



# Simulación de N Cuerpos Computacionales sobre Intel Xeon Phi KNL

Enzo Rucci<sup>1</sup> , Ezequiel Moreno<sup>2</sup>, Malek Camilo<sup>2</sup>, Adrián Pousa<sup>1</sup> and Franco Chichizola<sup>1</sup> 

<sup>1</sup> III-LIDI, Facultad de Informática, UNLP – CEA CICPBA. La Plata (1900), Bs As, Argentina  
{erucci, apousa, francoch}@lidi.info.unlp.edu.ar

<sup>2</sup> Facultad de Informática, UNLP. La Plata (1900), Bs As, Argentina  
{morenoezequiel8, malek.camilo}@gmail.com

**Resumen.** En la comunidad HPC, el uso de aceleradores se ha consolidado como estrategia para mejorar el rendimiento de los sistemas al mismo tiempo que la eficiencia energética. Recientemente, Intel introdujo Knights Landing (KNL), la segunda generación de aceleradores Xeon Phi. Entre sus características destacadas, se puede mencionar su gran cantidad de núcleos, la incorporación de las instrucciones vectoriales AVX-512 y la integración de una memoria de alto ancho de banda. Este trabajo se enfoca en la paralelización de la simulación de N cuerpos computacionales sobre un acelerador Xeon Phi KNL. Además de representar la base de un gran número de aplicaciones de la astrofísica, esta simulación requiere de alto poder computacional para ser procesada con un tiempo de respuesta aceptable. Comenzando por una implementación secuencial, se muestra cómo es posible que la implementación paralela alcance 2355 GFLOPS a través de diferentes optimizaciones.

**Keywords:** Xeon Phi, Knights Landing, N body, AVX-512, MCDRAM.

## 1 Introducción

En la actualidad, el problema del consumo energético se presenta como uno de los mayores obstáculos para el diseño de sistemas de cómputo de alto rendimiento (HPC) que sean capaces de alcanzar la escala de los Exaflops. En ese sentido, el uso de aceleradores (como pueden ser las GPUs de NVIDIA o los procesadores Xeon Phi de Intel) se ha consolidado como estrategia para mejorar la eficiencia energética. El factor clave en estos dispositivos radica en que ofrecen un pico de rendimiento muy superior al de las CPUs al mismo tiempo que limitan el consumo de potencia, lo que les permiten obtener mejores cocientes FLOPS/Watt [1].

Recientemente, Intel ha presentado la segunda generación de sus procesadores Xeon Phi, con nombre clave Knights Landing (KNL). Entre sus principales características, se pueden mencionar la gran cantidad de núcleos con soporte para *hyper-threading*, la incorporación de las instrucciones vectoriales AVX-512 y la integración de una memoria de alto ancho de banda [2].

Entre las áreas que se ven afectadas por los problemas actuales de los sistemas HPC se encuentra la física, debido a que cuenta con un número creciente de aplicaciones que requieren de cómputo de altas prestaciones para alcanzar tiempos de respuesta aceptables. Una de esas aplicaciones es el clásico problema de la simulación de  $N$  cuerpos computacionales, la cual aproxima en forma numérica la evolución de un sistema de cuerpos en el que cada uno interactúa con todos los restantes [3].

El uso más conocido de esta simulación quizás sea en la astrofísica, donde cada cuerpo representa una galaxia o una estrella particular que se atraen entre sí debido a la fuerza gravitacional. Sin embargo, también se ha empleado en otras áreas muy diferentes. Por ejemplo, para el plegado de proteínas en la biología computacional [4] o para la iluminación global de una imagen en computación gráfica [5].

