas. Existen diferentes maneras de lograrlo: método de Euler, método de Runge-Kutta, algoritmo de Verlet, algoritmo velocity Verlet, algortimo predictor-corrector, entre otros

4. **Determinar las trayectorias:**

   Esto es necesario para poder después recurrir a la mecánica estadística y relacionar los resultados con propiedades microscópicas y macroscópicas. Algunas de las propiedades que podemos calcular a partir de tales simulaciones son: energía potencial, energía cinética, temperatura, presión, difusión, entre muchas otras. La idea de fondo es expresar cualquier observable en términos de las posiciones y las velocidades. Las propiedades más relevantes a calcular son aquellas que, de alguna manera, podemos correlacionar experimentalmente.

A partir de una simulación de MD logramos extraer información sobre el espacio de fase $6N$ dimesional: las posiciones nos permitirán hacer un análisis estructural, mientras que los momentos nos permitirán hacer un anásis dinámico.

Hasta ahora todo lo dicho refiere a una dinámica microcanónica (NVE-MD), la cual tiene una difícil correlación experimental. Sin embargo, existen diversas maneras de hacer dinámica en otros ensambles, mediante controles de presión (barostato de Berendsen o de Nosé-Hoover) y de temperatura (termostato de Berendsen, de Andersen o de Nosé-Hoover).

2. Tiny MD

En el presente trabajo se utiliza un cristal cúbico. Particularmente, trabajamos con una red FCC. A partir de esto se sabe que

- Habrá 4 átomos por celda unidad, siendo $(0;0;0)$, $(0;0{,}5;0{,}5)$, $(0{,}5;0;0{,}5)$ y $(0{,}5;0{,}5;0)$ los vectores de red.
- El parámetro de red será $a = \sqrt[3]{4/\rho}$, donde ρ es la densidad: hay 4 átomos por celda unidad y la densidad nos indica qué tan juntos van a estar. Al modificarla, estaríamos influyendo en el estado de agregación de la sustancia.

Para el caso de la celda FCC, tenemos la relación $N = 4m^3$, donde 4 es la cantidad de átomos por celda y N es la cantidad total de átomos presentes. El valor de m indica la cantidad de veces que debemos replicar la celda unidad en cada dirección del espacio.

La inicialización de las posiciones se realiza colocando átomos de argón perfectamente distribuidos en una celda FCC según la densidad impuesta al sistema. Por otra parte, la inicialización de las velocidades se hace de manera aleatoria, variando entre -0.5 y 0.5. Luego se resta la velocidad del centro de masa para evitar que el sistema se desplace y, además, se ajusta la temperatura mediante un factor de escala dado por $sf = \sqrt{T_0/T}$, donde T_0 es la temperatura de referencia y T es la temperatura actual (esta sería la idea base de un termostato de Berendsen).

Para el cálculo de fuerzas se utiliza el potencial de Lennard-Jones (LJ o 12-6), el cual viene dado por

$$V_{LJ}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

donde ϵ es la profundidad del pozo y σ es la posición de equilibrio. A partir de esto, la fuerza será

$$F_{LJ}(r) = 24\frac{\epsilon}{r} \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

Las ecuaciones de movimiento se resuelven considerando el algoritmo velocity Verlet, siendo uno de los más usados por los programas de MD.

3. Detalles computacionales

Se utilizó un procesador Intel Core i5-4460:

- Microarquitectura Haswell.
- Disponibilidad de 4 núcleos.
- Frecuencia de 3,20 GHz.
- Caché de 6 MB.

Dicho CPU se encuentra operado en Ubuntu 20.04.1 LTS, kernel 5.4.0-72-generic.

El poder de cómputo de un core medido con Empirical Roofline Toolkit (Fig. 1) fue de 64.3 GFLOPs/sec. Se tiene una memoria de 8 GB y 1600 MT/s, con 1 canal ocupado. El ancho de banda para un core medido con Empirical Roofline Toolkit fue de 11. 6GB/s.

Los compiladores que se utilizaron fueron GCC-9 y Clang-10.

