Universidad Nacional de Córdoba

Comparación de arquitectura AMD e Intel para ejecución de GADGET

Autor: Ruben Dario Graña

Directores: Federico Stasyszyn, Nicolás Wolovick

Córdoba, Argentina 2017

Índice general

Agradecimientos

Quiero dedicar este trabajo a mi familia, especialmente Nelly y Ruben, quienes estuvieron a mi lado y alentándome a lo largo de toda la carrera.
Le agradezco a todos mi amigos, quienes de las más diversas formas me acompañaron a llegar hasta aquí.
También quiero agradecer a todos los docentes que intervinieron en mi formación en el correr estos años y que, en aquellos momentos de desánimo, me apuntalaron para que continuara, me aconsejaron y ayudaron a continuar.
Un especial agradecimiento a todo el equipo humano del IATE, quienes de una u otra forma estuvieron pendientes de que este proyecto llegara a buen puerto.
Finalmente no quiero dejar fuera de este agradecimiento a todo el personal nodocente del departamento de alumnos y la biblioteca, quienes siempre estuvieron dispuestos a ayudar y resolver cualquier duda o problema con lo que me encontrara con un trato personal excepcional.

Resumen

El trabajo que se presenta a continuación fue realizado en la Universidad Nacional de Córdoba, en el marco de la Facultad de Matemática, Astronomía, Física y Computación, y el Instituto de Astronomía Teórica y Experimental dependiente también del CONICET. Dentro de dicho instituto se trabaja con simulaciones numéricas, gran parte de las cuales son generadas localmente, por lo que resulta de vital importancia conocer que es lo más conveniente al momento de adquirir nuevo equipamiento.

Por ello que este trabajo nace de la necesidad de determinar que arquitectura de computadora es la más apropiada al momento de ejecutar Gadget, un código ampliamente utilizado en astronomía al momento de realizar simulaciones cosmológicas. Desde el punto de vista de la computación también resulta muy interesante estudiar como trabajan las distintas arquitecturas frente al mismo tipo de carga, una compleja en este caso, cómo impacta en el desempeño la utilización de distintos compiladores y banderas de compilación.

A lo largo del mismo se presenta el código, su historia y componentes. A su vez se hace una breve introducción de lo que ha cambiado en los procesadores en los últimos 30 años. Luego se presentan y caracterizan los equipos que fueron utilizados. La caracterización incluye los cálculos teóricos y las pruebas realizadas para comprobar los valores teóricos. Finalmente se muestran distintas comparativas de ejecuciones de Gadget realizadas con diversos parámetros, compiladores y configuraciones del hardware, junto a otras estadísticas obtenidas.

Glosario

-O0 Desahbilita todo tipo de optimización por parte del compilador.. 37

-O3 Activa todas las opciones de optimización ajustadas a los estándar. 34

-ip Genera optimizaciones dentro de archivos individuales y símbolos para debugging. 34

-march Activa todas las optimizaciones específicas a la arquitectura indicada, pero no necesariamente en arquitecturas anteriores. 34

-mprefer-avx128 Instruye al compilador que utilice instrucciones AVX de 128 bits en lugar de 256 bits en la vectorización. 40

-mtune El compilador generará código que podrá ser ejecutado en todas las arquitecturas, pero favoreciendo secuencias de instrucciones que se ejecuten más velozmente sobre la arquitectura indicada. 34

-xHost Instruye al compilador para generar código específico a la arquitectura indicada. 34

Capítulo 1

Descripción de GADGET

Gadget [22] es un código para simulaciones cosmológicas N-Body/SPH de libre acceso preparado para ejecutarse en gran cantidad de núcleos de computadoras con memoria distribuida; utiliza un modelo de comunicaciones explícito a través de MPI y está preparado para ser compilado y ejecutado sobre el hardware de diversas arquitecturas como x86 o PowerPC, partiendo de una máquina de escritorio hasta un cluster con miles de núcleos.

Éste código realiza el cálculo de fuerzas gravitacionales con un algoritmo de árbol jerárquico, que brinda la opción de combinarlo con una malla para las fuerzas gravitaciones en grandes escalas. Además tiene en cuenta la acción de fluídos hidrodinámicos a través de métodos de partículas suavizadas, SPH, por sus siglas en inglés para *smoothed-particle hydrodynamics*.

El código puede ser utilizado para estudios de sistemas que incluyan o no la expansión cosmológica del universo, con o sin condiciones periódicas de contorno e inclusive con condiciones de contorno arbitrarias (i.e. vacío, tidales específicas, etc). En todo este tipo de simulaciones Gadget sigue la evolución de un sistema de N-cuerpos que gravita sobre sí mismo sin colisiones y permite que sean incluidas opcionalmente las dinámicas de gases. Tanto la integración temporal de la gravedad como de la hidrodinámica se realizan de forma simpléctica (de forma ideal), con pasos totalmente adaptativos permitiendo un rango dinámico que es, en principio, limitado a la precisión de la máquina utilizada.

1.1. Autor e Historia

Gadget fue escrito por Volker Springel en los últimos 17 años. La primera versión, presentada en marzo de 2000, fue escrita como parte de la tesis doctoral de Springel en el instituto de astrofísica Max Planck, Garching,

Alemania, bajo la supervisión de Simon White. Con el correr de los años el código ha sido mejorado durante el post-doctorado de Springel en CfA, *Harvard-Smithsonian Center for Astrophysics* [10] y el MPA, *Max-Planck-Institut für Astrophysik* [8], nuevamente en colaboración con Simon White y Lars Hernquist. La segunda versión pública, presentada en mayo de 2005, contiene gran parte de estas mejoras, excepto los numerosos módulos de física desarrollados para el código que van más allá de gravedad y dinámica de gases comunes. Las primeras versiones del código, en el año 1998, fueron diseñadas casi de manera específica para ejecución serial con el objetivo de atacar principalmente problemas de colisión de interacción galaxias, lo que explica en parte el porqué de su nombre, traducido del inglés pequeña herramienta. Las siglas corresponden a ***Ga****laxies with* ***D****ark matter and* ***G****as Int****e****rac****t***

La versión para cómputo paralelo de código de árbol, en inglés *Tree Code*, también fue desarrollada durante el año 1998 junto con la primera versión en paralelo de SPH agregada a finales de 1999. Para la última parte, Naoki Yoshida, por aquel entonces otro estudiante de pos-grado en el MPA, se sumó a los esfuerzos de Springel para el desarrollo y prueba de los algoritmos paralelos de SPH. Entre los años 2001 y 2003, fue creado Gadget-2. El nuevo código fue prácticamente una reescritura completa de la versión original, incluyendo reemplazos para todo los algoritmos centrales con nuevos métodos. Los cambios más importantes yacen en una nueva integración temporal, un nuevo módulo del código de árbol, un nuevo esquema de comunicaciones para fuerzas gravitacionales y SPH, una nueva estrategia de descomposición de dominio, una nueva formulación de SPH basada en entropía (y no energía interna) como una variable independiente y finalmente en la adición de la funcionalidad TreePM.

Volker Springel, junto a su equipo, se encuentra en desarrollo de la tercera versión del código. La última versión de Gadget mejora la búsqueda, construcción, actualización y generación de las claves de Peano-Hilbert. Asimismo, mejora la descomposición de dominios, permitiendo tener nodos con partículas no adyacentes y esto ayuda a mejorar el balance de cómputo llegando a escalar en supercomputadoras de más de 1000 núcleos eficientemente. Mejora el manejo de memoria y la caminata del árbol, eliminando cuellos de botella debido a la paralelización. Estas son algunas de las mejoras, que hacen que prefiera emplear ésta versión del código para los estudios realizados en el IATE, de todos modos si bien aún no se ha presentado una versión pública, se espera pueda ser liberada pronto.

1.2. N-Body

Muchos fenómenos físicos pueden, directa o indirectamente, ser simulados con sistemas de partículas, donde cada partícula interactúa con todo el resto de acuerdo a las leyes de la física. Algunos ejemplos pueden ser la interacción gravitacional entre las estrellas de una galaxia o las fuerzas de Coulomb ejercidas por los átomos de una molécula. El desafío de llevar a cabo de manera eficiente todos estos cálculos es conocido como el problema de N-cuerpos [12]. Matemáticamente puede ser formulado como

$$U(x_0) = \sum_i F(x_0, x_i) \tag{1.1}$$

donde $U(x_0)$ es una cantidad física en x_0 que puede ser obtenida sumando las interacciones sobre las partículas tomadas de a pares, para el caso asumimos un sistemas compuesto por N partículas, ubicadas en x_i y con una masa m_i. La fuerzas gravitacionales ejercidas sobre una partícula x que posee una masa m es expresada como

$$F(x) = \sum_{i=1}^{N} Gmm_i \frac{x - x_i}{|x - x_i|^3} \tag{1.2}$$

donde G es la constante gravitacional y m la masa de la partícula x.

La tarea de evaluar la función $U(x_0)$ para las N partículas usando 1.1 requiere $O(n)$ operaciones para cada partícula, resultando en una complejidad total $O(n^2)$. Esta complejidad puede ser reducida a $O(n \log(n))$ o $O(n)$ mediante el uso de métodos eficientes para aproximar la suma del término de la derecha de la ecuación 1.1, mientras se preservan propiedades físicas importantes como la energía y el momento.

1.3. El método de partícula-partícula

La metodología consistente en evaluar por fuerza bruta el término derecho de la ecuación 1.1 se vuelve impracticable para una gran cantidad de partículas ya que tiene una complejidad $O(n^2)$, pero para una cantidad pequeña puede ser un método efectivo que acabará en un valor exacto, donde la precisión del cálculo estará limitada por la capacidad de la máquina utilizada.

Es aquí donde aparecen los algoritmos jerárquicos de cálculo de fuerzas, que vienen a reducir la brecha entre los métodos de fuerza bruta y aquellos que realizan aproximaciones. Todos los métodos jerárquicos realizan un particionado de la masa distribuyéndola en una estructura de árbol, donde cada

nodo tiene una descripción concisa de la distribución de materia en algún volumen. En este caso se utiliza el algoritmo de Barnes & Hut de 1986 [3], de aquí en más BH86, que ha sido ámpliamente utilizado en simulaciones de astrofísica a lo largo de los últimos años.

1.4. El código de árbol (Tree Code)

La estrategia de este algoritmo consiste en reducir la sobrecarga que genera la búsqueda de vecinos a través de un árbol asumiendo que los cuerpos cercanos tienen una lista similar de interacciones. Las fuerzas sobre los cuerpos son calculadas durante un recorrido del árbol completo y almacenado por niveles. Este proceso mantiene y actualiza la lista de interacciones; puede verse que en cada nivel, un cuerpo b en una celda c es usado para separar el conjunto de posibles interacciones que b puede tener. El costo de este ordenamiento es distribuido sobre todo los cuerpos que se encuentran en la celda, posibilitando una aceleración en el código que construye la lista de interacciones para cada cuerpo. Cuando el recorrido recursivo llega al cuerpo b, la lista de interacciones es utilizada para obtener la fuerza gravitacional y el potencial en b. Una forma práctica de entenderlo, consiste en visualizar la interacción entre un satélite y el planeta tierra, en este caso no se calcula la fuerza que ejerce cada átomo sobre el satélite, sino que se hace una estimación de la fuerza total que ejercería una partícula con la masa del planeta tierra y con ese resultado se realiza el cálculo de la fuerza. En este caso la partícula se encontraría en el nodo del árbol que reúne a todos los átomos, por lo que cualquier interacción que se desee calcular con la tierra, se computará a través de dicho nodo.

Típicamente los algoritmos de N-Cuerpos utilizan aproximadamente la mitad del tiempo en búsquedas dentro del árbol. Podría esperarse una ganancia de un factor dos generando listas de interacción para todos los cuerpos en un solo recorrido del árbol. En la práctica las listas generadas de esta manera no son específicas a cada partícula, deben ser más grandes para obtener una buena aproximación. Una aceleración en la ejecución del código se obtiene a través de una evaluación más rápida de las interacciones, la cual es alcanzada realizando ajustes en el algoritmo recorriendo el árbol. Las pruebas preliminares muestran la mejora en el tiempo de ejecución, reduciéndolo a la mitad sobre el utilizado en BH86 a un nivel de precisión determinado. Además muestra mayor robustez cuando se presentan ciertas distribuciones patológicas de masa. En procesadores escalares la mayor parte del tiempo se utiliza en la creación de las listas de interacción, una gran ventaja es ejecutar este tipo de acciones en unidades vectoriales, ya que permiten la aplicación

de la misma operación sobre múltiples datos a la vez.

1.5. Malla de partículas (Particle Mesh)

Este algoritmo guarda algunas similitudes con los anteriormente vistos. Nuevamente se realiza una división del espacio, armando una grilla, en inglés *Mesh*, pudiendo luego calcular una transformada de Fourier sobre la misma. Al realizar esta transformada deja de trabajarse en el espacio real para hacerlo con frecuencias, conocido como espacio de Fourier, lo cual simplifica los cálculos, teniendo en cuenta por ejemplo, una derivada en espacio real es simplemente una multiplicación en espacio de Fourier, teniendo luego que anti-transformar los resultados para ser utilizados en el espacio real. Es en este punto donde se utilizan bibliotecas específicas para la transformación de un espacio a otro. Una vez calculada la masa y aceleración para cada región del espacio se puede interpolar a las posiciones de las partículas y añadir los cálculos de las interacciones entre ellas, para obtener la fuerza aplicada sobre la misma.