Existen diferentes métodos para computar la simulación de los  $N$  cuerpos [6]. La forma más sencilla se denomina de *all-pairs* (o directa) y consiste en evaluar todas las interacciones entre todos los pares de cuerpos. Es un método de fuerza bruta que posee alta demanda computacional ( $O(n^2)$ ). Debido a su complejidad, la versión directa sólo es empleada cuando la cantidad de cuerpos es moderada, o bien para computar las interacciones entre cuerpos cercanos en combinación con una estrategia para los que se encuentran lejanos entre sí. Esta segunda opción es el enfoque empleado por métodos avanzados que permiten simular la interacción entre una gran cantidad de cuerpos, como pueden ser el de Barnes-Hut ( $O(n \log n)$ ) o el Fast Multipole Method ( $O(n)$ ). Por lo tanto, al acelerar la versión directa no solo se mejora a la misma sino también a las otras que la emplean como componente.

Este trabajo se enfoca en la paralelización de la simulación de  $N$  cuerpos computacionales sobre un acelerador Intel Xeon Phi KNL. Comenzando por una implementación secuencial, se muestra cómo es posible que la implementación paralela alcance 2355 GFLOPS a través de diferentes técnicas de optimización.

El resto del artículo se organiza de la siguiente forma. La Sección 2 introduce el problema de los  $N$  cuerpos computacionales mientras que la Sección 3 describe la arquitectura del Intel Xeon Phi KNL. La Sección 4 detalla las implementaciones realizadas. En la Sección 5 se analizan los resultados experimentales mientras que en la Sección 6 se discuten los trabajos relacionales. Finalmente, la Sección 7 resume las conclusiones junto a los posibles trabajos futuros.

## 2 Simulación de $N$ Cuerpos Computacionales

El problema consiste en simular la evolución de un sistema compuesto por  $N$  cuerpos durante una cantidad de tiempo determinada. Dada la masa y el estado inicial (velocidad y posición) de cada cuerpo, se simula el movimiento del sistema a través de instantes discretos de tiempo. En cada uno de ellos, todo cuerpo experimenta una aceleración que surge de la atracción gravitacional del resto, lo que afecta a su estado.

La física subyacente a la simulación es fundamentalmente la mecánica Newtoniana [7]. La simulación se realiza en 3 dimensiones espaciales y la atracción gravitacional entre dos cuerpos  $C_1$  y  $C_2$  se computa usando la ley de gravitación universal de Newton:

$$F = \frac{G \times m_1 \times m_2}{r^2}$$

donde  $F$  corresponde a la magnitud de la fuerza gravitacional entre los cuerpos,  $G$  corresponde a la constante de gravitación universal<sup>1</sup>,  $m_1$  corresponde a la masa del cuerpo  $C_1$ ,  $m_2$  corresponde a la masa del cuerpo  $C_2$ , y  $r$  corresponde a la distancia Euclídea<sup>2</sup> entre los cuerpos  $C_1$  y  $C_2$ .

Cuando  $N$  es mayor a 2, la fuerza de gravitación sobre un cuerpo, se obtiene con la sumatoria de todas las fuerzas de gravitación ejercidas por los  $N-1$  cuerpos restantes. La fuerza de atracción se traduce entonces en una aceleración del cuerpo mediante la aplicación de la segunda ley de Newton, la cual está dada por la siguiente ecuación:

$$F = m \times a$$

donde  $F$  es el vector fuerza, calculado utilizando la magnitud obtenida con la ecuación de gravitación y la dirección y sentido del vector que va desde el cuerpo afectado hacia el cuerpo que ejerce la atracción.

De la ecuación anterior, se despeja que se puede calcular la aceleración de un cuerpo dividiendo la fuerza total por su masa. Durante un pequeño intervalo de tiempo  $dt$ , la aceleración  $a_i$  del cuerpo  $C_i$  es aproximadamente constante, por lo que el cambio en velocidad es aproximadamente:

$$dv_i = a_i dt$$

El cambio en la posición de un cuerpo es la integral de su velocidad y aceleración sobre el intervalo de tiempo  $dt$ , el cual es aproximadamente

$$dp_i = v_i dt + \frac{a_i}{2} dt^2 = \left( v_i + \frac{dv_i}{2} \right) dt$$

Esta fórmula emplea el esquema de integración Leapfrog [8], en el cual una mitad del cambio de posición emplea la velocidad *vieja* mientras que la otra considera la velocidad *nueva*.

