Laboratorio II de tiny MD: vectorización

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

En la primera entrega se estudiaron optimizaciones secuenciales simples sobre una simulación de dinámica molecular, modificando diferentes flags y opciones de optimización para dos compiladores (gcc y clang), cada uno en 3 versiones diferentes. Se tuvieron 3 etapas de análisis:

1. Mantener la cantidad de partículas constantes y utilizar únicamente gcc-9 y clang-10, combinando diferentes optimizaciones y flags.
2. Mantener la cantidad de partículas constantes y utilizar las mejores optimizaciones y flags para estudiar diferentes versiones de los compiladores.
3. Utilizar compilador, optimización y flags óptimos para estudiar lo que ocurría con el desempeño del cálculo cuando variábamos la cantidad de partícuas simuladas.

A partir de un análisis de desempeño (perf) se supo que la función forces era la de mayor impacto. La métrica entonces utilizada fue GFLOPS, con la cual se analizó forces para cada caso, observándose que escala correctamente con el tamaño del problema. La conclusión fue entonces que el mejor compilador resultaba ser gcc-9 con $-Ofast$, $-march = native$, $-ffast - math$, $-funroll - loops$ y $-floop - block$.

Hasta ahora todo lo hecho no fue meramente secuencial, sino que sabemos que tuvo lugar cierto paralelismo implícito que no podíamos controlar finamente, originado por el compilador. En estas segunda entrega se pretende hacer un paralelismo más explícito mediante vectorización, tanto modificando el código para favorecer la autovectorización como reescribiendo parte del programa de manera SPMD (Single Program Multiple Data) para compilar con ISPC.

2. Detalles computacionales

En la primera entrega se utilizó una computadora de escritorio (Desktop), pero para esta segunda entrega se utilizó un cluster (Jupiterace). En la Tabla 1 se tienen los datos acerca de los procesadores en cada caso.

Tabla 1: Características de los procesadores de las computadoras utilizadas.

Computadora	ID	Microarq.	Cores	Base	Turbo	Cache
Desktop	Intel Core i5-4460	Haswell	4	3.20 GHz	3.40 GHz	6 MB
Jupiterace	Intel Xeon E5-2680 v4	Broadwell	14	2.40 GHz	3.30 GHz	35 MB

Los compiladores que se utilizaron fueron clang-10, gcc-9 e ispc-1.15. De ahora en más se referirá a ellos directamente como clang, gcc e ispc, respectivamente.

3. Resultados y discusión

3.1. Nuevo punto cero

Dado que cambiamos abruptamente de poder de cómputo al pasar a Jupiterace, se decidió hacer algunas pruebas previas en esta máquina para poder tener un punto de comparación entre los resultados actuales y los anteriores.

Primero se comparó gcc y clang con las 2 mejores combinaciones de optimizaciones y flags para cada uno, teniéndose:

- **A:** -O2 -march=native -funroll-loops.
- **B:** -O3 -march=native -funroll-loops.
- **C:** -O3 -march=native -funroll-loops.
- **D:** -Ofast -march=native -ffast-math -funroll-loops -floop-block

Dicha comparación se puede ver en la Fig. 1, donde se confirma que lo anteriormente concluido sigue en pie: gcc D es la combinación que da lugar a la mejor perfomance.

En la Fig. 2 se comparan los resultados obtenidos para Desktop y Jupiterace en función de la cantidad de partículas simuladas. Observamos cómo el uso de Jupiterace da lugar a una menor variabilidad de los resultados frente a Desktop, teniendo además una mejor performance: su poder de cómputo arroja resultados casi 4 veces mayores. Destacamos además que en el mismo tiempo de cálculo se pudo llegar a simular 5324 partículas en Jupiterace frente a las 2916 partículas en Desktop.

En la Tabla 2 se tienen los resultados de hacer perf con gcc -O0 en ambas computadoras y, además, en Jupiterace con los flags D. Vemos cómo cambia el tiempo que ocupa forces en la corrida al cambiar de -O0 a D. El — en minimum image es porque no aparece la función en los primeros registros del reporte de perf.