1.6. Partición de dominio

Unos de los puntos que tuvo que ser mejorado a lo largo del desarrollo de Gadget es la partición de dominio [23], es decir, distribuir las partículas desde su posición original de forma tal que la carga de los procesadores sea pareja y no se produzca desbalanceo, quedando algunos procesadores sin carga y otros con exceso. En la imagen 1.1 podemos ver como se ubican las partículas de un disco exponencial. La distribución no homogénea de partículas y los distintos pasos de tiempo como función de densidad convierten en un desafío importante obtener un ordenamiento de partículas que tenga buen balance de carga de procesadores y memoria.

En la primera versión de Gadget se contaba con una bisección recursiva ortogonal simple de dominio. Cabe recordar que la primera versión de Gadget era de ejecución serial y en su versión en paralelo se asignaba cada una de las regiones a un procesador, por lo tanto con la evolución en la ejecución, se produce un desbalance imposible de manejar, ya que es estático.

En la segunda versión de Gadget se logró desarrollar otra función de descomposición, una curva de Peano-Hilbert [16] que resulta más flexible a la hora de particionar el dominio. Esta decisión también tiene una justificación desde el punto de vista de ejecución del código, cuando se comenzó a desarrollar no se contaba con conectividad de alta velocidad de baja latencia como

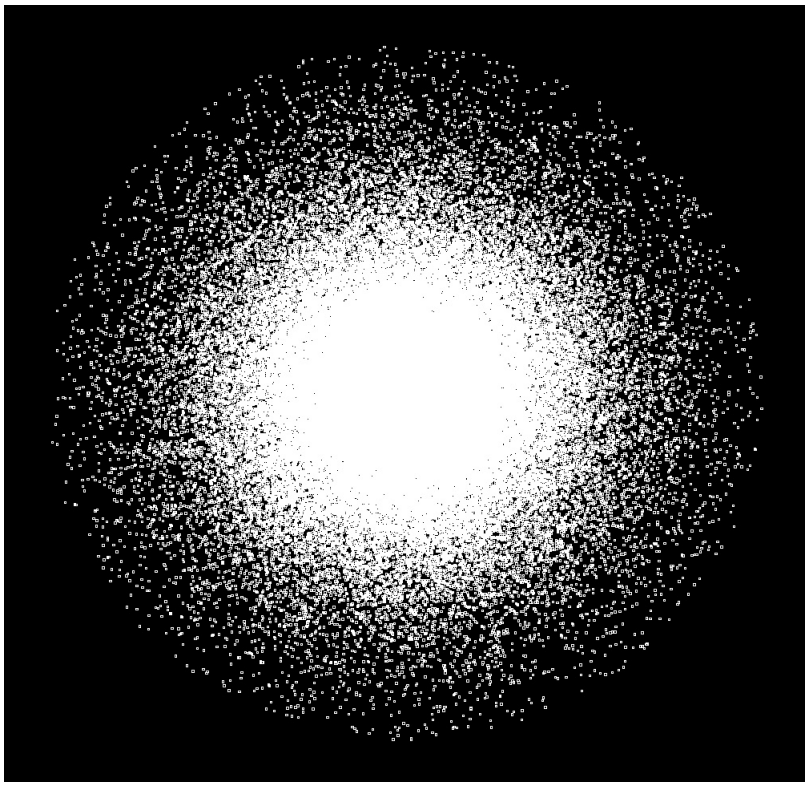

Figura 1.1: Distribución de partículas en un disco exponencial

pueden ser en la actualidad Omnipath o InfiniBand, por lo que el método de Peano-Gilbert permite que las regiones adyacentes permanezcan adyacentes en los nodos también, reduciendo el tiempo y la necesidad de comunicaciones. En este caso resulta de mayor importancia este particionamiento debido a que que Gadget-2 ya se ejecuta en paralelo, por lo que una distribución poco óptima puede llevar a tener importantes desbalanceos de carga y por lo tanto tiempos excesivos de ejecución.

Como puede verse en la figura 1.3 es notable el cambio que se realiza en la función de descomposición.

Se proyecta que para la tercera versión de Gadget se desarrolle una partición de dominio aún mejor, también basado en una curva de Peano-Hilbert, como puede verse en la figura 1.4. Se espera, nuevamente, que este cambio tenga impacto en un mejor desempeño en la ejecución del código.

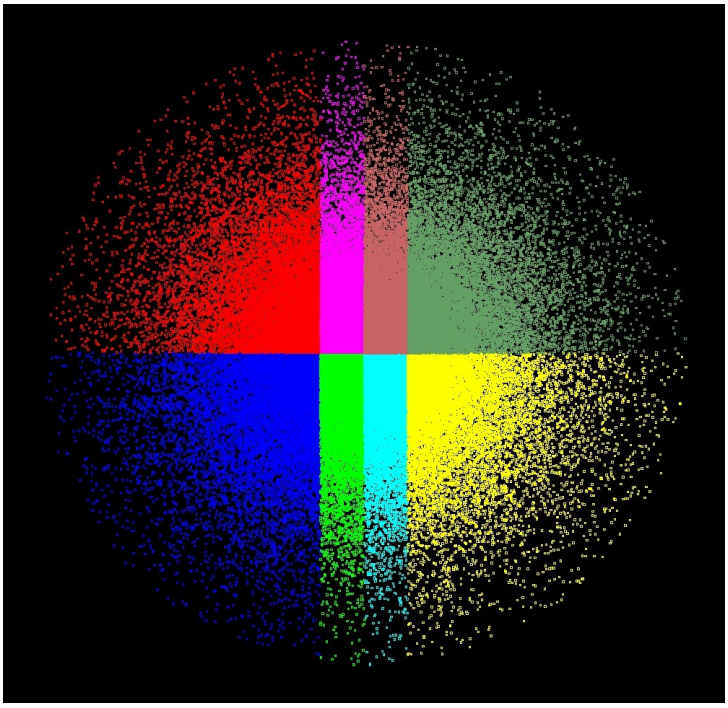

Figura 1.2: Bisección recursiva ortogonal simple

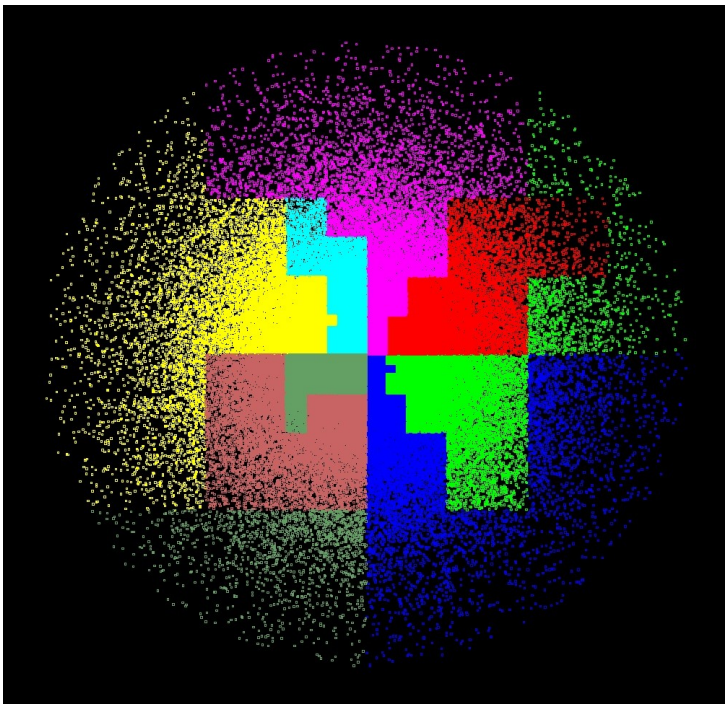

Figura 1.3: Curva de Peano-Hilbert

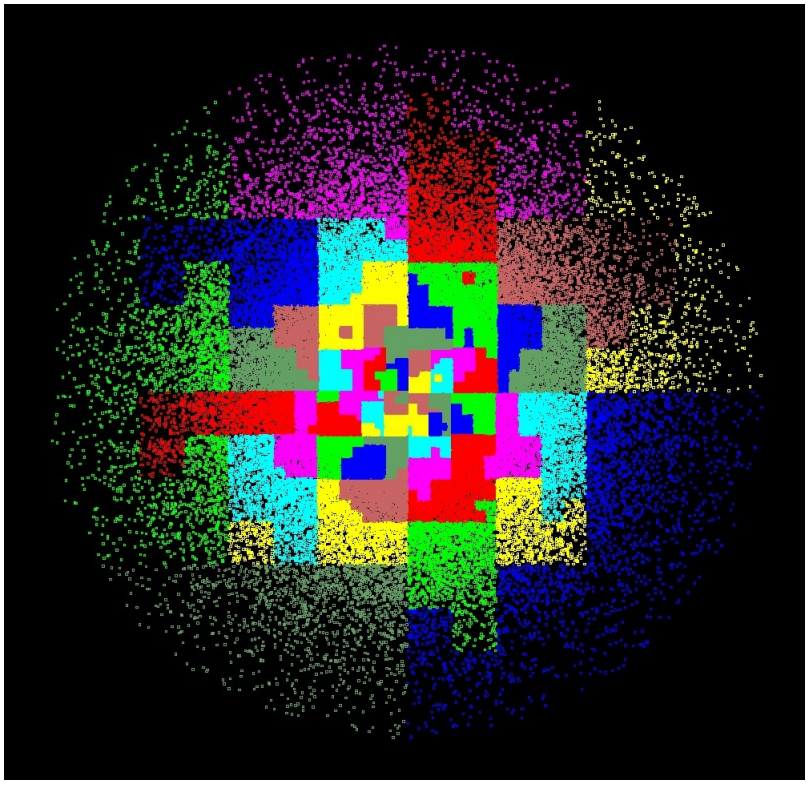

Figura 1.4: Nueva curva de Peano-Hilbert

Capítulo 2

Breve historia de los CPU

¿Qué fue lo que cambió en los microprocesadores en los últimos 30 años? [13]Esta pregunta tiene varias aristas que abordaremos en esta sección.

2.1. Memoria principal

Este es uno de los cambios que se ha vuelto más complejo en los últimos años. Pensemos que, por ejemplo, en una computadora con un procesador 80286 de finales de la década de 1980 y principios de 1990 un acceso a memoria no tomaba más que un par de ciclos de reloj. En el año 2005 a un procesador Pentium 4 le tomaba más de 400 ciclos. Esto nos muestra claramente que la velocidad de los procesadores ha crecido mucho más allá de lo que creció la velocidad de la memoria. Es para resolver este problema que aparecen las memorias cache, memorias de muy alta velocidad pero pequeñas, en las que se almacenan los datos más frecuentemente utilizados y que se cargan por adelantado. Este sistema es conocido en inglés como *prefetching* y está encargado de almacenar información en la memoria cache según patrones de acceso frecuente. Viendo estos datos, se podría pensar que se han tomado malas decisiones de diseño, ya que pasamos de unos pocos ciclos de reloj a 400 en arquitecturas modernas para un acceso a memoria. Sin embargo, si un programa posee un ciclo iterativo en el que se accede secuencialmente a una porción de memoria y el procesador logra detectar el patrón de acceso para hacer la precarga de datos, es claro que la pérdida de rendimiento es notablemente inferior. Existen casos en los que no es deseable dejar a interpretación del procesador el manejo de los datos que se cargan en caché, por lo que teniendo en cuenta ciertos parámetros de la arquitectura se puede diseñar el código de manera tal que aproveche al máximo las capacidades disponibles.

2.2. Unidad de gestión de memoria

Como se verá posteriormente, en la descripción de los procesadores utilizados, existen varios niveles de memoria caché con diversas funciones. Conocer sobre estos aspectos de la arquitectura es necesario en aquellos casos en que el desarrollador esté diseñando un programa que se ejecute de manera óptima sobre la misma, caso contrario puede abstraerse de ellas. Sin embargo aún hay que tener un factor en cuenta, la necesidad de traducir direcciones lógicas a direcciones físicas y como este proceso funciona.

La unidad de gestión de memoria, conocida por sus siglas en inglés *MMU* correspondientes a *memory management unit*, es el dispositivo encargado de administrar los accesos a memoria de parte del procesador. Las funciones de este dispositivo incluyen, el control de caché, la traducción de las direcciones lógicas a direcciones físicas y la protección de la memoria, entre otras.

Cuando el procesador intenta acceder a una dirección de memoria lógica, la *MMU* hace una búsqueda en una tabla especial llamada TLB, por sus siglas en inglés *Translation Lookaside Buffer*, donde se almacenan los accesos más recientes. Cuando la dirección solicitada se encuentra en la *TLB*, su traducción a dirección física es enviada, este hecho es conocido como acierto de TLB o *TLB hit*. En caso que la dirección solicitada no se encuentre en la TLB, hecho conocido como fallo de TLB o *TLB miss*, el procesador realiza la búsqueda en la tabla de páginas del proceso, utilizando el número de página buscada como entrada a ella. En la entrada de la tabla de páginas del proceso hay ubicado un bit de presencia que indica si la página buscada está alojada en la memoria principal. Si dicho bit está activado, se carga esta entrada en la TLB y se devuelve la dirección física. Caso contrario, debe informarse al sistema operativo mediante un fallo de página. El sistema operativo es el encargado de ubicar en la memoria física la página solicitada, usando alguno de los algoritmos de reemplazo de páginas.