### 3 Intel Xeon Phi KNL

KNL es la segunda generación de aceleradores Xeon Phi y la primera capaz de funcionar en forma autónoma (no requiere de un *host*). El diseño de su arquitectura se basa en el concepto de *tile* (unidad básica de replicación), donde cada uno cuenta con dos núcleos y una caché L2 de 1MB compartida entre ellos. Los *tiles* se encuentran físicamente replicados hasta 38 veces<sup>3</sup>, estando interconectados por una malla 2D con

<sup>1</sup> Equivalente a  $6,674 \times 10^{-11}$

<sup>2</sup> Se calcula utilizando la fórmula  $\sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2}$ , siendo  $(x_1, y_1, z_1)$  las coordenadas de  $C_1$  y  $(x_2, y_2, z_2)$  las coordenadas de  $C_2$ .

<sup>3</sup> Aunque sólo 36 de ellos pueden estar activos al mismo tiempo

coherencia de caché. A nivel de núcleo, la micro-arquitectura se basa en la de los Intel Atom con 4 hilos hw y 2 unidades de procesamiento vectorial (VPU). Estas VPU introducen las nuevas instrucciones Intel AVX-512 de 512 bits, además de dar soporte a las ya conocidas SSEx (de 128 bits) y AVXx (de 256 bits). Como una instrucción AVX-512 puede realizar 8 operaciones de suma/multiplicación con operandos de doble precisión (DP) o 16 operaciones con operandos de simple precisión (SP), el pico de rendimiento se encuentra por encima de 3(6) TFLOPS en DP(SP) [2].

Además de una memoria DDR4 convencional, KNL cuenta con una memoria de alto ancho de banda denominada MCDRAM. Esta memoria puede ser configurada en uno de entre tres modos (1) modo *Cache*, donde se la utiliza como memoria cache L3 (transparente al programador); (2) modo *Flat*, donde es tratada como una memoria direccionable con alto ancho de banda y baja latencia; y (3) modo *Híbrido*, donde una parte es utilizada en modo *Cache* y el resto en modo *Flat*. Si bien el modo *Flat* permite alcanzar un mayor rendimiento, requiere de intervención del programador.

Por su parte, la malla 2D configurable le permite a KNL ofrecer tres modos de operación de cluster diferentes: (1) *All-to-all*; (2) *Quadrant*; y (3) *sub-NUMA*. La diferencia principal entre estos modos radica en si los núcleos tendrán acceso UMA o NUMA a una memoria particular.

Desde el punto de vista de la programación, KNL soporta modelos tradicionales en HPC como MPI y OpenMP, lo que representa una ventaja frente a otros aceleradores que requieren del aprendizaje de lenguajes específicos como CUDA u OpenCL. Sin embargo, para obtener alto rendimiento, resulta necesario que los programas sean capaces de explotar eficientemente la jerarquía de memoria, además de las capacidades vectoriales de los núcleos [9].

## 4 Implementación

En esta sección se describen las optimizaciones realizadas a la implementación para el Xeon Phi KNL.

### 4.1 Implementación *Naive*

Inicialmente se desarrolló una implementación *naive*, la cual servirá como referencia para evaluar las mejoras introducidas por las técnicas de optimización posteriores. En la Fig. 1 se muestra el pseudo-código de la implementación *naive*.