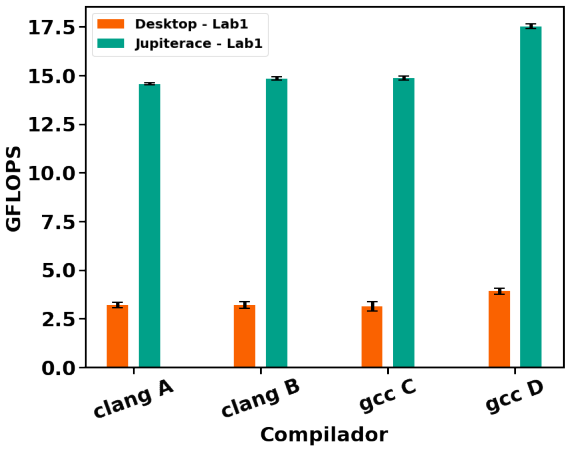
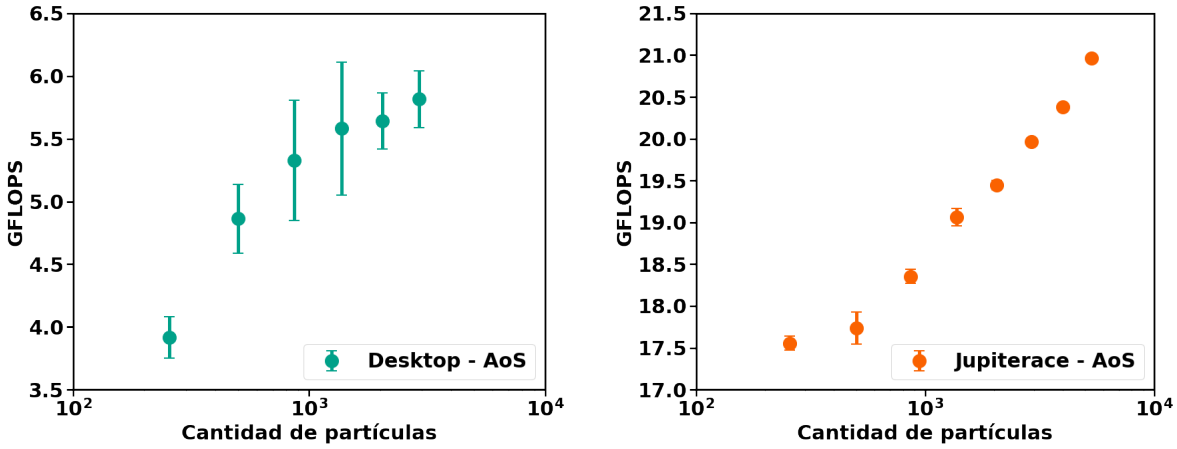


Figura 1: Comparación de los GLOPS de forces obtenidos utilizando las mejores optimizaciones, considerando diferentes versiones de los compiladores. El código utilizado es el resultante del laboratorio 1, corriendo tanto en Desktop como en Jupiterace.



(a) (b)

Figura 2: GFLOPS en función de la cantidad de partículas simuladas: código resultante del laboratorio 1 en Desktop (a) y en Juipterace (b).

Tabla 2: Resultados de perf con gcc en ambas computadoras utilizadas, considerando diferentes banderas.

Función	Desktop (-O0)	Jupiterace (-O0)	Jupiterace (D)
Forces	67.88 %	68.19 %	96.12 %
Minimum image	29.44 %	29.50 %	—
Velocity verlet	0.90 %	0.86 %	0.35 %

3.2. De AoS a SoA

Sabemos que el código resultante del laboratorio 1 estaba escrito en formato AoS (Array of Structures), sin embargo el formato SoA (Structure of Arrays) es mucho más compatible con el paralelismo que el formato AoS ya que facilita la vectorización y maximiza la localidad espacial. Con esto en mente se procedió a modificar el código y se probó ambas versiones utilizando la autovectorización que nos ofrece el compilador, confirmando lo mencionado anteriormente por los resultados de GFLOPS.
En la Fig. 3 se observan partes del código antes y después del cambio de AoS a SoA para las funciones init_pos y forces; mientras que en la Fig. 4 se comparan los resultados de utilizar el código en AoS o en SoA en Jupiterace, quedando en evidencia una mejora en el desempeño.

```
// Coordenadas xyz de la partícula 1
rxyz[idx + 0] = i * a;
rxyz[idx + 1] = j * a;
rxyz[idx + 2] = k * a;
// Coordenadas xyz de la partícula 2
rxyz[idx + 3] = (i + 0.5) * a;
rxyz[idx + 4] = (j + 0.5) * a;
rxyz[idx + 5] = k * a;
// Coordenadas xyz de la partícula 3
rxyz[idx + 6] = (i + 0.5) * a;
rxyz[idx + 7] = j * a;
rxyz[idx + 8] = (k + 0.5) * a;
// Coordenadas xyz de la partícula 4
rxyz[idx + 9] = i * a;
rxyz[idx + 10] = (j + 0.5) * a;
rxyz[idx + 11] = (k + 0.5) * a;
```