Debido a que el primer nivel de la TLB debe ser extremadamente rápido se encuentra limitada en su tamaño, en las arquitecturas que se verán más adelante, constan de 64 entradas, mientras que arquitecturas posteriores como Intel Broadwell [26] [2] constan de 128 entradas. Si se utilizan páginas de 4 kilobytes de tamaño, esto limita la cantidad de memoria que puede ser direccionada sin incurrir en un fallo de búsqueda. La arquitectura de x86 también soporta tamaños de página mayores, de 2 Megabytes y 1 Gigabyte, de lo cual algunas aplicaciones pueden beneficiarse, sobre todo aquellas que necesitan largos tiempos de ejecución y gran cantidad de memoria.

A su vez la memoria caché de primer nivel usualmente se encuentra limitada por los tamaños de página. Si el tamaño de la memoria cache es muy pequeño los bits utilizados para la indexación dentro de la caché son

los mismos, independientemente de estar mirando la dirección física o lógica. No es necesaria la traducción entre ambas. Si la cache es demasiado grande, debe hacerse primero una búsqueda en la *TLB*, que aumentará el costo, o crear un índice virtual, lo cual es posible pero incrementa la complejidad del programa.

2.3. Ejecución fuera de orden

Desde hace algunas décadas, los procesadores de arquitectura x86 han sido capaces de ejecutar de una manera especulativa e incluso de reordenar la ejecución para prevenir el bloqueo de la misma debido a la no disponibilidad de algún recurso. Esto puede resultar en un rendimiento dispar pero la arquitectura es estricta por lo que, para un solo núcleo y observando el estado de memoria y registros desde el exterior, debe verse todo como si la ejecución fuera en el orden original, caso contrario se vería alterada la semántica del programa [13].

Esta restricción que hace ver que la ejecución se realiza en orden significa que, para la mayor parte, se puede ignorar la existencia de la ejecución fuera de orden a no ser que se esté intentando el mayor rendimiento posible del procesador. Las mayores excepciones se dan cuando es necesario asegurar que la ejecución no solo aparenta haber sido realizada en el orden correspondiente, sino que efectivamente se ejecuta en ese orden.

Es importante notar que para que este tipo de ejecución sea posible, se debe contar con un procesador que sea *segmentado* y *superescalar* ya que al detenerse la ejecución de instrucciones de un programa en un núcleo, la ejecución del mismo está sujeta a la liberación de los recursos necesarios y deberá ser llevada a cabo en otro núcleo. Para garantizar el resultado de la computación, hay una unidad dentro del procesador encargada de asegurar que la semántica del programa no sea alterada. Un procesador se llama segmentado cuando divide una tarea compleja en otras más simples, a fin de que la salida de una tarea sea la entrada de la siguiente y de esta manera poder explotar en mejor medida el paralelismo dentro del procesador. En consonancia, un procesador superescalar es aquel capaz de realizar distintas tareas sobre distintos datos en un mismo pulso de reloj.

2.4. Una instrucción, múltiples datos

En la actualidad la mayoría de los procesadores x86 tienen soporte para instrucciones *AVX*, correspondiente a *Advanced Vector Extensions*, y *SSE*,

por sus siglas en ingles *Streaming SIMD Extensions*, en sus versiones 2, 3, 4, 4.1 y 4.2, vectores de registros e instrucción de 128, 256 y 512 bits. En vista de que es común querer aplicar la misma operación sobre múltiples datos más de una vez, tanto Intel como AMD agregaron instrucciones que permiten operar sobre porciones de datos de, por ejemplo, un bloque de 128 bits, o dos de 64 bits, o 4 de 16 bits, etcétera. Es habitual obtener una aceleración que reduce el tiempo de ejecución a la mitad o a la cuarta parte del tiempo, utilizando este tipo de instrucciones. No hay que olvidar en este caso la ley de Ambdahl, que establece que un programa puede reducir su tiempo de ejecución en tanto parte de ese programa pueda ser ejecutada en paralelo. La sección del programa que no pueda ser paralelizada, se mantendrá ocupando la misma porción de tiempo. Los compiladores tienen buenas capacidades al momento de analizar el código fuente y determinar que partes del código pueden ser reemplazadas por instrucciones vectoriales, pero no siempre consiguen los mejores resultados. Es por este motivo que se recomienda hacer un análisis del código *assembly* obtenido de la compilación y verificar que se estén utilizando dichas instrucciones. En caso de no lograr que el compilador reconozca donde aplicar estas instrucciones, deberá ajustarse el código, de manera que pueda ser capaz de identificarlo, ya que programar la vectorización a mano haría poco portable el código.

Capítulo 3

Equipamiento utilizado

A continuación se presentará el equipamiento utilizado para este trabajo y sus características técnicas.

3.1. Nodo con procesadores Intel

Para la comparación que se presenta en este trabajo, se utilizó un nodo del cluster Mendieta perteneciente al CCAD-UNC [5]. Se trata de un servidor marca Dell modelo Poweredge R720, que en su interior cuenta con una placa madre Dell 0VWT90, dos procesadores Intel Xeon E5 2670 y 256 Gigabytes de memoria principal.

3.1.1. Procesador

El procesador Intel Xeon E5 2670 en su primera generación fue lanzado en el primer trimestre del año 2012, forma parte de la familia de procesadores Sandy Bridge EP, consta de 8 núcleos, 16 hilos, una frecuencia base de 2.60 GHz, una frecuencia turbo de 3.30 GHz, 3 niveles de memoria cache estructurados de la siguiente manera:

- Cache L1
  - 8 x 32 KB 8-vías conjunto asociativo de caché de instrucciones
  - 8 x 32 KB 8-vías conjunto asociativo de caché de datos
- Cache L2
  - 8 x 256 KB 8-vías conjunto asociativo de caché
- Cache L3

- 20 MB 20-vías conjunto asociativo de caché

Su consumo, trabajando a frecuencia base con carga plena es de 115W según la hoja de datos del fabricante.

3.1.2. Diagrama de bloque

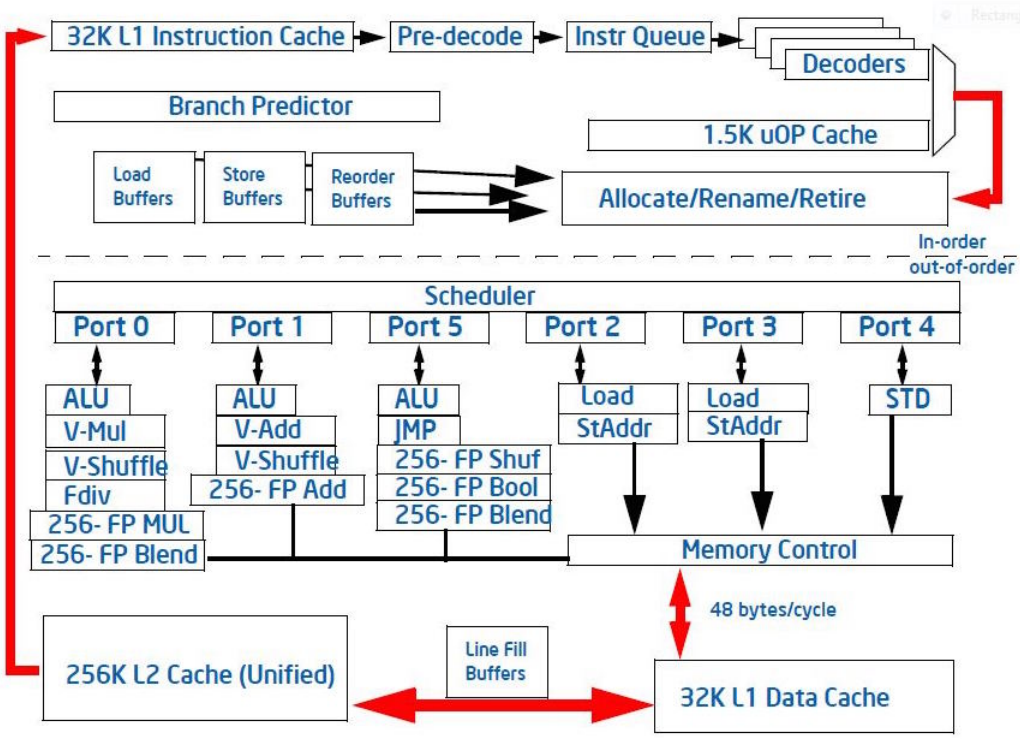

Figura 3.1: Representación de un núcleo Sandy Bridge [25]

Como puede verse en el diagrama 3.1, cada uno de los núcleos de este procesador cuenta con sus propias unidades de punto flotante, lo que asegura en una primera instancia y mientras no se encienda el sistema de *HyperThreading*, que el uso del núcleo puede dedicarse de manera exclusiva a cálculo. El sistema de *HyperThreading* [29] [11] Intel exhibe un núcleo virtual además del físico ya existente, por lo que el sistema operativo visualiza el doble de procesadores de los que verdaderamente existen de manera física.

3.1.3. Memoria

El microprocesador instalado en este nodo es capaz de direccionar un máximo de 384 GB de memoria de tipo DDR3 con velocidades de 800MHz, 1066MHz, 1333MHz y 1600MHz, 4 canales de memoria con un ancho de banda máximo teórico de 51.2GB/s y código de corrección de errores (ECC). El nodo cuenta con un total de 24 conectores para memoria de los cuales 16 se encuentran ocupados con módulos de memoria DDR3 de 16GB de capacidad y 1333MHz de velocidad. Los canales se encuentran balanceados en cantidad de memoria, garantizando que no se produzca saturación en ningún canal.

3.2. Nodo con procesadores AMD

La contraparte de esta comparación, fue realizada utilizando un equipo perteneciente al Instituto de Astronomía Teórica y Experimental [4] cuya descripción técnica se verá a continuación. Se trata de un servidor Supermicro AS-4022G-6F que consta de una placa madre Supermicro H8DGi, dos procesadores AMD Opteron 6282SE y 256 Gigabytes de memoria principal.

3.2.1. Procesador

El procesador AMD Opteron 6282SE fue lanzado en el último trimestre del año 2011, consta de una microarquitectura Bulldozer de la plataforma Maranello, consta de 8 módulos, cada uno de los cuales cuenta con dos unidades de cálculo entero y una unidad de punto flotante de 256 bits, una frecuencia base de 3.0GHz, una frecuencia turbo de 3.30 GHz, 3 niveles de memoria cache estructurados de la siguiente manera

- Cache L1
  - 8 x 64 KB 2-vías conjunto asociativo compartido de caché de instrucciones
  - 16 x 16 KB 4-vías conjunto asociativo de caché de datos
- Cache L2
  - 8 x 2 MB 16-vías conjunto asociativo compartido exclusivo de caché
- Cache L3
  - 2 x 8 MB hasta 64-vías conjunto asociativo compartido de caché

Su consumo, trabajando a frecuencia base con carga plena es de 140W según la hoja de datos del fabricante.

3.2.2. Diagrama de bloque

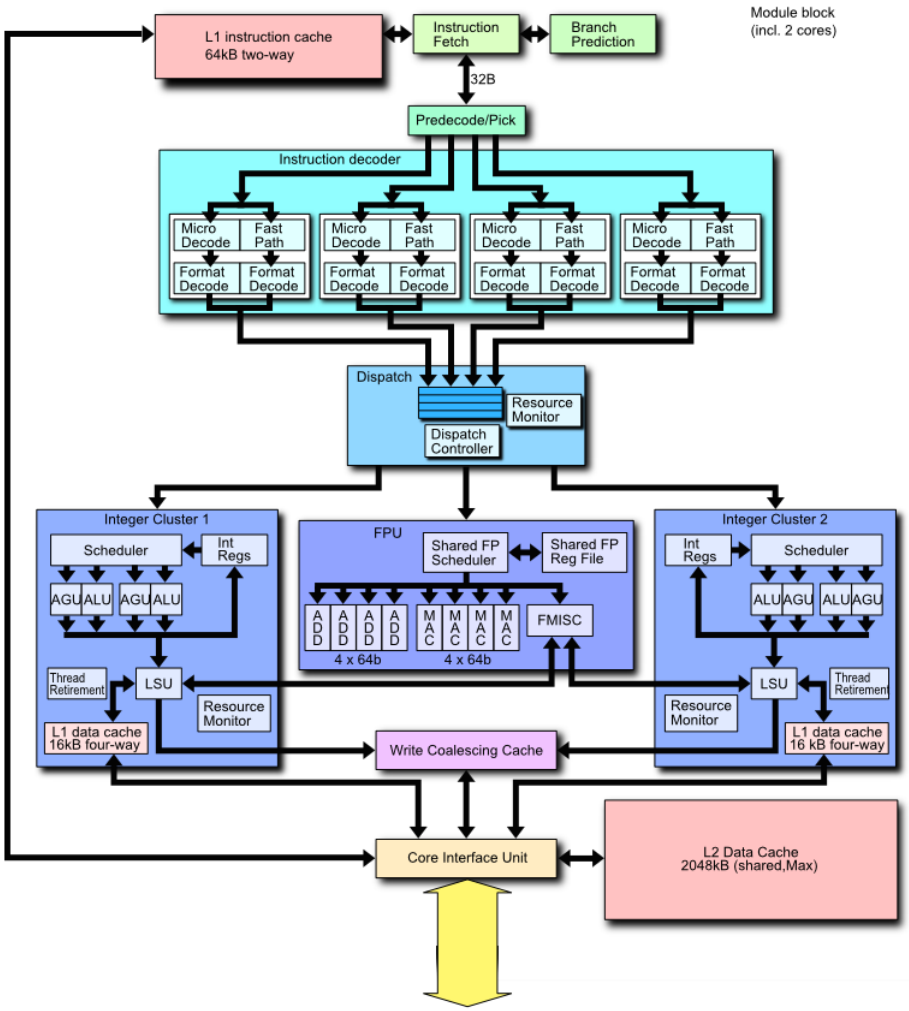

Figura 3.2: Representación de módulo AMD Bulldozer [28]

Como puede verse en la figura 3.2, el módulo Bulldozer consta de dos unidades para cálculo entero y comparte una unidad de punto flotante de 256 bits y el *dispatcher*. Este hecho supone que a pesar de contar en total con 32 núcleos para cálculo según muestra el sistema operativo, solo cuenta con 16 unidades de punto flotante.