### 4.2 Multi-hilado

La primera optimización consiste en introducir paralelismo a nivel de hilos a través de directivas OpenMP. Los bucles de las líneas 4 y 26 son paralelizados mediante la inserción de directivas *parallel for*. De esta forma, se respetan las dependencias del problema ya que un cuerpo no se puede mover hasta que el resto no haya terminado de calcular sus interacciones y tampoco puede avanzar al paso siguiente hasta que los restantes no hayan completado el paso actual. Por último, la opción *static* para la

cláusula *schedule* permite distribuir equitativamente la cantidad de cuerpos entre los hilos logrando un balance de carga de costo mínimo.

```

1 // Para cada instante discreto de tiempo
2 for (t = 1; t <= D; t += DT){
3     // Para cada cuerpo que experimenta una fuerza
4     for (i = 0; i < N; i++){
5         // Componentes de la fuerza del cuerpo i
6         forcesx[i] = forcesy[i] = forcesz[i] = 0.0;
7         // Para cada cuerpo que ejerce una fuerza
8         for (j = 0; j < N; j++){
9             // Ley de atracción gravitacional de Newton
10            dx = xpos[j] - xpos[i]; dy = ypos[j] - ypos[i]; dz = zpos[j] - zpos[i];
11            dsquared = (dx*dx) + (dy*dy) + (dz*dz) + SOFT;
12            F = G * masses[i] * masses[j]; d32 = 1/POW(dsquared,1.5);
13            // Calcular la fuerza total
14            forcesx[i] += F*dx*d32; forcesy[i] += F*dy*d32; forcesz[i] += F*dz*d32;
15        }
16        // Calcular aceleración
17        ax = forcesx[i] / masses[i]; ay = forcesy[i] / masses[i]; az = forcesz[i] / masses[i];
18        // Calcular velocidad
19        dvx = (xvi[i] + (ax*DT*0.5)); dvy = (yvi[i] + (ay*DT*0.5)); dvz = (zvi[i] + (az*DT*0.5));
20        // Calcular posición
21        dpv[i] = dvx * DT; dpy[i] = dvy * DT; dpz[i] = dvz * DT;
22        // Actualizar velocidad
23        xvi[i] = dvx; yvi[i] = dvy; zvi[i] = dvz;
24    }
25    // Actualizar posiciones
26    for(i = 0; i < N; i++){
27        xpos[i] += dpv[i]; ypos[i] += dpy[i]; zpos[i] += dpz[i];
28    }
29 }

```

Fig. 1. Pseudo-código para la implementación *naive*

### 4.3 Vectorización

El reporte de optimización del compilador ICC permite identificar qué bucles son vectorizados en forma automática. A partir del mismo, se pudo saber que el compilador detecta dependencias falsas en algunas operaciones, imposibilitando la generación de instrucciones SIMD. En consecuencia, para garantizar la vectorización de operaciones se optó por un enfoque guiado a través del uso de la directiva *simd* de OpenMP 4.0. En particular, los bucles vectorizados son los de las líneas 8 y 26, siendo el último combinado con la directiva *parallel for*, como se mencionó en la sección anterior. Por último, para favorecer el uso de instrucciones SIMD, se alinearon los datos a 64-bytes en su alocaión, agregando la cláusula *aligned* a la directiva *simd*.

### 4.4 Procesamiento por Bloques

Para explotar la localidad de datos, es posible implementar un procesamiento por bloques. En la Fig. 2 se muestra el pseudo-código de la implementación paralela por bloques. El cambio con respecto a la implementación *naive*, consiste en desdoblar el bucle *i* (línea 4) y colocarlo dentro del bucle *j* (línea 8). En consecuencia, se reemplaza un bucle por otros dos: uno que itera sobre todos los bloques (línea 5) y uno más interno que itera sobre los cuerpos de cada bloque (línea 12). De esta manera se mi-