(a) init_pos: AoS

```
// Coordenada x de las 4 partículas
rxyz[idx + 0] = i * a;
rxyz[idx + 1] = (i + 0.5) * a;
rxyz[idx + 2] = (i + 0.5) * a;
rxyz[idx + 3] = i * a;
// Coordenada y de las 4 partículas
rxyz[idx + 0 + N] = j * a;
rxyz[idx + 1 + N] = (j + 0.5) * a;
rxyz[idx + 2 + N] = j * a;
rxyz[idx + 3 + N] = (j + 0.5) * a;
// Coordenada z de las 4 partículas
rxyz[idx + 0 + 2*N] = k * a;
rxyz[idx + 1 + 2*N] = k * a;
rxyz[idx + 2 + 2*N] = (k + 0.5) * a;
rxyz[idx + 3 + 2*N] = (k + 0.5) * a;
```

(b) init_pos: SoA

```
fxyz[i + 0] += fr * rx;
fxyz[i + 1] += fr * ry;
fxyz[i + 2] += fr * rz;

fxyz[j + 0] -= fr * rx;
fxyz[j + 1] -= fr * ry;
fxyz[j + 2] -= fr * rz;
```

(c) forces: AoS

```
fxyz[i] += fr * rx;
fxyz[i + N] += fr * ry;
fxyz[i + 2*N] += fr * rz;

fxyz[j] -= fr * rx;
fxyz[j + N] -= fr * ry;
fxyz[j + 2*N] -= fr * rz;
```

(d) forces: SoA

Figura 3: Porciones de código de las funciones init_pos y forces antes (AoS) y desupés (SoA) de modificar el código.

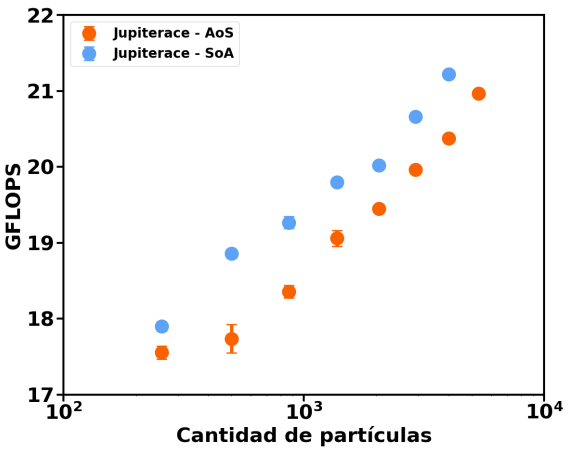


Figura 4: GFLOPS en función de la cantidad de partículas simuladas, comparando el código en formato AoS con el código en formato SoA.

3.3. Vectorización con ISPC

Sabemos que el compilador puede autovectorizar. Para ello debemos utilizar la flag $-ftree-vectorize$. Para conocer qué pudo y que no pudo vectorizar por sí mimso se necesitan dos flagas más: $-fopt-info-vec$ y $-fopt-info-vec-missed$, respectivamente. La información más relevante de todo aquello que no pudo autovectorizar concierne a la función forces, ya que es el nudo principal del código en términos de desempeño. Lo que no pudo vectorizar el compilador de forces es el loop anidado que, además, tiene control de flujo dentro.

Para lograr la vectorización del código, se recurrió a ISPC. Se reescribió la parte problemática de la función forces usando funciones propias del lenguaje como reduce_add() y foreach(). Este código se compiló con ISPC para obtener un nuevo header que permitiera la implementación vectorial dentro del código original.

Teniendo el código vectorizado, se analizó cómo se alteraba nuestra métrica en función de la cantidad de partículas simuladas. Todos los puntos se obtuvieron con gcc D, sumando la flag $-ftree-vectorize$, haciendo 10 corridas en cada caso.