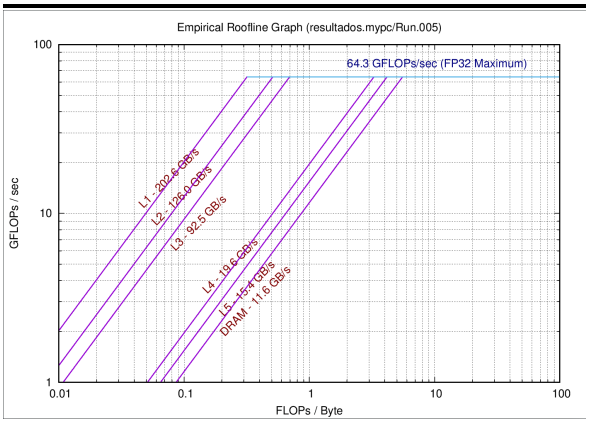


Figura 1: Resultado del análisis del poder de cómputo de un core medido con Empirical Roofline Toolkit.

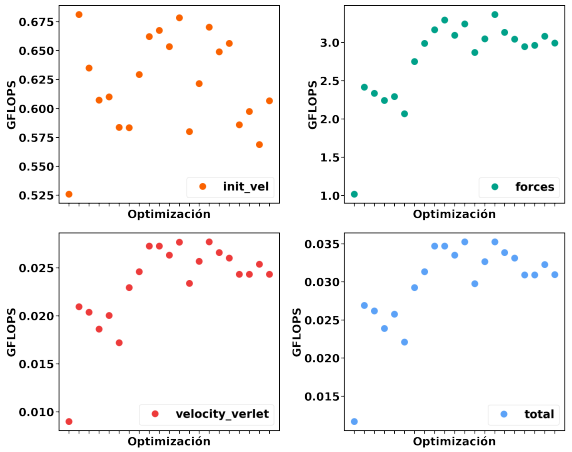


Figura 2: GFLOPS en función de la optimización utilizada junto a las diferentes flags para las distintas situaciones analizadas, compilando con Clang-10.

4. Resultados y discusión

4.1. Análisis de optimizaciones para una misma versión de los compilador

A partir de realizar perf con GCC-9 $-O0$, las tres contribuciones principales fueron:

1. 67,88 % del tiempo de cálculo consumido por la función de forces (cálculo de fuerzas).
2. 29,44 % del tiempo de cálculo consumido por la función minimum_image (determinación de imágen mínima).
3. 0,90 % del tiempo de cálculo consumido por la función velocity_verlet (algoritmo de integración).

Se decidió que la métrica adecuada para analizar el problema era GFLOPS ya que escalaría con el problema sin mayores dificultades. Consdierando esto, se determinó la cantidad de GFLOPS para las funciones forces y velocity_verlet. También se calcularon los GFLOPS para la función init_vel (inicialización de velocidades) y el total de la corrida. No se calcularon los GFLOPS para minimum_image ya que no está directamente expuesta dentro del archivo tiny_md.c, sino que es llamada por forces.

Todos estos valores fueron adquiridos utilizando GCC-9 y Clang-10 con optimizaciones $-O1$, $-O2$, $-O3$ y $-Ofast$, incluyendo las siguientes flags combinadas de diferentes maneras: $-march = native$, $-ffast - math$ $-funroll - loops$ y $-floop - block$ (esta última sólo disponible en GCC).

Como primer pantallazo de la situación, en las Figs. 2 y 3 se tiene un barrido de todos estos parámetros de optimización compilando con Clang-10-10 y GCC-9, respectivamente. Si reescaleamos los perfiles (4), vemos que:

- La función init_vel tiene un perfil completamente distinto a las otras dos.
- Las funciones forces y velocity_verlet tienen perfiles muy similares, lo cual está directamente asociado a cómo están asociadas estas funciones en el código.
- El perfil del tiempo total coincide con las dos anteriores ya que, según vimos con perf, estas dos funciones son las que se llevan el mayor porcentaje de uso del procesador.