nimiza el tráfico entre cache y memoria principal al aumentar la cantidad de veces que se usa cada dato en el bucle más interno.

```

1 // Para cada instante discreto de tiempo
2 for (t = 1; t <= D; t += DT){
3 // Para cada bloque de cuerpos de tamaño BS
4 #pragma omp parallel for schedule(static) private(i,j)
5 for(ii = 0; ii < N; ii+=BS){
6 // Componentes de la fuerza del cuerpo i
7 forcesx[BS] = forcesy[BS] = forcesz[BS] = {0.0};
8 // Para cada cuerpo que ejerce una fuerza
9 for(j = 0; j < N; j++){
10 // Para cada cuerpo que experimenta una fuerza
11 #pragma omp simd aligned
12 for (i = ii; i < ii+BS; i++){
13 // Ley de atracción gravitacional de Newton
14 dx = xpos[j] - xpos[i]; dy = ypos[j] - ypos[i]; dz = zpos[j] - zpos[i];
15 dsquared = (dx*dx) + (dy*dy) + (dz*dz) + SOFT;
16 F = G * masses[i] * masses[j]; d32 = 1/POW(dsquared,1.5);
17 // Calcular la fuerza total
18 forcesx[i-ii] += F*dx*d32; forcesy[i-ii] += F*dy*d32; forcesz[i-ii] += F*dz*d32;
19 }
20 }
21 #pragma omp simd aligned
22 for (i = ii; i < ii+BS; i++){
23 // Calcular aceleración |M = F * A| ---> |A = F / M|
24 ax = forcesx[i-ii] / masses[i]; ay = forcesy[i-ii] / masses[i]; az = forcesz[i-ii] / masses[i];
25 // Calcular velocidad
26 dvx = (xvi[i] + (ax*DT*0.5)); dvy = (yvi[i] + (ay*DT*0.5)); dvz = (zvi[i] + (az*DT*0.5));
27 // Calcular posición
28 dpv[i] = dvx * DT; dpy[i] = dvy * DT; dpz[i] = dvz * DT;
29 // Actualizar velocidad
30 xvi[i] = dvx; yvi[i] = dvy; zvi[i] = dvz;
31 }
32 }
33 // Actualizar posiciones
34 #pragma omp parallel for simd aligned
35 for(int i = 0; i < N; i++){
36 xpos[i] += dpv[i]; ypos[i] += dpy[i]; zpos[i] += dpz[i];
37 }
38 }

```

Fig. 2. Pseudo-código para la implementación paralela por bloques

#### 4.5 Desenrollado de Bucles

El desenrollado de bucles es otra técnica de optimización que puede mejorar el rendimiento de un programa. En particular, se encontró beneficioso desenrollar completamente el bucle más interno de la implementación presentada en la Fig. 2 (línea 9), además del que actualiza las posiciones de los cuerpos posteriormente (línea 35).

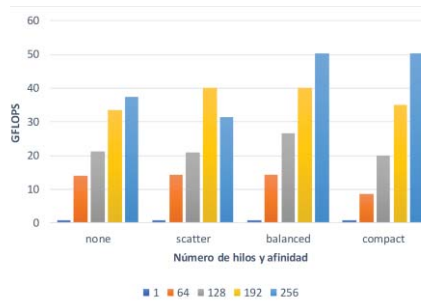
## 5 Resultados Experimentales

### 5.1 Diseño Experimental

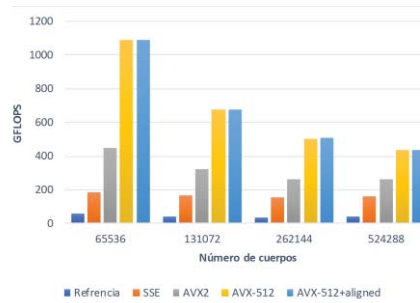
Todas las pruebas fueron realizadas en un sistema equipado con un Intel Xeon Phi 7230 de 64 núcleos (4 hilos hw por núcleo), 192 GB de memoria RAM y 16 GB de

memoria MCDRAM. En ese sentido, el procesador fue usado en modo cluster *All-to-all* y la memoria MCDRAM en modo *Flat*<sup>4</sup>.