Cuando vectorizamos nuestro código, los resultados de la métrica aumentan aún más (Fig. 5): crecen más del 50 % respecto a los valores obtenidos en Jupiterace con el código en formato AoS y sin vectorizar. Destacamos además que en el mismo tiempo de cálculo se pudo llegar a simular 6912 partículas con SoA más ISPC frente a las 5324 partículas en AoS. Se desconocen lor orígenes del zig-zag que se aprecia en los últimos puntos.

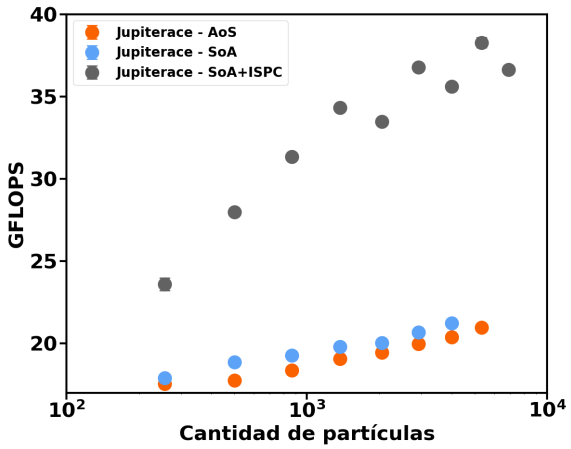


Figura 5: GFLOPS en función de la cantidad de partículas simuladas.

3.4. Energía potencial: una segunda métrica

Sabemos que nuestro código fue compilado con éxito y la métrica de GFLOPS ha mejorado. Sin embargo esta métrica no nos asegura que no hemos *roto* nuestro código, en el sentido de que no nos indica que el nuevo código necesariamente esté cumpliendo el mismo objetivo que antes. Dado que estamos modificando la parte del código encargada de calcular las fuerzas (gradiente del potencial), se puede utilizar la energía potencial para comparar los resultados arrojados por el nuevo código en relación a los resultados del código original.

En la Fig. 6 se tiene la energía potencial en función de la densidad para diferente cantidad de partículas, considerando el código original y el actualmente generado. Vemos que el resultado buscado no se ve alterado, por lo que las modificaciones realizadas no alteran el fin para el que fue escrito el código.

c

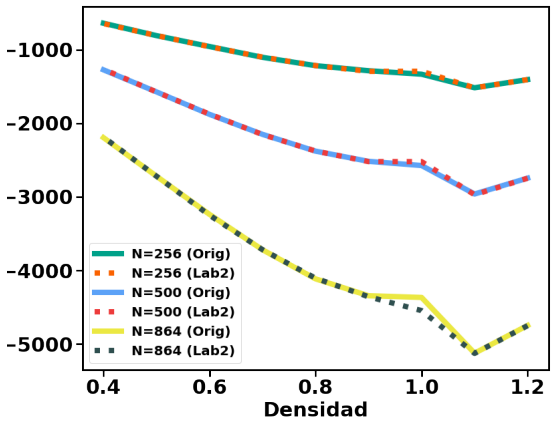


Figura 6: Energía potencial en función de la densidad del sistema. Se consideran tres tamaños de problema (256, 500 y 864), comparando el código original (Orig) con el generado en esta entrega (Lab2).

4. Conclusiones

En el presente trabajo se estudió la manera de obtener un código vectorizado para una simulación de dinámica molecular. En una primera etapa se reescribió el codigo, pasándolo de formato AoS a SoA, lo cual facilita tanto la autovectorización por parte del compilador como la vectorización explícita.

Luego se recurrió a ISPC para vectorizar el código, reescribiendo exclusivamente partes de la función forces, ya que es la función que más tiempo ocupa en la corrida. Utilizando el código en formato SoA junto a la vectorización con ISPC se logra que los GFLOPS crezcan más del 50 % respecto a los valores obtenidos cuando el código se encontraba en formato AoS y sin vectorizar. Asimismo se logran simular hasta 6912 partículas en el mismo tiempo de corrida disponible.

Por último, se definió una nueva métrica, basada en la energía potencial, que sirve de control para saber que el código no ha cambiado en su esencia: sigue siendo un código para una simulación de dinámica molecular.