Para hacer un análisis más detallado, se hicieron 10 corridas con cada una de las opciones ya mencionadas. Los resultados para Clang-10 y GCC-9 se encuentran en las Figs. 5 y 6, respectivamente. En ambos compiladores vemos que la función forces se lleva la mayor cantidad de GFLOPS, siguiéndole la función init_vel. Considerando los resultados hasta ahora expuestos y que init_vel sólo se llama una vez por iteración, pero que forces se llama dos veces dentro del cálculo (una de manera explícita en el loop principal y otra de manera implícita dentro de la definición de la función velocity_verlet), vemos

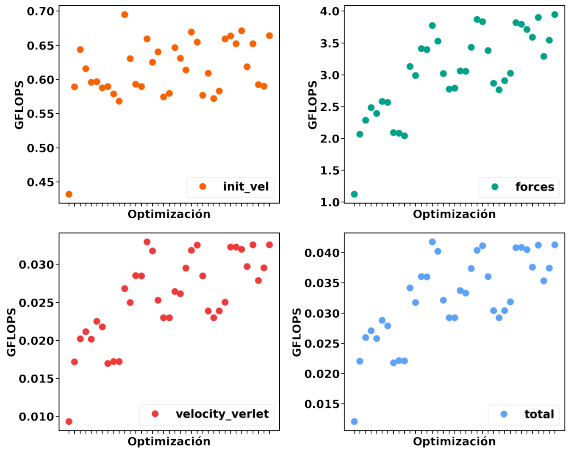


Figura 3: GFLOPS en función de la optimización utilizada junto a las diferentes flags para las distintas situaciones analizadas, compilando con GCC-9.

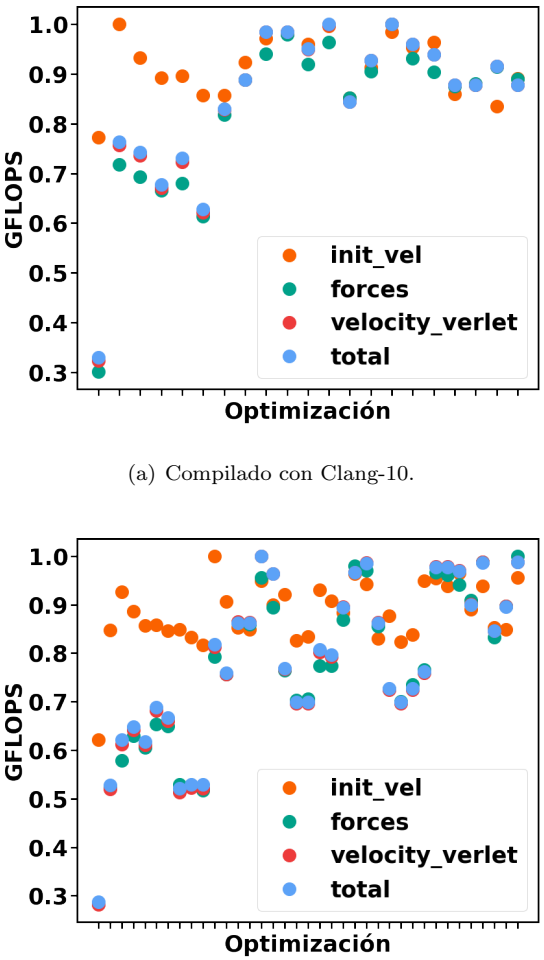
que queda en evidencia lo que perf nos había mostrado. Destacamos además el notorio aumento en performance para todos los casos de optimización con respecto al caso $-O0$.

Para continuar con las comparaciones, primero se analizaron estadísticamente los resultados para forces de cada una de las Figs. 5 y 6. Para el caso de Clang, vemos que:

- **-O1:** no existen diferencias significativas entre C1, C2, C3 y C4, siendo C4 el de mayor precisión. El valor alcanzado con C4 es un 11 % mayor que C5.
- **-O2:** no existen diferencias significativas entre C2, C3, C4 y C5, siendo C4 el de mayor precisión. El valor alcanzado con C4 es un 20 % mayor que C1.
- **-O3:** no existen diferencias significativas entre C1, C3, C4 y C5, siendo C4 el de mayor precisión. El valor alcanzado con C4 es un 17 % mayor que C2.
- **-Ofast:** no existen diferencias significativas, siendo C5 el de mayor precisión.