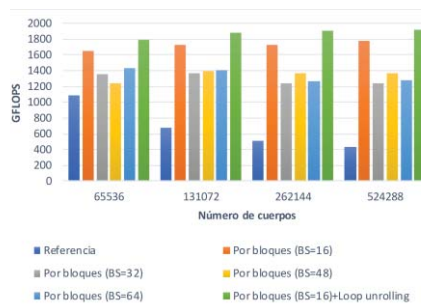
Respecto al software, el sistema operativo es Ubuntu 16.04.3 LTS mientras que el compilador es el ICC (versión 19.0.0.117). Para acelerar el cómputo de operaciones en punto flotante se usó el *flag -fp-mode fast=2* mientras que para utilizar las instrucciones vectoriales AVX2 y AVX-512, se emplearon los *flags -xAVX2* y *-xMIC-AVX512*, respectivamente. Además, se empleó el comando *numactl* para poder explotar la memoria MCDRAM (no requiere modificaciones al código fuente). Por último, se varió la carga de trabajo al usar diferentes números de cuerpos:  $N = \{65536, 131072, 262144, 524288, 1048576\}$ <sup>5</sup>.



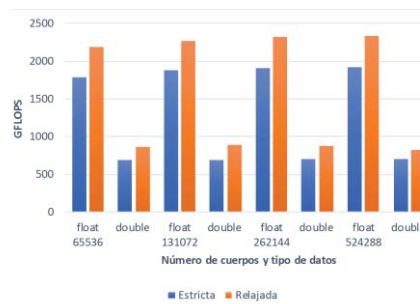
**Fig. 3.** Rendimientos obtenidos para diferentes tipos de afinidad y número de hilos cuando  $N=65536$ .



**Fig. 4.** Rendimientos obtenidos para diferentes niveles de vectorización al variar el número de cuerpos ( $N$ ).



**Fig. 5.** Rendimientos obtenidos para la técnica de bloques y desenrollado de bucles al variar la carga de trabajo ( $N$ ).



**Fig. 6.** Rendimientos obtenidos para la relajación de precisión al variar el tipo de dato y la carga de trabajo ( $N$ ).

<sup>4</sup> Debido a que los módulos de memoria DDR tienen diferente tamaño, no fue posible configurar el procesador en un modo diferente.

<sup>5</sup> El número de pasos de simulación se mantuvo fijo ( $I=100$ ).

## 5.2 Resultados de Rendimiento

Para evaluar el rendimiento se emplea la métrica GFLOPS (mil millones de FLOPS), utilizando la fórmula  $GFLOPS = \frac{20 \times N^2 \times I}{T \times 10^9}$ , donde  $N$  es el número de cuerpos,  $I$  es el número de pasos,  $T$  es el tiempo de ejecución (en segundos) y el factor 20 representa la cantidad de operaciones en punto flotante requerida por cada interacción<sup>6</sup>.

En la Fig. 3 se pueden ver los rendimientos al variar el tipo de afinidad y el número de hilos cuando  $N=65536$ . Al emplear paralelismo a nivel de hilos, el rendimiento mejora considerablemente, notando que un mayor número de hilos lleva a mejores prestaciones (excepto con afinidad *scatter*). Respecto a la afinidad, se puede apreciar que resulta conveniente elegir alguna de las estrategias disponibles en lugar de delegar la distribución en el sistema operativo (*none*). A diferencia de *scatter*, *balanced* y *compact* garantizan la contigüidad de hilos OpenMP con identificadores consecutivos, lo que favorece a la comunicación de los datos que cada uno requiere<sup>7</sup>.

Como se mencionó en la Sección 4.3, el compilador no es capaz de vectorizar todas las operaciones por su cuenta. Se puede notar en la Fig. 4 que, al forzar la vectorización de operaciones, se produce una mejora de aproximadamente  $3.9\times$ . Al agregar los flags `-xAVX2` y `-xMIC-AVX512`, el compilador emplea las instrucciones AVX2 y AVX-512, respectivamente. Como estas extensiones tienen mayor ancho vectorial, el rendimiento se incrementa aún más, logrando mejoras de  $7.4\times$  para AVX2 y de  $15.1\times$  para AVX-512. Por lo tanto, resulta claro que este problema se beneficia de instrucciones vectoriales más anchas. Adicionalmente, no se observan mejoras significativas por el alineamiento de datos a memoria.