3.2.3. Memoria

El microprocesador instalado en este nodo soporta memoria de tipo DDR3 con velocidades de 1333MHz y 1600MHz, cuenta con 2 controladores de memoria, cada uno con capacidad de manejar 2 canales, totalizando 4 canales y un ancho de banda máximo teórico de 51.2GB/s y código de corrección de errores (ECC). El nodo cuenta con un total de 16 conectores para memoria los cuales se encuentran ocupados con módulos de memoria DDR3 de 16GB de capacidad y 1600MHz de velocidad.

Capítulo 4

Caracterización

Con la finalidad de caracterizar el equipamiento que sería utilizado para las ejecuciones de Gadget se realizaron cálculos teóricos de ancho de banda de memoria y potencia de cálculo. A su vez estos parámetros fueron medidos a fin de corroborar cuanto se correspondían los cálculos teóricos con el uso efectivo del equipo.

4.1. Potencia de cálculo

La potencia de cálculo es medida en cantidad de operaciones por segundo. Específicamente en este caso nos interesa la cantidad de operaciones de punto flotante por segundo, conocido por su acrónimo en inglés FLOPS correspondiente a *Floating point Operations Per Second*, ya que también se podría medir la cantidad de operaciones de números enteros.

Una forma muy habitual de medir la potencia de cálculo teórica es a través de la siguiente ecuación:

$$FLOPS = fpus \times ops/cycle \times (vector\,width/64) \times base\,clock\,frequency \tag{4.1}$$

Siendo fpus la cantidad de unidades de punto flotante de doble precisión, *64 bits*, de las que dispone el procesador, ops/cycle es la cantidad de operaciones de punto flotante que es capaz de realizar por ciclo de reloj, (vector width / 64) es la cantidad de valores de doble precisión con los que puede trabajar en simultáneo y base clock frequency es la frecuencia de base del procesador. Es importante resaltar que en este último parámetro se debe tener en cuenta la frecuencia base y no la turbo o similares, ya que en principio el procesador no es capaz de tener todos los núcleos trabajando a esa frecuencia.

Sin embargo en este caso la fórmula utilizada es otra, debido a dos factores. En el caso del procesador Intel, este cuenta con una unidad de suma y

otra de multiplicación, cada una en un puerto independiente, por lo que puede realizar una operación de suma y otra de multiplicación en paralelo. Por el otro lado, el procesador AMD, cuenta con capacidad de realizar operaciones de multiplicación y suma en un mismo pulso de reloj; sus siglas en inglés son FMA correspondientes a *Fused Multiply-Add*. Este tipo de operaciones resulta en extremo útil al momento de trabajar, por ejemplo con multiplicación de matrices. La fórmula que calcula la cantidad de operaciones de punto flotante por segundo con este tipo de capacidades, es la siguiente:

$$FLOPS = fpus \times \text{ops FMA}/cycle \times (vector\ width/64) \times base\ clock\ frequency \tag{4.2}$$

En base a la descripción que vista anteriormente de cada uno de los procesadores y teniendo en cuenta la ecuación 4.2 podemos calcular la potencia pico teórica en cada caso.

CPU	fpus	opsFMA/cycle	vector w	clock freq	GFLOPS
Xeon E5 2670	8	2	256	2.6GHz	166.4
Opteron 6282SE	8	2	256	2.6GHz	166.4

Tabla 4.1: Potencia pico teórica para los procesadores utilizados

Por lo tanto, si se tiene en cuenta que cada nodo posee dos procesadores y cada uno cuenta con una potencia de cálculo teórica igual al doble de la vista en la tabla 4.1, obtenemos como resultado la tabla 4.2.

Nodo	fpus	opsFMA/cycle	vector w	clock freq	GFLOPS
Intel	16	2	256	2.6GHz	332.8
AMD	16	2	256	2.6GHz	332.8

Tabla 4.2: Potencia pico teórica para los nodos utilizados

4.2. DGEMM

Con el objetivo de caracterizar el equipamiento utilizado, se realizaron varias tareas; en primer lugar se realizó una prueba de cálculo denso, se trata de multiplicación de matrices de doble precisión de la forma

$$C = \alpha A \times B + \beta C \tag{4.3}$$

siendo la matriz A de la forma m filas por k columnas, B de k filas por n columnas y C de m filas por n columnas. El valor α es un escalar que multiplica

al resultado de $A \times B$ y β es otro escalar utilizado para multiplicar la matriz C. Para realizar la ejecución del código se le deben pasar tres parámetros que corresponden al tamaño de las matrices que se desea multiplicar, ya que para este caso, los valores α y β no serán tenidos en cuenta. El programa solicita la cantidad de memoria correspondiente al tamaño de las matrices por el tamaño de un valor de doble precisión, almacena en las matrices valores pseudoaleatorios, y luego llama a la función de multiplicación de matrices, tomando previamente el tiempo de inicio. Una vez terminada la multiplicación, toma el tiempo de finalización y evalúa la diferencia entre tiempo final y tiempo de inicio. Con este tiempo y la cantidad de operaciones de punto flotante realizadas puede calcular la potencia utilizando la ecuación 4.2.

DGEMM es un acrónimo para *Double-precision General Matrix Multiply*, que en este caso se convierte en la función utilizada para multiplicar matrices. Para las pruebas realizadas en esta sección, se tomaron en cuenta todas matrices cuadradas, es decir que lo valores m, k y n son iguales. La biblioteca utilizada para esta prueba fue *OpenBLAS* [30] compilada de manera específica para la arquitectura de cada equipo utilizado. Se decidió la utilización de este método ya que también se utiliza para caracterizar las computadoras de alto desempeño a través del uso de *Linpack*. En la actualidad la lista de las 500 supercomputadoras más poderosas del mundo, conocida como TOP500 [27], se arma a través de la ejecución de un código similar, denominado HPL [6].

4.2.1. DGEMM en AMD

El cálculo de DGEMM en el nodo AMD se realizó en dos modalidades. La primera, cuyos resultados podemos ver en la tabla 4.3, realizado con 16 núcleos y fijando a través del administrador de trabajos la afinidad de los procesos con los diferentes procesadores, de manera que no existan dos procesos dentro del mismo módulo y así poder destinar la total disponibilidad de la unidad de punto flotante al cálculo sin interrupciones.

Tamaño	Tiempo total utilizado	GFLOPS
2048	0.083221 s	206.436
4096	0.591606 s	232.315
8192	4.710643 s	233.410
16384	36.920581 s	238.243
18432	57.239284 s	218.802
20480	70.275109 s	244.465
24576	168.516876 s	176.165
32768	313.201853 s	224.675

Tabla 4.3: Potencia de cálculo para el nodo AMD con 16 núcleos

Como puede verse, los resultados obtenidos se encuentran lejos de alcanzar los valores teóricos vistos en la tabla 4.2 ya que el valor más próximo al teórico, solo llega al 73.45 %.

Es por esto, que se decide realizar una nueva evaluación del nodo, esta vez con los 16 módulos y los 32 núcleos. Previo realizar la prueba es necesario compilar nuevamente la biblioteca *OpenBLAS* a fin de que utilice eficientemente toda la capacidad del nodo.

Tamaño	Tiempo total utilizado	GFLOPS
2048	0.077408 s	221.939
4096	0.542884 s	253.164
8192	4.055642 s	271.106
16384	33.414981 s	263.238
18432	44.919970 s	278.809
20480	61.102392 s	281.165
24576	106.968454 s	277.528
32768	265.165076 s	265.377

Tabla 4.4: Potencia de cálculo para el nodo AMD con 32 núcleos

Aquí puede verse una leve mejora en los resultados obtenidos, alcanzando un 84.58 % del pico teórico calculado.

4.2.2. DGEMM en Intel

Al igual que para el nodo con arquitectura AMD, también se utilizaron distintos tamaños de matrices en el cálculo de DGEMM. Los resultados se muestran en la tabla 4.5

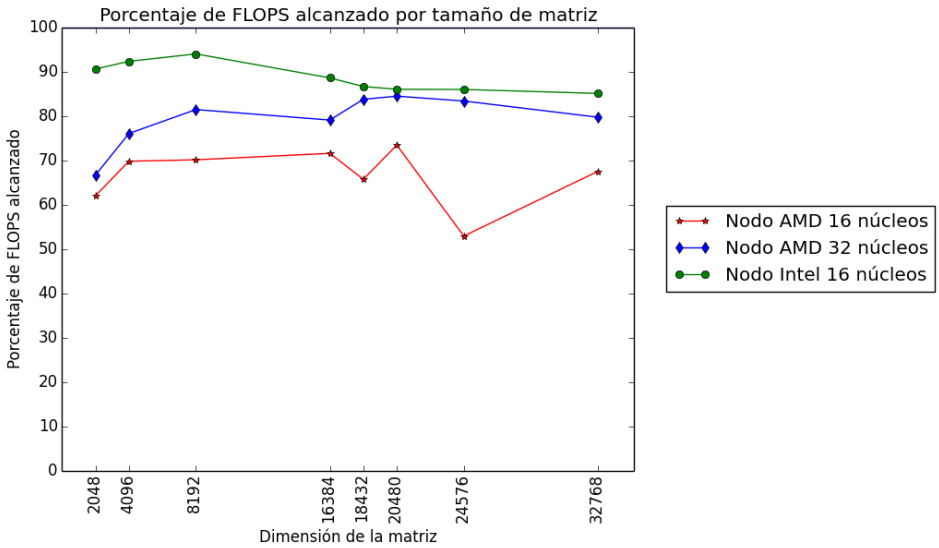

Figura 4.1: Porcentaje de FLOPS obtenido por cada nodo

Tamaño	Tiempo total utilizado	GFLOPS
2048	0.056978 s	301.517
4096	0.447248 s	307.299
8192	3.513998 s	312.894
16384	29.829968 s	294.874
18432	43.422726 s	288.423
20480	60.000443 s	286.329
24576	103.723717 s	286.210
32768	248.453719 s	283.226

Tabla 4.5: Potencia de cálculo para el nodo Intel

Como queda expuesto en la tabla, el mejor tiempo es obtenido para una matriz de 8192 elementos de lado alcanzando el 94 % de la capacidad teórica calculada previamente.

En el gráfico 4.1 se puede ver la comparación de los valores obtenidos en la ejecución de DGEMM en todos los nodos y porcentaje respecto del pico teórico obtenido por cada uno. Se puede notar que cuando incrementamos el tamaño de la matriz el desempeño en AMD mejora. Lo contrario ocurre con Intel, acabando en un desempeño similar. Esto puede deberse a la diferencia en el tamaño de la memoria caché de cada procesador.

4.3. Ancho de banda de memoria

Para ambos equipos utilizados, también se realizaron mediciones de ancho de banda de memoria. En la ecuación 4.4 se puede observar como es calculada de manera teórica dicha magnitud.

$$BW = channels \times 8\,(64bits) \times F \tag{4.4}$$

En la fórmula, *channels* hace referencia a la cantidad de canales de memoria con los que se cuenta y F es la frecuencia a la que trabaja la memoria. En la tabla 4.6 se pueden ver los valores teóricos calculados para ambos procesadores.

CPU	Cant. canales	Frec. memoria	BW teórico
Xeon E5 2670	4	1333MHz	42.6GB/s
Opteron 6282SE	2 × 2	1600MHz	51.2GB/s

Tabla 4.6: Ancho de banda teórico por CPU

Las pruebas se realizaron en ambos equipos, cada uno de los cuales cuenta con dos procesadores de los antes descriptos y distintas configuraciones de memoria como muestra la tabla 4.7:

Equipo	Cant. canales	Frec. memoria	BW teórico
Nodo Intel	8	1333MHz	85.2GB/s
Nodo AMD	8	1600MHz	102.4GB/s

Tabla 4.7: Ancho de banda teórico por nodo

4.4. STREAM

Este código [14] [15] un test sintético utilizado ampliamente para medir el ancho de banda efectivo en MB/s y la correspondiente tasa de cómputo para vectores. La motivación original para el desarrollo de este código consistió en que la velocidad de los procesadores fue en constante ascenso en las últimas décadas, sin embargo la velocidad de acceso a memoria no creció al mismo ritmo y la brecha entre ambos se hizo cada vez más grande. Con el aumento de esta brecha, cada vez mayor cantidad de programas vieron ralentizada su ejecución. Este código ha sido específicamente diseñado para poder ser ejecutado de manera que los datos utilizados no entren en la memoria caché obligando al sistema a buscar datos en la memoria principal, por lo que los resultados obtenidos son, un indicativo del rendimiento

Function	Best Rate GB/s	
	`gcc`	`icc`
Copy	23.6252	32.3689
Scale	28.0051	40.6936
Add	27.5941	38.9338
Triad	27.9189	35.1233

Tabla 4.8: stream en AMD con 16 núcleos y 16.000.000 elementos `gcc` e `icc`

de la memoria. Se realizan cuatro pruebas, en primer lugar *copy* que mide la capacidad de transferencia en ausencia de operaciones aritméticas, *scale* agrega operaciones aritméticas simples, *sum* agrega un tercer operando para poder medir procesadores con múltiples puertos de carga y almacenamiento, *load/store*, *triad* realiza la prueba para operaciones de suma y multiplicación encadenada, solapada o conjunta *chained/overlapped/fused*. Todos los resultados mostrados por stream corresponden a valores de 64 bits y cabe resaltar que se valoran las operaciones de lectura y escritura. Las compilaciones del código para esta prueba se hicieron inicialmente con el compilador GCC versión 5.3.0. Luego se agregaron pruebas con el compilador `icc` de la suite Intel 2016 [21].