Por otro lado, para el caso de GCC se tiene que:

- **-O1:** no existen diferencias significativas entre G3, G5 y G6, siendo G6 el de mayor precisión. El valor alcanzado con G6 es un 7 % mayor que G4.



(a) Compilado con Clang-10.

(b) Compilado con GCC-9.

Figura 4: Reescaleo y comparación de lo expuesto en las Figs. 2 y 3. Por razones de espacio y comodidad visual, se omiten intencionalmente los nombres particulares de las optimizaciones.

- **-O2:** no existen diferencias significativas desde G1 hasta G6, siendo G3 el de mayor precisión. El valor alcanzado con G3 es un 13 % mayor que G7.
- **-O3:** no existen diferencias significativas entre G3, G4 y G5, siendo G4 el de mayor precisión. El valor alcanzado con G4 es un 14 % mayor que G6.
- **-Ofast:** no existen diferencias significativas, salvo G1 y G7, siendo G9 el de mayor precisión. El valor alcanzado con G9 es un 20 % mayor que G7.

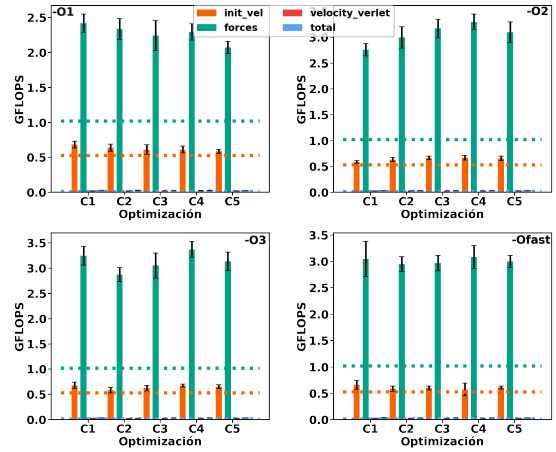


Figura 5: GFLOPS en función de los diferentes flags de optimización considerando $-O1$ (arriba a la izquierda), $-O2$ (arriba a la izquierda), $-O3$ (arriba a la izquierda) y $-Ofast$ (arriba a la izquierda). Se compiló con Clang-10. Código: C1 = ninguna flag, C2 = $-march = native$, C3 = $-march = native \quad -ffast-math$, C4 = $-march = native \quad -funroll-loops$ y C5 = $-march = native \quad -ffast-math \quad -funroll-loops$. Las líneas horizontales indican los GFLOPS para la situación $-O0$.

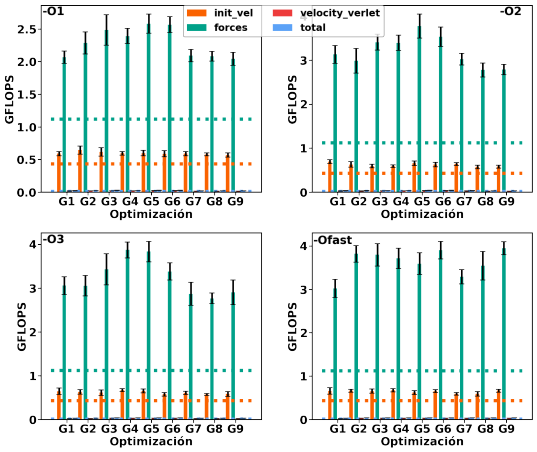


Figura 6: GFLOPS en función de los diferentes flags de optimización considerando $-O1$ (arriba a la izquierda), $-O2$ (arriba a la izquierda), $-O3$ (arriba a la izquierda) y $-Ofast$ (arriba a la izquierda). Se compiló con GCC-9. Código: G1 = ninguna flag, G2 = $-march = native$, G3 = $-march = native \quad -ffast-math$, G4 = $-march = native \quad -funroll-loops$, G5 = $-march = native \quad -floop-block$, G6 = $-march = native \quad -ffast-math \quad -funroll-loops$, G7 = $-march = native \quad -ffast-math \quad -floop-block$, G8 = $-march = native \quad -funroll-loops \quad -floop-block$ y G9 = $-march = native \quad -ffast-math \quad -funroll-loops \quad -floop-block$. Las líneas horizontales indican los GFLOPS para la situación $-O0$.