Como se puede apreciar en la Fig. 5, la técnica de procesamiento por bloques aumenta considerablemente el rendimiento, siendo mayor la ganancia a medida que la cantidad de cuerpos crece. En particular, se logra una mejora promedio de  $2.9\times$  y una máxima de  $4.1\times$  ( $BS=16$ ). En forma adicional, las prestaciones mejoran aproximadamente un 9% al desarrollar los bucles mencionados en la sección 4.5.

En la Fig. 6 se muestran los rendimientos obtenidos para la relajación de precisión al variar el tipo de dato y la carga de trabajo ( $N$ ). Se puede notar que las prestaciones mejoran un 22% en promedio al aplicar la optimización del



Fig. 7. Rendimientos obtenidos para la explotación de MCDRAM al variar la carga de trabajo ( $N$ ).

<sup>6</sup> Convención ampliamente aceptada en la literatura disponible sobre este problema.

<sup>7</sup> Como todos los núcleos del procesador se encuentran en el mismo paquete, *balanced* y *compact* producen la misma asignación cuando se emplean 4 hilos por núcleo.



compilador para este fin. En sentido contrario, el rendimiento decae aproximadamente un 60% al duplicar la precisión numérica usando el tipo de dato *double*.

Por último, a partir de la Fig.7, se puede notar que el rendimiento se mantiene (casi) constante al aumentar el número de cuerpos, obteniendo como máximo 2355 GFLOPS. También se puede apreciar una pequeña mejora cercana al 2% por el uso de la memoria MCDRAM. Este resultado es similar al observado en [10], estando relacionado con el hecho de que el rendimiento de esta aplicación se ve más influenciado por la latencia de la memoria que por el ancho de banda<sup>8</sup>.

## 6 Trabajos Relacionados

La aceleración de la simulación de N cuerpos ha sido ampliamente estudiada en la literatura. Sin embargo, pocos trabajos lo hacen sobre la arquitectura Xeon Phi, usando en su mayoría la primera generación de esta familia (KNC) [6] [11] [12]. Con respecto a KNL, sólo se puede mencionar el trabajo [10], el cual presenta algunas similitudes y diferencias con la presente investigación. Al igual que en este artículo, los autores estudian la paralelización de la versión directa de la simulación mostrando las mejoras obtenidas a través de diferentes optimizaciones, aunque la implementación final alcanza un pico de rendimiento mayor (2875 GFLOPS). A diferencia de esta investigación, el trabajo prioriza las optimizaciones para KNL, haciendo algunas simplificaciones en el cálculo de la simulación, como emplear un mecanismo de integración más sencillo (menos operaciones) y sólo computar un único paso en el tiempo. Al usar la misma métrica de evaluación<sup>9</sup>, sobreestiman la cantidad de FLOPS obtenidos. En relación con el análisis de prestaciones, en este artículo se consideraron varios aspectos adicionales como la cantidad y la afinidad de los hilos, el uso de todos los conjuntos de instrucciones vectoriales del KNL, la búsqueda del tamaño de bloque óptimo, el impacto del aumento de precisión por el uso del tipo *double*, además de la variación en la cantidad de cuerpos.

## 7 Conclusiones y Trabajos Futuros

Este trabajo se enfoca en la paralelización de la simulación de N cuerpos computacionales sobre un acelerador Intel Xeon Phi KNL. Comenzando por una implementación secuencial, se mostró cómo es posible que la implementación paralela alcance 2355 GFLOPS a través de diferentes técnicas de optimización. Entre las principales conclusiones de esta investigación se pueden mencionar:

- En general, un número mayor de hilos llevó a un mejor rendimiento. Respecto a la afinidad, se encontraron diferencias significativas entre las distintas opciones, por lo que es un factor que no se debe omitir al momento de ejecutar una aplicación paralela.

---

<sup>8</sup> La latencia de la memoria DDR4 es similar a la de la MCDRAM.