4.4.1. STREAM en AMD

Como se vió previamente en la tabla 4.7 el ancho de banda de memoria teórico para este nodo es de 102.4GB/s. Se probaron distintas formas de ejecución para tratar de alcanzar ese límite. En el primer caso se ejecutó con un arreglo de 16.000.000 elementos y se compiló con `gcc`.

Como puede verse en la tabla 4.8 los resultados obtenidos con `gcc` están muy por debajo del límite teórico, alcanzando sólo un 27.34 % del mismo, por lo que se procedió a realizar la compilación con `icc` de la suite Intel con la misma cantidad de elementos y las banderas correspondientes para ajuste de código a la plataforma.

Se observa una mejora en el resultado obtenido, aunque sigue estando muy lejos del resultado esperado. Ante estos valores, y teniendo en cuenta la arquitectura de módulos del procesador, se decidió hacer la misma ejecución con los dos compiladores utilizados anteriormente, pero en este caso con 32 núcleos y la misma cantidad de elementos.

Nuevamente, los valores obtenidos están muy por debajo de lo esperado y no registran diferencia con los obtenidos al ejecutar en 16 núcleos.

Aquí se obtiene una leve mejora respecto de los valores anteriores, siendo

Function	Best Rate GB/s	
	`gcc`	`icc`
Copy	28.3751	37.4714
Scale	28.3751	42.9033
Add	27.8624	39.5913
Triad	27.9170	36.7711

Tabla 4.9: stream en AMD con 32 núcleos y 16.000.000 elementos `gcc` e `icc`

los mejores valores obtenidos hasta el momento para esta plataforma con un 41.89 % del ancho de banda teórico. Como se verá a continuación en el gráfico 4.2, hay una leve mejora en la medida cuando se utiliza el compilador `icc`, pero sigue siendo muy por debajo del límite teórico. Sin embargo no deja de ser llamativo, el hecho de que en una arquitectura AMD el compilador de Intel exhiba mejores resultados.

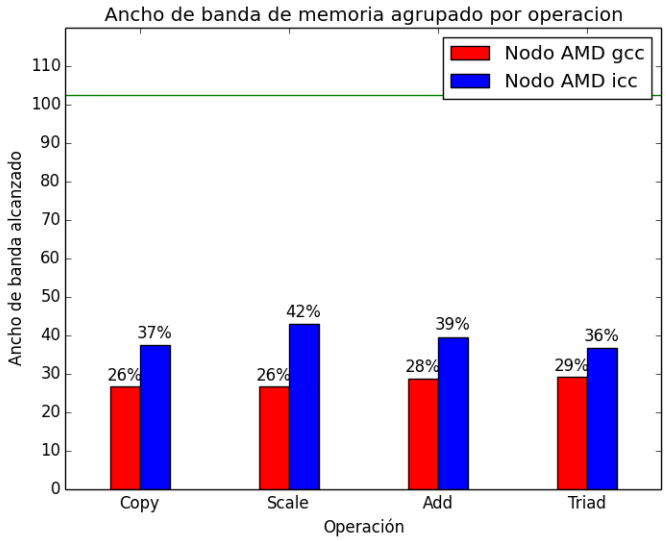

Figura 4.2: Comparación de ancho de banda obtenido en nodo AMD

Todo este análisis logró demostrar que el manejo de memoria en este procesador no es el óptimo y la ejecución de programas con gran necesidad de acceso a memoria se vería seriamente afectada.

4.4.2. STREAM en Intel

Para este nodo el ancho de banda teórico es un poco menor ya que la velocidad de las memorias que tiene instaladas es un poco más baja, alcanzando un pico teórico de 85.2GB/s. Como ocurrió con el otro nodo, se realizaron las pruebas con los dos compiladores disponibles.

Function	Best Rate MB/s	
	`gcc`	`icc`
Copy	45.4021	49.6422
Scale	45.6492	66.1986
Add	50.0725	67.1716
Triad	50.0315	66.5476

Tabla 4.10: stream en Intel con 16 núcleos y 20.000.000 elementos `gcc` e `icc`

Para el caso de `gcc` puede notarse que se alcanzó más de la mitad del valor teórico. La cantidad de elementos difiere ya que debió ser ajustada al tamaño de memoria caché del procesador. En la prueba realizada con `icc` puede verse como se logró obtener un valor incluso más cercano al teórico, alcanzando un 78.83 % del mismo. En base a estas mediciones y por lo anteriormente explicado de Gadget, que se trata de un problema ligado fuertemente a la memoria, podemos plantear la hipótesis de que el mismo experimento debería tardar más tiempo en el nodo con procesadores AMD que en aquel que posee procesadores Intel.

En la figura 4.3 puede verse la comparación de la ejecución de STREAM sobre el nodo con procesador Intel con compiladores `gcc` e `icc`.

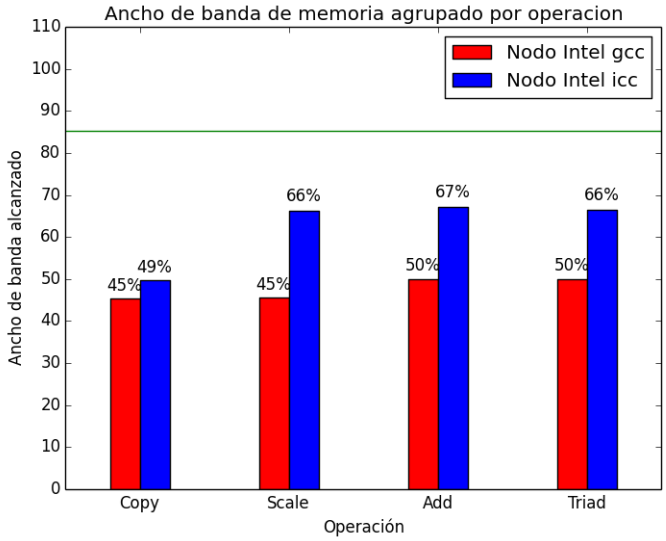

Figura 4.3: Comparación de ancho de banda obtenido en nodo Intel

En el gráfico 4.4 se puede observar la comparación de todos los valores obtenidos. Resulta notable a simple vista el mejor ancho de banda que presenta el nodo Intel sobre el AMD a pesar de contar con memorias con menor frecuencia de trabajo.

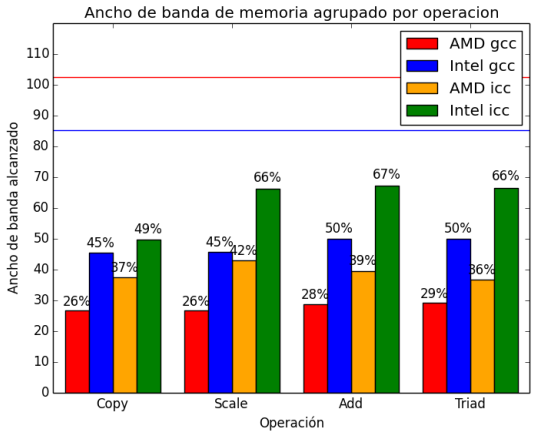

Figura 4.4: Comparación de ancho de banda obtenido en ambos nodos

Capítulo 5

Ejecuciones realizadas

En este capitulo se expondrán las distintas ejecuciones realizadas del código GADGET. Aquí se podrá ver la utilización de distintos parámetros de compilación, tanto para Gadget como para las bibliotecas y resultados obtenidos en cada caso y sus comparaciones. Ademas del código de Gadget y sus dependencias, se utilizó el administrador de trabajos llamado SLURM [20] al cual se le indicó según correspondiera, instrucciones específicas como pueden ser el uso exclusivo del nodo o la afinidad de un determinado proceso a un núcleo.

5.1. Descripción de la simulación

Para este trabajo, se decidió simular un cubo cosmológico periódico de 250 megapársecs[1] y una resolución de 512^3 partículas. Con esta configuración se puede resolver halos de materia oscura de 30 kilopársecs donde se albergan las galaxias. Debido a que la finalidad del presente trabajo es comparar el desempeño del código en casos concretos de uso ejecutado sobre las distintas arquitecturas antes descriptas, se decidió solo trabajar con el módulo de gravedad a través de la interacción de partículas de materia oscura. Esta decisión también tuvo que ver con poder realizar las ejecuciones en tiempos acotados, caso contrario cada ejecución se podría tomar semanas. La resolución y tamaños elegidos permiten medir el desempeño del código en dos regímenes que el algoritmo calcula el potencial de gravedad en gran escala a través de la transformada de Fourier y en escalas pequeñas a través del método de árbol. La integración de la evolución de un cubo cosmológico requiere tener en cuenta la expansión del universo, esto se verá manifestado en diferentes regímenes a los largo de la simulación. Al comenzar, la distri-

[1] El pársec es una unidad de medida de distancia equivalente a 3.2616 años luz.

bución de fuerzas es homogénea prácticamente en su totalidad, por lo que la integración a gran escala será la que más tiempo de cómputo consuma. En la medida que el sistema evolucione en el tiempo, se formarán diferentes estructuras cosmológicas como vacíos, filamentos y halos, entre otros. Con la aparición de estas estructuras, la evolución de las partículas se desacopla de la expansión del universo y comienzan a tener mayor importancia las interacciones entre partículas a pequeña escala, por lo que el algoritmo de árbol será más requerido y siendo este último el que al final de la simulación definirá la dinámica de las mismas.

En la figura 5.1, puede verse el logaritmo de la densidad de materia oscura, para un corte del cubo perpendicular al eje z, donde se puede apreciar la homogeneidad en la distribución inicial de las partículas (factor de expansión del universo igual a 0.05). A continuación, en la figura 5.2, se puede observar la estructura del universo en gran escala y la menor homogeneidad de la distribución de las partículas al final de la simulación.

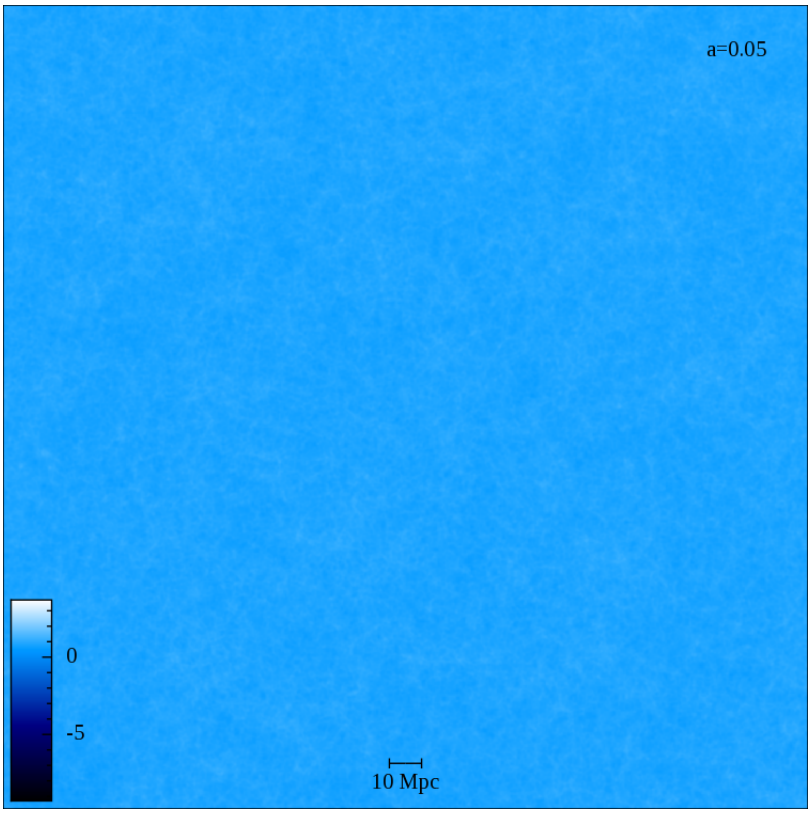

Figura 5.1: Distribución de las partículas al iniciar la simulación

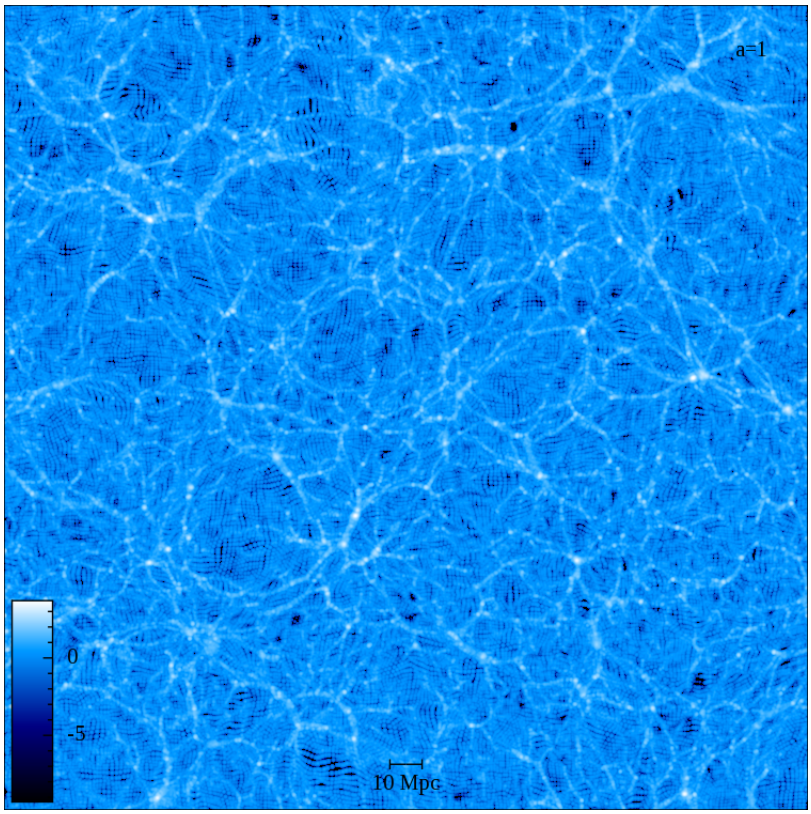

Figura 5.2: Distribución de las partículas al finalizar la simulación

5.2. Dependencias

Las dependencias de Gadget para este tipo de ejecuciones son pocas, pero es importante ver que cada una de ellas esté correctamente configurada. Para empezar, se decidió utilizar el compilador GNU GCC 5.3.0 [9] último disponible al momento de comenzar con las ejecuciones para este trabajo y capaz de crear código óptimo para ambas arquitecturas.