Estas 8 situaciones de mayor precisión se comparan en la Fig. 7. Se tiene entonces que lo mejor para Clang es $-O2 \quad -march = native \quad -funroll-loops$ y $-O3 \quad -march = native \quad -funroll-loops$, siendo un 12 % mayor que $-Ofast \quad -march = native \quad -ffast-math \quad -funroll-loops$. En el caso de GCC lo mejor es $-O3 \quad -march = native \quad -funroll-loops$ y $-Ofast \quad -march = native \quad -ffast-math \quad -funroll-loops \quad -floop-block$, siendo un 16 % mayor que $-O2 \quad -march = native \quad -ffast-math$. En cualquier caso, por ahora resulta más adecuado utilizar GCC ya que presenta valores un 17 % mayores que Clang.

4.2. Análisis de versiones de compiladores para las mejores optimizaciones

Hasta ahora hemos compilado con GCC-9-9 y Clang-10. Considerando las mejores condiciones de optimización, se analizaron distintas versiones para cada compilador: Clang-10, Clang-11, Clang-12, GCC-8, GCC-9 y GCC-10.

En la Fig. 8 se tienen los resultados de dicha comparación, quedando claramente en evidencia que GCC-9 y GCC-10 sobresalen respecto a los demás cuando compilamos utilizando $-Ofast \quad -march = native \quad -ffast-math \quad -funroll-loops \quad -floop-block$. Las diferencias que se presentan entre ambos compiladores no son significativas, pudiéndose elegir cualquiera de ellos. En números, tenemos que GCC-9 es un 16 % mayor que GCC-8 y un 22 % mayor que Clang-11, todo con $-Ofast$.

4.3. Impacto del número de partículas en la performance

Hasta este punto hemos logrado determinar que el compilador que ofrece el mejor desempeño para llevar a cabo la dinámica es GCC en sus versiones 9 ó 10 y compilando teniendo en cuenta $-Ofast \quad -march = native \quad -ffast-math \quad -funroll-loops \quad -floop-block$. Todos los cálcu-

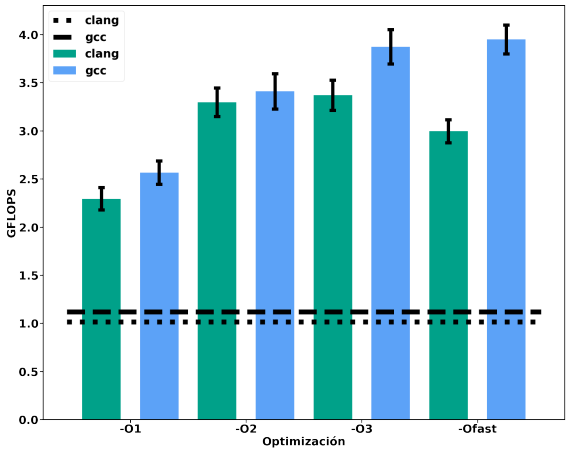


Figura 7: Comparación de los picos de GLOPS de forces obtenidos utilizando las optimizaciones de mejor precisión.

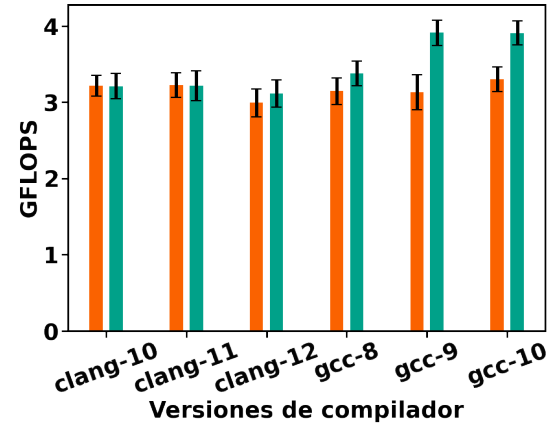


Figura 8: Comparación de los GLOPS de forces obtenidos utilizando las mejores optimizaciones, considerando diferentes versiones de los compiladores. Código de colores Clang: naranja para $-O2 \quad -march = native \quad -funroll-loops$ y verde para $-O3 \quad -march = native \quad -funroll-loops$. Código de colores GCC: naranja para $-O3 \quad -march = native \quad -funroll-loops$ y verde para $-Ofast \quad -march = native \quad -ffast-math \quad -funroll-loops \quad -floop-block$.)

los hasta ahora realizados se hicieron manteniendo constante el numero de partículas simuladas: siempre fueron 256 partículas. En el presente apartado mostraremos cómo varía la performance del cálculo cuando variamos dicha cantidad de partículas.