<sup>9</sup> También asumen que se realizan 20 operaciones de punto flotante por interacción pero el número real es inferior por usar un esquema de integración más simple (menos operaciones).

- La vectorización de operaciones representó un factor fundamental para la mejora de rendimiento. En ese sentido, se lograron aceleraciones cercanas al número de operaciones simultáneas que cada repertorio SIMD permite, a un bajo costo de programación.
- La explotación de la localidad de datos mediante el procesamiento por bloques resultó otro aspecto clave para obtener alto desempeño. No sólo permitió incrementar los FLOPS obtenidos en cada caso sino también que el rendimiento escale al aumentar la cantidad de cuerpos.
- Si la precisión en el resultado final no es una prioridad, el compilador puede ofrecer mejoras significativas en el rendimiento. En sentido contrario, duplicar la precisión puede reducir el rendimiento por debajo de la mitad.
- El uso de la memoria MCDRAM puede no proveer mejoras significativas en el rendimiento, especialmente cuando la aplicación no tiene una alta demanda de ancho de banda.

Entre los trabajos futuros, se pueden mencionar dos posibles líneas de investigación:

- Considerando los resultados obtenidos, se espera avanzar en la implementación de métodos avanzados para esta simulación.
- Dado que las GPUs son el acelerador dominante en la actualidad, interesa realizar una comparación de rendimiento y eficiencia energética entre estas arquitecturas.

## Referencias

- [1] M. B. Giles y I. Reguly, «Trends in HPC for engineering calculations,» *Phil. Trans. of the Royal Soc. of London A: Mathematical, Physical and Engineering Sciences*, vol. 372, n° 2022, 2014.
- [2] J. Reinders, J. Jeffers y A. Sodani, Intel Xeon Phi Processor High Performance Programming Knights Landing Edition, Boston, MA, USA: Morgan Kaufmann , 2016.
- [3] G. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley, 2000.
- [4] P. L. Freddolino, C. B. Harrison, Y. Liu y K. Schulten, «Challenges in protein-folding simulations,» *Nature Physics*, vol. 6, pp. 751-758, 2010.
- [5] R. Goradia, «Global Illumination for Point Models,» 2008. [En línea]. Disponible en: <https://tinyurl.com/y5nqel39>.
- [6] R. Yokota y M. Abduljabbar, «N-Body methods,» de *High Performance Parallelism Pearls - Multicore and Many-core Programming Approaches*, Morgan-Kaufmann, 2015, pp. 175-183.
- [7] P. Tipler, Physics for Scientists and Engineers: Mechanics, Oscillations and Waves, Thermodynamics, Freeman & Co, 2004.
- [8] P. Young, «The leapfrog method and other "symplectic" algorithms for integrating Newton's laws of motion,» 21 04 2014. [En línea]. Disponible en:

- <https://young.physics.ucsc.edu/115/leapfrog.pdf>. [Último acceso: 23 07 2019].
- [9] E. Rucci, M. Naiouf y A. D. Giusti, «Blocked All-pairs Shortest Paths Algorithm on Intel Xeon Phi KNL Processor: A Case Study,» de *Computer Science - CACIC 2017*, La Plata, Argentina, Springer, 2018, pp. 47-57.
  - [10] A. Nikolaev y R. Asai, «N-Body simulation,» de *Intel Xeon Phi Processor HPC - KNL edition*, Morgan Kaufmann, 2016, p. 638.
  - [11] A. Vladimirov y V. Karpusenko, «Test-driving Intel Xeon Phi coprocessors with a basic N-Body simulation,» 2013. Reporte técnico Colfax Research International. Disponible en: <https://tinyurl.com/y5vtj34a>
  - [12] B. Lange y P. Fortin, «Parallel and dual tree traversal on multi-core and many-core architectures for astrophysical N-body simulations,» de *Euro-Par 2014 Parallel Processing. Lecture Notes in Computer Science, vol 8632*, 2014, pp. 716-727.