Luego fue necesaria la instalación de una interfaz de paso de mensajes, en adelante MPI por sus siglas en inglés *Message Passing Interface*; se optó por MPICH [18] versión 3.2. Esta decisión tuvo que ver con pruebas realizadas con OpenMPI [24], pero se notó un comportamiento extraño, donde se lanzaba más de un hilo por proceso, el cual no resultó fácil de resolver; teniendo en cuenta que las ejecuciones se hicieron dentro de un nodo, los administradores de proceso de ambos tienen un desempeño similar y no hubo necesidad de comunicaciones externas, por lo cual MPICH resultó la opción más acertada.

La siguiente dependencia es FFTW [7], por sus siglas en inglés *Fastest Fourier Transform in the West*, encargada de realizar la transformada de

Fourier dentro del código. En este caso se utilizó la versión 2.1.5. Esto se debe a que las nuevas versiones, de la 3 en adelante, se volvieron incompatibles con las anteriores. En las nuevas versiones hubo importantes cambios de semántica que responden a cuestiones de mejoras en el rendimiento, dichas mejoras no pueden emular de manera eficiente la interfaz anterior. Debido a que hasta este momento no hay versiones estables de Gadget que utilicen la nueva versión de FFTW, las ejecuciones de este trabajo debieron ser realizadas con una versión desactualizada. A su vez, FFTW tiene como dependencia MPI.

Finalmente queda la biblioteca GSL [17], *GNU Scientific Library*, de la cual se utilizó la versión 2.1. Al revisar el código y ver que influencia podría tener GSL en la óptima ejecución del programa, nos encontramos que sólo se utilizaban constantes numéricas de la misma, por lo que las banderas con las que fuera compilada, no afectarían a la ejecución.

En todos los casos, las bibliotecas fueron compiladas con las banderas `-O3 -march=sandybridge -mtune=sandybridge` para la arquitectura Intel y `-O3 -march=bdver1 -mtune=bdver1` para la arquitectura AMD. Con esta configuración inicial se midieron los primeros valores que se tomaron como patrones.

Para tener otro parámetro de comparación, también se realizaron pruebas con compiladores de Intel, al igual que se realizó previamente con la caracterización del equipamiento disponible. A tal fin también debieron ser compiladas las bibliotecas. Para el caso patrón del compilador de Intel se utilizaron las banderas `-O3 -xHost -ip`.

5.3. Ejecución patrón

Para ambas arquitecturas se tomó una ejecución patrón que, como se mencionó antes, se realizó compilando con GCC 5.3.0 y las bibliotecas necesarias. También el código de Gadget fue compilado utilizando las mismas banderas de compilación. La ejecución se realizó en ambos casos con 16 núcleos, en el caso Intel son todos los disponibles, en el caso AMD se dio instrucciones al administrador de trabajos para que mantuviera los procesos ligados a un procesador y que ejecutara este trabajo de manera exclusiva, es decir que a pesar de haber recursos disponibles, no permitiera el ingreso de otros trabajos al nodo. En el gráfico que veremos a continuación, se puede observar la comparación de los resultados obtenidos.

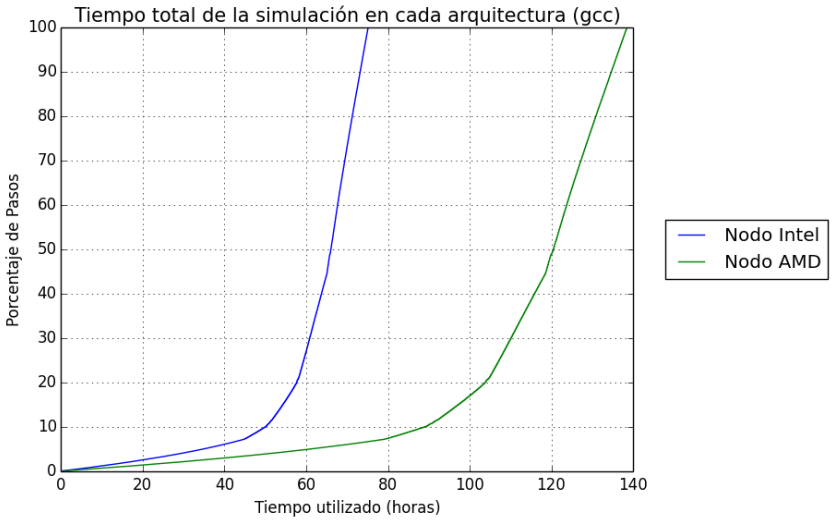

Figura 5.3: Tiempo total utilizado en cada arquitectura

Este gráfico muestra el porcentaje de avance de la simulación versus el tiempo acumulado, en horas, hasta ese momento. Desde un comienzo resulta notoria la diferencia en el tiempo de ejecución en ambas arquitecturas a iguales condiciones de trabajo.

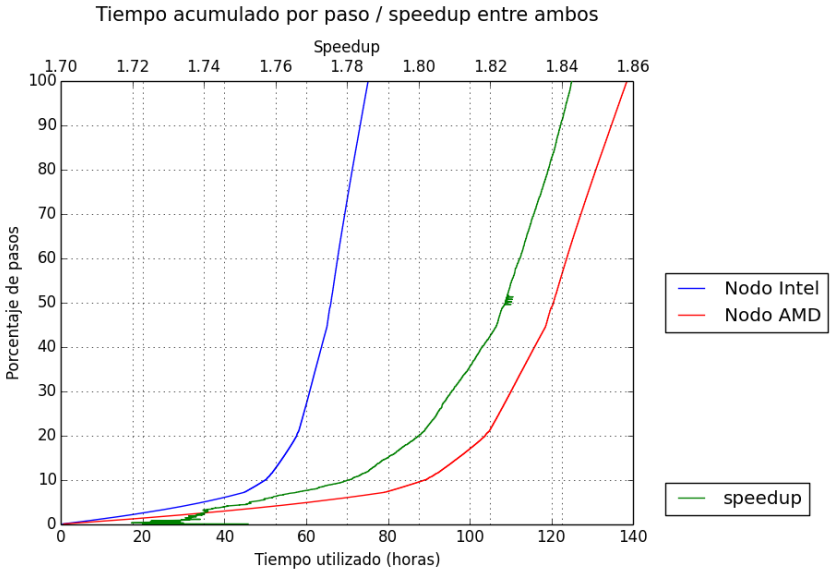

Figura 5.4: Tiempo total utilizado en cada arquitectura y el speedup

Aquí puede verse la comparación de la ejecución en ambas arquitecturas, compilando con `gcc` y las optimizaciones antes mencionadas, mostrando en

el eje x el tiempo utilizado expresado en horas, en el eje y el porcentaje de pasos realizados. En verde se puede ver la división de los tiempos, en este caso AMD dividido por los tiempos de Intel, para obtener que cantidad de veces más rápido ha sido el cálculo, dicho valor puede verse en el eje x superior

En vista de estos dos resultados, se decidió revisar cuanto tiempo de la ejecución se estaba asignando a cada una de las tareas del código.

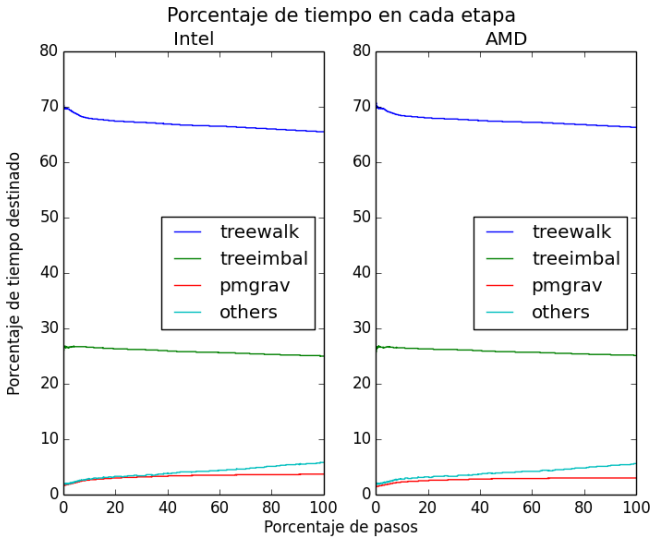

Figura 5.5: Tiempo total utilizado en cada arquitectura por etapa

Aquí puede verse que la mayor parte del tiempo en cada uno de los pasos se dedicó a recorrer el árbol en memoria. Acorde a lo propuesto por el *roofline model* [19], el algoritmo de Gadget queda ubicado a la izquierda, con una baja intensidad computacional, es decir que se realiza poca cantidad de operaciones por cada byte leído de memoria y además hay una alta tasa de acceso a memoria. Este es el primer resultado que comienza a explicar el gráfico 5.3, ya que como se vió previamente en la caracterización de cada nodo, el sistema AMD no tiene un manejo óptimo del acceso a la memoria principal.

Una vez obtenidos estos valores que serían utilizados como patrones, se decidió hacer otras ejecuciones con algunas variaciones, tanto en los compiladores utilizados, las banderas utilizadas para la compilación del código de Gadget y las bibliotecas.

5.4. Ejecución sin optimizaciones

La primera pregunta que surgió fue, qué tanto podría variar el tiempo de ejecución si la compilación del código se realizara con la bandera *-O0* y de esta manera poder obtener la aceleración que se estaba adquiriendo.

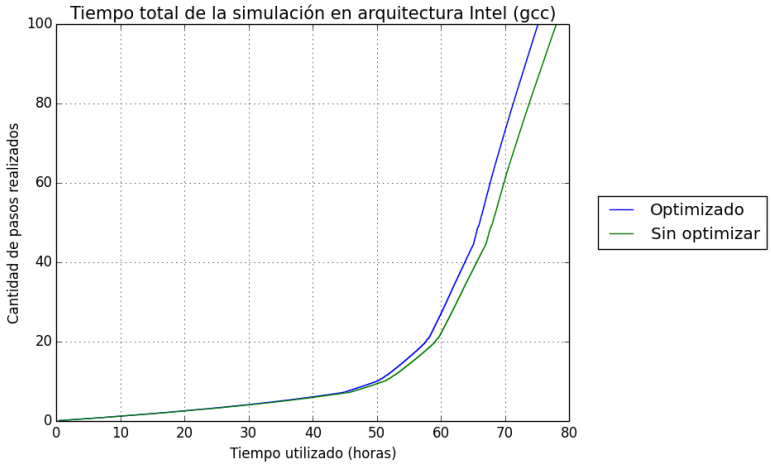

Figura 5.6: Tiempo de ejecución para la arquitectura Intel sin optimizaciones

Es claro que en el caso de Intel, la ganancia no ha sido sorprendente. Como puede observarse, la mayor parte de la diferencia entre ambas ejecuciones, ocurre en la primera etapa, donde se utiliza el cálculo con FFTW. Para esta ejecución también fue compilada con la bandera `-O0`, por lo que el cálculo tarda un poco más de tiempo.

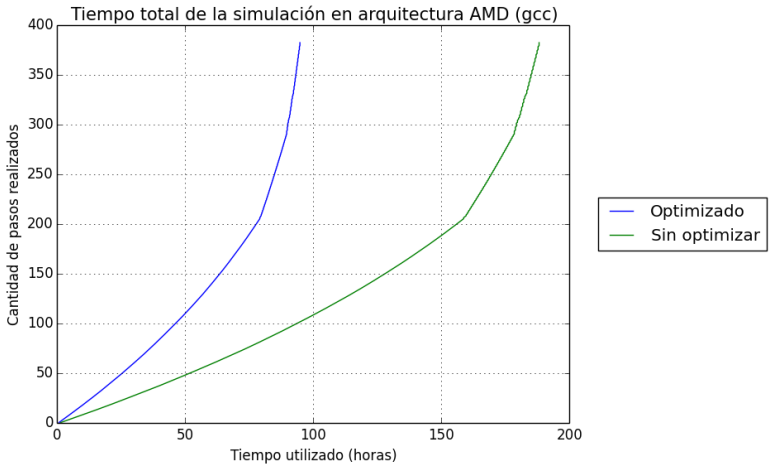

Figura 5.7: Tiempo de ejecución para la arquitectura AMD sin optimizaciones

Sin embargo, en el caso de AMD, es en extremo notoria la diferencia; en primer lugar la simulación no logró ser terminada luego de 7 días de ejecución, alcanzado el paso número 382 de un total de 2788, mientras que en el caso patrón la ejecución completa se realizó en poco más de 5 días.