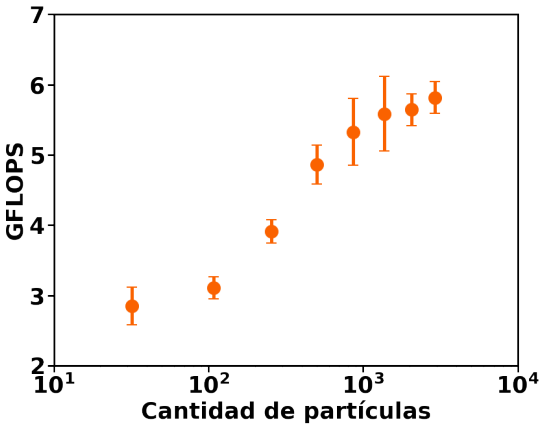


Figura 9: GFLOPS en función de la cantidad de partículas simuladas. Todos los puntos obtenidos con $GCC - 9 - Ofast \quad -march = native \quad -ffast-math \quad -funroll-loops \quad -floop-block$.

Como se puede ver en la Fig. 9, se varío la cantidad de partículas entre 32 y 2916. El comportamiento errático del primer punto podemos atribuirlo a que el tamaño de la caja de simulación para esa cantidad de partículas es comparable al radio de corte, introduciendo errores en la medición. Dejándolo de lado, notamos dos situaciones en la gráfica:

1. Una primera etapa de crecimiento, donde notamos que la métrica elegida fue buena ya que escala con el tamaño del problema.

2. Una segunda etapa de saturación, la cual refleja que estamos empezando a llegar al límite de nuestros caches y la jerarquía de memoria se ve comprometida.

Destacamos además que el quiebre entre una etapa y la otra presenta gran variabilidad en las mediciones.

5. Conclusiones

En el presente trabajo se estudiaron optimizaciones secuenciales simples sobre un simulación de dinámica molecular, modificando diferentes flags y opciones de optimización para dos compiladores (GCC y Clang), cada uno en 3 versiones diferentes.

Manteniendo la cantidad de partículas constantes ($N = 256$) y utilizando únicamente GCC-9 y

Clang-10, primero se estudiaron todas las compilaciones posibles entre $-O1$, $-O2$, $-O3$ y $-Ofast$, y ciertas flags ($-march = native$, $-ffast-math$ $-funroll-loops$ y $-floop-block$). La situación más favorable resulto ser $GCC-9 \quad -Ofast \quad -march = native \quad -ffast-math \quad -funroll-loops \quad -floop-block$. Comparándola con otras combinaciones relevantes, vemos que es:

- Un 17 % mayor que $Clang-10 \quad -O3 \quad -march = native \quad -funroll-loops$.
- Un 289 % mayor que $Clang-10 \quad -O0$.
- Un 252 % mayor que $GCC-9 \quad -O0$.

Como en esta primera etapa sólo se habían utilizado una versión para cada compilador, se decidió contrastar los resutlados de varias versiones tanto de Clang como de GCC. A partir de esto ratificamos que la mejor oppción dentro de las estudiadas era $GCC-9 \quad -Ofast \quad -march = native \quad -ffast-math \quad -funroll-loops \quad -floop-block$.

Finalmente, utilizando esta combinación óptima se estudio lo que ocurría con el desempeño del cálculo cuando variábamos la cantidad de partícuas simuladas. Dentro del rango de análisis se destacan una situación de crecimiento, dejando ver que la métrica elegida para el trabajo resultó adecuada, y una situación de saturación, evidenciando el límite de los caches.