5.5. Cambio de compiladores

Ante la disponibilidad de otra suite de herramientas para realizar la compilación del código, en este caso la suite de compiladores de Intel, se decidió utilizarla a fin de observar que mejoras se obtenían. Para poder llevar el experimento adelante, nuevamente fue necesario compilar Gadget y todas sus dependencias con las banderas correspondientes. En el gráfico 5.8 pueden verse los resultados obtenidos en la ejecución en el nodo Intel.

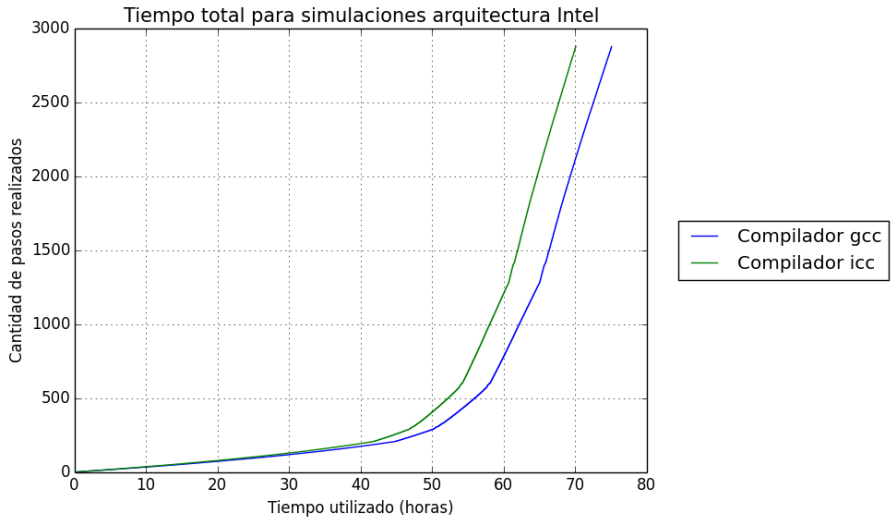

Figura 5.8: Tiempo de ejecución para la arquitectura Intel con distintos compiladores

Si bien puede observarse una mejora en el tiempo, lo cual era esperable teniendo en cuenta los resultados obtenidos con STREAM en la sección de caracterización, la mejora obtenida es en torno al 6 % lo cual dista de la medida anteriormente, que era en promedio del 30 %.

Por otro lado la misma prueba se realizó en el nodo AMD, donde debió ser nuevamente compilado Gadget y sus dependencias, en este caso siguiendo la guía de optimizaciones provista por AMD [1] para esta arquitectura. Lamentablemente, en este caso no fue posible obtener resultados concretos debido a que la ejecución fue lanzada en repetidas ocasiones sin conseguir

que ninguna de ellas llegara a terminar de manera correcta, en algunos casos por cortes de suministro eléctrico y en otros por una excesiva extensión en el tiempo de ejecución. De los tiempo parciales observados, puede notarse un tremendo incremento en el tiempo de ejecución. Investigando sobre este tema, se puede observar que el compilador `icc` realiza una comprobación del procesador sobre el cual está realizando la compilación. Al detectar que no se trata de un procesador Intel, desactiva todas las optimizaciones y genera un archivo binario muy general, motivo por el cual el código se ejecutó de manera similar a la realizada con el compilador `gcc` sin optimizaciones.

5.6. Cambio de hardware

En vista de los resultados obtenidos, teniendo en cuenta la disponibilidad de mayor cantidad de núcleos, aunque no de unidades de punto flotante y buscando reducir el tiempo total de espera por parte del usuario, se decidió hacer la ejecución de código con las mismas banderas de la prueba patrón, pero esta vez ejecutando sobre el total de unidades de cómputo disponibles en el nodo. El gráfico 5.9 ilustra los resultados.

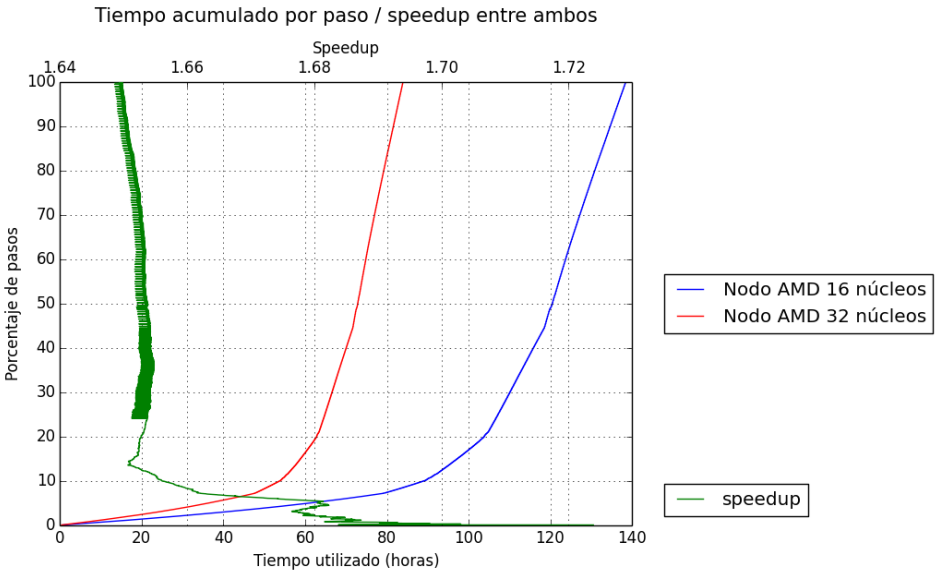

Figura 5.9: Tiempo de ejecución para la arquitectura AMD con 16 y 32 núcleos

Como puede verse de manera muy clara, se consigue una notable mejora del 60 %. Aunque no se consigue una mejora del 100 % como se esperaría

habitualmente al incrementar al doble la cantidad de núcleos en un código que ha sido largamente probado y se sabe que para esta pequeña cantidad de núcleos escala de manera lineal.

Este comportamiento hace suponer que, la unidad de punto flotante de cada módulo tienen alta latencia, por lo que es importante mantenerla siempre con trabajo pendiente, a fin de poder esconder de alguna manera dicha latencia.

Ante este resultado se decide compilar nuevamente el código de Gadget, agregando a las banderas de compilación, `-mprefer-avx128`. Este cambio se hace, esperando que al utilizarse la mitad del ancho del vector, este sea capaz de computar mayor cantidad de elementos por unidad de tiempo. Sin embargo el resultado obtenido no refleja ese comportamiento, sino que mantiene el mismo que al utilizar el ancho de vector de 256 bits.

Este caso es un poco más llamativo, ya que con el doble de núcleos no se obtiene una mejora en el tiempo de ejecución, sino que por el contrario, puede verse como la arquitectura Intel se mantiene por debajo de AMD en el tiempo de ejecución, a pesar de reducirse la brecha que existía entre ambos.

Para realizar una experiencia similar en el nodo Intel, se decidió encender el sistema de HyperThreading. El gráfico 5.10 muestra los resultados.

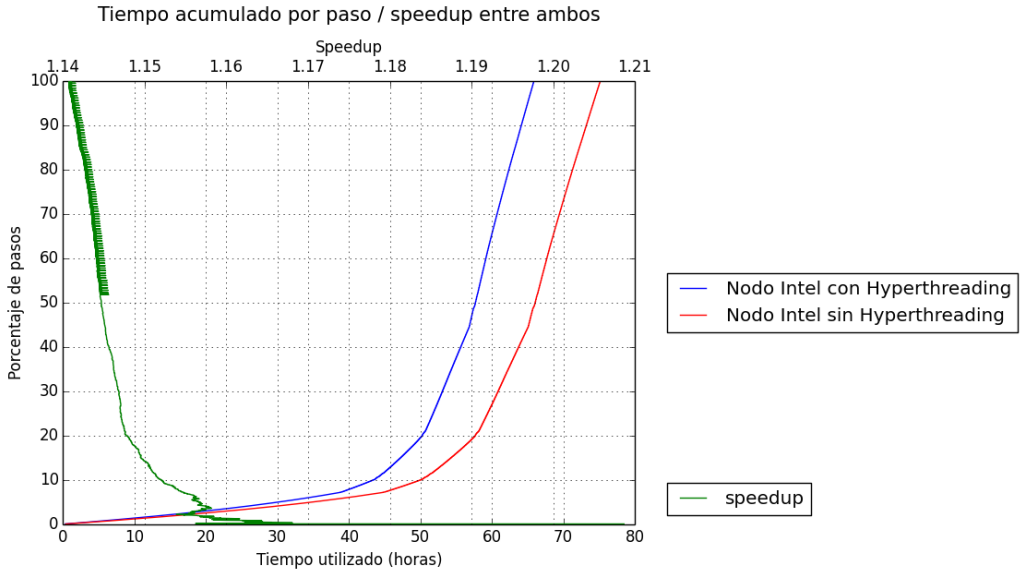

Figura 5.10: Tiempo de ejecución para la arquitectura Intel HT y sin HT

Se puede observar que se obtiene una mejora. Sin embargo, al igual que ocurrió con el nodo AMD, no se obtiene una reducción equivalente a la cantidad de núcleos en los que se ejecuta. Aunque puede ser un dato de utilidad al

realizar una simulación de grandes dimensiones, ya que a pesar de tener una ganancia baja en términos de aceleración, resulta una ganancia muy aceptable en términos de tiempo, teniendo en cuenta que en este caso la ejecución demoró 9 horas menos.

En el gráfico 5.11 se puede ver un concepto que optamos por llamar speedup instantáneo, consiste en agrupar los tiempos en grupos de 17 valores y sacar el promedio de cada uno, luego se divide el tiempo para obtener la comparación de speedup.

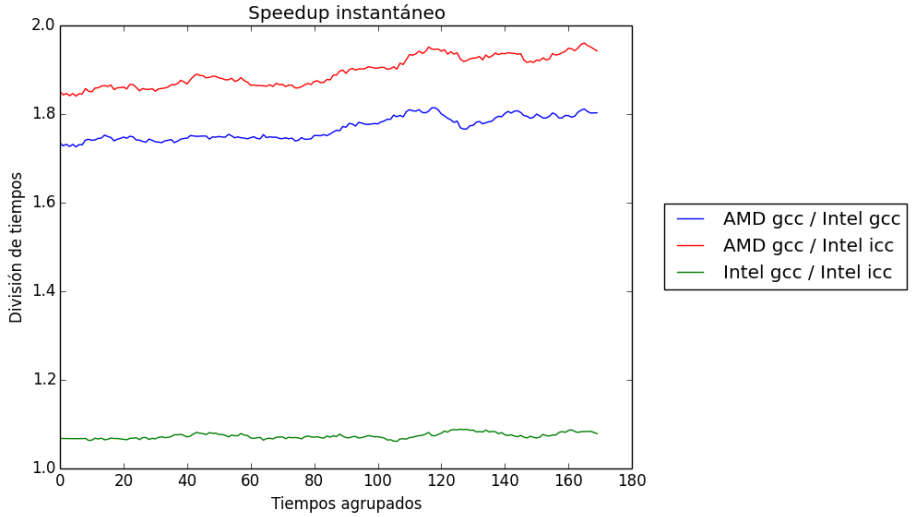

Figura 5.11: Diferencias de velocidad entre distintas ejecuciones

Resulta interesante analizar este gráfico al poder observarse la diferencia entre arquitectura y compiladores. Se obtiene una mejora notable en el uso del compilador `gcc` sobre arquitectura Intel respecto de AMD. Así mismo, también es de resaltar que no hay gran diferencia, sobre arquitectura Intel, en el uso de distintos compiladores, lo que quiere decir, que al menos en la ejecución, no tendría gran sentido invertir en la compra del compilador de Intel.

5.7. Ejecuciones en simultáneo

Un interrogante que surgió de medir el desempeño de los distintos equipos, tanto de manera específica como a través de la ejecución de Gadget y teniendo presente cuanto tiempo ocupa cada sección del código, fue cronometrar la ejecución en simultáneo de dos instancias de Gadget. En el gráfico 5.12 pueden verse comparados el tiempo de una sola instancia de Gadget, la

que denominamos ejecución patrón, y el tiempo de dos instancias de Gadget trabajando en el mismo nodo. Queda a la vista que el tiempo no se duplica,

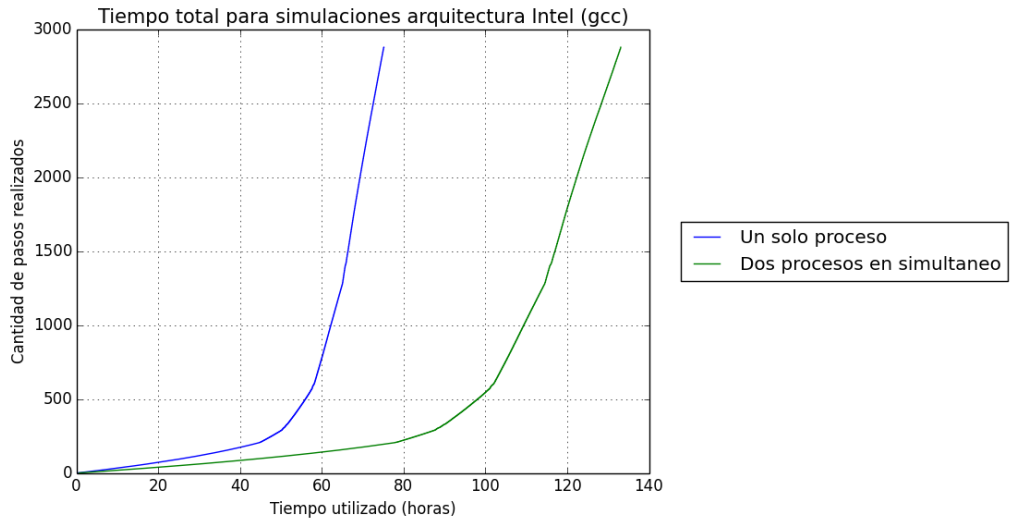

Figura 5.12: Ejecución de dos instancias simultáneas de Gadget en nodo Intel

solo tarda un 77 % más, lo que casualmente coincide casi de manera exacta con el desbalanceo informado por Gadget del 25.3 %. La diferencia de tiempo entre ambas ejecuciones corriendo en simultáneo es de 20 segundos, teniendo en cuenta que el tiempo total es de 133 horas, esta diferencia resulta insignificante.

5.8. Resumen de ejecuciones

En la tabla 5.1 podemos ver un resumen de todas las ejecuciones realizadas.

El tiempo total indicado en la entrada de la tabla señalado con AMD* debió ser estimado, ya que dichas ejecuciones no consiguieron ser finalizadas por cuestiones de suministro eléctrico. La entrada indicada como AMD32 hace referencia a la ejecución realizada con los todos los núcleos del nodo. Para el caso de la arquitectura AMD, donde esta indicado `-march -mtune` corresponde a `-march=bdver1 -mtune=bdver1`; para el caso de la arquitectura Intel corresponde a `-march=sandybridge -mtune=sandybridge`.

Arquitectura	Compilador	Banderas	Tiempo	Speedup
Intel	`gcc`	`-march -mtune -O3`	75.16	1
AMD	`gcc`	`-march -mtune -O3`	138.52	1.842
Intel	`icc`	`-xHost -ipo -O3`	70.12	0.932
Intel	`gcc`	`-O0`	78.01	1.037
AMD*	`gcc`	`-O0`	275.00	3.658
AMD 32	`gcc`	`-march -mtune -O3`	83.95	1.117
Intel HT	`gcc`	`-march -mtune -O3`	65.89	0.876
AMD* Superpuesto	`gcc`	`-march -mtune -O3`	163.66	2.177
Intel Superpuesto	`gcc`	`-march -mtune -O3`	133.05	1.770

Tabla 5.1: Resumen todas las ejecuciones realizadas

5.9. Discusiones

La principal discusión surgida durante la elaboración de este trabajo tiene que ver con el aprovechamiento del ancho de banda de memoria, cuello de botella en todas las ejecuciones. Teniendo en cuenta que el recorrido del árbol es la principal tarea que realiza el código y se lleva el 70 % del tiempo de ejecución, termina siendo una barrera muy importante que debe ser tenida en cuenta al momento de elegir el nuevo equipamiento a ser adquirido. Este parámetro no debe ser solo aplicado a Gadget sino a cualquier tipo de aplicación que, al igual que Gadget, esté altamente ligado a memoria.

Por otro lado también resulta de importancia analizar el impacto del desbalanceo de carga en la ejecución del código, que se encuentra alrededor del 25 %. Como pudo verse en las ejecuciones realizadas, a mayor es la carga que se genera sobre el nodo, menor es tiempo que tarda el mismo en terminar la ejecución, contrario a lo que uno esperaría, por ejemplo, en un programa que tiene alta demanda de procesador, conocido en inglés como *cpu-bound*. Este comportamiento pudo verse al ejecutar el código de Gadget sobre los 32 núcleos disponibles del nodo AMD y al utilizar el nodo Intel con HyperThreading, el cual quedó evidenciado en la ejecución que se realizó superponiendo dos simulaciones en simultáneo sobre el mismo nodo. En este punto resulta notable que las mediciones que se hicieron con *DGEMM* y *STREAM*, donde se vio que el desempeño se incrementa al exigir en mayor medida al equipo con arquietectura AMD. Esto puede deberse a que, utilizando todos los recursos disponibles, se obtiene una mejora en el uso de los canales de memoria o de los controladores de la misma, ya que como se describió previamente, esta arquitectura cuenta con dos controladores y dos canales por cada controlador en cada procesador. A pesar de esto, el peor desempeño medido en

la arquitectura Intel siempre resulta superior a la arquitectura AMD.

Igualmente importante resultó, sobre todo en el caso del nodo AMD, tener en cuenta utilizar la banderas de compilación adecuadas al momento de generar el binario, como pueden ser `-march=bdver1 -mtune=bdver1` en el caso de estar utilizando el compilador `gcc`. También en este punto pudo verse, hecho que resulta notable en los gráficos que ilustran la ejecución, lo importante de utilizar dichas banderas en la compilación de dependencias, caso contrario, por más que el código pueda ser optimizado su ejecución se verá afectada en desempeño por las funciones externas que el código utiliza. Ejemplos de esto pueden verse en la compilación de *OpenBLAS*, donde uno de los parámetros específicos indica la cantidad total de núcleos sobre los que se utilizará. En casos como puede ser *FFTW*, debe analizarse el tipo de procesos a ejecutar y configurar la misma con `-mprefer-avx128` según corresponda.

Adicionalmente se hicieron análisis de la ejecución con la herramienta `perf` determinando cuales eran las funciones más frecuentemente utilizadas. Una vez identificadas, se realizó un examen del código *assembly* obtenido a través de `gcc` y se pudo determinar que no se estaban utilizando instrucciones *AVX* o *SSE*. Se probaron algunos cambios sobre el código para ver si se observaban cambios, pero el resultado fue siempre negativo, lo que lleva a pensar que posiblemente ciertas partes del código original, deban ser reescritas para aprovechar las unidades vectoriales.

Capítulo 6

Conclusiones

De este trabajo de desprenden varias conclusiones. En primer lugar pudo observarse que los equipos con arquitectura Intel tienen un mejor manejo de memoria respecto de los AMD, sobre todo para casos como el de Gadget, que realizan grandes recorridos de estructuras de árbol en memoria y no hacen gran aprovechamiento de la memoria cache.

También es necesario resaltar la importancia de las banderas que se le pasan a los compiladores, no solo al momento de construir el binario de Gadget, sino también cuando se compilan las dependencias necesarias para el mismo, debido a que un bajo desempeño en, por ejemplo FFTW, afectará a la ejecución completa del código.

De la misma manera hay que tener en cuenta, en el caso de querer utilizar el compilador `icc` de la suite Intel sobre una plataforma AMD, es necesario revisar la documentación provista por el fabricante del procesador, para que el binario obtenido resulte correctamente compilado aprovechando todas las funciones disponibles en la arquitectura y no un binario general que terminará en un aumento del tiempo total de ejecución.

En relación a este punto, también cabe resaltar que las compilaciones realizadas con `icc` no dieron una marcada aceleración al momento de ejecutarlo. En base a este resultado podría cuestionarse si siempre es verdaderamente necesario desembolsar el costo de las licencias de la suite Intel. Según los resultados de este trabajo, puede decirse que no.

Otro ítem a tener en cuenta es la vectorización, en el caso de Gadget se realizaron algunos análisis del código *assembly* que producen los compiladores y pudo observarse que no son capaces de vectorizar de manera automática, por lo que en algún momento debe plantearse la necesidad de escribir nuevamente el código con éstas mejoras de manera manual o modificar lo necesario del código actual para que vectorice automáticamente. De la misma manera resulta notable la necesidad de prestar especial atención en la función de des-

composición de dominio, ya que, como se vió en la presentación del código, el resultado de la misma tiene un impacto directo en la distribución de carga sobre los procesadores y la evolución de la misma a lo largo de la ejecución. Es recomendable, al momento de ejecutar el código en una nueva máquina, prestar especial atención al parámetro de desbalanceo o *imbalance* en inglés, para poder realizar los ajustes necesarios.

Por otro lado la carga de los procesadores mostró funcionar diferente en ambos casos. El caso de AMD fue uno de los resultados que más resaltó, ya que mejoró el desempeño del mismo al utilizar el módulo completo y no utilizando la mitad del mismo para tener total disponibilidad de la unidad de punto flotante. Casi por el contrario en el caso de Intel al activar el sistema de HyperThreading, si bien mejoró levemente el desempeño, no se logró una diferencia tan notable. En ambos casos, usando todos los núcleos AMD o utilizando el sistema de HyperThreadng de Intel hay que resaltar que pudo reducirse el tiempo de ejecución, como pudo verse en la sección de ejecuciones.

En las proyecciones a futuro, se deben abordar varios aspectos. Uno de los principales, al momento de adquirir nuevo equipamiento, se debe tener en cuenta cuando se los elija que los mismos tengan el mayor ancho de banda posible en memoria, no solo en los microprocesadores, sino tener en cuenta la cobertura total de los canales de memoria disponibles en la placa madre. Por otro lado, los nuevos procesadores cuentan cada vez con más y mejoras conjuntos de instrucciones vectoriales, lo que el código de Gadget debería ser adecuado a dichas instrucciones. En el momento que la tercera versión de Gadget sea liberada, debe ser evaluada respecto de su desbalanceo y función de descomposición de dominio, para evaluar si es necesario encender el sistema de HyperThreading a fin de reducir el tiempo total de ejecución. Como complemento a estos puntos también es necesario resaltar la necesidad de utilizar compiladores actualizados, que estén en capacidad de utilizar de la mejor manera posible los recursos del equipamiento utilizado.

Resulta de interés analizar en profundidad algunos aspectos del programa con herramientas como likwid[1] o perf[2] en busca de posibles cuellos de botella o deficiencias en la ejecución. Este trabajo podría ser continuado en un futuro, ya que escapa a los objetivos originalmente planteados en esta tesis.

[1] https://github.com/RRZE-HPC/likwid
[2] https://perf.wiki.kernel.org/index.php/Main_Page

Bibliografía

[1] AMD. Compiler options quick reference guide. http://developer.amd.com/wordpress/media/2012/10/CompilerOptQuickRef-62004200.pdf.

[2] Anandtech.com. Intel broadwell architecture. https://www.anandtech.com/show/8355/intel-broadwell-architecture-preview/2.

[3] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[4] Instituto de Astronomía Teórica y Experimental CONICET. IATE - Cluster Mirta. http://iate.oac.uncor.edu.

[5] Universidad Nacional de Córdoba. CCAD - Cluster Mendieta. http://ccad.unc.edu.ar/.

[6] Jack Dongarra. What's HPL? http://www.netlib.org/utk/people/JackDongarra/faq-linpack.html#_What_is_HPL?

[7] Matteo Frigo and Steve G. Johnson. FFTW Fastest Fourier Transform in the west. http://www.fftw.org.

[8] Max Planck Gesellschaft. Max-Planck Institut für Astrophysik. http://www.mpa-garching.mpg.de/.

[9] GNU Project. GCC Compiler Collection. https://gcc.gnu.org.

[10] Smithsonian Institution Harvard University. Harvard-Smithsonian Center for Astrophysics. https://www.cfa.harvard.edu/.

[11] Intel. Intel hyper-threading technology. https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

[12] Tandred Lindholm. N-body algorithms. http://www.cs.hut.fi/ ctl/NBody.pdf.

[13] Dan Luu. New cpu features. https://danluu.com/new-cpu-features/, Enero 2015.

[14] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[15] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[16] Giuseppe Peano and David Hilbert. Peano-Hilbert filling curve. https://en.wikipedia.org/wiki/Space-filling_curve.

[17] GNU Project. Gnu scientific library. https://www.gnu.org/software/gsl.

[18] MPICH Project. High-performance portable mpi. http://http://www.mpich.org.

[19] David Patterson Samuel Williams, Andrew Waterman. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM - A Direct Path to Dependable Software*, pages 65–76, april 2009.

[20] SchedMD. SLURM. https://slurm.schedmd.com.

[21] Intel Software. Parallel Studio XE. https://software.intel.com/en-us/parallel-studio-xe.

[22] Volker Springel. Gadget-2. http://wwwmpa.mpa-garching.mpg.de/gadget/.

[23] Volker Springel. Summer school on cosmological numerical simulations. http://www.aip.de/summerschool2006/lectures/volker/monday.pdf.

[24] Open MPI Team. A high performance message passing library. https://www.open-mpi.org/.

[25] techpowerup.com. Intel sandy bridge (microarchitecture). https://www.techpowerup.com/forums/attachments/sb_pipeline-jpg.47843/.

[26] Techreport.com. Intel broadwell architecture. https://techreport.com/review/26896/intel-broadwell-processor-revealed/3.

[27] top500 List. top500 list. https://www.top500.org.

[28] wikipedia.org. Amd bulldozer (microarchitecture). https://en.wikipedia.org/wiki/Bulldozer_(microarchitecture).

[29] wikipedia.org. Intel hyper-threading technology. https://en.wikipedia.org/wiki/Hyper-threading.

[30] Zhang Xianyi. OpenBLAS. https://github.com/xianyi/OpenBLAS.

Los abajo firmantes, miembros del tribunal de evaluación de tesis, damos fe que el presente ejemplar impreso se corresponde con el aprobado por este tribunal